THE WINDOW OBJECT:

Every JavaScript environment has a **global object**. Any variables that are created in the global scope are actually properties of this object, and any functions are methods of it.

the global object is the window object, which represents the browser window that contains a web page.

THE BROWSER OBJECT MODEL:

There is no official standard for the BOM, although there are a number of properties and methods that are supported by all the major browsers, making a sort of de facto standard. These properties and methods are made available through the window object. Every browser window, tab, popup, frame, and iframe has a window object.

Captura de pantalla de un celular

Descripción generada automáticamente

GOING GLOBAL:

DIALOGS:

BROWSER INFORMATION:

WHICH BROWSER?

LOCATION, LOCATION, LOCATION:

The window.location property is an object that contains information about the URL of the current page. It contains a number of properties that provide information about different fragments of the URL.

The href property returns the full URL as a string:

Texto

Descripción generada automáticamente con confianza baja

This property (as well as most of the others in this section) is a read/write property, which means it can also be changed by assignment. If this is done, the page will be reloaded using the new property. For example, entering the following line into the browser console will redirect the page to the SitePoint JavaScript channel:

Texto

Descripción generada automáticamente

The protocol property returns a string describing the protocol used (such as http, https, pop2, ftp etc.). Note that there is a colon (:) at the end:

Texto

Descripción generada automáticamente

The host property returns a string describing the domain of the current URL and the port number (this is often omitted if the default port 80 is used):

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

The hostname property returns a string describing the doma

in of the current URL:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

The port property returns a string describing the port number, although it will return an empty string if the port is not explicitly stated in the URL:

Texto

Descripción generada automáticamente

The pathname property returns a string of the path that follows the domain:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

The search property returns a string that starts with a “?” followed by the query string parameters. It returns an empty string if there are no query string parameters. This is what I get when I search for “JavaScript” on SitePoint:

Texto

Descripción generada automáticamente

The hash property returns a string that starts with a “#” followed by the fragment identifier. It returns an empty string if there is no fragment identifier:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

The origin property returns a string that shows the protocol and domain where the current page originated from. This property is read-only, so cannot be changed:

Texto

Descripción generada automáticamente

The window.location object also has the following methods:

* The reload() method can be used to force a reload of the current page. If it’s given a parameter of true, it will force the browser to reload the page from the server, instead of using a cached page.
* The assign() method can be used to load another resource from a URL provided as a parameter, for example:

Texto

Descripción generada automáticamente

* The replace() method is almost the same as the assign() method, except the current page will not be stored in the session history, so the user will be unable to navigate back to it using the back button.
* The toString() method returns a string containing the whole URL:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

THE BROWSER HISTORY:

The window.history property can be used to access information about any previously visited pages in the current browser session. Avoid confusing this with the new HTML5 History API. (See http://www.sitepoint.com/javascript-history-pushstate/ post for details.)

The window.history.length property shows how many pages have been visited before arriving at the current page.

The window.history.go() method can be used to go to a specific page, where 0 is the current page:

Texto

Descripción generada automáticamente

There are also the window.history.forward() and window.history.back() methods that can be used to navigate forwards and backwards by one page respectively, just like using the browser’s forward and back buttons.

CONTROLLING WINDOWS:

A new window can be opened using the window.open() method. This takes the URL of the page to be opened as its first parameter, the window title as its second parameter, and a list of attributes as the third parameter. This can also be assigned to a variable, so the window can then be referenced later in the code:

Texto

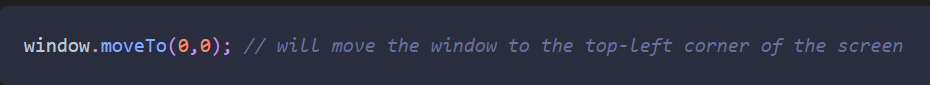
Descripción generada automáticamente

The close() method can be used to close a window, assuming you have a reference to it:

Texto

Descripción generada automáticamente con confianza media

It is also possible to move a window using the window.moveTo() method. This takes two parameters that are the X and Y coordinates of the screen that the window is to be moved to:



You can resize a window using the window.resizeTo() method. This takes two parameters that specify the width and height of the resized window’s dimensions:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

SCREEN INFORMATION:

The window.screen object contains information about the screen the browser is displayed on. You can find out the height and width of the screen in pixels using the height and width properties respectively:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media

The availHeight and availWidth can be used to find the height and width of the screen, excluding any operating system menus:

Texto

Descripción generada automáticamente

The colorDepth property can be used to find the color bit depth of the user’s monitor, although there are few use cases for doing this other than collecting user statistics:

Texto

Descripción generada automáticamente

THE DOCUMENT OBJECT:

The write() method simply writes a string of text to the page. If a page has already loaded, it will completely replace the current document:

This would replace the whole document with the string Hello, world!. It is possible to include HTML in the string and this will become part of the DOM tree. For example, the following piece of code will create an <h1> tag node and a child text node:

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

The document.write() method can also be used within a document inside <script> tags to inject a string into the markup. This will not overwrite the rest of the HTML on the page. The following example will place the text "Hello, world!" inside the <h1> tags and the rest of the page will display as normal:

Texto

Descripción generada automáticamente

COOKIES:

A restriction of cookies is that they can only be read by a web page from the same domain that set them. This is to stop sites being able to access information about users, such as other sites they have visited. Cookies are also limited to storing up to 4KB of data, although 20 cookies are allowed per domain, which can add up to quite a lot of data.

Cookies can be used for personalizing a user’s browsing experience, storing user preferences, keeping track of user choices (such as a shopping cart), authentication and tracking users. The use of cookies for tracking purposes has been much maligned in recent years. Their use for data storage is starting to be replaced in many cases by the new HTML5 localStorage API as it allows more data to be stored. This is covered in Chapter 14. Cookies are still useful for retaining state information (such as if a user is logged in) because they’re passed between the client and server on every HTTP request.

Cookies take the form of a text file that contain a list of name/value pairs separated by semicolons. For example, a cookie file might contain the following information:

CREATING COOKIES:

To create a cookie, you assign it to JavaScript’s “cookie jar”, using the document.cookie property, like so:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

The document.cookie property acts like a special type of string. Assigning another cookie to it won’t overwrite the entire property, it will just append it to the end of the string. So we can add more cookies by assigning them to document.cookie:

CHANGING COOKIE VALUES:

A cookie’s value can be changed by reassigning it to document.cookie using the same name but a different value. The following code will update the value of two of the cookies that we set in the previous section:

READING COOKIES:

To see the current contents of the cookie jar, simply enter document.cookie:

We can use the split method to break the string into an array containing each name/value pair, then use a for of loop to iterate through the array:

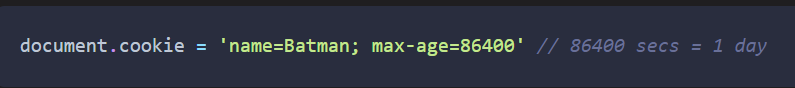
COOKIE EXPIRY DATES:

Cookies are session cookies by default. This means they are deleted once a browser session is finished (when the user closes the browser tab or window). Cookies can be made persistent ― that is, lasting beyond the browser session ― by adding "; expires=date" to the end of the cookie when it’s set, where date is a date value in the UTC String format Day, DD-Mon-YYYY HH:MM:SS GMT. The following example sets a cookie to expire in one day’s time:

Una captura de pantalla de un celular

Descripción generada automáticamente

An alternative is to set the max-age value. This takes a value in seconds, but it wasn’t supported in Internet Explorer before version 10:



THE PATH AND DOMAIN OF COOKIES:

By default, cookies can only be read by pages inside the same directory and domain as the file was set. This is for security reasons so that access to the cookie is limited.

The path can be changed so that any page in the root directory can read the cookie. It’s done by adding the string ; path=/ to the end of the cookie when it is set:

Texto

Descripción generada automáticamente

It’s also possible to set the domain by adding "; domain=domainName" to the end of the cookie:

Texto

Descripción generada automáticamente

A cookie can only be read by the domain that created it anyway, but doing this will allow all subdomains of sitepoint.com (such as javascript.sitepoint.com and books.sitepoint.com) to read it.

SECURE COOKIES:

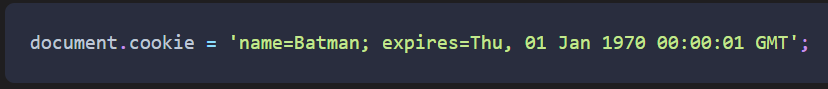
Adding the string ; secure to the end of a cookie will ensure it’s only transmitted over a secure HTTPS network:

Texto

Descripción generada automáticamente con confianza baja

DELETING COOKIES:

To remove a cookie, you need to set it to expire at a time in the past:



TIMING FUNCTIONS:

SETTIMEOUT():

The window.setTimeout() method accepts a callback to a function as its first parameter and a number of milliseconds as its second parameter. Try entering the following example into a console. It should show an alert dialog after three seconds (that’s 3000 milliseconds):

Texto

Descripción generada automáticamente

Notice that the method returns an integer. This is an ID used to reference that particular timeout. It can also cancel the timeout using the window.clearTimeout() method. Try calling the code again and make a note of the number that is returned:

SETINTERVAL():

The window.setInterval() method works in a similar way to window.setTimeout(), except that it will repeatedly invoke the callback function after every given number of milliseconds.

The previous example used an anonymous function, but it is also possible to use a named function like so:

Now we can set up the interval and assign it to a variable:

*const* summon = window.setInterval(chant,1000);<< 6

This should show the message “Beetlejuice” in the console every second (1,000 milliseconds).

To stop this, we can use the window.clearInterval() method and the variable repeat as an argument (this is because the window.setInterval() method returns its ID, so this will be assigned to the variable repeat):

window.clearInterval(summon);

Be careful when using a method that uses the this keyword with either of these timing methods. The binding of this is set to the window object, rather than the method’s object, so it can get some unexpected results:

ANIMATION:

REQUESTANIMATIONFRAME:

This method of the window object works in much the same way as the window.setInterval() method, although it has a number of improvements to optimize its performance. These include making the most of the browser’s built-in graphics-handling capabilities, and not running the animation when the tab is inactive, resulting in a much smoother performance. It’s supported in all major browsers, including Internet Explorer from version 10 onwards. Change the code in main.js to the following:

This is similar to the earlier code, but this time we place the rotation code inside a function called rotate. The last line of this function uses the window.requestAnimationFrame() method and takes the rotate() function as an argument. This will then call the rotate() function recursively. The frame rate cannot be set using requestAnimationFrame(); it’s usually 60 frames per second, although it’s optimized for the device being used.

To start the animation, we need to call the requestAnimationFrame() method, giving the rotate() function as an argument. This will return a unique ID that can be employed to stop the animation using the window.cancelAnimationFrame() method:

HTML5 API’s:

HTML5 API’S:

HTML5:

THE DATA- ATTRIBUTE:

The data- attribute is a way of embedding data in a web page using custom attributes that are ignored by the browser. They’re also private to a page, so are not intended to be used by an external service – their sole purpose is to be used by a JavaScript program. This means they’re perfect for adding data that can be used as a hook that the program utilizes to access information about a particular element on the page.

Texto

Descripción generada automáticamente

The information contained in the attributes can be used to identify particular elements. For example, all the elements with an attribute of data-dropdown could be identified as dropdown menu. The values of the attributes can also be used to filter different elements. For example, we could find all the elements that have a data-rating value of 3 or more.

Each element has a dataset property that can be used to access any data- attributes it contains. Here’s an example of some markup:

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza baja

The data-powers attribute can be accessed using the following code:

Texto

Descripción generada automáticamente

Notice that the data- prefix is dropped. To access the attribute, powers is used as if it’s a property of the dataset object. If a data- attribute’s name contains hyphens, they are replaced with camel-case notation, so data-max-length would be accessed using dataset.maxLength.

The support for the data- attribute is generally very good in modern browsers. Even Internet Explorer 8 has partial support! Some older browsers are unable to understand the dataset property, however, but any data- attribute can be found using the standard getAttribute method. So the previous code could be replaced with the following if you still need to support older browsers:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

The restriction of only using a string value can be overcome by encoding any JavaScript object or value as a JSON string, then performing type-conversion later, as required. For example, the value of data-max-length will return a string, but can easily be converted into a number using the following code:

Texto

Descripción generada automáticamente con confianza media

Data attributes provide a convenient way of adding data directly into the HTML markup, enabling a richer user experience. More information

QUESTIONS:

IS THIS BETTER THAN USING CLASSES OR ID’S TO ACCESS ELEMENTS?

HTML5 APIS:

HTML5 WEB STORAGE:

The Web Storage API has some crucial differences with cookies:

* Information stored is *not* shared with the server on every request.
* Information is available in multiple windows of the browser (but only if the domain is the same).
* Storage capacity limit is much larger than the 4KB limit for cookies ( There is no actual limit in the specification, but most browsers have a limit set at 5GB per domain.).
* Any data stored does not automatically expire as it does with cookies. This potentially makes cookies a better choice for something like showing a popup once a day.

There is also a sessionStorage object that works in the same way, although the data is only saved for the current session.

Every time a value is saved to local storage, a storage event is fired. Note that this event is only fired on any other windows or tabs from the same domain, and only if the value of the item being saved changes. The event object sent by the event listener to the callback has a number of properties that provide information about the updated item:

* key tells us the key of the item that changed
* newValue tells us the new value to which it has been changed
* oldValue tells us the previous value before it was changed
* storageArea tells us if it is stored in local or session storage.

The code following will add an event listener that logs information about any changes to the Web Storage (note that this example won't work locally as it needs to be running on a server):

Texto

Descripción generada automáticamente

GEOLOCATION:

The Geolocation API is used to obtain the geographical position of the device. This means it can be used to find the user’s exact location, then link to nearby places or measure the speed at which the user is moving. This information can then be used to filter data based on the user's location or speed and direction of travel. An example of this might be a search function that returns results based on your location. Because of privacy concerns, permission to use this has to be granted by the user first.

If geolocation is available, it will be a property of the navigator object that we met in Chapter 9. This property has a method called getCurrentPosition() that will return a position object to a specified callback function, called youAreHere() in the example:

The position object passed to the youAreHere() function has a coords property with a latitude and longitude property, which together give the coordinates of the device. These coordinates can then be used in conjunction with other applications or web services (such as a mapping service) to obtain the user’s exact location. In this example, we simply show an alert dialog that displays the user’s coordinates:

The position object has several other properties that can be used to find out information about the location and movement of the device:

* position.speed property returns the ground speed of the device in meters per second.
* position.altitude property returns an estimate of the device’s altitude in meters above the [WGS84](http://en.wikipedia.org/wiki/World_Geodetic_System) ellipsoid, which is a standard measurement for the center of the Earth.
* position.heading property returns the direction the device is moving in. This is measured as a bearing in degrees, clockwise from North.
* position.timestamp property returns the time that the position information was recorded.

The position object also has properties that calculate the accuracy of the measurements. These can be useful, as sometimes you only need to know the town or city users are in, while at other times you may need their exact position. position.accuracy property returns the accuracy of the latitude and longitude properties in meters. The lower the returned value, the more accurate the measurements are, as is the case for the position.altitudeAccuracy property, which returns the accuracy of the altitude property in meters.

In addition, the geolocation object has a watchPosition() method that will call a callback function every time the position of the device is updated. This method returns an ID that can be used to reference the position being watched:

The clearWatch() method can be used to stop the callback being called, using the ID of the watch as an argument:

WEB WORKERS:

To get started, use the Worker() constructor function to create a new worker:

This function takes the name of another JavaScript file as an argument. In the example, this is a file called 'task.js'. If this file exists, it will be downloaded asynchronously. The worker will only start once the file has finished downloading completely. If the file doesn’t exist, an error is thrown.

The variable that’s assigned to the constructor function (worker in our example) can now be used to refer to the worker in the main program. In the worker script ('task.js'), the keyword self is used to refer to the worker.

Web workers use the concept of messages to communicate back and forth between the main script and worker script.

The postMessage() method can be used to send a message and start the worker working. The argument to this method can send any data to the web worker. To post a message to the worker, the following code is used inside the main script:

worker.postMessage('Hello');

To post a message from the worker, the following is used in the worker script:

self.postMessage('Finished');

When a message is posted, a message event is fired, so they can be dealt with using an event listener. The data sent with the message as an argument is stored in the data property of the event object that’s passed to the callback function. The following example would log any data returned from the worker to the console:

Texto

Descripción generada automáticamente

When a worker has completed its task, it can be stopped using the terminate() method from within the main script:

worker.terminate();

Or using the close() method from inside the worker script:

self.close();

A FACTORIZING EXAMPLE:

Firstly, we create a new file called factors.js; and save it in the same folder as main.js. Then we remove the factorsOf() function from the main.js file and add it into our new factors.js; file. We’ll be adding more to this file later, but first we need to edit the factorize() function in the main.js file so it looks like the following:

Texto

Descripción generada automáticamente

Now we go back to the factors.js; file and add this event listener code to the end of the file:

Pantalla negra con letras blancas

Descripción generada automáticamente

This will fire when the worker receives a message, occurring when the form is submitted. The number to be factorized is stored in the event.data property. We use the factorsOf() function to find the factors of the number, then convert it into a string and send a message back containing the answer. We then use the close() method to terminate the worker, since its work is done.

Now if we test the code, it will still take a long time to factorize a long number, but the page will not freeze. You can also continue to change the background color while the factors are being calculated in the background.

SHARED WEB WORKERS:

The examples we have seen so far are known as dedicated web workers. These are linked to the script that loaded the worker, and are unable to be used by another script. You can also create shared web workers that allow lots of different scripts on the same domain to access the same worker object. [You can read more about shared web workers in this post on SitePoint.](http://www.sitepoint.com/javascript-shared-web-workers-html5/)

Web workers allow computationally complex operations to be performed in a separate thread, meaning that the flow of a program won’t suffer interruptions, and an application will not freeze or hang. They are a useful feature that help to keep sites responsive, even when complicated operations are being carried out. You can find [more information about them at the Mozilla Developer Network.](https://developer.mozilla.org/en-US/docs/Web/Guide/Performance/Using_web_workers)

SERVICE WORKERS:

The Service Worker API allows a worker script to run in the background with the added benefit of being able to intercept network requests. This allows you to take alternative action if the network is offline, and effectively create app-like offline experiences. Service workers also allow access to push notifications and background syncing. Service workers require a secure network to run on HTTPS to avoid any malicious code hijacking network requests.

WEBSOCKETS:

Websocket is a new protocol that allows two-way communication with a server – also known as push messaging. This means that a connection is kept open and responses are ‘pushed’ to the client as soon as they are received.

To see this in action, we’ll create a mini-messaging application that uses the websocket protocol to communicate with an Echo server. This sends a response that is exactly the same as the message it receives.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

This sets up some variables to store information. The first is URL, which is the URL we’ll be using to connect to the websocket. Notice that it starts 'wss://' instead of 'https://' This is the secure protocol used by websockets instead of HTTP. The site is the Echo server hosted at [websocket.org](http://websocket.org/). This accepts messages then returns the same message (like an echo).

When the code new WebSocket(URL) runs, it creates an instance of a WebSocket object and tries to connect to the URL. When this is successful, it fires an event called 'open'. This is one of a number of events that a WebSocket object can emit. To deal with it, we can add an event handler to main.js:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media

This works in the same way as the event handlers we’ve seen previously, and is called on the connection object. In this case, we call a function called output() with the string 'CONNECTED' provided as an argument. The output() is used to output messages to the screen. We need to add that function next:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

The last line is an important one. This calls a method of the connection object called send(). This sends the message to the URL that the websocket is connected to. When this message is received, the server will process it and send a response. The connection object waits for the response, and when it receives one, a 'message' event is fired. The 'echo.websocket.org' server simply responds with the same message, but any message could be processed in a variety of ways before sending a response.

Let's create an event handler to deal with the response:

Texto

Descripción generada automáticamente

This uses the event object that is provided as an argument to the event, and we can use the data property to access the data that was returned. It's then a simple case of using the output() function again to add this message to the growing stream of messages in the 'output'<div>, but this time with the phrase 'RESPONSE:' added to the beginning.

There are a couple of other events that the connection object responds to that are worth knowing about: The close event occurs when the connection is closed, which can be done using the close() method. The error event is fired when any sort of error occurs with the connection. The information about the error can be accessed in the data property of the event object.

Pantalla de computadora con letras

Descripción generada automáticamente

NOTIFICATIONS:

Before you can send notifications, you need to get permission granted by the user. This can be achieved using the requestPermission() method of a Notification global object. To try this out, visit any website in your browser (https://sitepoint.com for example), and enter the following code in the console:

Texto

Descripción generada automáticamente

This returns a promise with the permission property of the Notification object set to either 'granted' or 'denied'. If it’s set to granted, you can create a new notification using a constructor function, like so:

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

This will produce a system notification with the title 'Hello JavaScript!'.

The constructor function's first parameter is the title of the notification, and is required. The function also accepts a second parameter, which is an object of options. These include body that specifies any text that you want to appear below the title, and icon where you can specify a link to an image that will be displayed as part of the notification:

Texto

Descripción generada automáticamente

Depending on your browser and operating system, some notifications close automatically after a short period of time, and some will stay on the screen until the user clicks on them. You can close the notification programmatically using the close() method:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

The notification instance has a number of events that it can react to, including click (when a user clicks on it), show (when the notification appears) and close (when the notification is closed).

Texto

Descripción generada automáticamente

MULTIMEDIA:

Before HTML5, it was notoriously difficult to display audio and video in browsers, and plugins such as Flash often had to be used. HTML5 introduced the <audio> and <video> tags used to insert audio and video clips into a web page. It also introduced a Media API for controlling the playback of the clips using JavaScript.

An audio clip can be inserted into a page with the <audio> tag, using the src attribute to point to the audio file:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

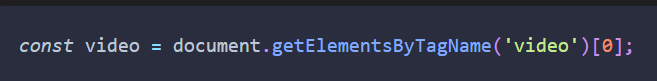
A video clip can be inserted with the <video> tag, using the src attribute to point to the movie file:

Texto

Descripción generada automáticamente

Any content inside the <audio> or <video> tags will only display if the browser does not support them; hence, it can be used to display a message to users of older browsers without support for these features. The controls attribute can be added (without any value) and will display the browser’s native controls, such as play, pause, and volume control, as can be seen in the screenshot below.

The audio or video element can be referenced by a variable using one of the DOM methods we saw in Chapter 6:



Audio and video elements have a number of properties and methods to control the playback of the clip.

The play() method will start the clip playing from its current position:

The pause() method will pause the clip at its current position:

The volume property is a number that can be used to set the audio volume:

video.volume = 0.9;

The muted property is a boolean value that can be used to mute the audio:

video.muted = true;

The currentTime property is a number value that can be used to jump to another part of the clip:

video.currentTime += 10; *// jumps forward 10 seconds*

The playbackRate property is used to fast-forward or rewind the clip by changing its value. A value of 1 is playback at normal speed:

video.playbackRate = 8; *// fast-forward at 8 times as fast*

The loop property is a boolean value that can be set to true to make the clip repeat in a loop:

video.loop = true;

The duration property can be used to see how long the clip lasts:

video.duration;<< 52.209

Some of the properties are only available once the browser has received all the metadata associated with the video. This means that, in order to ensure a value is returned, you should use an event listener that fires once the metadata has loaded, like the one shown below:

video.addEventListener('loadedmetadata', () => { console.log(video.duration); });

Audio and video clips also have a number of events that will fire when they occur, including:

* The play event, which fires when the clip starts and when it resumes after a pause.
* The pause event, which fires when the clip is paused.
* The volumechange event, which fires when the volume is changed.
* The loadedmetadata event, which we saw in the note above, and which fires when all the video's metadata has loaded.

These events allow you to respond to any interactions the user has with the video. For example, the following event listener can be added to check whether the user has paused the video:

Texto

Descripción generada automáticamente

OTHER APIS:

DRAWING WITH CANVAS:

Straight lines can be drawn employing the moveTo() and lineTo() methods. These methods can be used together to produce a path. Nothing will actually be drawn onto the canvas until the stroke() method is called. The following example will draw a thick red T shape onto the canvas by moving to the coordinates (150,50), then drawing a horizontal line 30 pixels long, and finally moving to the middle of that line and drawing a vertical line 40 pixels long:

Texto

Descripción generada automáticamente

The strokeRect() method works in the same way, but produces a rectangle that is not filled in. This will draw the outline of a rectangle underneath the last one:

context.strokeRect(10,100,100,50);

The fillText() method is used to write text onto the canvas. The first parameter is the text to be displayed, while the next two parameters are the x and y coordinates, respectively. The font property can be used to set the font style used, otherwise the style is inherited from the canvas element’s CSS setting (note that it needs to be changed before the fillText() method is used to draw the text). The following example will draw the text “Hello” in green at coordinates (20,50), as shown below.

The fillText() method is used to write text onto the canvas. The first parameter is the text to be displayed, while the next two parameters are the x and y coordinates, respectively. The font property can be used to set the font style used, otherwise the style is inherited from the canvas element’s CSS setting (note that it needs to be changed before the fillText() method is used to draw the text). The following example will draw the text “Hello” in green at coordinates (20,50), as shown below.

Texto

Descripción generada automáticamente

SHIMS AND POLYFILLS:

HTML5 APIs progress at a rapid rate ― new APIs are constantly being introduced, and existing APIs often change. Modern browsers are very quick to update and implement many of the changes, but you can’t always guarantee that users will have the most up-to-date browser. This is where a shim or a polyfill comes in handy. These are libraries of code that allow you to use the APIs as usual. They then fill in the necessary code that's not provided natively by the user’s browser.

The terms shim and polyfill are often used interchangeably. The main difference between them is that a shim is a piece of code that adds some missing functionality to a browser, although the implementation method may differ slightly from the standard API. A polyfill is a shim that achieves the same functionality, while also using the API commands that would be used if the feature was supported natively.

This means that your code can use the APIs as normal and it should work as expected in older browsers. The advantage here is that the same set of standard API commands can be used ― you don’t need to write additional code to deal with different levels of support. And when users update their browsers, the transition will be seamless, as their experience will remain the same. Once you are confident that enough users have up-to-date browsers, you can remove the polyfill code without having to update any actual JavaScript code.

**Managing the modern Front-end workflow**

Development workflow has become quite complicated for Web development. Lets take a medium sized project for example. It could have dozens of javascript files, several css files, 3rd party libraries, and who knows how many icons, fonts, images, etc are involved. It might be using a CSS preprocessor like SASS or it could be transpiling the Javascript to make sure that new features will work in older browsers.

A potential list of tasks that could need to happen everytime something changes is this:

1. Lint CSS and Javascript to find any coding issues.
2. Run unit tests to make sure nothing got broken with your last changes
3. compile all SCSS/LESS to CSS
4. concatinate all CSS into one file for faster loading,
5. Minify the CSS to reduce file size.
6. Transpile Javascript with Babel for wider support for older browsers
7. Concatinate all JS files into one file for faster loading
8. Minify and Uglify Javascript to reduce size
9. Do the same for any 3rd party CSS or JS
10. Move all the production assets into a distribution directory to separate them from the development stuff

It's no wonder that developers have produced tools to help manage all of this. This activity will introduce a simple implementation of some of these tools. The tools fall into three catagories:

1. Package managers: These keep track of all of the external dependencies for our app. Both development tools and libraries we might be using. It knows not only which packages to download...but tracks versions as well. We are using **npm** for our package manager.
2. Bundlers: bundlers handle the compiling, transpiling, concatinating, minifying, and moving around of assets in our project. We are using **Parcel** as our bundler. Another common bundler is Webpack
3. Task managers: these keep track of what needs to be done and when. There will generally be scripts defined in the task manager for each phase of development. Our project is fairly simple so we are just using **npm** again for task manager. Other common managers are Grunt or Gulp