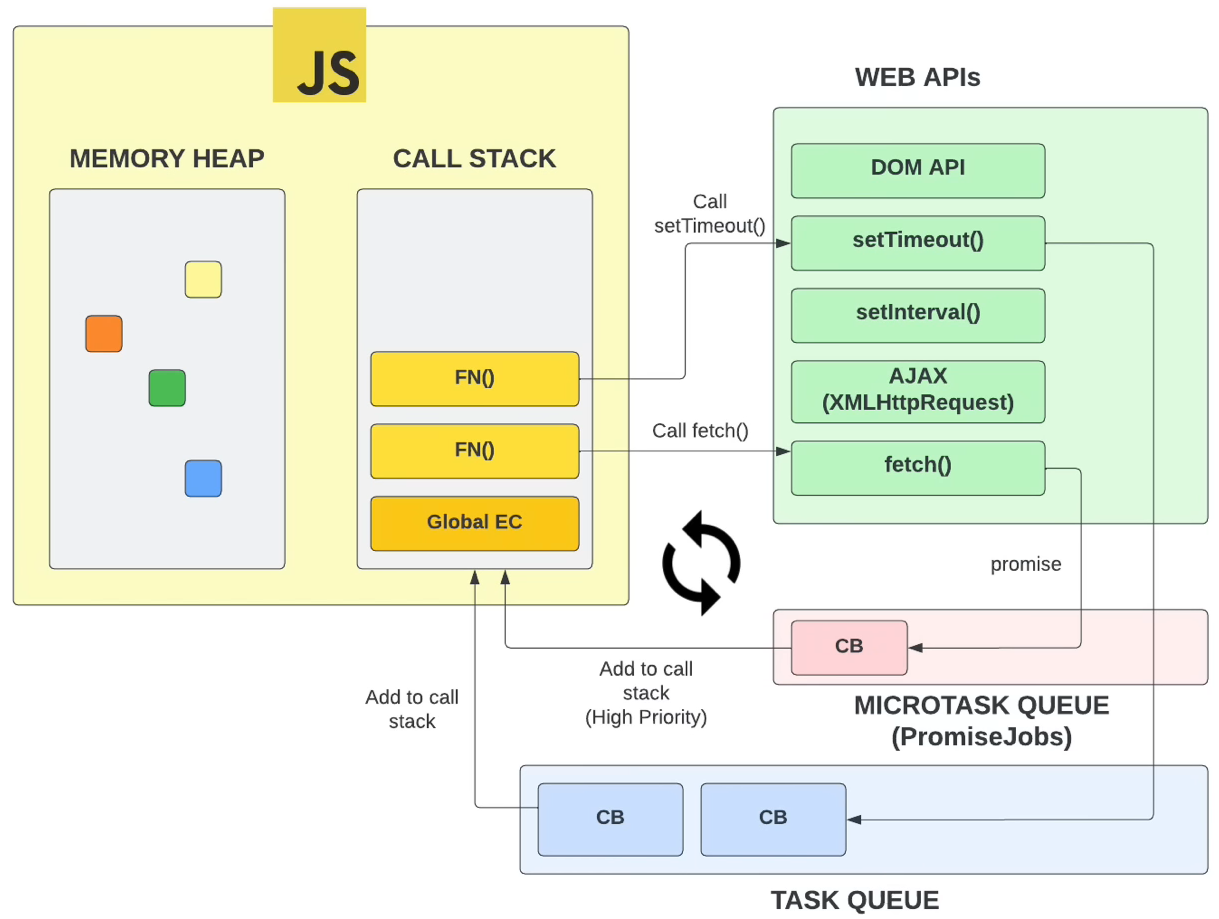
**Web APIs [**<https://developer.mozilla.org/en-US/docs/Web/API>**] &**

**Document Object Model (DOM) [**<https://www.w3schools.com/js/js_htmldom.asp>**]**

Last week we looked at the JS syntax and statements. As we discussed, JS can be used to implement code that runs at the browser (front-end programming), at the server, e.g., Node.js (back-end programming) (<https://nodejs.org/en/>), and even to implement desktop applications with frameworks such as Electron.js (<https://www.electronjs.org/>).

This week we will look at how JS is used at the front-end to interact with the browser to implement dynamic Web pages. As we mentioned in Week 1, when JS is run inside the browser, it can make use of the Web APIs (<https://developer.mozilla.org/en-US/docs/Web/API>) exported by the browser to interact with the browser and the DOM. Here is the interaction between the JS main loop (called the event loop) and the Web APIs:



**Execution Context Creation (**[**https://www.youtube.com/watch?v=Fd9VaW0M7K4**](https://www.youtube.com/watch?v=Fd9VaW0M7K4)**)**

When the browser loads your Web page and the scripts that come with it (all code included inside the <script> tags), it first creates an execution context and then lets the JS engine start from the top of the first script and execute the JS code line-by-line from top to bottom. The execution context creation has two phases: (1) Memory Creation Phase and (2) Execution Phase.

Memory Creation Phase consists of the following steps:

1. Create the global object (in browser, the global object is called “**window**”; in Node.js, it is called “**global**”)
2. Create “**this**” object and bind it to the global object (in browser this == window; in Node.js REPL (Read-Evaluate-Print-Loop) this == global)
3. Set up memory heap for storing variables and function references
4. Store functions and variables inside the global execution context (window/global object) and set them to “undefined”

Execution Phase consists of the following steps:

1. Execute the code line by line
2. When a function is invoked (called), create a new execution context for that function call to store the parameters and local variables (function invocation frame)

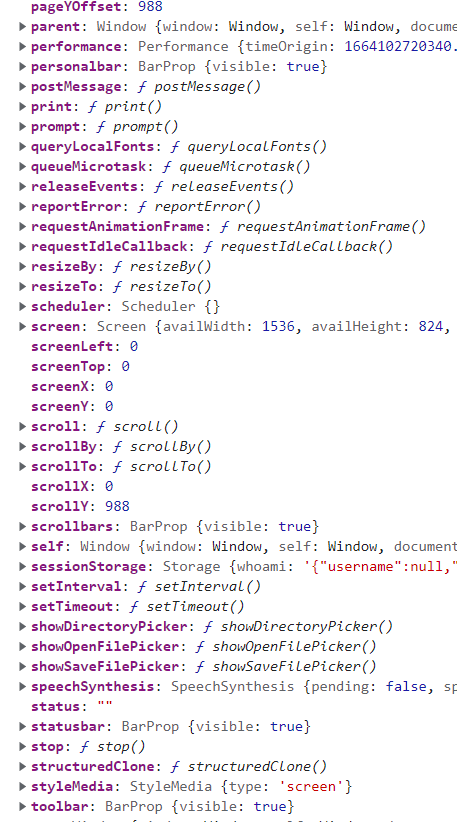
**The Global Window Object**

**[**<https://www.w3schools.com/js/js_window.asp>**]**

[<https://developer.mozilla.org/en-US/docs/Web/API/Window>]

Everything inside a browser revolves around the “window” object. It represents the browser's window containing a DOM document. All global JS objects, functions, and variables (declared with the var keyword in the main function) automatically become members of the window object. In the figure shown above, the window object is the “green” box shown on the right. All Web API methods are exported by the window object.

To see the details of the window object, open a Web browser to go to the developer tools (*press F12 in Chrome*) and open the console. Then type “**window**” and look at the details of the window object. Here is a snapshot of some of the member variables and functions from my browser. You can see some of the Web API functions such as alert, setTimeout, setInterval, clearTimeout, clearInterval, fetch, etc. There are many other objects and methods that you can use in your programs.

The window object also has a member variable named “window” that cyclically points back to the window object! This lets you refer to the members of the window objects directly using the name of the member or using window.member. For example, the current HTML DOM is stored inside a member variable (property) named **document**. This means that you can refer to the DOM simply as “**document**” or “**window.document**” as follows:

|  |
| --- |
| **window**.*document*.getElementById("header");  // is the same as:  *document*.getElementById("header"); |

Notice that you do not have write the “**window**” to refer to its members as it is the default object, a global container for everything within the window including the DOM, which is stored inside the “**document**” object.

**Execution Context Creation (**<https://www.youtube.com/watch?v=Fd9VaW0M7K4>)

To see how the execution context creation phase and the execution phase works, consider the following example code:

|  |
| --- |
| var x = 100  var y = 50  function add(n1, n2){    var sum = n1 + n2    return sum  }  var sum1 = add(x, y)  let sum2 = add(10, 5)  console.log(`sum1: ${sum1}, sum2: ${sum2}`) |

06-DOM/01-Window/index1.html

To run this code, load the Web page in a browser and then go to the developer tools and the source tab. There, select index1.js and put a breakpoint at the first statement. Now, reload the page and you will that the JS engine will stop at the first statement. You can think of all this code to be part of a hypothetical “main” function representing the entry point into your code. It is called “anonymous” in the call stack. Notice also that the global window object has already been created and the variables declared x, y, and sum1 declared with the “var” keyword has already been created and initialized to “undefined”. They are now part of the global **window** object. Furthermore, the function “add” is also part of the global object, that is, there is a member function named “add” inside the global window object. Notice also that variable sum2 declared with the “let” keyword has a local scope and is NOT part of the global object. However, it is part of the execution context of the main function (i.e., anonymous).

Continue executing this code step by step using the down arrow button at the top-right debugger window, and you will see that when the “add” function is invoked, a new execution context is created for the “add” function and it is pushed to the execution stack. There, you can see the parameters n1, n2 and the local variable sum. When we return from the function, the function invocation frame is removed from the stack. This is how the JS engine (the main/event loop) works. This is why JS is called a single-threaded, asynchronous PL. We will look at the asynchronous features of JS later on in this chapter and next week when we cover AJAX. For more details on the execution context, please refer to Brad Traversy’s following video: <https://www.youtube.com/watch?v=Fd9VaW0M7K4>

**Some other important members of the window object**

Here are some important members of the window object:

**window.screen** object contains information about the user's screen (<https://developer.mozilla.org/en-US/docs/Web/API/Screen>)

* screen.width
* screen.height
* screen.availWidth
* screen.availHeight
* screen.colorDepth
* screen.pixelDepth

**window.location** object can be used to get the current page address (URL) and to redirect the browser to a new page (<https://developer.mozilla.org/en-US/docs/Web/API/Location>)

* location.href returns the href (URL) of the current page
* location.hostname returns the domain name of the web host
* location.pathname returns the path and filename of the current page
* location.protocol returns the web protocol used (http: or https:)
* location.assign() loads a new document

**window.history** object contains the browsers history (<https://developer.mozilla.org/en-US/docs/Web/API/History>).

* history.back() - same as clicking back in the browser
* history.forward() - same as clicking forward in the browser
* history.pushState() – adds an entry to the browser's session history stack
* history.replaceState () - modifies the current history entry, replacing it with the state object and URL passed in the method parameters

**window.navigator** object contains information about the browser (<https://developer.mozilla.org/en-US/docs/Web/API/Navigator>).

* navigator.appName
* navigator.appCodeName
* navigator.platform
* navigator.cookieEnabled

**Window Popup Boxes**

The window object has three kinds of popup boxes: Alert box, Confirm box, and Prompt box.

An **alert box** is often used if you want to make sure information comes through to the user. When an alert box pops up, the user will have to click "OK" to proceed.

A **confirm box** is often used if you want the user to verify or accept something. When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed. If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.

A **prompt box** is often used if you want the user to input a value before entering a page. When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value. If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

Look at 06-DOM/01-Window/index2.html

**Timing Events**

The window object allows the execution of code at specified time intervals. These time intervals are called timing events. The key methods to use with JS are:

* **setTimeout**(function, milliseconds): Executes a function after waiting for a specified number of milliseconds
* **setInterval**(function, milliseconds): Repeats the execution of the function continuously after every milliseconds
* **clearTimeout**() method stops the execution of the function specified in setTimeout().
* **clearInterval**() method uses the variable returned from setInterval().

Look at 06-DOM/01-Window/index3.html

The methods illustrate the asynchronous nature of the JS. When you call setTimeout or setInterval, these are Web APIs exported by the **window** object. After you call one of these methods, the call immediately returns and continues executing the next statement. However, after “timeout” milliseconds passes, the timer of the browser (the window object) puts the callback function into the macro task queue of the event loop. When the event loop stack of function calls become empty, the event loop finally picks up the first callback function in the task queue and executes it. We will look into the details of this asynchronous feature of JS next week when we cover AJAX.

**Document Object Model (DOM) [**<https://www.w3schools.com/js/js_htmldom.asp>**]**

Recall from Week 1 that when a Web page is loaded, the browser creates a tree that represents the HTML elements and their hierarchy in a tree called the **Document Object Model (DOM)**. The DOM is stored inside the **document** member of the **window** object, i.e., window.document.

For example, consider the following HTML document. Here is its corresponding DOM.

|  |  |
| --- | --- |
| <!DOCTYPE html>  <html>  <head>  <title>My title</title>  </head>  <body>  <h1>My header</h1>  <a href="...">My link</a>  </body>  </html> | DOM HTML tree |

All nodes in the DOM tree can be accessed by JS, and they can be modified, deleted, or new nodes can be created.

The nodes in the node tree have a hierarchical relationship to each other. The terms parent, child, and sibling are used to describe the relationships.

* In a node tree, the top node is called the root (or root node)
* Every node has exactly one parent, except the root (which has no parent) (parentElement)
* A node can have a number of children (firstElementChild, lastElementChild, children[0], …)
* Siblings (brothers or sisters) are nodes with the same parent (nextElementSibling, previousElementSibling)

|  |  |
| --- | --- |
| <!DOCTYPE html>  <html>  <head>  <title>DOM Tutorial</title>  </head>  <body>  <h1>Header1</h1>  <p>Hello world!</p>  <h2>Header2</h2>  </body>  </html> | From the HTML above you can read:   * <html> is the root node * <html> has no parents * <html> is the parent of <head> and <body> * <head> is the first child of <html> * <body> is the last child of <html>   and:   * <head> has one child: <title> * <title> has one child (a text node): "DOM Tutorial" * <body> has three children: <h1>, <p> and <h2> * <h1> has one child: "Header1" * <p> has one child: "Hello world!" * <h2> has one child: “Header2” * <h1>, <p> and <h2> are siblings |

When you look at document.all, you will see that it is a collection of all HTML elements in the DOM. We can simply loop through this array, and print all elements to the console:

|  |
| --- |
| const all = document.all;  for (let i=0; i<all.length; i++){  console.log(all[i]);  } //end-for |

You can also access different elements of the page using the corresponding document properties. For example, document.title refers to the title of the document, document.head refers to the head section, document.body refers to the body section etc. Also note that although h1 (p or h2 for that matter) seems not to have any children, that’s not the case. In fact, h1 is an HTML element that has one child, an HTML text node that stores “Header 1” text. You can access the children of an HTML element using firstChild, lastChild, children, etc.

Look at 06-DOM/02-DOM-Nodes

**DOM Manipulation**

Given a DOM, it is possible to dynamically manipulate it using JS. But to manipulate the DOM using JS, we must be able to get access to (i.e., select) each and every HTML element, i.e., node, in DOM. The simplest way to select an element from the DOM is to assign the element a unique id, and use document.getElementById(elementId) method to get a handle to the element in the DOM.

|  |
| --- |
| let myElement = document.getElementById("myImage"); |

After you get a handle to the HTML element in the DOM, we can:

* add/change/remove HTML attributes (e.g., change “src” attribute of an image: Look at 06-DOM/03-DOM-Manipulation/index1.html). Any attribute belonging to the element can be changed.
* add/change/remove CSS styles (e.g., change font size, color, border etc. Look at 06-DOM/03-DOM-Manipulation/index2.html). Any CSS style of the element can be changed.
* change the internal content of an HTML element (Look at: 06-DOM/03-DOM-Manipulation/index3.html). In addition to innerHTML, you can also use textContent and innerText properties to change the contents of the element.
* add/change/remove HTML elements (Look at: 06-DOM/03-DOM-Manipulation/index4.html).

|  |  |
| --- | --- |
| document.createElement(element) | Create an HTML element |
| document.removeChild(element) | Remove an HTML element |
| document.appendChild(element) | Add an HTML element |
| document.replaceChild(new, old) | Replace an HTML element |
| document.write(text) | Write into the HTML output stream |

* add/change/remove HTML events (Look at: 06-DOM/03-DOM-Manipulation/index5.html). All HTML events have default handlers. You can add new handlers, change the existing handlers or remove a handler by setting its callback to null. For a list of all events associated with an HTML element, you have to look into its reference document, or just console.dir(element) and look at all of its event handler methods. Typical event handlers are: onclick, onmouseenter, onmouseleave, onmousemove, onkeypress, onkeydown, onkeyup, …, etc. You can either directly assign the corresponding event handler to point to your handler function, or you can use *element.addEventListener(event, function, useCapture);* method to assign a handler. For a list of event names, refer to [<https://www.w3schools.com/jsref/dom_obj_event.asp>]. The callback function can be a named function, or an unnamed arrow function:

|  |
| --- |
| btn1.onclick = Button1ClickHandler; // A named function  function Button1ClickHandler(e){…}  btn2.onclick = function(e){…}; // unnamed function as an event listener. “e” is the event that fires  btn3.addEventListener(“click”, (e)=>{…}); // An unnamed arrow function as event listener |

Inside the event handler function, the first thing you should do is to call “e.preventDefault()” to prevent the event from running the default handler. Then you write your own handler code. “e” contains a lot of useful information that you may make use of. For example,

|  |  |
| --- | --- |
| e.type | Event type: click, dblclick, mousedown, mouseup, mousemove, keyup, keydown, keypress, focus, blur, cut, paste, input (this fires any time you do something with an input element, i.e., when a key is pressed, focus, blur, cut, paste, etc.), etc. You can also associate a “change” event with a select list, which gets fired when the user selects an item from the select list. For the form element, you can catch the “submit” event, but inside the submit event handler, you must prevent the default behaviour by calling e.preventDefault(), which submits the form element values into a server web page. |
| e.target | Is the object on which the event has occurred (the button object in our example) |
| e.target.id | Id if the object on which the event has occurred |
| e.target.className | button |
| e.taget.classList | List of classes that this object belongs to |
| e.clientX | X-axis coordinate of the mouse position from the left side of the window |
| e.clientY | Y-axis coordinate of the mouse position from the top of the window |
| e.offsetX | X-axis coordinate of the mouse position from the left border of the target element |
| e.offsetY | Y-axis coordinate of the mouse position from the top of the target element |
| e.altKey | Is the alt key pressed when this event has occurred? |
| e.ctrlKey | Is the ctrl key pressed when this event has occurred? |
| e.shiftKey | Is the shift key pressed when this event has occurred? |

When you are inside an event handler that does not have the current event as its parameter, you can get a hold of this current event from the window object. That is, “**window.event**” stores the current event.

Here is another example from <https://www.foolishdeveloper.com/2022/04/eyes-follow-mouse-cursor-javascript.html> that puts two eyes in the middle of the window and have these eyes follow the mouse by handling the “mousemove” event. Inside the event handler, the current (x, y) coordinates of the mouse are used to calculate where the eyes should look like.

**DOM Selectors**

document.getElementById() method is a single element selector. That is, it returns a handle to a single matching HTML element. Another, and more commonly used, selector method is “document.querySelector(cssSelectorString)”. This method is very generic in the sense that you can write any CSS selector as the cssSelectorString and get a handle to those elements that you would get with CSS. If there are more than one matching element, this method returns the first one. For example, assume you have a paragraph with id=”main-para”. You can get an access to this element using either:

|  |
| --- |
| const para1 = document.getElementById(“main-para”); // Get handle to paragraph with id = “main-para”  const para2 = document.querySelector(“#main-para”); // This returns the same handle, and is preferred. |

There are also many methods that select multiple elements from the DOM. The following methods return an HTMLCollection as their result, which can be iterated over like an array:

|  |
| --- |
| getElementsByClassName("…"); |
| getElementsByTagName(“ …”); // Using the tag name |
| getElementsByName(“…”); // Using the name attribute |

But the preferred way to select multiple elements from the DOM is to use document.querySelectorAll(“cssSelectorString”); For example, the following selects all elements with class name “btn”:

|  |
| --- |
| const btns = document.querySelectorAll(“.btn”); |

This method returns a **NodeList**, which can also be iterated over like an array. In addition to these, recall that the document object already has several collections as properties that you may directly use as needed.

|  |  |
| --- | --- |
| document.links | A collection of link elements in the document |
| document.anchors | A collection of anchor elements in the document |
| document.forms | A collection of form elements in the document |
| document.images | A collection of image elements in the document |
| document.all | A collection of all elements in the document |

**Examples:** Assume that you have the following Web page:

|  |
| --- |
| <section class="container">  <form id="my-form">  <h1>Add Item</h1>  <div class="msg"></div>  <div>  <label for="item">Item:</label>  <input type="text" id="itemInput">  </div>  <input class="btn" type="submit" value="Submit">  </form>  <ul class="items">  <li class="item">Item 1</li>  <li class="item">Item 2</li>  <li class="item">Item 3</li>  </ul>  </section> |

Here is how we select different elements from the page using the above methods:

|  |
| --- |
| document.getElementById('my-form'); // Selects the form element  document.querySelector('.container'); // Selects the container element  document.querySelector(‘input[type=”text”]’); // Select the input element with type=”text”  // Multiple Element Selectors  document.getElementsByTagName('li'); // Selects all <li> elements. Returns a HTMLCollection  document.getElementsByClassName('item'); // Selects all elements with class = “item”. Returns an HTMLCollection  document.querySelectorAll('.item'); // Selects all elements with class = “item”. Returns a NodeList  const item1 = document.querySelector(“ul :nth-child(1)”); // Select the first child  const oddItems = document.querySelectorAll(“ul:nth-child(odd)”); // Select all odd children  const evenItems = document.querySelectorAll(“ul:nth-child(even)”); // Select all even children  oddItems.forEach((item)=>{item.style.backgroundColor = "#ccc";}); // Make the background dark-gray for all items  // Can also use firstElementChild, lastElementChild, nextElementSibling, previousElementSibling, parentElement etc.  // to traverse over the DOM  const item1 = document.querySelector('ul').firstElementChild; // First item  const item4 = document.querySelector('ul').lastElementChild; // Last item  const item2 = item1.nextElementSibling; // Next sibling of item1, i.e., item2  const item3 = item4.previousElementSibling; // Previous sibling of item4, i.e., item3  const list = item1.parentElement; // parent element of item1: ul |

Here is how we can create a new <li> node and add it in between item2 & item 3:

|  |
| --- |
| const newItem = document.createElement('li');  newItem.className = "item";  newItem.id = "newItemId"; // Give it a hypothetical id  // Create a new text item  const newTextItem = document.createTextNode("New Item");  newItem.style.fontWeight = "bold"; // Make the text bold  newItem.appendChild(newTextItem); // Append the text item to the new item  // Get a handle to item3  const item3 = ul.children[2];  // Add the new item after item2  ul.insertBefore(newItem, item3); |

Look at: 06-DOM/04-DOM-Selectors/index1.html

It is important to know that you can use querySelector()/querySelectorAll() methods with any HTML element, not just with document to select elements only within that container. In the following example, we have two containers with ids “container1” and “container2” with some paragraphs. If we call document.querySelectorAll(‘p’), this will select ALL ‘p’ elements within the document. However, if we get a handle to a container first and then call querySelectorAll(‘p’) for that container element, we will only get the ‘p’ elements within that container. The following example shows this:

Look at: 06-DOM/04-DOM-Selectors/index2.html

**ItemListApp**

Assume that we extend our simple application given above such that not only would we be able to add new items, but we would also like to delete items from the list and also filter the items. Here is how the application looks like:

Look at: 06-DOM/05-BradTraversy-ItemListApp

The code for this project was taken from Brad Traversy’s DOM Manipulation using vanilla JS tutorial series. Here is the link to that playlist: <https://www.youtube.com/playlist?list=PLillGF-RfqbYE6Ik_EuXA2iZFcE082B3s>. I recommend that you go through all vides on the playlist and see how he manipulates the DOM using vanilla JS.

**Client-Side Form Validation**

**(**[**https://developer.mozilla.org/en-US/docs/Learn/Forms/Form\_validation#what\_is\_form\_validation**](https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation#what_is_form_validation)**)**

**(**[**https://www.w3schools.com/js/js\_validation.asp**](https://www.w3schools.com/js/js_validation.asp)**,** [**https://www.w3schools.com/js/js\_validation\_api.asp**](https://www.w3schools.com/js/js_validation_api.asp)**)**

In many Web applications there are forms that contain some input fields where you need to enter some values (text, numbers, date, password, selection, radio buttons, etc.). It is important to ensure that all required form controls are filled out in the correct format. This is called client-side form validation.

There are two different types of client-side validation that you'll encounter on the web:

1. **Built-in form validation** uses HTML form validation features and generally doesn't require much JS. Built-in form validation lets you validate the input fields without the need to write any JS code, but is not as customizable as JS validation.
2. **JavaScript validation** is coded using JS. This validation is completely customizable, but you need to create it all by yourself (or use a JS library).

**Built-in form validation**: One of the most significant features of modern form controls is the ability to validate most user data without relying on JS. This is done by using **validation attributes** on form elements. Here are some validation attributes that you can use with form elements:

* ***type***: Specifies whether the data needs to be *text*, a *number*, an *email* address, a *password*, a *date*, etc.
* ***required***: Specifies whether a form field needs to be filled in before the form can be submitted (at least one char)
* ***minlength*** and ***maxlength***: Specifies the minimum and maximum length of textual data (strings)
* ***min*** and ***max***: Specifies the minimum and maximum values of numerical input types
* **pattern**: Specifies a **regular expression** that defines a pattern the entered data needs to follow (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions>)

If the data entered in a form field follows all of the rules specified by the above attributes, it is considered valid. If not, it is considered invalid.

When an element is **valid**, the following things are true:

* The element matches the *:valid* CSS pseudo-class, which lets you apply a specific style to valid elements.
* If the user tries to send the data, the browser will submit the form, provided there is nothing else stopping it from doing so (e.g., JS).

When an element is **invalid**, the following things are true:

* The element matches the :invalid CSS pseudo-class, and sometimes other UI pseudo-classes (e.g., :out-of-range) depending on the error, which lets you apply a specific style to invalid elements.
* If the user tries to send the data, the browser will **block** the form from getting submitted and display an error message.

The following example has a form with a text field and a number field. We want the text field to contain at least 3 and at most 8 characters, so we use minlength and maxlength attributes. We also want the entered number to be in between 200 and 750, so we use min and max attributes. If both of these conditions are satisfied, then the validation is successful and the form is submitted to the server. Otherwise, the browser will display some default error messages on the screen. Also notice that we target :valid and :invalid pseudo-classes on the input elements so that their CSS styles change as we type in values into the input fields.

06-DOM/06-FormValidation/index1.html

Sometimes we want the input field to be validated against a stronger validation criteria based on a regular expression (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions>). For example, in the previous example, we specified that the name will have at least 3 chars and at most 8 using minlength and maxlength attributes. However, the user may enter non-letter chars (digits, spaces, punctuation chars, etc.) and that will be accepted. What if we want to make sure that the user can only enter lowercase and uppercase chars in the name field? Then we put a pattern attribute and specify a regular expression so that the input is NOT accepted if the user enters any char other than letters. Here is that example:

06-DOM/06-FormValidation/index2.html

**JS validation**: Built-in form validation might be good enough for many applications, but sometimes you would like to take control of the validation process yourself and do it in JS. For example, you can see that the error messages displayed in the case of invalid entry is too generic: “Please match the requested format”. You may want the browser to display more specific error messages in the case of a pattern mismatch. To do this, you may need to make use of the Constraint API (<https://developer.mozilla.org/en-US/docs/Web/API/Constraint_validation>), which makes some properties and methods available on the form elements.

One of the most important properties pertaining to the validated input element is the “**validity**” object, which contains several properties describing the validity state of the element. An important method is **setCustomValidity(message**), which adds a custom error message to the element. If you set a custom error message, the element is considered to be invalid, and the specified error is displayed. This lets you use JS code to establish a validation failure other than those offered by the standard HTML validation constraints. The message is shown to the user when reporting the problem. If the input is valid, you need to set **setCustomValidity** to **“”**. In the following example, we add an event listener to the ‘input’ event and have the browser display more customized messages in the case of an invalid input.

06-DOM/06-FormValidation/index3.html

In the previous example we were able make the browser display customized messages inside a tooltip popup box if the input is invalid. It is usually the case that you do not want the browser to display messages as such, but you would like to show these messages below the input element as the user types in some value. In that case, we need to first prevent the browser from displaying these messages in a tooltip popup by adding the “novalidate” attribute to the form element as follows:

|  |
| --- |
| <form novalidate> |

This prevents the browser from validating the inputs and displaying the error messages in tooltip popups. Instead, we get the full control and must handle validation and error message display in HTML + JS. To do this, the usual practice is to put an empty span/div element under each input element in the form, where we can display the error message. Here is an example HTML for an input element for this:

|  |
| --- |
| <div>              <label for="name">Name:</label>              <input type="text"                     name="name"                     id="name"                     minlength="3"                     maxlength="8"                     pattern="[a-zA-Z]{3,8}"                     placeholder="John"                     required/><br>              <div class="invalid-feedback"></div>          </div> |

Then, we add an event listener for the ‘input’ event for this input element and display the error message inside the div.invalid-feedback as follows:

|  |
| --- |
| const nameEl = form.querySelector('#name')  const nameElErrMsg = form.querySelector('#name ~ div.invalid-feedback')      nameEl.addEventListener('input', ()=>{          if (nameEl.validity.tooShort)              nameElErrMsg.textContent = "Name must have at least 3 letters";          else if (nameEl.validity.patternMismatch)              nameElErrMsg.textContent = "Name can only contain letters";          else              nameElErrMsg.textContent = "";      }) // end-addEventListener |

Finally, we need to add an event listener for the form’s ‘submit’ event and check whether all elements inside the form is valid. Notice that since we set the form’s *novalidate* attribute, it is now possible to submit this form without the form elements being valid. Therefore, it is now our job to check if all elements are valid inside the ‘submit’ event and then submit the form to the server (or whatever we want to do with the data). If at least one element of the form is not valid, we should prevent the form from getting submitted (e.preventDefault()). The details of this example can be found here:

06-DOM/06-FormValidation/index4.html

It is also possible to do all error checking in JS without making use of any of the HTML5 form input elements attributes and the Constraint Validation API. The following example shows how this can be done. Here we have two CSS classes, valid and invalid, that are used to style a form element depending on whether their content is valid or not. We then check the validity of each form element and set its class appropriately. We also display the appropriate error message in the div below the input element.

06-DOM/06-FormValidation/index5.html

Client-side form validation is such an important concept that CSS frameworks such as Bootstrap has form validation built into their form components. The following example shows how you can use Bootstrap forms and validation. Notice that the approach taken by Bootstrap is very similar to the approach we have taken in 06-DOM/06-FormValidation/index4.html: The form’s novalidate is set so that JS can full control. The error messages are displayed below the input element in div.invalid-feedback.

06-DOM/06-FormValidation/index6.html

Notice that in the above example, the form’s default class is ‘needs-validation’. So, initially no error messages are displayed. Inside the ‘submit’ event, we add ‘was-validated’ class to the form element, which makes the error messages for all invalid input fields to appear in ‘red’ color. Since the form now has ‘was-validated’ class, we will continue seeing the error message until we enter a valid input. The error message for an input field will disappear as soon as the input becomes valid. If, however, we want the error messages to get displayed by default and then disappear after we enter a valid value for an input field, we need to make the form’s default class ‘was-validated’. The following example illustrates this:

06-DOM/06-FormValidation/index7.html

**Single Page Applications (SPA)**

A Single Page Application (SPA) is one that gets loaded from the Web Server once and then runs within the Browser Tab until it is closed without ever being re-loaded from the server. The ItemListApp that we just looked at (06-DOM/05-BradTraversy-ItemListApp) is an example of an SPA. Notice that after this application is loaded from the server, it never gets re-loaded. All page updates are handled by JS running inside the browser’s JS engine. Such applications may asynchronously contact REST APIs or Database servers running in the cloud to send/receive data through the browser’s AJAX API (the fetch api), but they will NEVER re-load themselves from the server. The reason for this is the following: When a page is first loaded, the JS code that comes with the page inside <script> tags gets run by the browser. This creates some sort of a state for the page, which is used to render the page, and then make changes in it as the app runs. If, however, the page is re-loaded or another browser page/tab is created, then the current state is thrown away, everything is initialized and the JS code that comes with the new page is executed. This means that we will lose the existing state holding our data. That’s why such applications have to run inside a single browser page/tab without re-loading themselves from the Web server.

While the ItemListApp given in 06-DOM/05-BradTraversy-ItemListApp works correctly, it is quite complicated. The data being rendered (the current set of items) are stored inside the HTML elements (the DOM), which makes manipulating the data very complicated as we have seen. The reason that application is implemented that way was to explain what DOM is, how DOM elements are structured and how you can manipulate DOM.

There is a must easier and common way to implement such SPAs: The idea is to store the app data (called the app state) in JS objects and to dynamically create the HTML elements directly from the data and then render the dynamically generated HTML elements on the page. To do this, the initial HTML page has a single <div> element (usually called the “root” element) in which we will render the dynamically generated HTML elements. This is how most JS frameworks such as React, Vue, and Svelte work.

In the following example, we will convert Brad Travery’s ItemListApp (06-DOM/05-BradTraversy-ItemListApp) to use this model. You will now see how easy it is to implement the same SPA using vanilla JS. Here is the heart of this code where we manipulate the current app state (items) and render them on the screen. As we will see in the following weeks, this is exactly how React, Vue and Svelte work.

|  |
| --- |
| /\* The state: Current list of items \*/  let items = [    {id: 1, title: "Item 1"},    {id: 2, title: "Item 2"},    {id: 3, title: "Item 3"},    {id: 4, title: "Item 4"},    {id: 5, title: "Item 5"},  ]  let nextItemId = 6;  function addItem(title){items.push({id: nextItemId++, title: title});}  function deleteItem(id){items = items.filter(item=>item.id != id)}  function filterItems(filterStr){    if (filterStr == '') return items;    const filtered = items.filter(item=>item.title.toLowerCase().indexOf(filterStr) >= 0)    return filtered  } // end-filterItems  function renderItems(filterStr=''){    const filteredItems = filterItems(filterStr);    let template = '';    filteredItems.forEach(item => {      template += `      <li class="listItem">        <div class="listItemText">${item.title}</div>        <button class="delbtn" onclick={deleteButtonHandler(${item.id})}>X</button>      </li>      `    })    document.querySelector('#root').innerHTML = template;  } // end-renderItems |

As you can see, we simply store the current set of items in a JS array and manipulate them using some helper functions. We also have a function to render the current set of items on the HTML page. Notice that this is a quick and dirty implementation. Ideally, you would want to encapsulate the items data and the functions that manipulate it inside a JS class, create a state object from it and then call the object’s methods to manipulate the current set of items. I will leave this as an exercise to you. Here is the entire app:

Look at: 06-DOM/07-MyItemListApp

**Local Storage [**<https://www.w3schools.com/html/html5_webstorage.asp>**]**

So far, our apps stored all of its data in memory. However, when the browser tab running our app is closed, all this data is lost. Ideally you would want to persist your data; that is, you want your data to be stored somewhere so that the next time your app starts, it can retrieve the last snapshot of the data and continue from there.

There are two ways to ways to make your data persistent:

1. Store your data in a database. To do this, we need to send the data to a backend-server, which can then store the data in a database. The backend-server can be a backend app implemented in a backend PL such as node.js, python, PHP, java, asp.net, etc. running inside a Web server (a.k.a. an app server), or it can also be a database service such as Firebase, supabase called the serverless architecture. In both cases, we need a way to send our data to this third-party server. This is called Ajax and we will look into this next week.
2. Store your data locally in the browser’s cache. This is what we will be looking at in this lecture.

It is possible for web applications to store data **locally** within the user's browser as **key/value** pairs. Both the key and the value must be strings. HTML web storage provides two objects for storing data on the client:

* **window.localStorage** - stores data with no expiration date
* **window.sessionStorage** - stores data for one session (data is lost when the browser tab is closed)

To store a key/value pair in localStorage, do:

|  |
| --- |
| localStorage.**setItem**("key", "value"); |

To read data from the localStorage, do:

|  |
| --- |
| let value = localStorage.**getItem**("key"); |

To remove data from the localStorage, do:

|  |
| --- |
| localStorage.**removeItem**("key"); |

Since the value associated with a key must be a string, how do we then store JS objects? The idea is to convert JS objects into a string using JSON.stringify(). To convert from a string to a JS object, we then use the reverse conversion function: JSON.parse(). It is also important to note that the data stored locally is not made permanent in the server. If you want your data to persist permanently on a remote server, you have to push the data there. The data stored locally is only temporarily available, and can be cleaned up by the user.

In the following example, we change our previous ItemListApp (06-DOM/07-MyItemListApp) so that the added items are also stored in the local storage. So when we restart/reload the application, the previously-created items are retrieved from the local storage, and thus they do not get lost. You can see the contents of the localStorage by going to the developer tools (F12)->Application->Local Storage

Look at: 06-DOM/08-MyItemListAppWithLocalStorage

**Pagination**

In many applications we have a lot of data that does not fit into a single page. For example, let’s say that you have 100 todo items that you want to display. You have two alternatives:

1. You can put all items into a single page and have the user scroll down the page to see all items
2. You can split your data into several pages and display the data one page at a time. In this case, you should also have a control at the bottom of the page that shows all pages with numbers so that the user can switch between different pages and see the items on that page.

06-DOM/09-Pagination shows how you can implement pagination using vanilla JS. Here we have 55 todos stored in a JS array and we want to display the items 10 at a time. Since we have 55 todos, we have 5 pages; so we display the first 10 todos and then show the page numbers at the bottom. The user can go to any page by clicking on the page and the items will dynamically be updated.

**jQuery [**<https://www.w3schools.com/jquery/default.asp>**]**

jQuery is a lightweight, "write less, do more", JavaScript library. The purpose of jQuery is to make DOM manipulation easier. jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that you can call with a single line of code.

**Adding jQuery to your projects**

There are two ways to add jQuery on your web site. You can:

* Download the jQuery library from jQuery.com, and then include it to your Web page inside <script> tags:

<head>  
<script src="jquery-3.4.1.min.js"></script>  
</head>

* Include jQuery from a CDN:

<head>  
<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.min.js"></script>  
</head>

**The Document Ready Event**

To prevent any jQuery code from running before the document is finished loading (is ready), it is good practice to wait for the document to be fully loaded and ready before working with it. This also allows you to have your JavaScript code before the body of your document, in the head section. To achieve this, always enclose your jQuery code inside $(document).ready() event handler as follows:

|  |
| --- |
| $(document).ready(function(){    *// jQuery methods go here...*  }); |

**Using jQuery: Selectors**

To use jQuery, you need to select HTML elements and perform "actions" on them. The basic syntax is: **$(selector).action()**, where:

* $ sign defines/accesses jQuery
* (selector) selects HTML elements
* action() specifies the action to be performed on the element(s)

Here are some examples:

|  |
| --- |
| $(this).hide() - hides the current element.  $("p").hide() - hides all <p> elements.  $(".test").hide() - hides all elements with class="test".  $("#test").hide() - hides the element with id="test". |

In fact, jQuery uses CSS selectors to select HTML elements from the document. For a list of jQuery selectors, refer to <https://www.w3schools.com/jquery/jquery_selectors.asp>

**Using jQuery: Events**

Many mouse and keyboard events occur on the DOM and can be handled as we have already seen. It is possible to catch and handle these events with jQuery. You simply need to add an event listener for the event you want to handle. For example, to handle mouse click events, you do the following:

|  |
| --- |
| $("p").click(function(){   // action goes here!! }); |

Alternatively, you can add the “click” event listener as follows:

|  |
| --- |
| $("p").on(“click”, function(){   // action goes here!! }); |

Look at: 06-DOM/10-jQuery1

**DOM Manipulation using JQuery**

jQuery has many methods for DOM manipulation. Three simple, but useful, jQuery methods for DOM manipulation are:

* text() - Sets or returns the text content of selected elements
* html() - Sets or returns the content of selected elements (including HTML markup)
* val() - Sets or returns the value of form fields

|  |
| --- |
| $("#btn1").click(function(){   alert("Text: " + $("#test").text()); });  $("#btn2").click(function(){   $("#test").text(“Changed”); }); |

The jQuery attr() method is used to get/set attribute values as follows:

|  |
| --- |
| alert($("#w3s").attr("href"));  $("#w3s").attr("href", "https://www.w3schools.com/jquery/"); |

The following four jQuery methods are used to add new content:

* append() - Inserts content at the end of the selected elements
* prepend() - Inserts content at the beginning of the selected elements
* after() - Inserts content after the selected elements
* before() - Inserts content before the selected elements

|  |
| --- |
| $("p").prepend("Some prepended text.");  $("p").append("Some appended text.");  $("img").after("Some text after"); $("img").before("Some text before"); |

To remove elements and content, there are mainly two jQuery methods:

* remove() - Removes the selected element (and its child elements)
* empty() - Removes the child elements from the selected element

|  |
| --- |
| $("#div1").remove();  $("#div1").empty(); |

The css() method sets or returns one or more style properties for the selected elements.

|  |
| --- |
| $("p").css("background-color");  $("p").css("background-color", "yellow"); |

Finally, jQuery has several methods for CSS class manipulation.

* addClass() - Adds one or more classes to the selected elements
* removeClass() - Removes one or more classes from the selected elements
* toggleClass() - Toggles between adding/removing classes from the selected elements

|  |
| --- |
| $("button").click(function(){   $("#div1").addClass("important blue"); });  $("button").click(function(){   $("h1, h2, p").removeClass("blue"); }); |

06-DOM/10-jQuery1 is an example that shows how we make use of JQuery.

06-DOM/11-jQuery2 is a re-implementation of our ItemListApp using JQuery.

**Using jQuery vs plain (vanilla) JS [**<https://www.w3schools.com/js/js_jquery_selectors.asp>**]**

jQuery was designed when it was quite difficult to select elements from the DOM and manipulate the DOM. But with new functionality added after ES6 (e.g., document.querySelector, document.querySelectorAll, etc.) it is also as easy to select elements from the DOM and manipulate them. So unless you are doing something that really requires the use of jQuery, it might be an overkill to use jQuery for many simple mundane tasks. However, you may come across some old code written using jQuery, so it is necessary to have a grasp of jQuery as much as we described here.

As a reference, we give some of the jQuery methods and how you can easily emulate them in plain JS:

|  |  |
| --- | --- |
| **Selecting Elements from the DOM** | |
| $(‘someElement’) | document.querySelector(‘someElement’) |
| $(‘someClass’) | document.querySelectorAll(‘someclass’) |
|  | |
| **DOM Manipulation** | |
| $(element).remove() | element.remove() |
| $(element).prepend(otherElement) | element.prepend(otherElement) |
| $(element).before(otherElement) | element.before(otherElement) |
| $(element).addClass(‘someClass’) | element.classList.add(‘someClass’) |
| $(element).removeClass(‘someClass’) | element.classList.remove(‘someClass’) |
| $(element).toggleClass(‘someClass) | element.classList.toggle(‘someClass’) |
| const parent = $(element).parent(); | const parent = element.parentElement |
| const cloned = $(element).clone(); | const cloned = element.cloneNode(true) |
|  | |
| **Events** | |
| $(element).on(‘click’, function(){  // Some logic  }); | element.addEventListener(‘click’, (e)=>{  // Some logic  }); |
|  | |
| **Utilities** | |
| $.isArray(someValue) | Array.isArray(someValue) |
| $.inArray(item, anArray) | someArray.indexOf(item)>-1 |
| $.each(someArray, (index, value)=>{}) | someArray.forEach((value, index)=>{}) |
| $.map(someArray, (value, index)=>{}) | someArray.map((value, index)=>{}) |
| $.grep(someArray, (value, index)=>{}) | someArray.filter((value, index)=>{}) |
| $.parseJSON(str) | JSON.parse(str) |
|  |  |
| **jQuery’s Main Uses** | |
| DOM Selection | querySelector() & querySelectorAll() |
| DOM Manipulation | Native Browser APIs |
| HTTP/AJAX | Fetch or Axios (or other specialized libs) |
| Utilities | Map, filter, reduce, etc. OR lodash |
| Animation | CSS3 or 3rd party specialized lib |
| Browser Support | Modern browsers + Babel & Polyfills |

06-DOM/12-jQuery1WithVanillaJS shows how we can re-implement 06-DOM/10-jQuery1 using vanilla JS. As you can see, it is not that difficult to implement the same code using vanilla JS. This is why jQuery is starting to lose its meaning in current Web applications. People are starting to rid of jQuery and doing all their work in plain vanilla JS as this lets them avoid yet another JS library.