DOM - 102

Prior Knowledge

- Comfortable with the nested structure of the DOM
- Comfortable setting up an HTML file and linking associated css and js files
- Understand how to use DOM methods to fetch elements
- Knowledge of a few types of events
- Understand what an event listener is & how to use them
- Understanding of asynchronous code and promises

Learning Objectives

- Understand how forms work
- Improve use of DOM methods and semantic html
- Be able to provide inline UI (user interface) validation
- Be able to use the 'action' and 'method' form attributes

1. Forms

Forms used to be the pre-eminent way of passing data around the internet.

The controls to collect data in an html form come in a variety - text input, radio buttons... They have developed to accept **types** which are parsed by browsers. Here's a full list of input types on w3schools.

The **action** attribute part of a form expects a url, an endpoint to which data is sent (like an API).

The **method** attribute expects a HTTP methods like `GET` or `POST`. `POST` requests don't display the URL so should be used for sensitive data.

These two attributes specify how and where the data is sent when a form is submitted (i.e. when a form experiences a 'submit' event). If a submit event is triggered but no action or method specified then the default is `action="/" method="GET"`.

Note: If each input box is given a name attribute, a GET request will append a query containing the input box names and the values to the url: e.g. `?fullname=izzi&username=izziNC` . This means that GET requests should not be used for sending sensitive data like passwords. POST requests do not show the data in the URL and instead send form-data in the HTTP body.

As http servers should have no state (a `RESTFUL` principle) all the information had to come from the client, even if it wasn't something the user had knowledge of. This meant that, in cases where the server needed information from the client but the front-end developers did not want this information to be visible to the end-user, the 'hidden' attribute was added to input elements. This meant that the request to the server had all the data required but the user would not be able to see or modify this data. For more info, see the MDN page on `<input type="hidden">`

Semantic HTML & Accessibility

Now that we are designing front end applications, we need to bear in mind accessibility for users and browsers.

Semantic HTML means using the correct HTML elements for their correct purpose as much as possible: a `ul` tag for a list, not a series of nested `div`s

- **Labels** are a semantic html element that doesn't render as anything special but adds usability by allowing the user to click on the label to select the input. The `for` attribute of the `label` should be equal to the `id` attribute of the `input` , and the `input` has to be a child of the label.
- **Buttons** have some suitable styling applied by default (which you will probably want to override), they also have default keyboard accessibility — they can be tabbed between, and activated using Return/Enter.
- Other ones you might use might be `header` , `footer` or `article` tags rather than plan old `div` s!

Some useful articles on what semantic html is and why its needed:

The Importance of Semantic HTML

Semantic HTML

2. Validation

When developing online forms your job is to alert the user if their inputs are incorrect and would be rejected by the api. If the user has not completed the fields (or filled them incorrectly) then there is no point making an api call to sign them up. When you try and sign in without providing information websites will often prompt you to complete the fields before trying to submit. This is `front-end` or `client-side` validation.

A nice guide on form design can be found here.

In general here are some common rules for client side validation

Good	Bad
Confirmation of completed fields when you move on to the next	Not validating until the whole thing is completed and having to go back to it to correct mistakes.
Clear prompts of what's expected e.g. password restrictions	Unclear what the restrictions are.
Errors are pointed out before you submit.	Validating too early. When you're halfway through typing something.
Clear colour schemes, green is good, red is bad.	Placeholders disappearing when you start typing.
Good tabbing, sort-codes are a great example of this.	Everything nested so you can't tab between
Progress Bars for long forms.	Deleting all the input on submission so you have to re-type

|

Some things have to be validated `server-side`, e.g. a new username that hasn't been taken. Do as much as possible client-side to smooth out this experience. A form is the final point of a sale and should be treated as such.

Confirmations

We can use an event listeners to trigger responses to users filling in forms. Consider carefully how you want responses to be generated. If you want to evaluate a user's input as they type, use `change`. If you want to evaluate when they move to the next field, use `blur`. Other common ones that may be of interest are `change`, `input`, `submit`, `click`, `blur`, and `focus` - a list of events can be found here on w3schools.

```
const handleChange = (event) => {};

usernameInput.addEventListener('change', handleChange);
```

In the above example, `handleChange` is our event handler which we are attaching to the `usernameInput` element by using `addEventListener`. The `handleChange` will be invoked whenever the `usernameInput` experiences a `change` event. Importantly, this `handleChange` will be invoked with the HTML DOM Event that caused `handleChange` to be invoked.

This DOM Event contains lots of information - for example, at the `event.target` key we can find the HTML element that experienced that event. This means that, in the example above, we could access the `usernameInput`'s value through `event.target.value`.

3. Accessing the url

Forms are frequently used to post data to a server that will interpret the request. When using the `GET` method they will add our form data as a query. We can access current pages url and associated properties through the Location api. This object can be accessed through the `location` property of either the `document` or `window` global objects.

```
console.log(document.location); // A Location object
```

There are several useful properties on this object. For accessing the queries we can use `Location.search` to access the queries as a raw string.

Parsing these queries can be handled in a number of ways. Most browsers come with URLSearchParams which will parse the raw queryString and give us access to methods such as `get` to retrieve the values associated with certain queries.

```
// If the current url is https://example.com/?name=Paul&age=34

const searchStr = document.location.search; // '?name=Paul&age=34'

const queries = new URLSearchParams(searchStr);

const name = queries.get('name'); // 'Paul'
const age = +queries.get('age'); // 34
```

n.b. All query values are strings. So any other type of value, such as numbers, will need to be converted to their associated type before use.