Intro to JavaScript Modules

Goals

In this lesson, we're going to cover the following:
- JavaScript modules – what they are and why they're useful
- The important topics of encapsulation, code maintainability and scalability
- How to start using modules to better organize our code
>
Introduction
Thus far, you have been exposed to several elements of the JavaScript language: variables, different data types and operators, loops, conditionals, functions, arrays, plus new features introduced in ES6 (e.g. - arrow functions, template literals, destructuring, etc.). If this course is your first exposure to programming and your head is spinning a little at this point, don't worry... that's normal and it will pass
>
This week, we're going to dive yet deeper and look at more intermediate concepts and uses of the language, including modules, the use of template engines on the client-side (and why it can be very helpful), as well as some basic data structures (two of which we've already seen and two new ones), and a very brief primer on the important concept of Big O Notation (which is a standard way of measuring code efficiency).

Exporting Modules: Named

> Modules are a widely used method of organizing and encapsulating code in modern JavaScript. They are part of the ECMAScript 2015 (ES6) specification and provide a way to define code that can be shared across multiple files or projects. In this section, we will cover the fundamentals of modules, including exporting and importing variables and functions.
>
> ### Exporting Modules
>
> In ES6 modules, we use the **export** keyword to make variables or functions available for import in other files. There are two main ways to export a given piece of code: **named exports** and **default exports**.
>
> **Named Exports:**

Named exports allow us to export multiple values from a module, each with a unique name. We use the **export** keyword followed by either the variable or function we want to export.

Let's say you're building a recipe application, and you want to keep track of your favorite ingredients and a simple function to combine them into a recipe. We'll export an array of favorite ingredients and a function to combine them into a recipe. Go ahead and add the following to **module1.js**:
>

```
// module1.js
export const favoriteIngredients = ["Salt", "Pepper", "Olive
        Oil"];
export function combineIngredients(ingredients) {
  return `Combined ingredients: ${ingredients.join(", ")}`;
}
```

In this code, we are exporting two things: an array and a function. These can then be imported into another file using an **import** statement. Let's add the following to the top of **main.js**:

```
// main.js
import { favoriteIngredients, combineIngredients } from
        './module1.js'
```

Now, in **main.js**, in the section marked as NAMED EXPORTS, add the following:

```
console.log(favoriteIngredients); // Output: ["Salt", "Pepper",
        "Olive oil"]
console.log(combineIngredients(["Tomato", "Basil",
        ...favoriteIngredients])); // Output: "Combined
        ingredients: Tomato, Basil, Salt, Pepper, Olive oil"
```

In this example, we are importing **favoriteIngredients** and **combineIngredients** from the `module1.js` file, using the destructuring syntax. We then console log the favoriteIngredients variable and invoke the combineIngredients function, passing in as arguments the values “Tomato”, “Basil”, and everything from the favoriteIngredients variable (using the spread operator).

>

Exporting Modules: Default

> **Default Exports:**

Default exports allow us to export a single variable or function from a module as the default export. We use the **export default** keywords.

Imagine you're creating a chat application and there's a common phrase you want your chatbot to say. We create a function that returns that phrase, and make the function a default export. Add the following to **module2.js**:

```
// module2.js
const welcomeMessage = () => {
  return "Welcome to ConvoBox! How can I assist you today?";
};

export default welcomeMessage;
```

Note that a module can only have **one** default export.

Here we defined a function called **welcomeMessage** for our fictional chat application called 'ChatterBox', and use it as our default export from this module. Now, as with named exports, we can easily import it into our main script file. The syntax for importing default exports is slightly different, however:

```
// main.js
import welcomeMessage from './module2.js'
```

As you can see, we import the **welcomeMessage** function, without using destructuring syntax this time. Next, let's actually call this function. Add the following to **main.js** in the section marked as DEFAULT EXPORTS:

```
console.log(welcomeMessage());  // Output: "Welcome to ConvoBox!
        How can I assist you today?"
```

Exporting Modules: Multiple

> **Multiple Exports:**
> We can also export both named and default exports from a module. Let's say you're working on a weather application. You have both temperature units and a function to convert between them. Add the following to **module3.js**:

```
// module3.js
export const Celsius = "Celsius";
export const Fahrenheit = "Fahrenheit";

function convertTemperature(value, fromUnit, toUnit) {
  if (fromUnit === Celsius && toUnit === Fahrenheit) {
    return (value * 9/5) + 32;
  }
  if (fromUnit === Fahrenheit && toUnit === Celsius) {
    return (value - 32) * 5/9;
  }
  return "Invalid units";
}


export default convertTemperature;
```

In this example we are exporting the **Celsius** and **Fahrenheit** variables as named exports, and a **convertTemperature** function as our *default* export. To use these exports, at the top of **main.js** add the following:

```
import convertTemperature, { Celsius, Fahrenheit } from
        './module3.js'
```

Now, in the output section marked as MULTIPLE EXPORTS, add:

```
console.log(Fahrenheit);  // Output: "Fahrenheit"
console.log(Celsius);  // Output: "Celsius"
console.log(convertTemperature(0, Celsius, Fahrenheit));  //
        Output: 32
```

Renaming Imported Modules

It is also possible to rename variables or functions when we import them, using the **as** keyword. This can be useful if we want to avoid naming conflicts or if we just want to use a different name for the import for some reason. To see this in action, let's modify the first import statement in **main.js** as follows:

```
import { favoriteIngredients, combineIngredients,
        favoriteIngredients as favs, combineIngredients as
        combine } from './module1.js'
```

Here we are tacking on the same imports, but renamed using the **as** keyword. We add the imports **favoriteIngredients** as **favs** and **combineIngredients** as **combine**

The reason we're *adding* the renamed imports in this example instead of replacing the original imports is so that our other code in **main.js** doesn't break!

Now, in **main.js** in the RENAMING EXPORTS section, add the following (which should produce the same output as in the NAMED EXPORTS section):

```
console.log(favs); // Output: ["Salt", "Pepper", "Olive oil"]
console.log(combine(["Tomato", "Basil", ...favs])); // Output:
        "Combined ingredients: Tomato, Basil, Salt, Pepper,
        Olive oil"
```

Default exports can also be renamed when we import them and there's no difference in syntax or a need to use the **as** keyword. The **welcomeMessage** function from one of the previous examples we could just as easily have called "pollywog" or anything we want; it would still point to the same function we defined in our module as the default export:

```
// main.js
import pollywog from './module2.js'


// RENAMING EXPORTS section
pollywog() // Output: "Welcome to ChatterBox! How can I assist
        you today?"
```

Conclusion & Takeaways

ES6 modules provide a powerful way to organize and encapsulate code in modern JavaScript. We can export and import variables and functions, renaming them if we need to. Knowing this will also come in handy when using third-party packages in our projects, as we will see later. By using modules, we can write more organized, encapsulated, and maintainable code that can be easily shared across multiple files or projects.

Resources

Room to Grow

Use these links to help with the basics.

JavaScript Modules

The basics of modules from W3Schools.

Lift Off!

Once you are ready for the next step, use these links to continue practicing and deepen your knowledge.

JavaScript modules

Take a deeper dive with this guide from MDN. It gives you all you need to get started with JavaScript module syntax. You can also work through an example.

Modern JavaScript Tooling: Introduction

> ### Goals
>
> By the end of this lesson, you will understand:
> - What JavaScript tooling is
> - Why tooling can be helpful in modern JavaScript projects
> - Some of the different tooling packages available, and their purpose
>
> ### Introduction
>
> JavaScript tooling refers to a set of various software packages and processes used by developers to streamline their workflows and improve the quality of their code. These tools are designed to automate certain processes which are crucial to developing robust applications, such as bundling, linting, and testing, so that developers can focus on writing code rather than performing repetitive tasks.
>
> Effective and efficient tooling has become increasingly important as the complexity of web applications has grown. With modern web applications, developers are often working with large codebases, multiple modules and libraries, and a variety of different browsers and devices. Manually managing all aspects of development these days can be overwhelming, very time consuming, and also quite error-prone. Instead, in this environment, using the right tools can make all the difference in the world.

Overview

> Modern JavaScript tooling is a complex ecosystem of tools and processes, each designed to solve a specific problem. Some of the most important categories of tools in this ecosystem include:
>
> - **Build Tools and Compilers**: These tools are used to bundle and optimize code for production, as well as to enable features like code splitting, hot module replacement, and tree shaking. Some popular build tools include **Webpack**, **Rollup**, **Parcel** and **Vite**.
> - **Package Managers**: These tools are used to manage dependencies and install packages. Some popular package managers include **npm** and **yarn**.
> - **Linters**: These tools are used to analyze code for errors and enforce coding standards. A very common linter is **eslint**.
> - **Testing Frameworks**: These tools are used to automate testing of code, including unit testing, integration testing, and end-to-end testing. Some popular testing frameworks include **Jest**, **Mocha**, and **Jasmine**.
> - **Task Runners**: These tools are used to automate repetitive tasks, such as building, testing, and deployment. Some popular task runners include **Gulp** and **Grunt**.

In the following sections, we will explore each of these categories of tools in more detail.
>

Starter Files

You will notice in the file tree that we have a starter project set up with several different files – many of them configuration files for the various tools we'll be using, namely:

- `vite.config.js` - Configuration file for Vite
- `.eslintrc.cjs` and `.eslintignore` - Respectively, the configuration file and ignore file for ESLint
- `babel.config.cjs` - Configuration file for Babel
- `jest.config.js` - Configuration file for Jest
- `Gruntfile.cjs` - File to define tasks the Grunt task runner

> **NOTE**: You may be wondering about the **.cjs** file extensions. These are being used for the ESLint, Babel, and Grunt configuration files to explicitly indicate that these files are CommonJS modules, ensuring compatibility in a Node.js environment since Node does not yet fully support ES6 modules.

In addition there is a `src` directory with a some basic files we're a little more used to: **index.html**, **style.css**, and **main.js**. Inside the same directory, we also have a `modules` directory which has both regular JavaScript files and **test** files (denoted by the extension '.test.js').

The code in these files is the same code we saw in the previous lesson. A very basic UI has been created in order to actually use the code we wrote in a practical way. To see what it looks like, run the following command in the terminal:

```
npm run dev
```

Now refresh the preview window!

Build Tools and Compilers

Built Tools

Build tools are software tools that help developers bundle and package their code for deployment. The main options for build tools, as mentioned above, are: **Webpack**, **Rollup**, and **Parcel**. There is also a relative newcomer here called **Vite** (pronounced “veet”), which is gaining a lot of traction in this space and is what we are using in this project.
>
That said, let’s go over the other options mentioned above, to get a little more familiar with them.

Webpack is the gold standard when it comes to build tools. It is extremely configurable and thus extremely powerful. It can be used to bundle code, split code into smaller chunks, and optimize performance. Webpack can be set up to handle not only JavaScript files, but also HTML, CSS, Sass, and images. The only “downside” to Webpack is that setting up the configuration file can get quite complex.
>
Parcel, a main contender with Webpack, made its claim to fame as a “zero configuration” bundler. It’s easy to set up and get started with Parcel relatively quickly, and doesn’t require a config file – although you can include one if a specific need arises. We’ll look at using Parcel in the next lesson.
>

Compilers

As far as compilers go, the main one which you will see a lot is **Babel**. The purpose of Babel and its ecosystem of plugins is to allow us to use the newest features of JavaScript which are not yet supported by most browsers, and thus it is used a lot in frontend frameworks like React, for example. In our project, we are using Babel to enable us to use ES6 import syntax in our test files.
>

Package Managers and Linters

Package Managers

Package managers are software tools that help developers manage and install external libraries and dependencies in their projects. Some popular package managers for JavaScript projects are **npm** and **yarn** (our project uses **npm**).

With either of these, we can install external libraries and other 3rd party packages using a command like the following:

`npm install {NAME OF PACKAGE}` OR `yarn add {NAME OF PACKAGE}`

This will download the latest version of package we specify (without the curly braces – this is just used here to enclose our placeholder package name) and add that package as a dependency in a project's **package.json** file (more on **package.json** files in the next lesson).
>

Linters

Linters are software tools that help developers identify errors and potential bugs in their code. One of the most popular linters out there is ESLint. The main benefit of using a linter is that it helps to ensure a consistent coding style or standard across an entire project by defining specific rules in a configuration file.

Let's add the following to **.eslintrc.cjs**:
>

```
// .eslintrc.cjs
module.exports = {
  parserOptions: {
    sourceType: 'module',
    ecmaVersion: 2015
  },
  rules: {
    'semi': ['error', 'always'],
    'quotes': ['error', 'double'],
  }
};
```

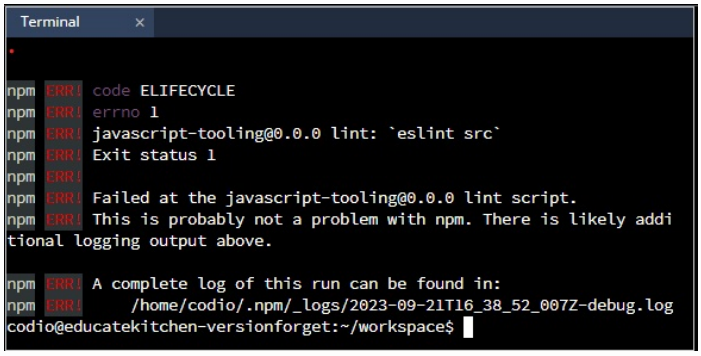
In this example, we start by defining `parserOptions` to set up ESLint for ES6 modules. Then, three rules are defined for linting our code: semicolons required at the end of statements, and double quotes required for strings. There are, of course, many other rules which can be defined here, as well as whether any code breaking a particular rule will throw an error (as defined here) or simply a warning; or, in certain cases, it can even be useful to tell ESLint to ignore a certain rule.

When ESLint is run in a project, any rules not explicitly defined in the config file will use the linter's default settings for that particular rule.
>

To test out ESLint, first we need to stop Vite's development server. To do this, click inside the terminal and press `CTRL + C`. Next, type the following command in the terminal and hit enter:

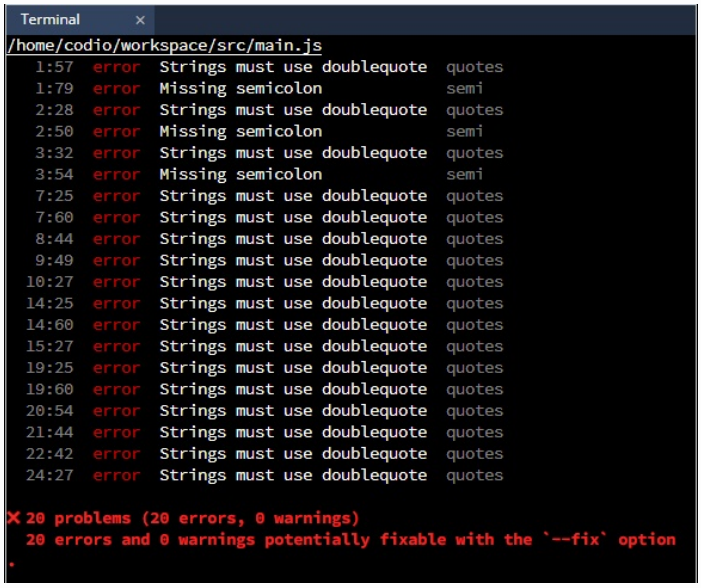
```
npm run lint
```

After running this command you will likely see something like the following in the terminal:



ESLint output

While it looks like something went wrong, it actually didn't. We're getting this error message at the end of ESLint's output because we set up our config to throw errors for the rules we defined. If we scroll up a little in the terminal we will find the output we're looking for, which looks like this:



alt text

Now let's run a different command from `package.json`, which adds the `--fix` flag to the **lint** command. In the terminal, type the following and press enter:

```
npm run lint:fix
```

While it may look like that did nothing, open up **main.js** and have a look at it now: all the single quotes have been replaced with double quotes, and semicolons have been added where they were missing (i.e. - the import

```
statements)!
>
```

Testing Frameworks

Testing frameworks are software tools that help developers write and run automated tests for their code to ensure that things are working as expected. Some popular testing frameworks include **Jest**, **Mocha**, and **Jasmine**. There are a couple of new testing libraries, however, such as **Vitest** (from the creators of Vite) and **Cypress**.

Our project here will use Jest.

Let's add some tests for our module files:

module1.test.js

```
import { favoriteIngredients, combineIngredients } from
        "./module1";



test("Should combine ingredients into a string", () => {
  const result = combineIngredients(favoriteIngredients);
  expect(result).toBe("Combined ingredients: Salt, Pepper, Olive
        Oil");
});

test("Should combine custom ingredients into a string", () => {
  const customIngredients = ["Garlic", "Onion"];
  const result = combineIngredients(customIngredients);
  expect(result).toBe("Combined ingredients: Garlic, Onion");
});

test("Should combine favorite ingredients and custom ingredients
        into a string", () => {
  const result = combineIngredients([...favoriteIngredients,
        "Garlic", "Onion"]);
  expect(result).toBe("Combined ingredients: Salt, Pepper, Olive
        Oil, Garlic, Onion")
});
```

This test file contains three tests for imports from **module1.js**.

`Should combine ingredients into a string`: This test verifies that calling the combineIngredients() function with the favoriteIngredients array returns a string with the ingredients combined and prefixed by "Combined ingredients:".

`Should combine custom ingredients into a string`: This test uses a custom list of ingredients (“Garlic”, “Onion”) to check if combineIngredients() works correctly with different inputs, again returning a string prefixed with “Combined ingredients:”.

`Should combine favorite ingredients and custom ingredients into a string`: This test concatenates favoriteIngredients and a custom list (“Garlic”, “Onion”) to test if combineIngredients() correctly combines and outputs all the ingredients in a string.

module2.test.js

```
import welcomeMessage from "./module2";

test("Should return the welcome message", () => {
  expect(welcomeMessage()).toBe("Welcome to ConvoBox! How can I
        assist you today?");
});
```

This test file contains a single test for the `welcomeMessage` function imported from **module2.js**.

`Should return the welcome message`: This test checks if calling welcomeMessage() returns the expected welcome message: “Welcome to ConvoBox! How can I assist you today?”.

module3.test.js

```
import convertTemperature, { Celsius, Fahrenheit } from
        "./module3";

test("Should convert Celsius to Fahrenheit", () => {
  expect(convertTemperature(0, Celsius, Fahrenheit)).toBe(32);
});

test("Should convert Fahrenheit to Celsius", () => {
  expect(convertTemperature(32, Fahrenheit, Celsius)).toBe(0);
});

test("Should return 'Invalid units' for invalid units", () => {
  expect(convertTemperature(0, "invalid",
        "invalid")).toBe("Invalid units");
});
```

This test file contains three tests for the `convertTemperature` function and constants (`Celsius`, `Fahrenheit`) imported from **module3.js**.

`Should convert Celsius to Fahrenheit`: This test verifies that converting a temperature of 0 degrees Celsius to Fahrenheit returns 32 degrees.

`Should convert Fahrenheit to Celsius`: This test checks if converting a temperature of 32 degrees Fahrenheit to Celsius returns 0 degrees.

`Should return "Invalid units" for invalid units`: This test makes sure that passing invalid units to the convertTemperature() function will result in the string “Invalid units”.

Each of these tests is designed to exercise a particular “unit” of functionality in its respective module, hence why these kinds of tests are called “unit tests”.

Let’s now try to run our tests to see if everything is working as expected. In the terminal, type the following command and hit enter:

```
npm run test
```

Task Runners

Task runners are software tools that help developers automate various tasks in the development process. Two main task runners out there are **Grunt** and **Gulp**. Our project uses Grunt.
>

What we're going to do here is automate the task of running our tests. To do that, add the following in **Gruntfile.cjs**:

```
module.exports = function (grunt) {

  grunt.initConfig({
    // Configure Jest
    jest: {
      options: {
        coverage: true,
        testPathPattern: '**/*.test.js'
      }
    },

    // Configure watch task
    watch: {
      js: {
        files: ['**/*.test.js'],
        tasks: ['jest']
      }
    }
  });

  // Load Grunt plugins
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.registerTask('jest', 'Run Jest', function () {
    const done = this.async();
    const exec = require('child_process').exec;
    exec('npx jest --config=jest.config.js', function (err,
        stdout, stderr) {
      console.log(stdout);
      console.log(stderr);
      done(err ? false : true);
    });
  });

  // Register Grunt tasks
  grunt.registerTask('default', ['jest', 'watch']);
  grunt.registerTask('test', ['jest']);
};
```

> Here, we are telling Grunt to run Jest tests with coverage information and targets files that match the pattern **/*.test.js. A custom Grunt task is defined to execute the Jest tests via Node's child_process module, allowing for more customized control over the test execution. Finally, the watch task is configured to monitor changes in all .test.js files and rerun the Jest tests automatically when changes are detected, making the development process more efficient.
>
> To test this out, type the following command in the terminal and press enter:

```
npm run grunt
```

Build for Production

Last but not least, pretending that we've finished developing our project, let's use Vite to build our code for production. To do this, we will use the included "build" command shown in our `package.json` file. In the terminal, type the following and then press enter:

```
npm run build
```

This will create a directory at the root of our project called `dist`, and output our production build there!
>

Conclusion & Takeaway

> JavaScript tooling is an essential part of modern web development. With the help of various software tools, developers can automate tasks, streamline workflows, and improve the quality of their code. In this lesson, we provided an overview of some of the most popular tools used in modern JavaScript development, including build tools, package managers, linters, testing frameworks, and task runners.
>
> As a developer, it's important to understand the various tools available and how they can be used to improve your workflow and productivity. By incorporating these tools into your development process, you can focus more on writing high-quality code and less on tedious, repetitive tasks.

Everything we have done in this lesson barely scratches the surface of what you can do with the tools we covered; what we went over was just to give you an introduction. It would be beneficial to spend some time further researching these tools, and to that end be sure to check out some of the resources listed below!
>
Resources

Build Tools and Compilers

- Webpack
- Rollup
- Parcel
- Vite
- Babel

Package Managers

- npm
- Yarn

Linters

- ESLint
- JSHint
- JSLint

Testing Frameworks

- Jest
- Mocha
- Jasmine
- Cypress
- Vitest

Task Runners

- Gulp
- Grunt

Templating Engines: Introduction

> ## Goals
>
> By the end of this lesson you will:
> - Have a basic understanding of what template engines are, and how they can be used on the client side
> - Be familiar with the Handlebars template engine
> - Be introduced to a rising star in the JavaScript world: Astro!
> ### Introduction
> JavaScript template engines allow developers to separate the presentation of the web app (i.e., the HTML, CSS, and JavaScript) from the business logic. It provides a way to define templates, which are a combination of static and dynamic content, and then use JavaScript to render the templates with data. This allows you to easily update the view of your application based on changes in the data.
>
> The other big advantage of template engines is that they can help reduce code repetition, which is the feature we will focus on in this lesson. We'll go over the basics of using Handlebars, which is one of the most popular template engines. In addition, we'll look at a relative newcomer in the JavaScript world called Astro which, although technically a framework for static site generation, can nonetheless be used (at its most basic) as a template engine.
>
> ## Business Context
>
> Most commonly, template engines are used on the server side in order to generate all the HTML and fill it with the relevant data before sending it to the browser. It is possible, however, to use template engines on the client side; it just takes a little extra configuration to make it work. This use case is what we will focus on in this lesson.

Handlebars - Getting Started

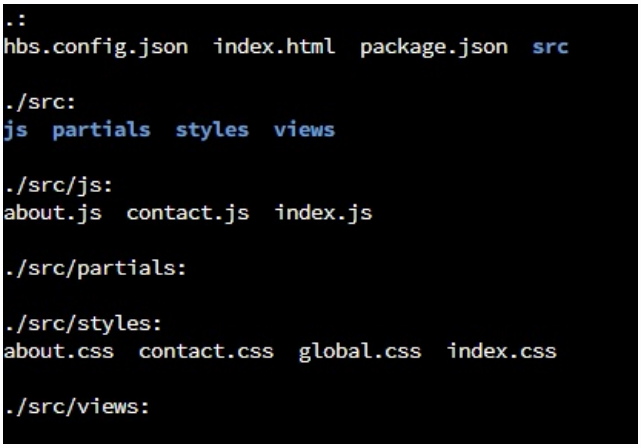
Starter files

> To save time, some starter files have already been created for our Handlebars project. Let's briefly go over some of these and what purpose they serve.

Run the following command in the terminal, which will print out all the contents of our project directory (including subdirectories) and add an empty line below everything so it's easier to read:

```
ls -R && printf "\n"
```

After running this command, you should see the following in the terminal:



src dir contents

Everything in blue denotes a directory.
>
You will notice in the root of our project directory (outside the **src** folder, alongside the config files and the **package.json** file) there is a file called **index.html** (click to open). This is the base template which we will convert to Handlebars, along with creating a couple additional pages so we can see why using a template engine can be helpful even on the client side – namely, in order to not have to repeat a lot of the same HTML, and so that we can generate pages more dynamically. Finally, we will use a bundler called Parcel to compile the template engine files to regular HTML.
>

Config Files

package.json

Let's take a quick look at the **package.json** file (click to open). In a nutshell, this file acts as a sort of manifest file for projects which use Node.js (which this one does), and holds various metadata about the given project – such as the name, version, description, any dependencies (external libraries or modules that the project depends on), any custom scripts to be executed from the command line, license and author info, etc.

The main part of **package.json** which is of interest to us at the moment is the scripts section, which should look like this:

```
"scripts": {
  "build:dev": "rm -rf .parcel-cache/ dist/ && parcel build --
no-source-maps --public-url /dist/ --no-optimize
src/views/**/*.hbs",
  "build:prod": "rm -rf .parcel-cache/ dist/ && parcel build --
no-source-maps --public-url /dist/ src/views/**/*.hbs"
}
```

This section contains all the commands we can execute in the terminal, prepended by "npm run". For example, running "npm run build:dev" from the terminal in the same directory as the **package.json** file will execute our build script, which first deletes the **.parcel-cache**/ and **.dist**/ directories if they exist, then tells Parcel to compile any Handlebars files in our **src** directory.
>
The main difference between the "dev" and "prod" scripts here is whether or not the output files will get optimized and minified (all whitespace removed and everything smushed together as a single line).

While still developing an application, minification isn't necessary. However, when we're ready to build everything for production and deploy our site to some hosting provider, it's generally a good idea to minify as much as possible because it reduces file size and thus increases page load times.
>
.parcelrc
The first config file we're going to look at is the file called **.parcelrc** (click to open). This file relates to the Parcel module bundler. Go ahead and open this file and you will see the following:

```
{
  "extends": "@parcel/config-default",
  "transformers": {
    "*.{hbs,handlebars}": ["@partiellkorrekt/parcel-transformer-
handlebars"]
  }
}
```

As mentioned in the previous lesson, Parcel is a “zero config” bundler; this means that in a majority of use cases a **.parcelrc** file is not even necessary and everything just works right out of the box. However, in our use case, since we’re dealing with a template engine and Parcel doesn’t automatically know what to do with those files, we need to use extensions called transformers and tell Parcel to use those extensions when compiling our project.

That’s what’s going on in **.parcelrc**: we’re telling Parcel which transformer to use in order to compile our Handlebars files.
>
hbs.config.json
This file is responsible, in turn, for telling the Parcel Handlebars transformer in which directory to look for what are called “partials” (partial templates), which we’ll go over shortly. Go ahead and click to open the file. Inside you should see this:

```
{
  "partials": "src/partials"
}
```

Here we’re saying that all our partials will be located inside a **partials** directory in **src**.
>

Handlebars - Syntax Overview

Syntax Overview

Handlebars uses a simple syntax based on double curly braces `{{ }}` to enclose expressions, and it provides a rich set of features such as conditionals, variables, loops, helpers, and what are called 'partials' to make it easy to generate dynamic HTML.
>
For our purposes here, we'll only be using a conditional, as well as variables and partials, but below is a general overview of the syntax:
>
Outputting a variable:

```
<div>{{title "Default Title"}}</div>
```

Outputting a variable with a default value (in case one isn't passed to the template):

```
<div>{{title "Default Title"}}</div>
```

Conditional template:

```
{{#if content}}
  <p>{{content}}</p>
{{/if}}
```

Loop template:

```
<ul>
{{#each items}}
  <li>{{this}}</li>
{{/each}}
</ul>
```

Using partials:

```
<div>
{{> myPartial}}
</div>
```

Including a partial and passing variables:

```
{{> myPartial
  title="Dynamic Title"
}}
```

Handlebars syntax is easy to learn and use, and it provides a powerful set of features that make it a popular choice for many web applications. It's easy to read, write and understand, and it's a good option for small and large projects.
>
To learn more, check out the documentation.
>

Handlebars - Create templates and partials

> First, let's install all of our project dependencies for Parcel and Handlebars. In the terminal, run the following command:

```
npm install
```

Next, in the terminal let's run the following command to create the files we'll need for our templates and partials:

```
touch src/partials/footer.hbs src/partials/header.hbs
src/partials/htmlBegin.hbs src/partials/htmlEnd.hbs
```

> We'll start by working on our header and footer templates. Open up **header.hbs** by clicking , then add the following:

```
<header class="header container">
  <a href="/dist" class="brand">Awesome <span>Brand</span></a>
  <nav class="main-nav">
    <ul class="main-nav__list">
      <a href="/dist/about" class="main-nav__item">
        <li>About</li>
      </a>
      <a href="/dist/contact" class="main-nav__item">
        <li>Contact</li>
      </a>
      <a href="#" class="main-nav__item cta">
        <li>Signup</li>
      </a>
    </ul>
  </nav>
</header>
```

> **NOTE:** For any of the files we create from the terminal, if the link to open them doesn't work try resyncing the file tree (**Project –> Resync File Tree**), then try clicking the link again.

Now, open up **footer.hbs** by clicking and add the following:

```
<footer id="footer" class="footer container">
  <div class="footer__content">
    &copy; 2023 AwesomeBrand. All rights reserved.
  </div>
</footer>
```

Next, let's write partials to hold the beginning part of any given HTML document (including our header) and the end part of any given HTML document (including our footer) so that we don't have to worry about writing either of these on every page we create:
>
Open up **htmlBegin.hbs** by clicking and add the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
        scale=1.0">
  <title>
    {{ title }}
  </title>


  {{#if global}}
  {{{ global }}}
  {{/if}}

  {{#if styles }}
  {{{ styles }}}
  {{/if}}
</head>
<body>
  {{> header }}
```

> Next, open **htmlEnd.hbs** by clicking and add the following

```
{{> footer}}


{{#if script }}
{{{ script }}}
{{/if}}
</body>
</html>
```

Handlebars - Create pages

Now it's time to create our pages inside the **src/views** directory. Inside this directory, first create a file called **index.hbs** by running the following in the terminal:

```
touch src/views/index.hbs
```

This will be our home page. Click to open the file.
>
Inside **index.hbs** add the following:

```
{{> htmlBegin
  title="Home"
  global="<link type='text/css' rel='stylesheet'
        href='../styles/global.css'>"
  styles="<link type='text/css' rel='stylesheet'
        href='../styles/index.css'>"
}}


  <main class="main container">
    <h1>HOME PAGE</h1>
  </main>


{{> htmlEnd
  script="<script src='../js/index.js'></script>"
}}
```

Note that in this file we're defining the page structure by including our **htmlBegin** partial at the top, followed by the main content of our home page and our **htmlEnd** partial. This will be the same on each page we create.

Next, create two other page files inside **src/views** – one called **about.hbs**, and one called **contact.hbs**:

```
touch src/views/about.hbs src/views/contact.hbs
```

Now, click to open the files. Inside **about.hbs** add the following:

```
{{> htmlBegin
  title="About"
  global="<link type='text/css' rel='stylesheet'
        href='../styles/global.css'>"
  styles="<link type='text/css' rel='stylesheet'
        href='../styles/about.css'>"
}}


  <main class="main container">
    <h1>ABOUT PAGE</h1>
  </main>


{{> htmlEnd
  script="<script src='../js/about.js'></script>\n"
}}
```

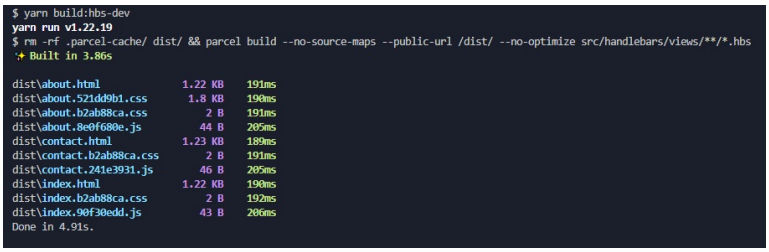
Inside **contact.hbs** add the following:

```
{{> htmlBegin
  title="Contact"
  global="<link type='text/css' rel='stylesheet'
        href='../styles/global.css'>"
  styles="<link type='text/css' rel='stylesheet'
        href='../styles/contact.css'>"
}}


  <main class="main container">
    <h1>CONTACT PAGE</h1>
  </main>


{{> htmlEnd
  script="<script src='../js/contact.js'></script>"
}}
```

Perfect! Now, we can go to the terminal and run the command to compile our Handlebars files (we'll use the **dev** command for now):

```
npm run build:dev
```

If everything compiled successfully and there were no errors, you should see something like this in the terminal:



terminal successful compile

Now in the file tree, expand the newly generated **dist**/ directory and have a look at the results of running the build command; in particular, open all three HTML files and look them over to the markup that was generated from our Handlebars files!

Astro - Getting Started

> Okay, now let's have a look at Astro!

Creating a new Astro project

Normally, to create an Astro project we would run the following in the terminal (DON'T ACTUALLY RUN THIS COMMAND):

```
npm create astro@latest
```

Then, we would follow the CLI prompts to finish generating all the project files. However, to save time, this has already been done.

Check Node.js Version ⚠

The first thing we will need to do in order to use Astro is to check our Node.js installation and make sure it is at least **v18.14.1**. To do that, run the following command in the terminal:

```
node -v
```

If our version is lower than the version specified above, we'll need to update it. To update the Node version, go to **Project -> Settings**, then click on the **Stack Settings** tab. From there, click the **Current Stack** input, then click the **Certified** tab and in the search box type "Node". The one we want is called **CODIO-CERTIFIED / Node 18 Ubuntu 22.04**; once you find it in the list (it should be the first one), click the "Select" button. Then, click "Save". If prompted to enter a code to confirm, enter the code shown in the popup.

Once all of that is done, run `node -v` in the terminal again, and if it now shows version **18.14.1** or higher, we are good to go!

Starter and config Files

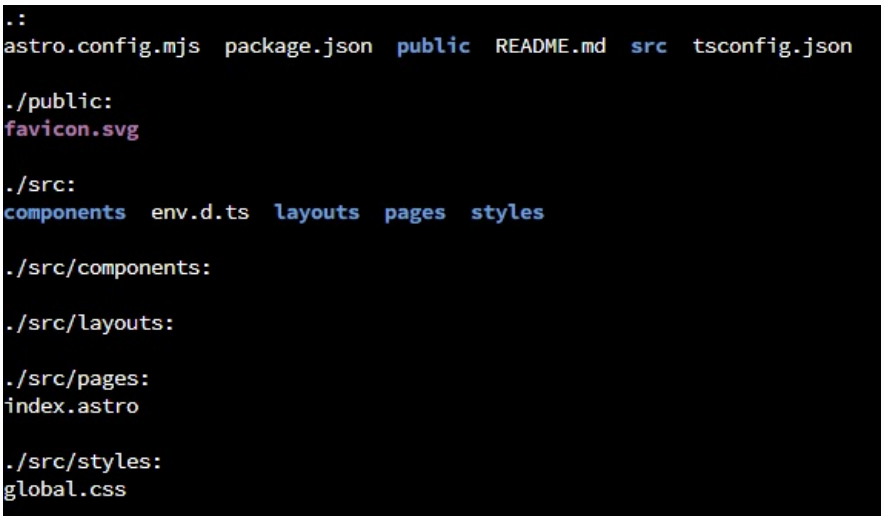
Before we go over the starter and config files, let's be sure to install the dependencies for our project. In the terminal, run the following:

```
npm install
```

Once everything finishes installing, let's take a look inside this directory. In the terminal, run that same command we used at the beginning of the Handlebars lesson:

```
ls -R --ignore='node_modules' && printf "\n"
```

After running this command, you should see the following in the terminal:



>

You'll notice a few config files throughout our project: **env.d.ts** (in the **src** folder), **astro.config.mjs**, and **tsconfig.json**. You don't need to worry about **env.d.ts** and **tsconfig.json**; these two files have to do with TypeScript (a superset of the JavaScript language) which Astro projects currently use by default. We won't be doing anything with TypeScript here, though, so you can ignore those two files.

>

The **astro.config.mjs** and **package.json** files, however, are important; go ahead and click to open them. Inside **astro.config.mjs**, you should see the following:

>

```
import { defineConfig } from 'astro/config';

// https://astro.build/config
export default defineConfig({});
```

And in **package.json**:

```
{
  "name": "astro-example",
  "type": "module",
  "version": "0.0.1",
  "scripts": {
    "dev": "astro dev",
    "start": "astro dev",
    "build": "astro build",
    "preview": "astro preview",
    "astro": "astro"
  },
  "dependencies": {
    "astro": "^2.5.6"
  }
}
```

> We already talked about the purpose of **package.json**, but let's briefly discuss **astro.config.mjs**. It is an important file in an Astro project because it allows you to configure various aspects of the project – such as build options, rendering settings, and custom plugins. It serves as the central configuration file for your Astro project, providing a way to tailor the build process and behavior of your application to your specific needs.

Astro - Syntax Overview

Syntax Overview

In order to give some idea of the syntax of Astro, it will be easiest to show an example of an Astro component and an Astro layout file.

An Astro component typically consists of three parts:

- **Component Script**: Defined at the top of a component file, enclosed by triple-dashes (`---`). This is where you can import other components, define variables, create functions, and set up component props.
- **Component Template**: The main body of the component, where you write your HTML markup and use the variables, functions, and imported components defined in the script section.
- **Scoped Styles** (optional): CSS styles specific to the component. These styles won't interfere with other components, even if you use the same class names.

Let's look at an example of an Astro component and an Astro layout file.

A basic component:

```
---
// Component Script
import Layout from '../layouts/Layout.astro'

const heading = "Home Page"

const someFunction = () => {
  // execute some code
}
---

<!-- Component Template -->
<Layout title="Home | My Awesome Website">
  <main slot="content">
    <h1 id="heading">{heading}</h1><br>
    <p>Lorem ipsum dolor sit amet</p>
  </main>
</Layout>


<!-- Scoped Component Styles -->
<style>
  #heading {
    color: steelblue;
  }

  p {
    font-size: 1rem;
  }
</style>
```

Layout:

```
---
// Component Script
import Header from '../components/header.astro'
import Footer from '../components/footer.astro'

const { title } = Astro.props;
---

<!-- Component Template -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width" />
    <meta name="generator" content={Astro.generator} />
    <title>{title}</title>
  </head>
  <body>
    <!-- header component -->
    <Header />

    <!-- Placeholder for each page's <main> element -->
    <slot name="content" />

    <!-- footer component -->
    <Footer />
  </body>

</html>
```

To learn more, check out the documentation.
>

Astro - Create layout, header and footer

> Let's begin by doing in our Astro project what we did with Handlebars!
>
> From the terminal, create the following two files: **footer.astro**, and **header.astro** inside the `src/components` directory, as well as a **Layout.astro** file (case-sensitive) in the `src/layouts` directory, using the following command:

```
touch src/components/header.astro src/components/footer.astro
src/layouts/Layout.astro
```

Next, click to open the newly created files, and enter the following in **header.astro**

```
<header class="header container">
  <a href="/" class="brand">Awesome <span>Brand</span></a>
  <nav class="main-nav">
    <ul class="main-nav__list">
      <a href="/about" class="main-nav__item">
        <li>About</li>
      </a>
      <a href="/contact" class="main-nav__item">
        <li>Contact</li>
      </a>
      <a href="#" class="main-nav__item cta">
        <li>Signup</li>
      </a>
    </ul>
  </nav>
</header>
```

Now enter the following in **footer.astro**

```
<footer id="footer" class="footer container">
  <div class="footer__content">
    &copy; 2023 CompanyName. All rights reserved.
  </div>
</footer>
```

Next, let's work on our layout template, which will hold the beginning part of any given HTML document (including our header) and the end part of any given HTML document (including our footer). Add the following in **Layout.astro**:
>

```
---
import Header from '../components/header.astro'
import Footer from '../components/footer.astro'

import '../styles/global.css'

const { title } = Astro.props;
---

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width" />
    <meta name="generator" content={Astro.generator} />
    <title>{title}</title>
  </head>
  <body>
    <Header />


    <slot />

    <Footer />
    <slot name="script"/>
  </body>


</html>
```

Astro - Create pages

Let's now create our page files inside the **src/pages** directory. We'll also add some scoped styles to them which would otherwise be conflicting, so we can see how that works. Create **index.astro**, **about.astro**, and **contact.astro** by running the following in the terminal:

```
touch src/pages/index.astro src/pages/about.astro
src/pages/contact.astro
```

Click to open the newly created files.

Inside **index.astro** add the following:

```
---
import Layout from '../layouts/Layout.astro'
---

<Layout title="Home">
  <main class="main container">
    <h1>HOME PAGE</h1>
  </main>


  <script is:inline slot="script" src="/js/index.js"></script>
</Layout>


<style>
  h1 {
    color: steelblue;
  }
</style>
```

Next, inside **about.astro** add the following:

```
---
import Layout from '../layouts/Layout.astro'
---

<Layout title="About">
  <main class="main container">
    <h1>ABOUT PAGE</h1>
  </main>

  <script is:inline slot="script" src="/js/about/index.js">
        </script>
</Layout>



<style>
  h1 {
    color: darkgreen;
  }
</style>
```

Finally, inside **contact.astro** add the following:

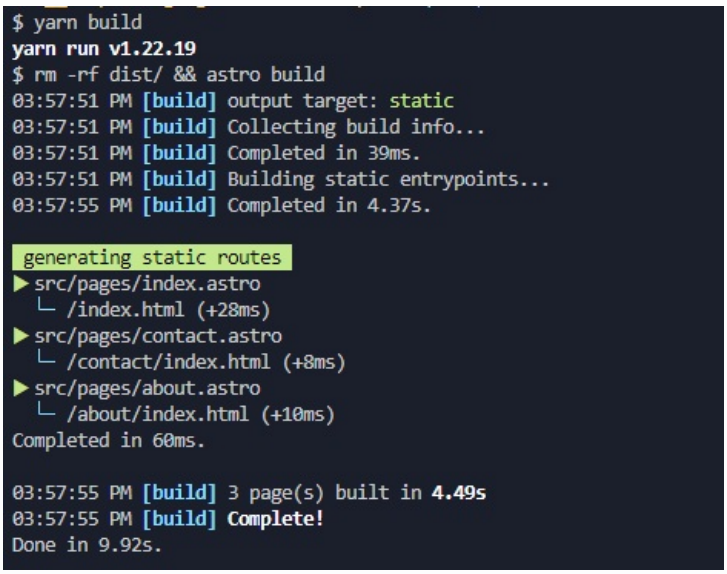
```
---
import Layout from '../layouts/Layout.astro'
---

<Layout title="Contact">
  <main class="main container">
    <h1>CONTACT PAGE</h1>
  </main>

  <script is:inline slot="script" src="/js/contact/index.js">
        </script>
</Layout>



<style>
  h1 {
    color: blueviolet;
  }
</style>
```

Build Astro Project

Now, we can go to the terminal and run the command to compile our Astro files.

```
npm run build
```

If everything compiled successfully, you should see something like this in the terminal:



terminal successfully compiled

Now in the file tree, as with the classic template engines we looked at previously, expand the newly generated **dist**/ directory and have a look at the results of running the build command!

We are not deploying this project, however Astro provides a tutorial to Build your First Astro Blog!

Conclusion & Takeaways

> In this lesson, you learned about one of the most popular template engines: Handlebars! Although template engines are traditionally used server-side they can be also be used client-side, namely for the benefit of reducing repetition of HTML. We also had a look at Astro – a new static-site-generating JavaScript framework that is steadily growing in popularity, and can be used in much the same way as a template engine.

Resources

Handlebars

- Handlebars.js

Astro

- Astro Documentation

Asynchronous Elective Learning

There is an optional assignment associated with this lesson which covers two additional popular template engines: EJS and Pug. If you would like to learn more about these, see the lesson called **More Template Engines (AEL)**!

More Template Engines: Introduction

> **Goals**
>
> By the end of this lesson you will:
> - Have a basic understanding of the EJS and Pug template engines and how to use them

Introduction

In this Asynchronous Elective Learning lesson, we will continue our exploration of JavaScript template engines and how to use them in frontend development. We will cover two more of the most popular template engines (EJS and Pug), including configuration, syntax and how to utilize templates, layouts, and partials to generate multiple HTML pages.

EJS - Getting Started

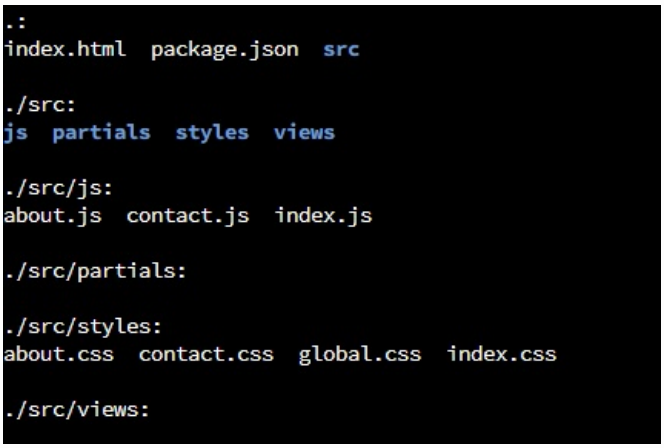
Starter files

> To save time, some starter files have already been created for our EJS project. Let's briefly go over a couple of these.

Run the following command in the terminal, which will print out all the contents of our project directory (including subdirectories) and add an empty line below everything so it's easier to read:

```
ls -R && printf "\n"
```

After running this command, you should see the following in the terminal:



src dir contents

Everything in blue denotes a directory.
>
You will notice in the root of our project directory (outside the **src** folder, alongside the **.parcelrc** file and the **package.json** file) there is the **index.html** file (click to open). This is the same base template we used in the Handlebars lesson, and which we will now convert to EJS, along with creating a couple additional pages as we did before. Finally, we will again use the Parcel bundler to compile the template engine files to regular HTML.
>

Config Files

package.json

Let's take a quick look at the **package.json** file (click to open). Everything in here is pretty much the same as in the Handlebars lesson, except the scripts section is slightly different to reflect the fact that we're working with EJS files now:

```
"scripts": {
  "build:dev": "rm -rf .parcel-cache/ dist/ && parcel build --
no-source-maps --public-url /dist/ --no-optimize
src/views/**/*.ejs",
  "build:prod": "rm -rf .parcel-cache/ dist/ && parcel build --
no-source-maps --public-url /dist/ src/views/**/*.ejs",
}
```

Again, to execute either of these commands in the terminal, we would simply prepend them with "npm run". For example, running "npm run build:dev" from the terminal in the same directory as the **package.json** file will execute our build script, which if you recall first deletes the **.parcel-cache**/ and **.dist**/ directories if they exist, then tells Parcel in this case to compile any EJS files in our **src** directory.
>
As a reminder, the main difference between the "dev" and "prod" scripts is whether or not the output files will get optimized and minified (all whitespace removed and everything smushed together as a single line).

> **.parcelrc**
>
> The only other config file we'll need here is **.parcelrc** (click to open). Inside this file you will see the following:

```
{
  "extends": "@parcel/config-default",
  "transformers": {
    "*.ejs": ["parcel-transformer-ejs"]
  }
}
```

This looks very similar to what we saw before with Handlebars, except now we're telling Parcel to use a different transformer to compile our EJS files.
>

EJS - Syntax Overview

> EJS (which stands for Embedded JavaScript) is a simple and popular templating engine that allows you to embed JavaScript code in HTML files to generate dynamic pages. It uses `<% %>` tags to enclose JavaScript code, and `<%= %>` tags to output expressions.
>
> Outputting a variable:

```
<div><%= title %></div>
```

Conditional template:

```
<% if (content) { %>
  <p><%= content %></p>
<% } %>
```

Loop template:

```
<ul>
  <% items.forEach((item) => { %>
    <li><%= item %></li>
  <% }) %>
</ul>
```

Including a partial:

```
<%- include('myPartial') %>
```

Passing variables:

```
<%- include('myPartial', {
  title: "Dynamic Title"
}) %>
```

Using functions:

```
<% const upper = (str) => str.toUpperCase() %>
<div><%= upper(title) %></div>
```

EJS syntax is also easy to learn and use, is good for projects of all sizes, and has the added bonus of being able to write regular JavaScript code directly inside HTM, as depicted above.

To learn more, check out the documentation.
>

EJS - Create templates and partials

> Now, let's do the same thing for EJS that we did for Handlebars!

First, let's install all of our project dependencies for EJS. In the terminal, run the following command:

```
npm install
```

Next, in the terminal let's run the following command to create the files we'll need for our templates and partials:

```
touch src/partials/footer.ejs src/partials/header.ejs
src/partials/htmlBegin.ejs src/partials/htmlEnd.ejs
```

> We'll start by working on our header and footer templates. Open up **header.ejs** by clicking , then add the following:

```
<header class="header container">
  <a href="/dist" class="brand">Awesome <span>Brand</span></a>
  <nav class="main-nav">
    <ul class="main-nav__list">
      <a href="/dist/about" class="main-nav__item">
        <li>About</li>
      </a>
      <a href="/dist/contact" class="main-nav__item">
        <li>Contact</li>
      </a>
      <a href="#" class="main-nav__item cta">
        <li>Signup</li>
      </a>
    </ul>
  </nav>
</header>
```

> **NOTE:** For any of the files we create from the terminal, if the link to open them doesn't work try resyncing the file tree (**Project –> Resync File Tree**), then try clicking the link again.

Now, open up **footer.ejs** by clicking and add the following:

```
<footer id="footer" class="footer container">
  <div class="footer__content">
    &copy; 2023 CompanyName. All rights reserved.
  </div>
</footer>
```

Next, let's write partials to hold the beginning part of any given HTML document (including our header) and the end part of any given HTML document (including our footer):
>
Open up **htmlBegin.ejs** by clicking and add the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
        scale=1.0">
  <title>
    <%= title %>
  </title>



  <% if (styles) { %>
    <%- styles %>
  <% } %>
</head>
<body>
  <%- include('header') %>
```

Next, open **htmlEnd.ejs** by clicking and add the following

```
<%- include('footer') %>



<% if (script) { %>
  <%- script %>
<% } %>
</body>
</html>
```

EJS - Create pages

Now it's time to create our pages inside the **src/views** directory. Inside this directory, first create a file called **index.ejs** by running the following in the terminal:

```
touch src/views/index.ejs
```

This will be our home page. Click to open the file.
>
Inside **index.ejs** add the following:

```
<%- include('../partials/htmlBegin', {
  title: "Home",
  styles: "<link rel='stylesheet' href='../styles/global.css'>
        \n <link rel='stylesheet' href='../styles/index.css'>"
}) %>


  <main class="main container">
    <h1>HOME PAGE</h1>
  </main>


<%- include('../partials/htmlEnd', {
  script: "<script src='../js/index.js'></script>"
}) %>
```

Note that in this file we're defining the page structure by including our **htmlBegin** partial at the top, followed by the main content of our home page and our **htmlEnd** partial. This will be the same on each page we create.

Next, create two other page files inside **src/views** – one called **about.ejs**, and one called **contact.ejs**:

```
touch src/views/about.ejs src/views/contact.ejs
```

Now click to open the files. Inside **about.ejs** add the following:

```
<%- include('../partials/htmlBegin', {
  title: "About",
  styles: "<link rel='stylesheet' href='../styles/global.css'>
        \n <link rel='stylesheet'
        href='../styles/about/index.css'>"
}) %>


  <main class="main container">
    <h1>ABOUT PAGE</h1>
  </main>


<%- include('../partials/htmlEnd', {
  script: "<script src='../js/about/index.js'></script>"
}) %>
```

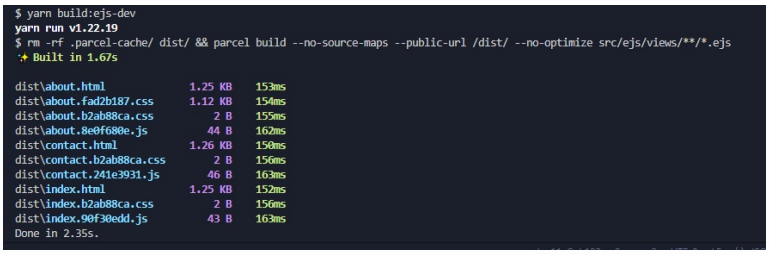
Inside **contact.ejs** add the following:

```
<%- include('../partials/htmlBegin', {
  title: "Contact",
  styles: "<link rel='stylesheet' href='../styles/global.css'>
        \n <link rel='stylesheet'
        href='../styles/contact/index.css'>"
}) %>


  <main class="main container">
    <h1>CONTACT PAGE</h1>
  </main>


<%- include('../partials/htmlEnd', {
  script: "<script src='../js/contact/index.js'></script>"
}) %>
```

Perfect! Now, we can go to the terminal and run the command to compile our EJS files (as with Handlebars, we'll use the dev command):

```
npm run build:dev
```

If everything compiled successfully, you should see something like this in the terminal:



terminal successful compile

Now in the file tree, as with Handlebars, expand the newly generated **dist**/ directory and have a look at the results of running the build command!

Pug - Getting Started

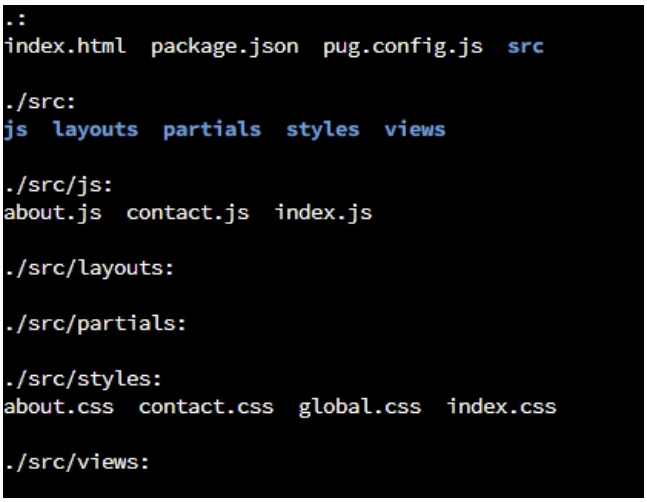
Starter files

> To save time, some starter files have already been created for our Pug project. As we did with EJS, let's briefly go over a couple of these.

Run the following command in the terminal, which will print out all the contents of our project directory (including subdirectories) and add an empty line below everything so it's easier to read:

```
ls -R && printf "\n"
```

After running this command, you should see the following in the terminal:



src dir contents

Everything in blue denotes a directory.
>
As before, you will notice in the root of our project directory we have our **index.html** file (click to open). This is the same base template we used in the Handlebars and EJS lessons, and which we will now convert to Pug, along with the About and Contact pages. Finally, we will once again use the Parcel bundler to compile the template engine files to regular HTML.
>

Config Files

package.json

Let's take a quick look at the **package.json** file (click to open). Everything in here is the same, except the scripts and dependencies sections. The scripts section should now looks as follows:

```
"scripts": {
  "build:dev": "rm -rf .parcel-cache/ dist/ && parcel build --
no-source-maps --public-url /dist/ --no-optimize
src/views/**/*.pug",
  "build:prod": "rm -rf .parcel-cache/ dist/ && parcel build --
no-source-maps --public-url /dist/ src/views/**/*.pug"
}
```

And the dependencies section:

```
"dependencies": {
  "@parcel/transformer-pug": "^2.8.2",
  "glob-parent": "^6.0.2",
  "highlight.js": "^11.7.0",
  "parcel": "^2.8.2"
}
```

> **.parcelrc**
>
> As with Handlebars and EJS, another necessary config file is **.parcelrc** (click to open). Inside this file you will see the following:

```
{
  "extends": "@parcel/config-default",
  "transformers": {
    "*.pug": ["@parcel/transformer-pug"]
  }
}
```

This looks very similar to what we saw before with Handlebars and EJS, except now we're telling Parcel to use a different transformer to compile our Pug files.

pug.config.js

Last but not least is **pug.config.js** (click to open):

```
module.exports = {
  pretty: true
}
```

This is simply an extra instruction for Pug not to minify any of the output HTML.
>

Pug - Syntax Overview Part 1

> Pug, formerly known as Jade, has a syntax that is markedly different from the previous two engines we looked at, and allows you to write templates in a more concise way. It uses indentation and whitespace to indicate nesting and structure, and it still includes a robust set of features such as variables, mixins (helper code similar to functions), loops, and conditionals.

In addition, it allows you to define a layout template to use for every page you create, in which you define blocks (placeholders) that you later "extend" with HTML for a given component or entire page. We achieved something close to this with the previous two engines, but Pug's layout functionality makes things a little easier.

Simple template:

```
div
  h1 Hello World
  p This is a basic template
```

Conditional template:

```
if content
  div= content
```

Loop template:

```
h1 title
ul
  each item in items
    li item.text
```

Mixin template:

```
mixin item(name)
  li name


h1 title
ul
  +item("item 1")
  +item("item 2")
  +item("item 3")
```

Using variables:

```
- let message = "Hello"
div message
```

Including a partial:

```
// someFile.pug

include path/to/partial
```

Using attributes:

```
input(type='text', name='username', value='username')
```

Layout template:

```
doctype html
html(lang="en")
  head
    meta(charset="UTF-8")
    meta(http-equiv="X-UA-Compatible", content="IE=edge")
    meta(name="viewport", content="width=device-width, initial-
        scale=1.0")
    block title
    block styles
  body
    include ../partials/header.pug
    main(class="container")
      block content
    include ../partials/footer.pug
    block scripts
```

Extending a layout:

```
extends ../layouts/[LAYOUT FILE NAME].pug


block title
  title Home | My Awesome Website


block styles
  link(rel="stylesheet", href="style.css")


block content
  h1 HOME PAGE


block scripts
  script(src="index.js")
```

To learn more, check out the documentation.
>

Pug - Syntax Overview Part 2

> One thing in Pug which we can do a bit easier than in Handlebars and EJS is creating a layout which we can then *extend*. Handlebars and EJS don't really have this functionality built in, so we had to work around that limitation in a different way. But with Pug, it's a little more straightforward.

Here's an example of creating a layout:

```
doctype html
html(lang="en")
  head
    meta(charset="UTF-8")
    meta(http-equiv="X-UA-Compatible", content="IE=edge")
    meta(name="viewport", content="width=device-width, initial-
        scale=1.0")
    block title
    block styles
  body
    include ../partials/header.pug
    main(class="container")
      block content
    include ../partials/footer.pug
    block scripts
```

Essentially, this establishes certain "blocks" which are kind of like placeholders for code we will write when we extend the layout. There are four blocks in this example (**title**, **styles**, **content**, and **scripts**). We can reference the block names in other Pug files.

Let's now take a look at an example of extending a layout, which adds in the code for the blocks defined in the layout:

```
extends ../layouts/[LAYOUT FILE NAME].pug


block title
  title Home | My Awesome Website


block styles
  link(rel="stylesheet", href="style.css")


block content
  h1 HOME PAGE


block scripts
  script(src="index.js")
```

To learn more, check out the documentation.
>

Pug - Create layout and partials

> Now, let's do the same thing for Pug that we did for Handlebars and EJS!

First, let's install all of our project dependencies for Pug. In the terminal, run the following command:

```
npm install
```

Next, in the terminal let's run the following command to create the files we'll need for our templates and partials:

```
touch src/partials/header.pug src/partials/footer.pug
```

> Open up **header.pug** by clicking , then add the following:

```
header(class="header container")
  a(href="/dist" class="brand") Awesome
    span Brand
  nav(class="main-nav")
    ul(class="main-nav__list")
      a(href="/dist/about" class="main-nav__item")
        li About
      a(href="/dist/contact" class="main-nav__item")
        li Contact
      a(href="#" class="main-nav__item cta")
        li Signup
```

Now, open up **footer.pug** by clicking and add the following:

```
footer(id="footer" class="footer container")
  div(class="footer__content") &copy; 2023 CompanyName. All
        rights reserved.
```

Next, inside the **src/layouts** directory, let's create a layout template called **main.pug** to hold the beginning part of any given HTML document (including our header) and the end part of any given HTML document (including our footer), along with some blocks:
>

```
touch src/layouts/main.pug
```

Now click to open the file, then add the following:

```
doctype html
html(lang="en")
  head
    meta(charset="UTF-8")
    meta(http-equiv="X-UA-Compatible", content="IE=edge")
    meta(name="viewport", content="width=device-width, initial-
        scale=1.0")
    block title
    block styles
  body
    include ../partials/header.pug
    main(class="main container")
      block content
    include ../partials/footer.pug
    block scripts
```

> **NOTE:** For any of the files we create from the terminal, if the link to open them doesn't work try resyncing the file tree (**Project –> Resync File Tree**), then try clicking the link again.

Pug - Create pages

Now it's time to create our pages inside the **src/views** directory. Inside this directory, first create a file called **index.pug** by running the following in the terminal:

```
touch src/views/index.pug
```

This will be our home page. Click to open the file.
>

Inside **index.pug** add the following:

```
extends ../layouts/main.pug


block title
  title Home


block styles
  link(rel="stylesheet" href="../styles/global.css")
  link(rel="stylesheet" href="../styles/index.css")


block content
  h1 HOME PAGE


block scripts
  script(src="../js/index.js")
```

Next, create two other page files inside **src/views** – one called **about.pug**, and one called **contact.pug**:

```
touch src/views/about.pug src/views/contact.pug
```

Open **about.pug** by clicking , and add the following:

```
extends ../layouts/main.pug


block title
  title Contact


block styles
  link(rel="stylesheet" href="../styles/global.css")
  link(rel="stylesheet" href="../styles/about.css")


block content
  h1 ABOUT PAGE


block scripts
  script(src="../js/about.js")
```

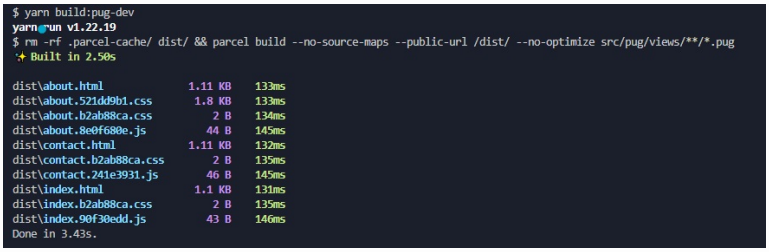
Next, open **contact.pug** by clicking , and add the following:

```
extends ../layouts/main.pug


block title
  title Contact


block styles
  link(rel="stylesheet" href="../styles/global.css")
  link(rel="stylesheet" href="../styles/contact.css")


block content
  h1 CONTACT PAGE


block scripts
  script(src="../js/contact.js")
```

Now, we can go to the terminal and run the command to compile our Pug files (as with Handlebars and EJS, we'll use the dev command):

```
npm run build:dev
```

If everything compiled successfully, you should see something like this in the terminal:



terminal successful compile

Now in the file tree, as with Handlebars and EJS, expand the newly generated **dist**/ directory and have a look at the results of running the build command!

Conclusion & Takeaways

> In this lesson, you learned about a couple more popular template engines (EJS and Pug), and how to make effective use of templates, layouts, and partials to generate multiple pages without having to repeat a lot of the same HTML.

Resources

Pug

- Pug.js

EJS

- EJS Documentation

Forms and Form Validation: Introduction

Goals

In this section you will:
- Be introduced to the form object
- Learn about client-side form validation and why it's important
- Explore two different types of validation (HTML based and JavaScript based)
>
Introduction
The web is full of forms: login forms, signup forms, subscribe-to-my-blog forms, etc. Being a frontend developer means you will deal with forms – whether in your own projects, projects assigned by your employer, or projects for a client if you become a freelancer.
>
And when it comes to forms, given the great amount of hackers out there who try to maliciously gain access to systems, steal information, or otherwise cause digital mayhem, it's extremely important to learn about form validation!
>
It's important not only as a first step to help protect against security breaches but also to ensure that the data your web app expects from a given form is the data it gets, so that your app doesn't break. You'll learn more about web security later in this course, but here we'll focus on getting a good foundation in form validation.
>
Business Context

Before diving in, it's important to note that in this section when we talk about form validation we will be dealing exclusively with client-side validation (recall the client-server model you learned about in the first week). Any web developer worth their salt, however, will tell you that relying solely on client-side validation is (to put it mildly) a mistake.
>
This is because client-side validation is not that difficult to circumvent. It's still important to implement, but it's always recommended to have server-side validation in place as well. Implementing server-side validation is part of the job of backend developers… or perhaps you one day, if you decide to branch out and become a fullstack developer!

Review of Form Elements - Part 1

> First things first, let's do a brief review of several common HTML form input elements (with their applicable events we can listen for in JavaScript), then we'll move on to cover the form object itself.

Text

The text input is probably the most basic of all form inputs, and is simply a single line input field which can have placeholder text if desired, and is usually preceded by the element. The text input can also have added attributes which are useful for validation, such as the required attribute, the minlength attribute, etc.
>
Events: input, change, focus, blur
>
Textarea
The textarea element, which is not a variation of the input element but has its own tag in HTML, is similar to the text input except it is designed for multiple lines of text.
>
Events: input, change, focus, blur
>
Checkbox
Checkboxes are a type of input element that render as boxes which are empty when in their unchecked state and contain a checkmark when in their checked state, which can be toggled by clicking the element. They can appear slightly different depending on the user's operating system.
>
If grouped together with other checkboxes, many can be selected at a time.
>
Events: change, focus, blur
>
Radio
Radio buttons are usually grouped together by setting the name attribute to the same value on all of them, so that they reference the same piece of information. When doing this, only one radio button can be selected at a time; if another radio button is selected, the previous one becomes unticked.
>
Events: change, focus, blur
>

Review of Form Elements - Part 2

> ### Select
>
> 'Select' provides a dropdown of choices. Like textarea, the select element is not an input element variant but its own HTML tag. Typically, select elements contain one or more 'option' elements, depending on how many choices you want to include for that field.
>
> **Events**: change, focus, blur
>
> ### Button
>
> This element can actually be utilized either by using the `<button>` tag, or with `<input type="button">`. It is a particular control designed to perform some explicit action when clicked, such as opening a modal, logging in, submitting a form, etc.

If using `<input type="button">`, the button's inner text (label) is set using the value attribute rather than by typing text in between opening and closing `<button>` tags.
>
Events: click, focus, blur

Here is a basic example using the elements noted above. Go ahead and add the following to the **index.html** file open in the top left panel:

```
<form action="#" method="POST">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" placeholder="Enter
        your name" required>
  <br /><br />

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" placeholder="Enter
        your email" required>
  <br /><br />

  <label for="password">Password:</label>
  <input type="password" id="password" name="password"
        placeholder="Enter your password" required>
  <br /><br />

  <label for="message">Message:</label>
  <textarea id="message" name="message" placeholder="Enter your
        message"></textarea>
  <br /><br />

  <label for="subscribe">Subscribe to our newsletter:</label>
  <input type="checkbox" id="subscribe" name="subscribe"
        value="1">
  <br /><br />

  <label for="country">Select your country:</label>
  <select id="country" name="country">
    <option value="usa">USA</option>
    <option value="canada">Canada</option>
    <option value="uk">UK</option>
    <option value="australia">Australia</option>
  </select>
  <br /><br />

  <button type="submit">Submit</button>
</form>
```

Now click the refresh button in the preview window!
>

The Form Object - Part 1

> The form object is simply a term that refers to the HTML `<form>` element. You can access the form object in the same way as any other HTML element – either by class, id, tag name, or other selector – but there are a few attributes which are specific to forms, some of which we'll look at here.
>
> ### action
>
> The action attribute is used to define the URL where the form data should be sent when it is submitted. Often this is a separate script file responsible for processing the form data, but this attribute can also be used to redirect a user to a specific page on your site after the form has been submitted. If the action attribute is not specified, the data will be sent to the same URL as the current page.
>
> ### method
>
> The HTML method attribute is used to specify the HTTP method that will be used to submit the form data (usually, to a server). The two most common methods are "GET" and "POST".
>
> The "GET" method sends the form data as part of the URL, and is typically used for forms intended to retrieve some kind of data from the server (e.g. - a search form). This data is visible in the URL and can be cached by the browser. Because of this, GET requests should not be used to submit sensitive data like passwords.
>
> The "POST" method sends the form data in the request body, and is typically used for forms that send data to the server (e.g. - a login, signup, or contact form). The data is not visible in the URL and is not cached by the browser, and thus is the method that should be used if sensitive data is being submitted.
>
> If the method attribute is not specified, the default method is "GET".
>
> ### target
>
> The HTML target attribute is used to specify where the form's response should be displayed after it is submitted. The target attribute can be used to open the response in a new window or in a specific frame or iframe.

- If the target attribute is set to "_blank", the response will be opened in a

new window or tab.

- If the target attribute is set to the name of a specific frame or iframe, the response will be opened in that frame.
- If the target attribute is not specified, the response will be displayed in the same window or frame as the form.
  >
  It's important to note that using target='_blank' without any security measures in place can present a serious security vulnerability, because it can be exploited by a malicious website to open a link to your site in a new tab, potentially stealing the user's session.
  >

The Form Object - Part 2

enctype

The enctype attribute is used to specify the format of the data sent to the server when the form is submitted. The enctype attribute can have one of the following values:
>
- “application/x-www-form-urlencoded”: This is the default value. With this enctype, data is sent in the same format as a query string. This is likely the preferable choice in a majority of scenarios.
>
- “multipart/form-data”: This value is used when the form includes file upload fields, and the data is sent in a multipart format. This format allows the form data and the files to be sent in the same request.
>
- “text/plain”: This value is rarely used, and the data is sent in plain text format. It should not be used for forms with sensitive information.
>
Note that the enctype attribute is only used when the method attribute is set to “POST”.
>
You probably won’t need to use this attribute much, unless your form includes one or more file uploads; in that case, the enctype attribute has to be set to “multipart/form-data” or the file upload(s) will not work.
>
autocomplete
The HTML autocomplete attribute is used to specify whether or not a form or form elements should have autocomplete functionality enabled, and can be set to “on” or “off.” Autocomplete functionality allows the browser to automatically fill in previously entered values for a form or form element, based on the user’s browsing history.
>
Note that this attribute can be used on the form as a whole, or just on specific elements (such as text or select fields). Autocomplete functionality can pose a security risk as it can reveal sensitive information to an attacker. Therefore, it’s recommended to set autocomplete=“off” on forms that collect sensitive information like login credentials or credit card information.

Here is a basic example using the attributes noted above. Go ahead and add the following to the **index.html** file open in the top left panel:

```
<form method="post" action="javascript:void(0);" target="_self"
        autocomplete="on" enctype="multipart/form-data">
  <label for="name">Name:</label>
  <input type="text" id="name_field" name="name"
        autocomplete="name" required>
  <br /><br />

  <label for="email">Email:</label>
  <input type="email" id="email_field" name="email"
        autocomplete="email" required>
  <br /><br />

  <label for="message">Message:</label>
  <textarea id="message" name="message"></textarea>
  <br /><br />

  <label for="attachment">Attachment:</label>
  <input type="file" id="attachment" name="attachment">
  <br /><br />

  <input type="submit" />
</form>
```

Now click the refresh button in the preview window!
>

Client-side form validation

> ‘Client side form validation’ is the process of checking user-submitted data, using JavaScript, before the data is sent to the server, and helps to ensure that the data is in the correct format, meets certain criteria, and is free of errors. Client side validation is a good way to improve user experience by providing immediate feedback when a user submits an incorrect or incomplete form.

There are typically two layers to client side validation: HTML-based validation, and JavaScript-based validation.
>
HTML-based validation
This layer uses HTML attributes to specify validation rules for form fields:
>
- **required** attribute: This attribute can be added to an input field to make it mandatory. A form field with this attribute cannot be submitted unless it has a value.
>
- **pattern** attribute: This attribute can be added to an input field to specify a regular expression that the field’s value must match.
>
- **min** and **max** attributes: These attributes can be added to number and date input fields to specify a minimum and maximum value that the field’s value must be between.
>
- **minlength** and **maxlength** attributes: These attributes can be added to input fields to specify a minimum and maximum number of characters that the field’s value must be between.
>
- **type** attribute: The type attribute can be set to different values like email, number, tel, etc. which enforces specific validation rules
>
JavaScript-based validation
JavaScript-based form validation can be implemented in several ways, such as:
>
- Using event listeners to trigger a validation function when the form is submitted. The validation function returns an error message if any errors are found, and can also prevent the form from being submitted until the errors are corrected.
>
- Using JavaScript libraries and frameworks: there are many libraries and frameworks that provide pre-built validation functions; two of note are Validator.js and JustValidate. These libraries (and others like them) make it

easy to implement client-side form validation and provide a wide range of validation options.
>
In addition, you can make use of other built-in features like the Constraint Validation API, which allows you to check the validity of form fields using various methods and properties:
- **checkValidity()**: This method returns a Boolean indicating whether the input is valid.
- **validity**: This property is an object containing several properties like valueMissing, typeMismatch, tooLong, tooShort, patternMismatch etc. that indicate the type of validation error, if any.
- **setCustomValidity()**: This method expects a string value as an argument and can be used to set a custom error message that will be displayed when the field is invalid.
- **validationMessage**: This property returns the error message that will be displayed when the field is invalid.
The API also provides some events that can be used to handle form validation, such as:
- **invalid**: This event is fired when an input field is invalid.
- **valid**: This event is fired when an input field is valid.
>

Exercise: Custom validation - Part 1

> We're going to look at a practical example in order to practice event listeners: a signup form. We'll also be adding some custom validation to our form using what we've learned thus far about different validation methods.
>
> Open **main.js**. First, we need to get access to all the DOM elements that we'll need:

```
// Get all the necessary DOM elements
const form = document.getElementById('exampleForm')
const submitButton = document.querySelector('.submit')
const successMessage = document.getElementById('form-submitted-
        msg')
```

Next we need to store all the form elements in an iterable. An easy way to do that is create a variable and set it to an array that spreads the elements property of the form object using the ES6 spread operator:

```
// Store all form elements in an array by spreading the elements
        property of the form
const formElements = [ ...form.elements ]
```

Next, we'll create a function to check if all the form inputs are valid:

```
// Create function to check if all form elements are valid
const allInputsValid = () => {
  const valid = formElements.every((element) => {
    if (element.nodeName === 'SELECT') {
      return element.value !== 'Please select an option'
    } else {
      return element.checkValidity()
    }
  })


  return valid
}
```

Exercise: Custom validation - Part 2

Now we need to create a handler function for the change event on any given input, which also enables the submit button if all the inputs are valid:

```javascript
// Define a function to handle changes to any form element
const handleChange = () => {
// Use the forEach() function to execute the provided function once for each element in the formElements array
formElements.forEach((element) => {
// If the element is invalid and is not a button, a select dropdown, a checkbox, or a radio button, style it with a red border and red text
if (!element.checkValidity()
&& element.nodeName !== 'BUTTON'
&& element.nodeName !== 'SELECT'
&& element.type !== 'checkbox'
&& element.type !== 'radio'
) {
element.style.borderColor = 'red'
element.nextElementSibling.style.color = 'red'
element.nextElementSibling.style.display = 'block'
element.previousElementSibling.style.color = 'red'
}
```

```
// If the element is valid, reset its style to the original colors
// The conditions are the same as above for excluding certain elements
if (element.checkValidity()
      && element.nodeName !== 'BUTTON'
      && element.nodeName !== 'SELECT'
      && element.type !== 'checkbox'
      && element.type !== 'radio'
) {
  element.style.borderColor = '#CED4DA'
  element.nextElementSibling.style.color = '#CED4DA'
  element.nextElementSibling.style.display = 'none'
  element.previousElementSibling.style.color = '#212529'
}

// If the element is a checkbox or a radio button and is invalid, style it with a red border and red text
if (!element.checkValidity()
```

```
      && (element.type === 'checkbox'
          || element.type === 'radio')
) {
  element.style.borderColor = 'red'
  element.nextElementSibling.style.color = 'red'
}

// If the checkbox or radio button is valid, reset its style to
the original colors
if (element.checkValidity()
      && (element.type === 'checkbox'
          || element.type === 'radio')
) {
  element.style.borderColor = '#CED4DA'
  element.nextElementSibling.style.color = '#212529'
}

// If the element is a select dropdown and the default option is
selected, style it with a red border and red text
if (element.nodeName === 'SELECT'
      && element.value === 'Please select an option'
) {
  element.style.borderColor = 'red'
  element.nextElementSibling.style.color = 'red'
  element.nextElementSibling.style.display = 'block'
  element.previousElementSibling.style.color = 'red'
}

// If an option other than the default is selected in the
dropdown, reset its style to the original colors
if (element.nodeName === 'SELECT'
      && element.value !== 'Please select an option'
) {
  element.style.borderColor = '#CED4DA'
  element.nextElementSibling.style.color = '#CED4DA'
  element.nextElementSibling.style.display = 'none'
  element.previousElementSibling.style.color = '#212529'
}
```

})

// If all form elements are valid, enable the submit button; otherwise, disable it
if (allInputsValid()) {
submitButton.removeAttribute(‘disabled’, ’‘)
} else {

```
        submitButton.setAttribute('disabled', '')
        }
        }
```

Exercise: Custom validation - Part 3

Now let's create a handler function for the submit event on the form. If any of the inputs is invalid, the handler should set the border color and label font color to 'red', as well as enabling the error message beneath whichever input is invalid. In addition, if all inputs are valid, it should make the success message appear, reset the form and disable the submit button, then after a period of time should make the success message disappear again:

```javascript
// Define a function to handle form submission
const handleSubmit = (e) => {
// Prevent the default form submission behavior
e.preventDefault()

// If all form elements are valid after the form submission, display a success message, reset the form, and disable the submit button
if (allInputsValid()) {
successMessage.style.display = 'block'
form.reset()
submitButton.setAttribute('disabled', '')

// Hide the success message after 3 seconds
setTimeout(() => {
  successMessage.style.display = 'none'
}, 3000)

}
}
```

Exercise: Custom validation - Part 4

Now, to finish up, let's attach our event listeners:

```
// Add event listener to each form element
formElements.forEach((element) => {
  element.addEventListener('change', handleChange)
})

// Add submit listener to the form
form.addEventListener('submit', (e) => handleSubmit(e))
```

Go ahead and hit the refresh button in the preview window!

Conclusion & Takeaways

> In this section you learned about the form object and its various properties. Additionally, we discussed client-side form validation, why it's important, and how to implement it using HTML and JavaScript. We also covered why it's important not to rely only on client-side validation in real-world websites, due to the fact that someone who knows what they're doing can easily get around it if they want to.

Basic Data Structures: Introduction

Goals

Throughout this section you will:
- Learn about a few of the data structures in JavaScript (you've used two of them already in the past couple JavaScript modules of this course)
- Understand why measuring efficiency of data structures and their methods using Big O Notation is important as a developer

Introduction

Data structures are an important concept in programming and they are fundamental to the way we store and access information. JavaScript has a number of built-in data structures that can be used to store, manipulate, and retrieve data. These data structures include arrays, objects, Maps, Sets, linked lists, trees, and graphs. In this section we will look at the basics of the first four in this list and how to work with their built-in properties and methods. We'll finish out this section talking about Big O Notation.

Business Context

Learning a variety of data structures and how to effectively use them and in a discerning way is very important both for your personal growth as a developer, but also, ultimately, for landing a job! Data structures are one of the foundational building blocks of programming (in any language), and thus in interviews you will more often than not be expected to have some knowledge of them which goes beyond just their practical use cases and individual properties and methods – in particular, their efficiency (which we'll delve into later today)

Sets

> A Set is a collection of unique values. Sets are similar to arrays, but they can be used to store multiple values while ensuring there aren't any duplicates. Add the following to the **index.js** file open in the top left panel:

```
// Create a new Set
let mySet = new Set();


// Add values to the Set
mySet.add("Toyota");
mySet.add("Chevrolet");
mySet.add("Dodge");


// Try to add a duplicate value
mySet.add("Dodge"); // This will not be added


console.log(mySet.size); // 3
console.log(mySet); // Set { "Toyota", "Chevrolet", "Dodge" }


// Check if a value is in the Set
console.log(mySet.has("Chevrolet")); // true
console.log(mySet.has("Tesla")); // false


// Get all the entries of the Set as an iterable
console.log(mySet.entries()); // [ "Toyota", "Chevrolet",
        "Dodge" ]


// Remove a value from the Set
mySet.delete("Chevrolet");
console.log(mySet); // Set { "Toyota", "Dodge" }


// Iterate over the values in the Set
mySet.forEach((value) => {
  console.log(value);
});
```

```
/* Output:
 Toyota
 Dodge
*/


// Clear the Set
mySet.clear();
console.log(mySet);


// Output:
//
```

In this example, we create a new Set called `mySet` and add three values to it: 'Toyota', 'Chevrolet', and 'Dodge'. We then try to add a duplicate value ('Dodge') to the Set, but it is not added because **Sets only store unique values**. We can check if a value is in the Set using the `has()` method, which returns true if the value is present and false otherwise. We can also remove a value from the Set using the `delete()` method, and iterate over the values in the Set using `forEach()`. Finally, we clear the Set using the `clear()` method."

In order to see the output for yourself, from the terminal run the following command:

```
node index.js
```

Maps

> A Map is a collection of key-value pairs. It can be used to store multiple values, but unlike sets, it allows duplicate values. Each value is associated with a unique key, allowing for easy and efficient lookups. Add the following to the **index.js** file open in the top left panel:

```
// Create a new Map
let myMap = new Map();


// Add key-value pairs to the Map
myMap.set("name", "Matthew");
myMap.set("age", 35);
myMap.set("favoriteMovie", "Lord of the Rings");


console.log(myMap.size); // 3
console.log(myMap); // Map { "name" => "Matthew", "age" => 35,
        "favoriteMovie" => "Lord of the Rings" }


// Get the value of a key
console.log(myMap.get("name")); // "Matthew"


// Check if a key exists in the Map
console.log(myMap.has("age")); // true
console.log(myMap.has("address")); // false


// Get all the keys in the Map as an iterable
console.log(myMap.keys()); // ["name", "age", "favoriteMovie"]


// Get all the values in the Map as an iterable
console.log(myMap.values()); // ["Matthew", 35, "Lord of the
        Rings"]


// Get all the entries in the Map as an iterable
console.log(myMap.entries()); // [["name", "Matthew"], ["age",
        35], ["favoriteMovie", "Lord of the Rings"]]
```

```
// Remove a key-value pair from the Map
myMap.delete("age");
console.log(myMap); // Map { "name" => "Matthew",
        "favoriteMovie" => "Lord of the Rings" }


// Iterate over the key-value pairs in the Map
myMap.forEach((value, key) => {
  console.log(key + ": " + value);
});


/* Output:
 name: Matthew
 favoriteMovie: Lord of the Rings
*/


// Clear the Map
myMap.clear();
console.log(myMap);


// Output:
//
```

> In this example, we create a new Map called `myMap` and add three key-value pairs to it: `'name' => 'Matthew'`, `'age' => 35`, and `'favoriteMovie' => 'Lord of the Rings'`. We can retrieve the value associated with a specific key using the `get()` method. We can also check if a key exists in the Map using the `has()` method. Additionally, we can get all the keys or values in the Map using the `keys()` and `values()` methods respectively, or get all the entries (key-value pairs) in the Map using the `entries()` method. We can remove a key-value pair from the Map using the `delete()` method, and iterate over the key-value pairs in the Map using `forEach()`. Finally, we can clear the Map using the `clear()` method.

In order to see the output, from the terminal run the following command as we did before:

```
node index.js
```

It's important to note that in a Map, keys can be any value (including objects and functions!), and the insertion order is preserved, unlike objects which we'll talk about in a moment.

>

Arrays

> We learned about arrays and objects a couple weeks ago, but let's go through a quick refresher. We'll start with arrays.

As we learned previously, an array is a collection of ordered values. It can be used to store multiple values, any of which can be accessed by their index. Add the following to the **index.js** file open in the top left panel:

```
// Create a new array
let myArray = ["Toyota", "Chevrolet", "Dodge"];


console.log(myArray.length); // 3
console.log(myArray); // ["Toyota", "Chevrolet", "Dodge"]


// Access an element in the array by its index
console.log(myArray[1]); // "Chevrolet"


// Add an element to the end of the array
myArray.push("Tesla");
console.log(myArray); // ["Toyota", "Chevrolet", "Dodge",
        "Tesla"]


// Remove an element from the end of the array
let lastElementFromArray = myArray.pop();
console.log(lastElementFromArray); // "Tesla"
console.log(myArray); // ["Toyota", "Chevrolet", "Dodge"]


// Add an element to the beginning of the array
myArray.unshift("BMW");
console.log(myArray); // ["BMW", "Toyota", "Chevrolet", "Dodge"]


// Remove an element from the beginning of the array
let firstElementFromArray = myArray.shift();
console.log(firstElementFromArray); // "BMW"
console.log(myArray); // ["Toyota", "Chevrolet", "Dodge"]


// Find the index of an element in the array
```

```
console.log(myArray.indexOf("Dodge")); // 2


// Slice the array to create a new sub-array
let subArray = myArray.slice(0, 2);
console.log(subArray); // ["Toyota", "Chevrolet"]


// Iterate over the elements in the array
myArray.forEach((value) => {
  console.log(value);
});


/* Output:
Toyota
 Chevrolet
 Dodge
*/
```

Let's give it a try! Once again, the command to run is:

```
node index.js
```

Objects

An object is a collection of key-value pairs and can be used to store multiple values, similar to a Map (except the insertion order is not preserved). Add the following to the **index.js** file open in the top left panel:

```
// Create a new object
let myObject = {
  name: "Matthew",
  age: 35,
  favoriteMovie: "Lord of the Rings",
  address: {
    street: "Main St",
    city: "New York",
    state: "NY",
  }
};

console.log(myObject);
/* Output:
 {
   name: "Matthew",
   age: 35,
   favoriteMovie: "Lord of the Rings",
   address: {
     street: "Main St",
     city: "New York",
     state: "NY",
   }
 }
*/


// Access a property of the object
console.log(myObject.name); // "Matthew"
console.log(myObject["age"]); // 35
console.log(myObject.address.city); // "New York"


// Add a new property to the object
myObject.phoneNumber = "222-222-2222";
console.log(myObject);

/* Output:
```

```
 {
   name: "Matthew",
   age: 35,
   favoriteMovie: "Lord of the Rings",
   address: {
     street: "Main St",
     city: "New York",
     state: "NY",
   }
   phoneNumber: "222-222-2222"
 }
*/
```

Try out the code! From the terminal, run the following command:

```
node index.js
```

Filtering data structures: Arrays

> Let's look at some examples of how we can filter the different data structures we've looked at so far. We'll start with arrays.

While there are couple different ways to filter an array in JavaScript, in this lesson we will look at the most straightforward way: using the *filter()* method!

Add the following to the **filtering.js** file open in the top left panel:

```
// Filtering an array
let numbersArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let evenNumbers = numbersArray.filter((number) => {
  return number % 2 === 0;
});
console.log(evenNumbers); // [2, 4, 6, 8, 10]
```

From the terminal, run the following command:

```
node filtering.js
```

Filtering data structures: Objects

> Filtering an object is a little more complicated than filtering an array. One possible way to filter an object is to convert it to an array of its key-value pairs using the *Object.entries()* method, then filter the array returned by that method using *filter()*, and then converting the filtered array back to an object using the *reduce()* method.

Add the following to the **filtering.js** file open in the top left panel:

```
// Filter an object
let myObject = {
  one: 1,
  two: 2,
  three: 3,
  four: 4
};


let filteredObject = Object.entries(myObject)
    .filter(([key, value]) => value > 2)
    .reduce((object, [key, value]) => {
        object[key] = value;
        return object;
    }, {});


console.log(filteredObject); // { three: 3, four: 4 }
```

The *reduce()* method takes in a couple different parameters: a callback function to execute which in turn takes in two parameters (one representing the previous value, and the other representing the current value to be operated on) and a second parameter defining the initial value – in this case, an empty object.
>
Then, *reduce()* iterates over the array returned by the *filter()* method and sets each of the new object's keys to the value given in the current iteration and returns the mutated object each time until it reaches the end of the array. What we end up with is a new object containing only those key-value pairs that matched our filtering condition.
>

From the terminal, run the following command:

```
node filtering.js
```

Filtering data structures: Sets

> Filtering Sets and Maps is a bit more complex than filtering an array or an object. Nothing too crazy, but it does involve an extra step. We'll start with Sets.

The extra step we need to do is to convert the set to an array, so that we can use the filter method.

Add the following to the **filtering.js** file open in the top left panel:

```
// Filter a Set


// Create a new Set
let mySet = new Set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])


// Create an array from mySet
let arrayFromSet = Array.from(mySet)


// Create a new variable to store the result of running the
        filter method
let filteredArray = arrayFromSet.filter((number) => {
  return number % 2 === 0
})


// Create a new Set from filteredArray
let newFilteredSet = new Set(filteredArray)


console.log(newFilteredSet) // Set(5) { 2, 4, 6, 8, 10 }
```

From the terminal, run the following command:

```
node filtering.js
```

Filtering data structures: Maps

> Filtering a map using this method looks very similar. Go ahead and add the following to the **filtering.js** file open in the top left panel:

```
// Filter a Map


// Create a new Map
let myMap = new Map([
  ["name", "Matthew"],
  ["age", 35],
  ["favoriteMovie", "Lord of the Rings"],
])


// Create an array from myMap
let arrayFromMap = Array.from(myMap)


// Create a new variable to store the result of running the
        filter method
let filteredMap = arrayFromMap.filter(([key, value]) => {
  return key !== "age"
})


// Create a new Map from filteredMap
let newFilteredMap = new Map(filteredMap)


console.log(newFilteredMap) // Map(2) { 'name' => 'Matthew',
        'favoriteMovie' => 'Lord of the Rings' }
```

From the terminal, run the following command:

```
node filtering.js
```

A Primer on Big O Notation - Part 1

> Now that we've spent some time talking about a few data structures in JavaScript (there are many more), it's fitting to address the question of *"which one's the best?"*

The answer is... it depends!
>
It depends on the situation and what operation is being performed on the given data structure.
>
While it may not be possible to definitively answer the question of "Is one data structure better than another," fortunately there *is* an unbiased way to answer which operation on a given data structure is most *efficient*. This is called **Big O Notation**.
>
Big O Notation is a concept in computer science which is language-agnostic, so it has become the standard to use when trying to measure the efficiency of a given piece of code (i.e. - algorithm, which is just a fancy word that means a defined set of steps to accomplish a given task). Big O measures code in terms of complexity in two different categories: time complexity and space complexity. Time complexity refers to how quickly and efficiently our code can run based on the input it is given, while space complexity refers to how many new spaces in memory are taken up (by variables created during the execution of the algorithm, etc.)
>
A lot of the time you will find yourself focusing more on time complexity, but it's also crucial to be aware of space complexity as well!
>
Big O Notation allows developers to compare the performance of different algorithms in terms of their scalability. An algorithm with a lower Big O complexity is generally more efficient than one with a higher Big O complexity. By using Big O Notation, developers can identify potential performance bottlenecks and choose the most appropriate algorithm for a given task.
>
And Big O, overall, is a very important concept to become familiar with even as a frontend JavaScript developer, because it comes up often in developer interviews – especially at the big tech companies like Google or Amazon.
>
The "O" in Big O Notation stands for "order of," and the notation is typically followed by a function that describes the algorithm's time or space

complexity. For example, O(1) represents **constant** complexity, O(n) represents **linear** complexity, and so on.
>

Here's a basic overview of the most common Big O complexities:

Name	Notation	Summary	Application in Code
Constant	O(1)	No matter how big your task is, it will take the same amount of time. A real-world example would be flipping a light switch.	Looking up a single value in an array by its index.
Logarithmic	O(log n)	As the size of your task grows, the time it takes grows slowly. A real-world example would be looking up a word in a dictionary.	Binary search in a sorted array.
Linear	O(n)	Time grows directly in relation to the task size. A real-world example would be reading a book page by page.	Looping over an array to find a value
Quadratic	O(n^2)	Time grows quickly in relation to the task size. A real-world example would be trying on every pair of socks in your drawer.	Nested loops; bubble sort
Factorial	O(n!)	Time grows extremely fast, even for small tasks. A real-world example of this would be trying every combination of a lock code.	Generating all possible combinations of a given array's elements.

A Primer on Big O Notation - Part 2

While the entirety of Big O Notation would be too much to cover in depth in this lesson, let's at least see a few examples of Big O in relation to certain operations in JavaScript.
>
Arrays
Here's a rundown of various array methods with their corresponding (time) complexities:
>
ARRAY OPERATIONS	BIG O COMPLEXITY
———	———-
Accessing an item - array[i]	O(1)
.push() - add item to end of array	O(1)
.pop() - remove item from end of array	O(1)
.shift() - remove first item from array	O(n)
.unshift() - add item to beginning of array
.concat() - merge 2 or more arrays into 1	O(n)
.slice() - returns a copy of part of an array	O(n)
.splice() - remove, replace, or add items to an array at a specified index	O(n)
.forEach() - performs an operation one time on each item in an array	O(n)
.map() - performs an operation on each item in an array then returns a new array with the results	O(n)
.filter() - returns a copy of part of an array that matches a condition	O(n)
.reduce() - runs a callback function on every item in array, passing in the return value of previous iteration until the result is a single value	O(n)
>
Objects
Now here's a look at some object methods:
>
OBJECT OPERATIONS	BIG O COMPLEXITY
———	———-
Accessing an item - object[i] = value	O(1)
.hasOwnProperty() - returns true or false, depending on whether the object has a given property	O(1)
.assign() - copy properties of one or more objects to a new object	O(n)
.keys() - returns array of object keys	O(n)
.values() - returns array of object values	O(n)
.entries() - returns array of object's key-value pairs	O(n)
>
Sets

SET OPERATIONS	BIG O COMPLEXITY
———	———-
new Set([i]) - create set from another iterable	O(n)
.add() - add item to Set	O(1)
.delete() - delete item from Set	O(1)
.has() - returns true or false depending on whether the Set has a specified value	O(1)
.keys() - returns an iterable of all the keys in the Set	O(n)
.values() - returns an iterable of all the values in the Set	O(n)
.entries() - returns iterable containing 2-value array for every element in the Set ([key, value])	O(n)
.forEach() - performs an operation one time on each item in the Set	O(n)
>
Maps
MAP OPERATIONS	BIG O COMPLEXITY
———	———-
new Map([i]) - create map from another iterable	O(n)
.set() - add item to Map	O(1)
.get() - get value from Map at the specified key	O(1)
.delete() - returns true if item removed from Map	O(1)
.has() - returns true or false depending on whether the Set has a specified value	O(n)
.keys() - returns an iterable of all the keys in the Map (preserving insertion order)	O(n)
.values() - returns an iterable of all the values in the Map (preserving insertion order)	O(n)
.entries() - returns iterable containing 2-value array for every element in the Map ([key, value]) (preserving insertion order)	O(n)
.forEach() - performs an operation one time on each item in the Map	O(n)
>

Conclusion & Takeaways

> In this section, we went over a few basic JavaScript data structures (arrays, objects, maps, and sets), as well as examples of many of their methods. You learned the very basics of Big O Notation and the time complexity measurement of many of the methods of the data structures we covered, as well as the various ways to filter them.
>
> For additional material covering Big O Notation, data structures, and algorithms, be sure to check out the following resources for further study:
>
> Big O Cheat Sheet – Time Complexity Chart
>
> Big O Notation Explained
>
> Learn Big O Notation In 12 Minutes
>
> JavaScript Algorithms Crash Course - Learn Algorithms & “Big O” from the Ground Up!
>
> Big O Notation - Full Course

Filtering with forEach(): Introduction

> ### Goals
>
> Throughout this section you will:
> - Learn about how to use the forEach() method to basic data structures

> ### Introduction
>
> In this Asynchronous Elective Learning lesson, we will learn an alternative way of filtering each of the data structures we looked at in the main lesson (arrays, objects, Sets, and Maps): using the **forEach()** method!

Arrays and Objects

> Now, let's look at filtering all four data structures again, this time using the *forEach()* method.
>
> ### Array

Here's an example of filtering an array using **forEach()**:

```
// Filtering an array with forEach()

let numbersArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


let evenNumbers = []


numbersArray.forEach((number) => {
  if (number % 2 === 0) {
    evenNumbers.push(number)
  }
})


console.log(evenNumbers) // [2, 4, 6, 8, 10]
```

Object

Here's an example of filtering an object using **forEach()**:

```
// Filter an object with forEach()


let myObject = {
  one: 1,
  two: 2,
  three: 3,
  four: 4
}


let filteredObject = {}


Object.keys(myObject)
  .forEach(key => {
    if (myObject[key] > 2) {
      filteredObject[key] = myObject[key]
    }
  })


console.log(filteredObject) // { three: 3, four: 4 }
```

To see the output, run the following command in the terminal:

```
node index.js
```

Sets and Maps

Set

Here's an example of filtering an object using **forEach()**:

```
// Filter a Set with forEach()


// Create a Set
let mySet = new Set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])


// Create a new Set to store the filtered results
let filteredSet = new Set()


// Filter the Set with forEach()
mySet.forEach((value) => {
  if (value > 5) {
    filteredSet.add(value)
  }
})


console.log(filteredSet) // Set(5) { 6, 7, 8, 9, 10 }
```

Map

```
// Filter a Map with forEach()


// Create a new Map
let myMap = new Map([
  ["name", "Matthew"],
  ["age", 35],
  ["favoriteMovie", "Lord of the Rings"],
])


let filteredMap = new Map();


myMap.forEach((value, key) => {
  if (key !== "age") {
    filteredMap.set(key, value);
  }
});


console.log(filteredMap); // Map(2) { 'name' => 'Matthew',
        'favoriteMovie' => 'Lord of the Rings' }
```

To see the output, run the following command in the terminal:

```
node index.js
```

Conclusion & Takeaways

> In this AEL lesson, we went over some simple examples demonstrating how to use the **forEach()** method as an alternative means of filtering arrays, objects, Sets and Maps in JavaScript.

Gamification: Introduction

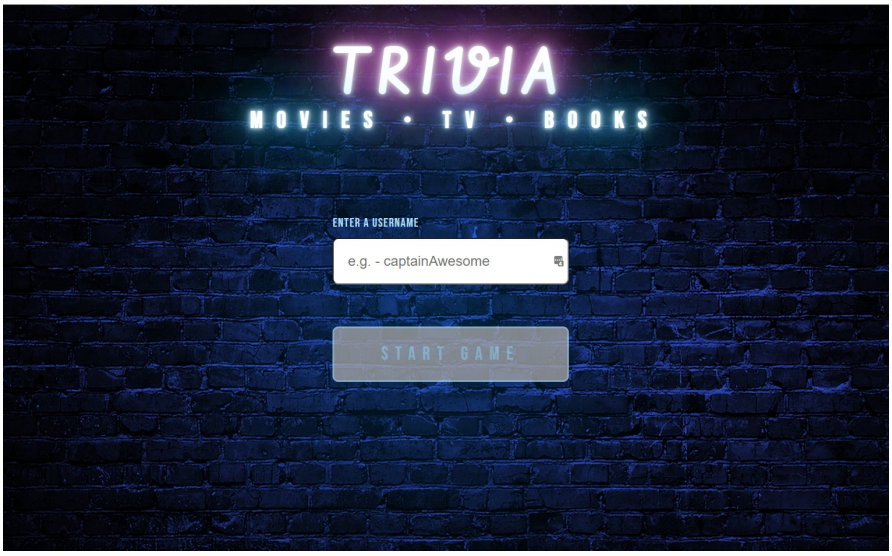
Goals

By the end of this section you will:
- Have begun implementing the logic for a simple trivia game
- Have practiced using Sets and Maps in a real application
>
Introduction
Did you know JavaScript can also be used to create games? There are a number of different libraries and frameworks in JavaScript which are specifically created to help build games. Games, at their core, are based on programming logic just like any application or website. A different use case, but many of the same concepts.
>
Over the next two lessons, we're going to create our own game! Nothing too fancy, and we won't be using any frameworks or libraries, but we will use what we've learned thus far about arrays, Sets, and Maps, to build a browser-based trivia game. While we won't have a backend to persist our game data (refreshing the browser will cause everything to revert back to its initial state), we will nonetheless mimic some of the same logic and behaviors. Without further ado, let's get to it!

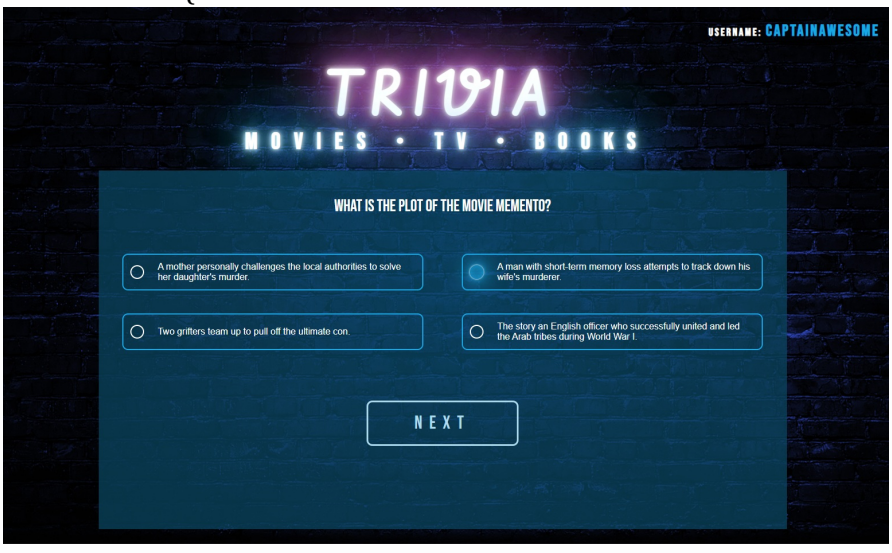
Build a Trivia Game part 1

> We're going to be making a trivia game with questions about movies, TV, and books. Here's some images of the finished product:

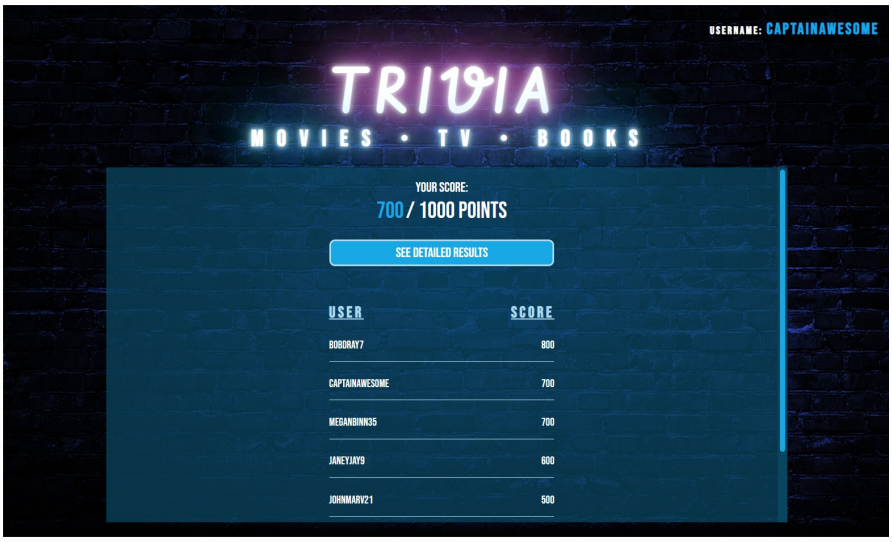
Start Screen:



Individual Question Screen:



Game end screen:



Results modal:

Take a look at the following starter code. The HTML and CSS have already been written, so we will just be focusing on writing the JavaScript to make the game work.

>

Before we jump in, let's take a look at a couple of the files we have here. You will notice two JSON files – one called **questions.json** and one called **users.json**. These two files have been preloaded with data so that our game has a good set of questions to select from (which it will do randomly), as well as some dummy user data to mimic user score ranking.

>

In addition, if you have a look at **index.html** you will notice we are

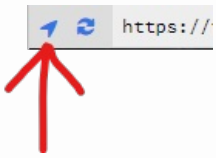
including the CDN links for two packages we've seen before: DOMPurify, and Validator.js. While we only have one field that receives user input in this project, we'll still utilize these packages to help implement some basic sanitization and validation.
>
Now, to get started, the first thing we're going to do is import the data from our JSON files so we will have access to it as we build the game logic. In **main.js**, add the following:
>

```
import questions from './questions.json' assert { type: 'json' }
import users from './users.json' assert { type: 'json' }
```

First, notice we are using the ES6 import syntax; we can do that because we set the **type** attribute on our script tag in **index.html** to 'module'. Also notice that at the end of our import statements, we included what is called an import assertion *(assert { type: 'json'})*, something which is relatively new and seems now to be required to import JSON in an ES6 module.
>
To verify that the imports worked, let's try console logging **users** and **questions**:
>

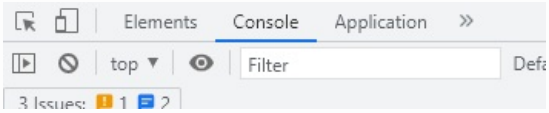
```
console.log(users)
console.log(questions)
```

Now, in the preview window on the bottom left, click the button to open the preview in a new browser tab:



new tab preview

Next, open the browser DevTools and click on the Console panel. (Note: this may look different depending on the browser you are using.)



console panel

If we've done everything correctly so far, you should see something like the following in the console:


Object
main.js:5
u1: {username: 'bobdray7', score: 800}
u2: {username: 'meganbinn35', score: 700}
u3: {username: 'janeyjay9', score: 600}
u4: {username: 'johnmarv21', score: 500}
[[Prototype]]: Object
main.js:6
Object
q1: {id: 'q1', question: "What was the name of the talking car ir
q2: {id: 'q2', question: 'Tilda Swinton plays the role of which
q3: {id: 'q3', question: "Who won the 2014 Academy Award for Best
q4: {id: 'q4', question: "Name the movie that matches the followi
q5: {id: 'q5', question: 'What are The Simpsons Halloween special
q6: {id: 'q6', question: "Which wizard was portrayed by Christopl
q7: {id: 'q7', question: "Name the movie that matches the followi
q8: {id: 'q8', question: 'Which film contains the character Oda
q9: {id: 'q9', question: "Name the movie that matches the followi
q10: {id: 'q10', question: 'Which actor plays the role of Michell
q11: {id: 'q11', question: 'Which actor plays the role of Bucky B
q12: {id: 'q12', question: 'In the Marvel Cinematic Unvierse, whi
q13: {id: 'q13', question: "What was the name of Dustin Hoffman's
q14: {id: 'q14', question: 'What is the plot of the movie Memento
q15: {id: 'q15', question: "Who won the 1998 Academy Award for Be
q16: {id: 'q16', question: 'Which film contains the character Jul
q17: {id: 'q17', question: "Who won the 2000 Academy Award for Be
q18: {id: 'q18', question: "Who played the role of James Bond in
q19: {id: 'q19', question: "Which director directed 'Inglourious
q20: {id: 'q20', question: "Which of these quotes is from the fil
q21: {id: 'q21', question: "Which director directed '2001: A Spac
q22: {id: 'q22', question: "From which London railway station doe
q23: {id: 'q23', question: "Who was the Defense Against the Dark
q24: {id: 'q24', question: 'In which year was Star Wars: Episode
q25: {id: 'q25', question: "Who is associated with the address '2

console

Build a Trivia Game part 2

Great! We have our fake users and our questions. Go ahead and delete the **console.log** statements. Next, we need to get access to all the DOM nodes we will need:

```
// Get all DOM elements
const usernameInput = document.getElementById('username')
const validationMsg = document.getElementById('validation-msg')
const startBtn = document.getElementById('start-btn')
const nextBtns = document.querySelectorAll('.next-question')
const playAgainBtn = document.getElementById('play-again')
const startSection = document.getElementById('start')
const currentUserDisplay = document.getElementById('user-
  display')
const questionGroups = document.querySelectorAll('.question')
const endSection = document.getElementById('game-end')
const finalScoreSpan =
  document.querySelector('span[id="score"]')
const answerButtons = document.querySelectorAll('.answer')
const modal = document.getElementById('modal');
const openModal = document.getElementById('show-details');
const closeModal = document.getElementById('modal-close');
const questionsInModal = document.querySelectorAll('.game-
  question')
const userStatsItems = document.querySelectorAll('.user-stat')
```

Build a Trivia Game part 3

Now let's define some variables that we'll need later:

```
// Create array from all the answer buttons
const answers = [...answerButtons]


// Create array from buttons which trigger the displayed
        <section> element to change
const nextSectionTriggers = [startBtn, ...nextBtns]


// Create an array from all the <section> elements
const sections = [startSection, ...questionGroups, endSection]


// Create an array from all question <li> elements in detailed
        results modal
const resultsQuestions = [ ...questionsInModal ]


// Create an array from all stat <li> elements at the end of the
        game
const resultsStats = [ ...userStatsItems ]


// Create array from the questions.json object keys, which will
        help in selecting random questions
const questionsKeysArray = Object.keys(questions)


// Create array from the users.json object values
const usersValuesArray = Object.values(users)


// Create a new set which will store 10 random questions
const randomTen = new Set()


// Create a set to store fake users
const gameUsers = new Set()


// Create a variable to store current user's chosen username
let currentUser
```

```
// Create a variable to store the user's running score
let runningScore = 0


// Declare necessary variables for cycling through the <section>
        elements
const lastSectionIndex = sections.length - 1
let displayedSectionIndex = 0
let sectionOffset


// Declare necessary variables to display a question and store
        the selected answer
let nextQuestionNumber = displayedSectionIndex + 1
let currentQuestion
let selectedAnswer
let correctAnswer
let userSelection = false


// Create map to store detailed results
const currentUserDetailedResults = new Map()
currentUserDetailedResults.set("results", [])


// Create map to store all users stats
const usersStats = new Map()
usersStats.set("stats", [])
```

Let's take a moment to break down what's going on here.
>
First, using the spread operator we are creating some arrays out of the HTML NodeLists we get from document.querySelectorAll so that they're easier to work with; then, we're creating an array of the keys from **questions.json**, as well as an array of values from **users.json**, using the Object.keys() and Object.values() methods, respectively.
>
After that, we're creating two Sets – one to store ten random questions from our questions file, and the other to store our fake users. Then, we're creating several variables which we will use in our game logic to keep track of the current user and the running score, cycle through the `<section>` elements, display a question/answer set and store the user's selected answer.
>
Lastly, we're creating two Maps – one to store the results of the user's

answers throughout the game and the other to store our fake users for
score ranking (in both cases setting the value argument to an empty array).
>

Build a Trivia Game part 4

Now, we will add our fake users' usernames to the gameUsers Set, and all the user data to the userStats Map:

```
// Add fake users' usernames to gameUsers Set and the full fake
        user objects to userStats Map
for (const user of usersValuesArray) {
  gameUsers.add(user.username)
  usersStats.entries().next().value[1].push(user)
}
```

Here, we're utilizing the **.entries()** Map method, then chaining on the built in **.next()** method, then accessing the array in each entry's value with **.value[1]**.

Doing it this way means we won't have to use nested loops (which, as we learned earlier today in the section on Big O Notation, is best to avoid).

It works because the **.entries()** method returns a new iterable object, and when the **.next()** method is chained on it selects the first entry in the iterable object and then "removes" and returns it while still retaining the remaining entries. Then each subsequent time **.entries().next()** is invoked it repeats this process until it reaches the end of the iterable object.
>
Now, let's move on and create a while loop to add ten random questions to the randomTen Set, then create a variable to store the Set's values:

```
// Add 10 random questions from JSON file to the randomTen array
while (randomTen.size < 10) {
  const randomIndex = Math.floor(Math.random() *
        questionsKeysArray.length)
  const randomObjectKey = questionsKeysArray[randomIndex]
  if (randomTen.has(questions[randomObjectKey])) {
    continue;
  } else {
    randomTen.add(questions[randomObjectKey])
  }
}


// Get access to the set's values
const randomQuestionSet = randomTen.values()
```

Now that we've got a decent start on building out our game's logic, we'll stop here for today. In the next lesson, we'll finish building our trivia game and have a chance to try it out!

Conclusion & Takeaways

> In this section we began building a simple entertainment trivia game, step by step, utilizing a lot of what we learned in the previous two sections about Sets and Maps. Using what you learn in the context of a project or exercise like this is always a good reinforcement to help solidify concepts that otherwise can seem very abstract and difficult to digest. As mentioned, in the next lesson we will finish building our trivia game's functionality and get to see the end result!

Gamification Continued: Introduction

Goals

By the end of this lesson you will:
- Have finished the logic for the trivia game

> ### Introduction
>
> In the previous lesson, we began writing the JavaScript for our entertainment trivia game. We started with importing some fake user data and a collection of question/answer sets, then created variables to store the relevant DOM elements. Additionally, we added our fake users to a Set and created a function to select 10 random questions. In this lesson, we'll complete the necessary logic to take a user all the way to our "end" screen where they will be able to view their score and, inside a modal, see which questions they answered correctly and which they didn't. Let's get started!

Build a Trivia Game part 5

Okay, we've got a lot of the preliminary building blocks out of the way. Now, before continuing to implement the game logic, let's first take care of a small housekeeping matter related to the beginning state of our game:

```
/*
   Check if DOM's readyState is "complete", then move all
        question sections
   out of view
*/


document.onreadystatechange = (e) => {
  if (document.readyState === "complete") {
    sections.forEach((section, index) => {
      section.style.transform = `translateX(${index * 100}%)`
    })
  }
}
```

Once again, let's break this down for a moment. We're utilizing the **readyStateChange** event to listen for when the browser has completed parsing of our page, then we're moving all of the question `<section>` elements out of the main view by looping over them and setting each one's **translateX** transform to a percentage that increases every time the loop runs.

In essence, we're creating a carousel effect, where each question/answer set is in fact lined up next to each other but visually hidden. Then, when a user clicks the button to move to the next question/answer set, the next `<section>` element will slide in from the right and become visible.

Normally, carousels are used for images but they can be utilized in this way as well!

Build a Trivia Game part 6

Now, let's get back to our main game logic and define a few functions we will need:

```
// Define functions to handle valid and invalid state at game
        start
const setStartGameInvalidState = () => {
  usernameInput.style.border = "2px solid rgb(211, 70, 70)"
  validationMsg.style.display = "block"
  startBtn.setAttribute('disabled', '')
}



const setStartGameValidState = () => {
  usernameInput.style.border = "2px solid black"
  validationMsg.style.display = "none"
  startBtn.removeAttribute('disabled')
}



// Create helper function to check if gameUsers Set already
        contains the username entered
const userExists = (username) => {
  if (gameUsers.has(username)) {
    return true
  } else {
    return false
  }
}



// Create helper function to check validity of usernameInput
        value using the Validator.js package
const isValid = (usernameInputValue) => {
  if (!validator.isEmpty(usernameInputValue) &&
        validator.isLength(usernameInputValue, { min: 5 })) {
    return {
      valid: true,
      msg: null
    }
  } else {

    if (validator.isEmpty(usernameInputValue)) {
      return {
        valid: false,
```

```
        msg: "Required"
      }
    } else if (!validator.isLength(usernameInputValue, { min: 5
        })) {
      return {
        valid: false,
        msg: "Minimum 5 characters"
      }
    } else {
      return {
        valid: false,
        msg: "Input invalid"
      }
    }
  }
}


// Create an event listener callback function to sanitize and
        validate the input value from the username field
const checkUsernameValidity = () => {
  const sanitizedInput = DOMPurify.sanitize(usernameInput.value)
  const trimmedInput = validator.trim(sanitizedInput)
  const escapedInput = validator.escape(trimmedInput)

  const validation = isValid(escapedInput)
  const usernameNotTaken = userExists(escapedInput)

  if (!validation.valid || usernameNotTaken) {
    setStartGameInvalidState()

    if (usernameNotTaken) {
      validationMsg.innerHTML = "Username already in use"
    } else {
      validationMsg.innerHTML = validation.msg
    }

  } else {
    currentUser = escapedInput
    setStartGameValidState()
  }
}
```

In the code above, first we're handling some styling concerns for valid and invalid states when the game begins. This gives visual indications to the user that they need to enter a valid username before they can start the game.
>

Next we created a couple helper functions to help in verifying that the username isn't already taken, and that the user entered a valid username value (which is a minimum of 5 characters).
>
Finally, we created an event listener callback function to sanitize the user's input using the DOMPurify package and perform all our validity checks using the helper functions and the Validator.js package. This function will later be attached to an 'input' event listener on the username field.
>

Build a Trivia Game part 7

Let's add the next bit of code to handle when the user selects an answer to a question:

```
// Define a function to toggle the select indicator on any given
        answer button
const toggleSelectIndicator = (e) => {

  userSelection = true

  if (e.target.id.includes("answer-selection")) {
    const childrenArray =
        Array.from(e.target.parentElement.children)
    childrenArray.forEach((answerBtn) => {
      answerBtn.children[0].style.border = "2px solid #fff"
      answerBtn.children[0].style.boxShadow = "none"
    })

    e.target.children[0].style.border = "none"
    e.target.children[0].style["box-shadow"] = "var(--blue-neon-
        box)"

    selectedAnswer = e.target.children[1].innerText

    if (userSelection) {

        e.target.parentElement.nextElementSibling.removeAttribut
        e('disabled')
    }

  } else if (e.target.id.includes("-indicator") ||
        e.target.id.includes("__text")) {

    const childrenArray =
        Array.from(e.target.parentElement.parentElement.children
        )
    childrenArray.forEach((answerBtn) => {
      answerBtn.children[0].style.border = "2px solid #fff"
      answerBtn.children[0].style.boxShadow = "none"
    })

    if (e.target.id.includes("-indicator")) {

      e.target.style.border = "none"
      e.target.style["box-shadow"] = "var(--blue-neon-box)"

      selectedAnswer = e.target.nextElementSibling.innerText
```

```
    } else {

      e.target.previousElementSibling.style.border = "none"
      e.target.previousElementSibling.style["box-shadow"] =
        "var(--blue-neon-box)"

      selectedAnswer = e.target.innerText
    }

    if (userSelection) {

        e.target.parentElement.parentElement.nextElementSibling.removeAttrib
        ute('disabled')

    }
  }
}


// Define a function to check whether a given answer is correct
        and update user score
const checkAnswer = (question, userAnswer, correct) => {
  const results =
        currentUserDetailedResults.entries().next().value

  if (results[1].length < 10) {
    if (userAnswer === correct) {
      results[1].push({
        question,
        selectedAnswer,
        outcome: "Correct"
      })

      runningScore+=100

    } else {
      results[1].push({
        question,
        selectedAnswer,
        outcome: "Incorrect"
      })
    }
  }
}
```

In the first function (**toggleSelectIndicator**), we're "turning on" an indicator on any given answer button which will show that that answer was selected.

We're also being careful to set it up in such a way that no matter where on the answer button element the user clicks the indicator will still light up. We're able to achieve this by checking the click event target inside the nested if/else block and traversing the DOM accordingly to change the style of the correct element. Then, at the end of the function, we activate the "next" button so the user can go to the next question.

In the second function (**checkAnswer**), which takes in three arguments ("question", "userAnswer", "correct"), we're implementing some logic to check if the answer selected by the user is correct. First, we're getting access to the array stored as the value in the currentUserDetailedResults Map using the same **.entries().next().value** syntax we used when adding our fake users to the userStats Map.
>
Then, we're creating a nested if/else block to first check that the second item in that **.value** array, which is another array, has a length less than 10 and, if it does, then we do another if check to see if the user's answer matches the correct answer. Lastly, the function pushes a result object to the results array which contains a property we'll access later to show the user whether their answer was correct or not.
>

Build a Trivia Game part 8

> Almost done with our trivia game! We now need to define some logic to handle the game end, as well as an event listener callback function to handle when the user clicks the 'Next' button after answering a question:

```
// Define function to handle game end logic
const gameEnd = () => {
  const score = runningScore.toString()
  const results =
        currentUserDetailedResults.entries().next().value
  const stats = usersStats.entries().next().value

  finalScoreSpan.innerHTML = score

  stats[1].push({ username: currentUser,  score: runningScore})

  const sortedStats = stats[1].sort((a, b) => (a.score <
        b.score) ? 1 : -1)

  resultsStats.forEach((rs, index) => {
    rs.children[0].innerHTML = sortedStats[index].username
    rs.children[1].innerHTML =
        sortedStats[index].score.toString()
  })

  resultsQuestions.forEach((rq, index) => {
    rq.children[1].style["font-family"] = "var(--accent-font)"
    rq.children[0].children[0].innerHTML = results[1]
        [index].question
    rq.children[0].children[1].children[0].innerHTML =
        results[1][index].selectedAnswer

    rq.children[1].innerHTML = results[1][index].outcome

    if (results[1][index].outcome === "Correct") {
      rq.children[1].style.color = "green"
    } else if (results[1][index].outcome === "Incorrect") {
      rq.children[1].style.color = "var(--error-color)"
    }
  })
}


// Define function to display question/answer set from randomTen
        Set
```

```
const loadQuestionAndAnswers = () => {

  if (nextQuestionNumber != lastSectionIndex) {

    currentQuestion = randomQuestionSet.next().value
    correctAnswer = currentQuestion.correctAnswer
    sections[nextQuestionNumber].children[0].innerHTML =
        currentQuestion["question"]

    const answerNodes =
        Array.from(sections[nextQuestionNumber].children[1].chil
        dren)

    answerNodes.forEach((node, index) =>
        node.children[1].innerHTML = currentQuestion["answers"]
        [index])

    setTimeout(() => {
      container.style.background = "rgba(11, 70, 96, 0.75)"
    }, 350)
  }
}



// Define function to progress to the next section
const goToNextSection = () => {
  sections.forEach((section, loopIndex) => {
    sectionOffset = loopIndex - displayedSectionIndex
    section.style.transform = `translateX(${sectionOffset *
        100}%)`
    section.style.opacity = 1
  })
}



// Create an event listener callback function to move to the
        next <section> element
const nextSectionClickListener = (e) => {
  if (e.target.id === "start-btn") {
    gameUsers.add(currentUser)
    currentUserDisplay.children[0].innerHTML = currentUser
    currentUserDisplay.style.display = "block"
  }

  if (correctAnswer && selectedAnswer) {
    checkAnswer(currentQuestion["question"], selectedAnswer,
        correctAnswer)
  }

  if (displayedSectionIndex === lastSectionIndex - 1) {
    userSelection = false
```

```
    displayedSectionIndex++
    gameEnd()
    goToNextSection()

  } else {
    loadQuestionAndAnswers()
    userSelection = false
    displayedSectionIndex++
    nextQuestionNumber++
    goToNextSection()
  }
}
```

Build a Trivia Game part 9

Final breakdown: in the **gameEnd** function we're turning the current user's score into a string and assigning it to its corresponding DOM element's innerHTML. Then, we're getting the array of question/answer results from the currentUserDetailedResults Map and the array of fake user stats from the userStats Map. Next, we're pushing the stats of the current user (username and score) to the user stats array and sorting it from highest score to lowest score, assigning the sorted array to a new variable. After that, we're performing all the necessary DOM and style updates for the game end **

** element.

In the **loadQuestionAndAnswers** function, if the nextQuestionNumber variable does not equal the lastSectionIndex variable, we're getting the current question, answer choices, and correct answer from the Set containing random questions, then updating the innerHTML of the next section's DOM elements and setting a background color on our `<main>` element, so that when the function is called before moving to the next `<section>` element, everything is ready to go.
>
In the **goToNextSection** function, we're creating some logic to handle moving our carousel to the appropriate `<section>` based on the sectionOffset variable to which we're assigning the value yielded by subtracting the displayedSectionIndex variable from the index of the loop over the `<section>` elements.
>
Here's how it works: if, for example, the displayedSectionIndex (which is initially set to **0** and will get incremented every time we move to a new section) equals **1** then subtracting that from **0** will give us **-1**; then, if you multiply that by 100 you get **-100** and if you do **translateX** of **-100%** that moves the entire current section to the left and out of view while the next section gets moved entirely into view.
>
In the **nextSectionClickListener** function, first we're using an `if` check to see if the event target is the start button, then assigning the username the user entered to the innerHTML of the DOM element which will display it in the upper right corner throughout the game. Then, we're checking to see if anything has been assigned yet to the correctAnswer and selectedAnswer variables and, if so, passing all of it to the checkAnswer function.
>
Next, if the displayedSectionIndex is equal to the lastSectionIndex minus 1 (meaning the next section is the game end section), we set the userSelection variable to **false**, increment displayedSectionIndex, call the gameEnd()

function and then the goToNextSection() function; if the result of the if check is false (meaning the next section is not the game end section) we do pretty much everything the same except we also increment nextQuestionNumber and don't call gameEnd().
>
Last but not least, let's set up all of our event listeners:

```
// Add listener to all nextSectionTrigger buttons
nextSectionTriggers.forEach((trigger) => {
  trigger.addEventListener('click', (e) =>
        nextSectionClickListener(e))
})



// Add listeners to all the answer buttons
answers.forEach((answer) => {
  answer.addEventListener('click', (e) =>
        toggleSelectIndicator(e))
})



// Add input and blur listeners to username input field
usernameInput.addEventListener('input', checkUsernameValidity)
usernameInput.addEventListener('blur', checkUsernameValidity)



// Add a click listener to the Play Again button
playAgainBtn.addEventListener('click', () =>
        window.location.reload())
```

Congratulations, you created a trivia game! Click the refresh button in the preview window and give it a try!

Conclusion & Takeaways

> In this lesson we finished building our trivia game's functionality, step by step. Even though this was a 'simple' trivia game, there was a fair amount of code we had to write and logic we had to work through. Learning frontend development is in some ways like weight lifting: at the beginning, lifting a given amount might seem overwhelming but with persistence it will gradually get easier as the muscles get stronger. Your 'coder' muscles just got a challenging workout, but later as those muscles get stronger the code we wrote in this lesson might well seem "trivial", as it were. :)

Browser Storage: Introduction

Goals

By the end of this lesson you will:
- Have learned what browser storage is and why it can be useful
- Understand how to work with one of the storage options available (called localStorage)
- Understand what (and what not to put in localStorage)
>
Introduction
So, what is browser storage exactly? It is a powerful tool for web developers to store and access data on the client side, used commonly for things like caching, authentication, session management and more. One big benefit of using browser storage is persisting data even if the user refreshes the page, or closes the tab or the browser entirely. When that user opens up your website again, all the data will still be there.
>
There are a few different options when it comes to browser storage, but in the lesson we will focus on one in particular: localStorage. One of the advantages of localStorage over some of the alternatives is that there really isn't that much to learn in order to use it (which we will in the project later today).

Working with localStorage - Part 1

> Here is a few brief overview of how to work with localStorage in JavaScript. We'll start by looking at how to store, retrieve and delete simple values:

Storing a value:

```
localStorage.setItem("name", "Matthew")
```

Retrieving a value:

```
const nameOfUser = localStorage.getItem("name")
console.log(nameOfUser) // "Matthew"
```

Removing a value:

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

Working with localStorage - Part 2

In addition to simple values, we can also store more complex values like objects:

Storing an object:

```
const userData = {
  name: "Matthew",
  age: 35,
  city: "New York"
}


localStorage.setItem("user", JSON.stringify(userData))
```

Note that localStorage only stores strings, so you need to use the JSON.stringify() method to convert the object to a string before storing it.
>

Retrieving an object:

Likewise, if you want to get the same user data back, and in the form of a JavaScript object again instead of a string, you'll need to use JSON.parse():

```
const user = JSON.parse(localStorage.getItem("user"))
console.log(user) // { name: "Matthew", age: 35, city: "New
        York" }
```

Clear all data in localStorage:

```
localStorage.clear()
```

There are a few additional things to note here: localStorage has a maximum size limit (usually around 5-10 MB), and data stored in localStorage is accessible to any script running on the same origin; thus, it is potentially vulnerable to an attack by a malicious user, so you should be careful not to store sensitive data in localStorage.

Conclusion & Takeaways

In the preceding lesson, you got an overview of localStorage and how to work with it (adding items, deleting items, retrieving items, etc.). In addition, we talked about some of the benefits of localStorage – and browser storage in general – as well as some potential risks it poses if we're not careful about the sort of data we put there.

Exercise: Issue Logger App

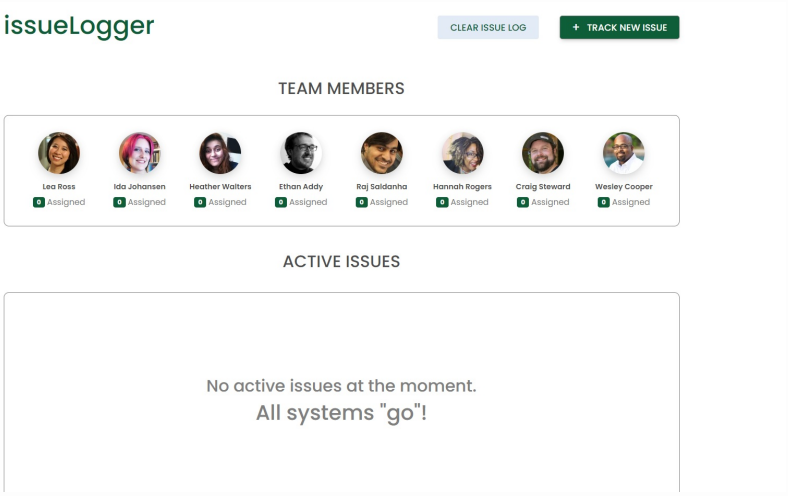
Goals

By the end of this lesson you will:
- Have practiced everything we've learned thus far about forms, validation, and using browser storage to persist data
- Have begun building an issue logger web app with an interactive form which adds elements to the DOM
>
Introduction
Time to put everything we've learned this week about forms and validation to use in a real project!
>
In this section we're going to start building an issue logger app which will utilize a form to collect information about an issue with a hypothetical software product, then assign this issue to be worked on by someone from an engineering team and will display a list of all currently assigned issues.
>
Business Context
As a developer working for a company, it's quite probable you will find yourself being assigned tasks through some kind of issue logging or ticketing system. Essentially, an issue is discovered with a piece of software or a website, sometimes reported by a user or multiple users, and a ticket will be created to investigate and resolve the issue. While we won't be building anything quite as robust as what you may encounter in the real world (e.g. - something like Atlassian's Jira software), nonetheless the general idea behind this project is in the same ballpark.

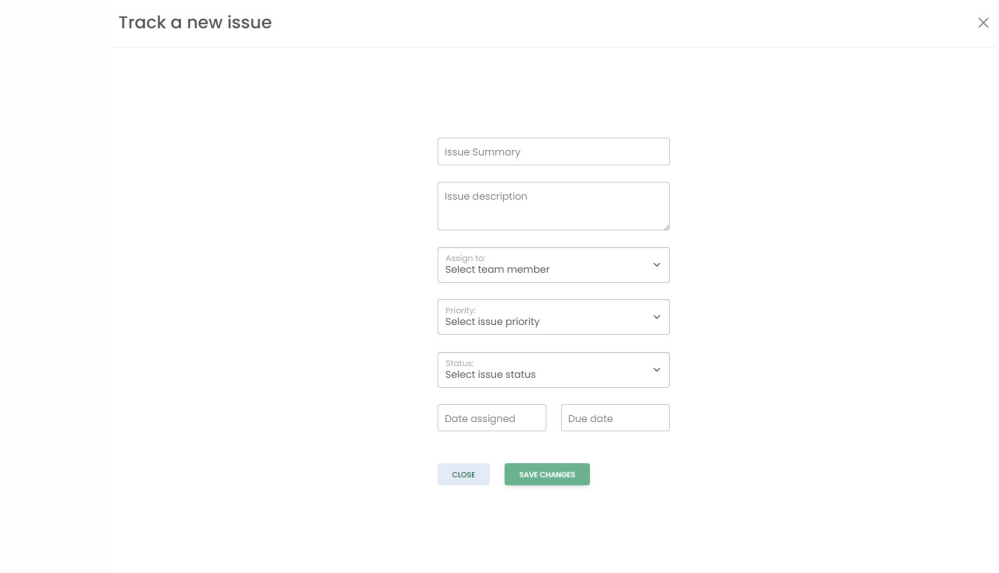
Issue Logger part 1

> For this project, we'll be using the two packages we learned about earlier (DOMPurify and Validator.js) to help us with our user input validation and sanitization logic. In addition, we'll be including data persistence so if the browser gets refreshed all the active issues that we create will remain in the list. We'll be using the browser's built-in localStorage which we learned about earlier.

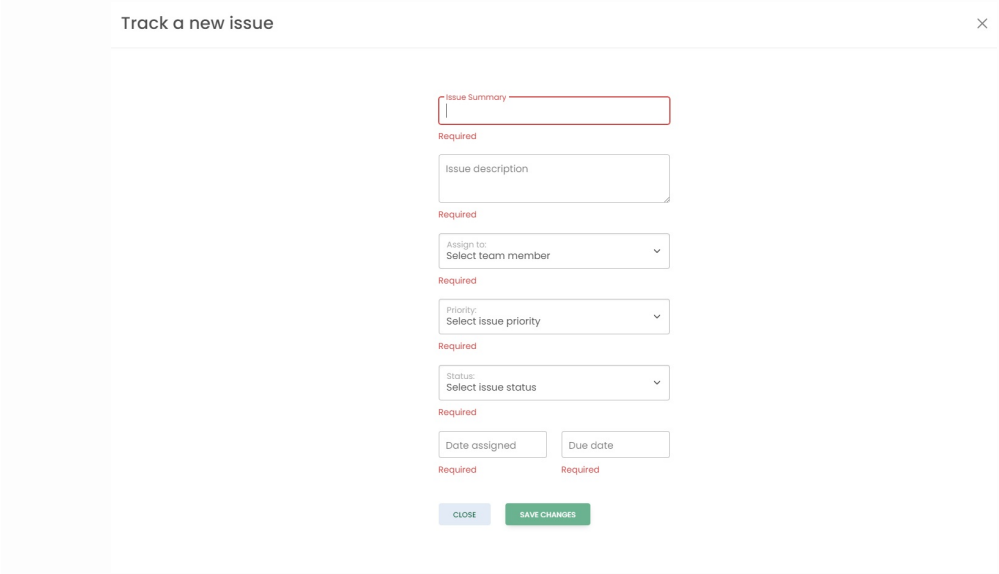
As in the Trivia Game project, all of the HTML and CSS has already been created, and we'll only focus on writing the JavaScript. Incidentally, this project also makes use of the MDBootstrap UI library (included via CDN), for the styling of some of the buttons and for the modal. Also, to generate a few fake engineering team members, the project makes use of some images from a random user API. If you would like to learn more about MDBootstrap or the API, you can follow the links below:
>
Bootstrap 5 & Vanilla JavaScript - Free Material Design UI KIT
>
Random User Generator
>
Before we start delving into the code, let's start off by seeing some screenshots of what we'll be building:
>
Main screen



Form modal



Form modal (with validation messages)



Alright, let's get started!

The first thing we need to do is import the team data from the JSON file (**team.json**). As you learned before, we need to use an import assertion for this to work properly:

```
// Import team data from JSON file
import team from './team.json' assert { type: 'json' }
```

To make sure there are no issues, do a console log of the team data:

```
console.log(team)
```

Open the **index.html** file in the browser, open DevTools, and go to the console panel. If nothing was wrong with the import, you should see the following:

>

>

If your console window looks like that, you're good to go! You can go ahead and erase the console log statement.

>

Issue Logger part 2

Now, as always, let's get access to all the DOM elements we'll need using getElementById, querySelector, or querySelectorAll:

```
// Get all DOM elements
const noIssuesMsg = document.getElementById('no-issues-msg')
const issuesListElement = document.getElementById('active-
        issues__list')
const teamListElement = document.getElementById('team-
        members__list')
const issueAssignSelect = document.getElementById('assign-to')
const form = document.getElementById('add-issues-form')
const issueSummary = document.getElementById('issue-summary')
const issueDescription = document.getElementById('issue-
        description')
const dateAssigned = document.getElementById('date-assigned')
const dueDate = document.getElementById('due-date')
const assignTo = document.getElementById('assign-to')
const issuePriority = document.getElementById('issue-priority')
const issueStatus = document.getElementById('issue-status')
const feedbackElements = document.querySelectorAll('.feedback')
const formSubmit = document.getElementById('modal-save-changes')
const modalDismissBtns =
        document.querySelectorAll('button:not(#modal-save-
        changes)[data-mdb-dismiss="modal"]')
const clearIssuesBtn = document.querySelector('.clear-issues-
        btn')
```

Next, let's create some variables that we'll need:

```
// Child elements of 'team-members__list' as an array
const teamListContents = [ ...teamListElement.children ]


// Team data from JSON file as an array
const teamDataFromJSON = Object.values(team)


/*
  Assign the value of each form field's 'id' attribute to
        variables,
  then store the variables in an object
*/
const issueSummaryId = issueSummary.id
const issueDescriptionId = issueDescription.id
```

```
const assignToId = assignTo.id
const issuePriorityId = issuePriority.id
const issueStatusId = issueStatus.id
const dateAssignedId = dateAssigned.id
const dueDateId = dueDate.id


const formInputs = {
  [issueSummaryId]: {
    notEmpty: false,
    sanitizedValue: ''
  },
  [issueDescriptionId]: {
    notEmpty: false,
    sanitizedValue: ''
  },
  [assignToId]: {
    notEmpty: false,
    sanitizedValue: ''
  },
  [issuePriorityId]: {
    notEmpty: false,
    sanitizedValue: ''
  },
  [issueStatusId]: {
    notEmpty: false,
    sanitizedValue: ''
  },
  [dateAssignedId]: {
    notEmpty: false,
    sanitizedValue: ''
  },
  [dueDateId]: {
    notEmpty: false,
    sanitizedValue: ''
  }
}
```

With the first two variables, we're creating arrays from the child nodes of the issues list element and our team data, respectively, then we're storing the id attribute's value for all of the form fields, then adding them to an object which will serve two purposes – one, it will provide an easy way for us to check that all the form fields are valid before enabling the user to click the 'submit' button; and two, it will store the sanitized values of each corresponding input and allow us to access them in an efficient way.
>

Issue Logger part 3

The next thing we need to do is define some functions to handle all of the operations our app will perform with regard to localStorage. We will look at these a few at a time:
>

```
/*
  Hydrate (load) all team details elements from team data in
        localStorage,
  create <option> elements for each team member and append them
        as children
  to the "Assign To" <select> element in the form
*/
const hydrateTeamElementsDOM = () => {
  const teamData = getTeamData()


  if (teamListContents.length === teamData.length) {
    teamData.forEach((teamMember, index) => {
      teamListContents[index].id = teamMember.id
      teamListContents[index].children[0].src =
        teamMember.imgSrc
      teamListContents[index].children[1].textContent =
        `${teamMember.firstName} ${teamMember.lastName}`
      teamListContents[index].children[2].children[0].textContent =
        teamMember.issuesAssigned.toString()




      const option = document.createElement('option')
      option.id = teamMember.id
      option.value = `${teamMember.firstName}
        ${teamMember.lastName}`
      option.textContent = `${teamMember.firstName}
        ${teamMember.lastName}`


      issueAssignSelect.appendChild(option)
    })
  }
}



/*
  Initialize localStorage with an entry for issues, and an entry
        for the team members
  (both arrays)
*/
```

```
const initDataStorage = () => {
  if (!localStorage.getItem('issueLogger.issues')) {
    localStorage.setItem('issueLogger.issues', '[]')
  } else {
    updateActiveIssuesDOM()
  }


  if (
      !localStorage.getItem('issueLogger.team') ||
      localStorage.getItem('issueLogger.team') === '[]'
    ) {


    localStorage.setItem('issueLogger.team',
        JSON.stringify(teamDataFromJSON))
  }
}


// Get all active issues from localStorage
const getActiveIssues = () => {
  const issues =
        JSON.parse(localStorage.getItem('issueLogger.issues') ||
        "[]")

  return issues
}


// Get all team data from localStorage
const getTeamData = () => {
  const team =
        JSON.parse(localStorage.getItem('issueLogger.team') || "
        []")

  return team
}
```

The first function will handle the initial “hydration” of the DOM elements related to our team members (names, images, etc.).
>
Next is a function which initializes our localStorage with two entries: one will hold an array of the active issues, and the other will store our team member data. Additionally, it does an initial update of the DOM elements related to active issues (which will be important for preserving our issues list and team data when the browser gets refreshed.
>

Then we have two functions which will handle getting issues and team data from localStorage.
>

Issue Logger part 4

Let's continue and create functions to update and delete issues:

```
// Update active issues data in localStorage when a new issue is
        added
const updateActiveIssues = (issue) => {
  const issues =
        JSON.parse(localStorage.getItem('issueLogger.issues') ||
        "[]")

  while (!issue.id) {
    const issueID = Math.floor(Math.random() * 99999)

    if (issues.find((item) => item.id === issueID)) {
      continue
    } else {
      issue.id = issueID
    }
  }

  const issueWithId = issue

  issues.push(issueWithId)
  localStorage.setItem('issueLogger.issues',
        JSON.stringify(issues))

  return issueWithId
}


// Update all DOM nodes related to active issues both when a new
        issue is added
// and when the page is reloaded
const updateActiveIssuesDOM = (newIssueId) => {
  const issues = getActiveIssues()
  let newIssue

  if (issues.length > 0 && newIssueId) {
    noIssuesMsg.style.display = "none"
    noIssuesMsg.parentElement.style.justifyContent = "flex-
        start"
    if (issues[issues.length - 1].id === newIssueId) {

      newIssue = issues[issues.length - 1]
```

```
    } else {
      newIssue = issues.find((issue) => {
        return issue.id === newIssueId
      })
    }

    const placeholderElement = document.createElement('div')
    const template = issueCardTemplate(newIssue)

    issuesListElement.appendChild(placeholderElement)

    placeholderElement.outerHTML = template

  } else if (issues.length > 0 && !newIssueId) {
    noIssuesMsg.style.display = "none"
    noIssuesMsg.parentElement.style.justifyContent = "flex-
        start"

    issues.forEach((issue) => {
      const placeholderElement = document.createElement('div')
      const template = issueCardTemplate(issue)

      issuesListElement.appendChild(placeholderElement)

      placeholderElement.outerHTML = template
    })
  }
}


// Delete all issue card elements from the DOM when issues data
        gets cleared
const deleteActiveIssuesDOM = () => {
    const nodesForDeletion = [ ...issuesListElement.children
        ].slice(1)

    nodesForDeletion.forEach((node) => {
      issuesListElement.removeChild(node)
    })

    noIssuesMsg.style.display = "block"
    noIssuesMsg.parentElement.style.justifyContent = "center"

}
```

We'll go ahead and pause here for now, then pick up where we left off in the next lesson.

Conclusion & Takeaways

In this section, we had the chance to put a lot of what we've learned into practice: form field validation, adding and removing elements from the DOM based on user input values, and of course persisting data using localStorage.
>
We used all of that to begin building a (very simplified) version of a real world application – namely, an issue logger/ticketing system which, as mentioned at the beginning of this lesson, is something you'll encounter a lot as a developer in the real world. In the next lesson, we will finish creating the remaining functionality.

Issue Logger App - Continued

Goals

By the end of this lesson you will:
- Have finished building the issue logger web app begun in the previous lesson
>
Introduction
In this lesson, we'll pick up right where we left off and finish building the functionality for our issue logger. Let's get started!

Issue Logger part 5

> These next two functions handle updating the team data and team DOM elements (specifically the number in the assigned issues badge):

```
// Update team data in localStorage
const updateTeamData = (assignedTeamMember) => {
  const teamData = getTeamData()


  const index = getTeamData().findIndex(
    (teamMember) => assignedTeamMember ===
        `${teamMember.firstName} ${teamMember.lastName}`
  )


  if (assignedTeamMember) {
    teamData[index].issuesAssigned++
  } else {
    teamData.forEach((teamMember) => teamMember.issuesAssigned =
        0)
  }


  localStorage.setItem('issueLogger.team',
        JSON.stringify(teamData))
}



// Update issues count for a team member when a new issue is
        added or when the issues data gets cleared
const updateTeamDataDOM = (assignedTeamMember) => {
  const teamData = getTeamData()


  const index = getTeamData().findIndex(
    (teamMember) => assignedTeamMember ===
        `${teamMember.firstName} ${teamMember.lastName}`
  )


  if (assignedTeamMember) {
    teamListContents[index].children[2].children[0].textContent =
        teamData[index].issuesAssigned.toString()


  } else {
    teamListContents.forEach((node) =>
        node.children[2].children[0].textContent = 0)
  }
}
```

Issue Logger part 6

> Now, we need to create a template for the issue card element so new issues can actually get added to the DOM in a structured way:

```
/*
  Create a string template for the issue card that will get
        added to the DOM
  when an issue is created
*/
const issueCardTemplate = (issue) => `
  <div class="card">
    <div class="card-body issue">
      <div class="issue-card-segment issue-id">
        <div class="issue-id__heading" >Issue ID: </div>
        <div class="issue-id__value">${issue.id}</div>
      </div>
      <div class="issue-card-segment issue-summary">
        <div class="issue-id__heading" >Summary: </div>
        <div class="issue-id__value issue-
        summary">${issue.issueSummary}</div>
      </div>
      <div class="issue-card-segment issue-priority">
        <div class="issue-id__heading" >Priority: </div>
        <div class="issue-id__value">${issue.issuePriority}
        </div>
      </div>
      <div class="issue-card-segment issue-status">
        <div class="issue-id__heading" >Status: </div>
        <div class="issue-id__value">${issue.issueStatus}</div>
      </div>
      <div class="issue-card-segment issue-assigned-date">
        <div class="issue-id__heading" >Date Assigned: </div>
        <div class="issue-id__value">${issue.dateAssigned}</div>
      </div>
      <div class="issue-card-segment issue-due-date">
        <div class="issue-id__heading" >Due Date: </div>
        <div class="issue-id__value">${issue.dueDate}</div>
      </div>
      <div class="issue-card-segment issue-asignee">
        <div class="issue-id__heading" >Assigned To: </div>
        <div class="issue-id__value">${issue.assignTo}</div>
      </div>
    </div>
  </div>
`.trim()
```

What we've done here is basically create a function which takes in the new issue data, and returns a string template literal with the issue data plugged in. Then just for good measure, we're running the trim function on the string to remove any leading and trailing white space.

>

Issue Logger part 7

This next set of functions will be somewhat similar to those we employed in the Trivia Game logic to handle checking and validating user input, as well as setting valid and invalid state (which includes an error message if invalid, and updates the 'notEmpty' property of its corresponding item in the 'formInputs' object we created at the beginning).
>
To keep things simple in terms of validation, we'll only have our form fields be invalid if empty:

```
// Define functions to handle displaying or hiding the
        validation message
const setInputInvalidState = (input, msg) => {

        input.parentElement.parentElement.nextElementSibling.sty
        le.display = "block"
  input.parentElement.parentElement.nextElementSibling.innerHTML
        = msg
}


const setInputValidState = (input) => {

        input.parentElement.parentElement.nextElementSibling.sty
        le.display = "none"
  input.parentElement.parentElement.nextElementSibling.innerHTML
        = ""
}



// Create a function to check the input value from the provided
        input field
const checkInput = (e) => {
  const validation = runValidation(e.value)
  let id = e.getAttribute('id')

  if (!validation.valid) {
    setInputInvalidState(e, validation.msg)
    formInputs[id].notEmpty = false
    return false
  } else {
    setInputValidState(e)
    formInputs[id].notEmpty = true

    return true
  }
}
```

```
// Create helper function to check validity of the provided
        input value using the Validator.js package
const runValidation = (inputValue) => {
  if (!validator.isEmpty(inputValue)) {
    return {
      valid: true,
      msg: null
    }
  } else {

    if (validator.isEmpty(inputValue)) {
      return {
        valid: false,
        msg: "Required"
      }
    } else {
      return {
        valid: false,
        msg: "Input invalid"
      }
    }
  }
}


// Create a function to check if all inputs are valid, then
        enable submit button
const enableSubmitIfInputsValid = () => {
  const allInputsValid = Object.keys(formInputs).every((input)
        => formInputs[input].notEmpty === true)


  if (allInputsValid) {
    formSubmit.removeAttribute('disabled')
  } else {
    formSubmit.setAttribute('disabled', '')
  }
}
```

Issue Logger part 8

Now it's time to create all of our event listener callback functions to handle the various user events of our app:

```
// Create function to handle the "blur" event of form unputs
const handleFieldBlur = (e) => {
  if (checkInput(e)) {
    const sanitizedInput = DOMPurify.sanitize(e.value)
    const trimmedInput = validator.trim(sanitizedInput)
    const escapedInput = validator.escape(trimmedInput)


    formInputs[e.id].sanitizedValue = escapedInput
    enableSubmitIfInputsValid()
  }
}


// Create function to handle the "input" event of form unputs
const handleFieldInput = (e) => {
  if (checkInput(e)) {
    enableSubmitIfInputsValid()
  }
}


// Create function to handle the "click" event of modal close
        buttons
const handleModalClose = (e) => {
  form.reset()
  feedbackElements.forEach((el) => el.style.display = "none")
  formSubmit.setAttribute('disabled', '')
}


// Clear all active issues
const clearActiveIssues = () => {
  localStorage.setItem('issueLogger.issues', '[]')
  updateTeamData()
  updateTeamDataDOM()
  deleteActiveIssuesDOM()
}
```

```
// Create function to handle the submit event on the form
const handleSubmit = (e) => {
  e.preventDefault()


  const issue = {
    issueSummary: formInputs[issueSummaryId].sanitizedValue,
    issueDescription:
        formInputs[issueDescriptionId].sanitizedValue,
    assignTo: formInputs[assignToId].sanitizedValue,
    issuePriority: formInputs[issuePriorityId].sanitizedValue,
    issueStatus: formInputs[issueStatusId].sanitizedValue,
    dateAssigned: formInputs[dateAssignedId].sanitizedValue,
    dueDate: formInputs[dueDateId].sanitizedValue
  }

  const issueWithId = updateActiveIssues(issue)
  updateActiveIssuesDOM(issueWithId.id)

  updateTeamData(issueWithId.assignTo)
  updateTeamDataDOM(issueWithId.assignTo)

  form.reset()
  formSubmit.setAttribute('disabled', '')
}
```

Issue Logger - Wrap Up

Okay, almost done! The last two things we need to do are: first, create our listener which will fire when our app first loads and initialize localStorage and update the DOM if needed; and second, attach all our event listener callbacks to their corresponding DOM elements:

```
/*
   Check if DOM's readyState is "complete", then call the
        initDataStorage
   and hydrateTeamElementsDOM functions
*/
document.onreadystatechange = (e) => {
  if (document.readyState === "complete") {
    initDataStorage()
    hydrateTeamElementsDOM()
  }
}

// Attach all event listeners
clearIssuesBtn.addEventListener('click', clearActiveIssues)


modalDismissBtns.forEach((btn) => btn.addEventListener('click',
        handleModalClose))


issueSummary.addEventListener('input', (e) =>
        handleFieldInput(e.target))
issueSummary.addEventListener('blur', (e) =>
        handleFieldBlur(e.target))


issueDescription.addEventListener('input', (e) =>
        handleFieldInput(e.target))
issueDescription.addEventListener('blur', (e) =>
        handleFieldBlur(e.target))


assignTo.addEventListener('input', (e) =>
        handleFieldInput(e.target))
assignTo.addEventListener('blur', (e) =>
        handleFieldBlur(e.target))


dateAssigned.addEventListener('input', (e) =>
        handleFieldInput(e.target))
```

```
dateAssigned.addEventListener('blur', (e) =>
        handleFieldBlur(e.target))



dueDate.addEventListener('input', (e) =>
        handleFieldInput(e.target))
dueDate.addEventListener('blur', (e) =>
        handleFieldBlur(e.target))



issuePriority.addEventListener('input', (e) =>
        handleFieldInput(e.target))
issuePriority.addEventListener('blur', (e) =>
        handleFieldBlur(e.target))



issueStatus.addEventListener('input', (e) =>
        handleFieldInput(e.target))
issueStatus.addEventListener('blur', (e) =>
        handleFieldBlur(e.target))



form.addEventListener('submit', (e) => handleSubmit(e))
```

And that's a wrap for this lesson. You should be proud of all of your hard work! Now, click the refresh button in the preview window and try out the app we just built!

Project: Add to personal website

Instructions

To incorporate what you learned this week into your personal website, choose one of the two options below – or both, if you want!

Option One

Build out client-side validation for your website's contact form. Use a combination of HTML validation and JavaScript validation. The functionality should be as follows:

- The border of any invalid inputs should change to a different color (usually a shade of red), in order to give a visual cue to the user that the value they entered is invalid. You can also have a brief error message appear below the input, if you want.
- The validity check can happen either as the user types in a given form field, or on submit (HINT: if implementing validation on submit, make use of the `preventDefault()` method on 'submit' event)
- The submit button should be disabled until all inputs are valid
- If the values of all inputs pass the validity check, after the form is submitted a success message should appear either above the first form field or below the last form field (but above the submit button). The success message text color is up to you, but the convention for this is typically a shade of green.

When you're finished, submit your form's HTML, CSS, and JavaScript below.

Option Two

In the GitHub repo for your personal website, create a new branch, and then try refactoring your personal website with Astro! Push all your code to GitHub (on the new branch you created). Then, submit a link to the repo below.

Happy coding!

> "Success is no accident. It is hard work, perseverance, learning, studying, sacrifice and most of all, love of what you are doing or learning to do."
> — Pele

Attribution

1. Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell. (n.d.). https://www.bigocheatsheet.com/
2. Checkbox - HTML: HyperText Markup Language | MDN. (2022, December 11). https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox
3. Constraint validation - HTML: HyperText Markup Language | MDN. (2022, December 19). https://developer.mozilla.org/en-US/docs/Web/HTML/Constraint_validation
4. Creative Commons — Attribution-ShareAlike 4.0 International — CC BY-SA 4.0. (n.d.). https://creativecommons.org/licenses/by-sa/4.0/
5. Document: readystatechange event - Web APIs | MDN. (2022, September 13). https://developer.mozilla.org/en-US/docs/Web/API/Document/readystatechange_event
6. EJS – Embedded JavaScript templates. (n.d.). https://ejs.co/
7. Element: blur event - Web APIs | MDN. (2022, September 29). https://developer.mozilla.org/en-US/docs/Web/API/Element/blur_event
8. Element: click event - Web APIs | MDN. (2022, September 22). https://developer.mozilla.org/en-US/docs/Web/API/Element/click_event
9. Element: focus event - Web APIs | MDN. (2022, September 29). https://developer.mozilla.org/en-US/docs/Web/API/Element/focus_event
10. Element: mouseout event - Web APIs | MDN. (2022, September 22). https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseout_event
11. Element: mouseover event - Web APIs | MDN. (2022, September 22). https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseover_event
12. Form validation meme. (n.d.). Imgflip. https://imgflip.com/i/76j1lg
13. Getting Started – Pug. (n.d.). https://pugjs.org/api/getting-started.html
14. GitHub - tc39/proposal-import-assertions: Proposal for syntax to import ES modules with assertions. (n.d.). GitHub. https://github.com/tc39/proposal-import-assertions
15. HTML attribute: autocomplete - HTML: HyperText Markup Language | MDN. (2022, December 25). https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/autocomplete
16. HTML DOM Event Object. (n.d.). https://www.w3schools.com/jsref/dom_obj_event.asp
17. HTMLElement: change event - Web APIs | MDN. (2022, September 14). https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/change_event
18. HTMLElement: input event - Web APIs | MDN. (2022, September 14). https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/input_event
19. HTMLFormElement.enctype - Web APIs | MDN. (2022, September 9). https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/enctype
20. HTTP request methods - HTTP | MDN. (2022, September 9). https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
21. Introduction | Handlebars. (n.d.). https://handlebarsjs.com/guide/

22. JavaScript Meme (ancient aliens guy). (n.d.). Imgflip. https://imgflip.com/i/76ipm5
23. Jaworski, A. (2022, January 4). Big O Notation Explained - JavaScript in Plain English. Medium. https://javascript.plainenglish.io/big-o-notation-explained-1f6a99328c82
24. Kms, V. (2022, January 7). Big O Performance of Arrays and Objects in JavaScript. Medium. https://javascript.plainenglish.io/performance-of-arrays-and-objects-in-javascript-through-the-lens-of-big-o-5a7c5891a43f
25. Olawanle, J. (2022, October 5). Big O Cheat Sheet – Time Complexity Chart. freeCodeCamp.org. https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/
26. Open Trivia DB. (n.d.). https://opentdb.com/
27. Paton, S. (n.d.). Custom Scrollbar Maker. CodePen.io. Retrieved December 28, 2022, from https://codepen.io/stephenpaton-tech/full/JjRvGmY
28. Radio - HTML: HyperText Markup Language | MDN. (2022, December 11). https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/radio
29. Random User Generator | Photos. (n.d.). https://randomuser.me/photos
30. Reacting to user events - GIF. (2022, September 6). GIPHY. https://giphy.com/gifs/cbc-schitts-creek-L2qukNXGjccyuAYd3W
31. Reacting to user events - GIF - alt. (2019, October 30). GIPHY. https://giphy.com/gifs/lightsout-comedy-central-lights-out-dXiqK1HQbl2s4ofwmT
32. tabindex - HTML: HyperText Markup Language | MDN. (2023, January 7). https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/tabindex
33. Text Input - HTML: HyperText Markup Language | MDN. (2022, December 11). https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/text
34. The Trivia API - Free API for Multiple Choice Quiz Questions. (n.d.). https://the-trivia-api.com/
35. Window: load event - Web APIs | MDN. (2022, November 15). https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event
36. Window: load event - Web APIs | MDN. (2022, November 15). https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event