JS Structure

Learning Objectives

- [ ] Understanding JavaScript modules
- [ ] Using import and export statements
- [ ] Understanding how to structure a JavaScript project

JavaScript Modules

JavaScript modules (sometimes also called "ECMAScript Modules" or "ESM") are a way to organize code into separate files. To use modules you have to let the browser know that you are using modules. This is done by adding the `type="module"` attribute to the `<script>` tag.

```
<script type="module" src="./my-module.js"></script>
```

> 💡 Modules allow you to use `import` and `export` statements but also change a few other things about how the browser treats your code that differ from standard scripts: They have their own scope and are not accessible from the global scope (unless exported). They also do not require the `defer` attribute as they are deferred by default. The scripts are executed in the more modern strict mode automatically.

> 💡 When reading about modules online you might stumble upon the file extension `.mjs` which is sometimes used for JavaScript modules. For modules both `.js` and `.mjs` work fine but we've decided to use the `.js` extension for consistency. There is a great section discussing `.js` vs `.mjs` on mdn.

Exporting using `export` Statements

Using the `export` statement you can export a variable or function to make it available to other modules. You can use named exports to export multiple variables or functions or one `default` export per module to export the main functionality of the module.

Named Exports

Usally named exports are created by using the keyword `export` dircectly before `const`, `let` or `function`.

```
export const name = "Alex";
export const age = 26;
export function sayHello() {
  console.log("Hello");
}
```

It is also possible to export functions or variables after they have been declared.

```
const name = "Alex";
const age = 26;
function sayHello() {
  console.log("Hello");
}

export { name, age, sayHello };
```

`default` Exports

Default exports are created by using the keyword `export default`. **You can only have one default export per module.**

Before a function declaration the syntax is similar to named exports.

```
export default function sayHello() {
  console.log("Hello");
}
```

For directly exporting variables as default you only declare the value of the thing you're exporting.

```
export default "Alex";
```

This is the same for arrow functions.

```
export default () => {
  console.log("Hello");
};
```

> 💡 Notice that there is no `const name =` or `const sayHello =` in the code above. Default exports are nameless and constant by default.

Just like with named exports you can export the default export after it has been declared.

```
const name = "Alex";

export default name;
```

> 💡 Notice that since default exports have no clear name they should semantically correspond to the name of the module. The above example should have a module name like `name.js`.

Mixing Named and `default` Exports

You can mix named and default exports.

```
export const name = "Alex";
export default function sayHello() {
  console.log("Hello");
}
```

Importing using `import` Statements

Code from other modules can be imported using the `import` statement. Import statements should always be placed at the top of the file. Everything that can be exported from a module can also be imported from another module.

Importing Named Exports

If another module exports a named export you can import it as such.

```
import { name, age } from "./my-module.js";
```

Now `name` and `age` are available in the current module.

Importing `default` Exports

If another module exports a `default` export you have to give it a name when importing it.

```
import myModule from "./my-module.js";
```

> 💡 Notice that the name you give it does not nessesarily have to match the name of the module or the original name of the thing that was exported. For example `myModule` could have the value of the `sayHello` function which is not clear from the name. Since you could import the same module in multiple files you can also give it a different name every time.

Mixing Named and `default` Imports

You can mix named and default imports.

```
import myModule, { name, age } from "./my-module.js";
```

Renaming Named Imports

You can rename named imports explicitly by using the `as` syntax.

```
import { name as firstName, age as yearsSinceBorn } from "./my-module.js";
```

The variables `firstName` and `yearsSinceBorn` are now available in the current module. This can be useful if the name of an import conflicts with a local variable name.

In contrast to default imports, you have to be explicit that you are renaming and what the original names were.

Structuring JavaScript Code

Utility Functions and Constants

Utility functions are functions that are used in multiple places in your code. They are usually smaller functions that are used to perform a specific task. They should be pure and not have any side effects.

Shared constants are constants that are used in multiple places in your code.

Functions and constants can be grouped into files that are named after the functionality they provide. For example `math.js` could contain functions like `add`, `subtract`, `multiply` and `divide`.

The file should have a named export for each function.

We recommend to create a `utils` folder in your project and put all utility functions in there.

Vanilla JavaScript Components

Vanilla JavaScript means that you are not using a framework like React (from vanilla being the most basic variant of ice cream).

Even though there is no set standard for structuring vanilla JavaScript components, we recommend the following:

- Create a folder for each component
- Make your component file and function names uppercase (PascalCase)
- Each component has a default export for the component function (e.g. `export default function ButtonGroup()`)
- Components can take arguments that are called props or properties a convention (e.g. `export default function ButtonGroup(props)`)
- Components should not depend on the outside world and create their own DOM elements
- Components should return a single DOM element

> 💡 **These are just recommendations**
>
> Another common convention is to use kabap-case names for component files. This way they are named after the BEM block class name. For example you could use `button-group/button-group.js` and `button-group/button-group.css` for a component that has a `.button-group` class. This is the same organizational style you are used to when working with just CSS.

> The name of the component function could also be `createButtonGroup()` or even `createButtonGroupElement()` to make it is clear that it is a function that creates a DOM element.
>
> Whatever style you choose, make sure you are consistent per project.

Here is an example of a component that creates a button:

```js
export default function Button(props) {
  const button = document.createElement("button");
  button.classList.add("button");
  button.textContent = props.text;
  return button;
}
```

An advanced use case are components that call other components (composition):

```js
import Button from "../Button/Button.js";

export default function ButtonGroup(props) {
  const buttonGroup = document.createElement("div");
  buttonGroup.classList.add("button-group");
  for (const buttonProps of props.buttons) {
    const button = Button(buttonProps);
    buttonGroup.append(button);
  }
  return buttonGroup;
}
```

Here is how these components could be used in another file:

```js
import ButtonGroup from "./ButtonGroup/ButtonGroup.js";
import Button from "./Button/Button.js";

const myButtonGroup = ButtonGroup({
  buttons: [{ text: "Button 1" }, { text: "Button 2" }, { text: "Button 3"
}],
});
document.body.append(myButtonGroup);

const myButton = Button({ text: "Button" });
document.body.append(myButton);
```

Resources

- Export and Import (javascript.info)

- A word against default exports (javascript.info)
- Case Styles: Camel, Pascal, Snake, and Kebab Case