JS Struktur

Lernziele

- [ ] Verstehen von JavaScript-Modulen
- [ ] Verwenden von `import`- und `export`-Anweisungen
- [ ] Verstehen, wie man ein JavaScript-Projekt strukturiert

JavaScript-Module

JavaScript-Module (manchmal auch "ECMAScript Modules" oder "ESM" genannt) sind eine Möglichkeit, Code in separate Dateien zu organisieren. Um Module zu verwenden, musst du dem Browser mitteilen, dass du Module verwendest. Dies geschieht durch Hinzufügen des `type="module"`-Attributs zum `<script>`-Tag.

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

> 💡 Module ermöglichen die Verwendung von import- und export-Anweisungen, ändern aber auch einige andere Dinge, wie der Browser deinen Code behandelt, die sich von normalen Skripten unterscheiden: Sie haben ihren eigenen Scope und sind nicht vom globalen Scope aus zugänglich (es sei denn, sie werden exportiert). Sie benötigen auch nicht das defer-Attribut, da sie standardmäßig verzögert geladen werden. Die Skripte werden automatisch im moderneren strict mode ausgeführt.

> 💡 Beim Lesen über Module online könntest du auf die Dateierweiterung .mjs stoßen, die manchmal für JavaScript-Module verwendet wird. Für Module funktionieren sowohl .js als auch .mjs, aber wir haben uns entschieden, die .js-Erweiterung aus Konsistenzgründen zu verwenden. Es gibt einen großartigen Abschnitt auf MDN, der .js vs .mjs diskutiert.

Exportieren mit export-Anweisungen

Mit der `export`-Anweisung kannst du eine Variable oder Funktion exportieren, um sie in anderen Modulen verfügbar zu machen. Du kannst benannte Exports verwenden, um mehrere Variablen oder Funktionen zu exportieren, oder einen `default` Export pro Modul, um die Hauptfunktionalität des Moduls zu exportieren.

Benannte Exports

Normalerweise werden benannte Exports erstellt, indem das Schlüsselwort `export` direkt vor `const`, `let` oder `function` gesetzt wird.

```
export const name = "Alex";
export const age = 26;
export function sayHello() {
```

```
  console.log("Hello");
}
```

Es ist auch möglich, Funktionen oder Variablen nach ihrer Deklaration zu exportieren.

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

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

`default` Exports

Default Exports werden mit dem Schlüsselwort `export default` erstellt. **Du kannst nur einen Default Export pro Modul haben.**

Vor einer Funktionsdeklaration ist die Syntax ähnlich wie bei benannten Exports.

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

Um Variablen direkt als Default zu exportieren, deklarierst du nur den Wert der exportierten Sache.

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

Dies gilt auch für Arrow Functions.

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

> 💡 Beachte, dass es in den obigen Codebeispielen kein const name = oder const sayHello = gibt. Default Exports sind standardmäßig namenlos und konstant.

Wie bei benannten Exports kannst du den Default Export auch nach seiner Deklaration exportieren.

```
const name = "Alex";

export default name;
```

> 💡 Da Default Exports keinen klaren Namen haben, sollten sie semantisch dem Namen des Moduls entsprechen. Das obige Beispiel sollte einen Modulnamen wie `name.js` haben.

Mischung aus Benannten und default Exports

Du kannst benannte und `Default` Exports mischen.

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

Importieren mit `import`-Anweisungen

Code aus anderen Modulen kann mit der `import`-Anweisung importiert werden. Import-Anweisungen sollten immer am Anfang der Datei platziert werden. Alles, was aus einem Modul exportiert werden kann, kann auch aus einem anderen Modul importiert werden.

Importieren von Benannten Exports

Wenn ein anderes Modul einen benannten Export exportiert, kannst du ihn als solchen importieren.

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

Jetzt sind `name` und `age` im aktuellen Modul verfügbar.

Importieren von `default` Exports

Wenn ein anderes Modul einen `default` Export exportiert, musst du ihm beim Importieren einen Namen geben.

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

> 💡 Beachte, dass der Name, den du ihm gibst, nicht unbedingt mit dem Namen des Moduls oder dem ursprünglichen Namen der exportierten Sache übereinstimmen muss. Zum Beispiel könnte myModule den Wert der sayHello-Funktion haben, was aus dem Namen nicht ersichtlich ist. Da du dasselbe Modul in mehreren Dateien importieren könntest, kannst du ihm auch jedes Mal einen anderen Namen geben.

Mischung aus Benannten und `default` Imports

Du kannst benannte und Default Imports mischen.

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

Umbenennen von Benannten Imports

Du kannst benannte Imports explizit umbenennen, indem du die `as`-Syntax verwendest.

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

Die Variablen `firstName` und `yearsSinceBorn` sind jetzt im aktuellen Modul verfügbar. Dies kann nützlich sein, wenn der Name eines Imports mit einem lokalen Variablennamen kollidiert.

Im Gegensatz zu Default Imports musst du ausdrücklich angeben, dass du umbenennst und wie die ursprünglichen Namen waren.

Strukturierung von JavaScript-Code

Utility Functions und Konstanten

Utility Functions sind Funktionen, die an mehreren Stellen in deinem Code verwendet werden. Sie sind normalerweise kleinere Funktionen, die eine bestimmte Aufgabe ausführen. Sie sollten rein sein und keine Seiteneffekte haben.

Gemeinsame Konstanten sind Konstanten, die an mehreren Stellen in deinem Code verwendet werden.

Funktionen und Konstanten können in Dateien gruppiert werden, die nach der von ihnen bereitgestellten Funktionalität benannt sind. Zum Beispiel könnte `math.js` Funktionen wie `add`, `subtract`, `multiply` und `divide` enthalten.

Die Datei sollte einen benannten Export für jede Funktion haben.

Wir empfehlen, in deinem Projekt einen `utils`-Ordner zu erstellen und alle Utility Functions dort abzulegen.

Vanilla JavaScript-Komponenten

Vanilla JavaScript bedeutet, dass du kein Framework wie React verwendest (ausgehend von Vanilla als der grundlegendsten Variante von Eiscreme).

Auch wenn es keinen festen Standard für die Strukturierung von Vanilla JavaScript-Komponenten gibt, empfehlen wir Folgendes:

- Erstelle einen Ordner für jede Komponente
- Benenne deine Komponentendateien und Funktionsnamen in Großbuchstaben (PascalCase)
- Jede Komponente hat einen Default Export für die Komponentenfunktion (z.B. `export default function ButtonGroup()`)
- Komponenten können Argumente entgegennehmen, die als Props oder Properties bezeichnet werden (z.B. `export default function ButtonGroup(props)`)

- Komponenten sollten nicht von der Außenwelt abhängen und ihre eigenen DOM-Elemente erstellen
- Komponenten sollten ein einziges DOM-Element zurückgeben

> 💡 **Dies sind nur Empfehlungen**

> Eine weitere gängige Konvention ist die Verwendung von kebap-case-Namen für Komponentendateien. Auf diese Weise werden sie nach dem BEM-Blockklassennamen benannt. Zum Beispiel könntest du `button-group/button-group.js` und `button-group/button-group.css` für eine Komponente verwenden, die eine `.button-group`-Klasse hat. Dies ist derselbe Organisationsstil, den du gewohnt bist, wenn du nur mit CSS arbeitest.

> Der Name der Komponentenfunktion könnte auch `createButtonGroup()` oder sogar `createButtonGroupElement()` lauten, um klarzustellen, dass es sich um eine Funktion handelt, die ein DOM-Element erstellt.

> Unabhängig davon, welchen Stil du wählst, achte darauf, dass du innerhalb des Projekts konsistent bleibst.

Hier ist ein Beispiel für eine Komponente, die eine Schaltfläche erstellt:

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

Ein fortgeschrittener Anwendungsfall sind Komponenten, die andere Komponenten aufrufen (Komposition):

```
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;
}
```

Hier ist ein Beispiel dafür, wie diese Komponenten in einer anderen Datei verwendet werden könnten:

```
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);
```

Ressourcen

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