JS Fetch

Lernziele

- Verstehen, wie asynchroner Code funktioniert
- Verstehen, wie man mit Promises und `async/await` arbeitet
- Die Fetch API verstehen
    - mit async/await
    - JSON
    - HTTP-Statuscodes
    - REST-API
    - Fehlerbehandlung

Asynchroner Code

Asynchroner Code ist Code, der im Hintergrund läuft. Dies ist nützlich für Aufgaben, die lange dauern können, aber den Hauptthread nicht blockieren müssen.

JavaScript ist eine Single-Threaded-Sprache, was bedeutet, dass immer nur eine Sache gleichzeitig passieren kann.

Das Blockieren des Hauptthreads ist schlecht, weil es den Benutzer daran hindert, mit der Seite zu interagieren, da kein anderer JavaScript-Code ausgeführt werden kann. Beispiele für asynchronen Code sind: Netzwerk-Anfragen, Dateisystemzugriffe, Animationen und Timer.

Promises

Asynchrone Funktionen geben ihren Rückgabewert nicht direkt zurück, sondern stattdessen ein Promise. Ein Promise ist ein Objekt, das den endgültigen Abschluss (oder Misserfolg) einer asynchronen Operation und deren resultierenden Wert darstellt. Meistens wird es von einer Funktion zurückgegeben, die eine asynchrone Operation durchführt. Es gibt zwei Hauptmethoden, um mit asynchronen Funktionen umzugehen: Arbeiten mit Promises oder Arbeiten mit `async/await`.

Asynchroner Code mit Promises

Du kannst die `then`-Methode mit einer Callback-Funktion verwenden, um auf den Abschluss der asynchronen Operation zu reagieren.

```
asynchronousFunction().then((value) => {
  console.log(value);
});
```

> 💡 Promises werden fast immer von anderen asynchronen APIs für dich erstellt. Nur selten erstellst du sie selbst. Wenn du ein Promise selbst erstellst (new Promise()), weißt du entweder genau, was du tust, oder du machst wahrscheinlich etwas falsch.

Asynchroner Code mit `async/await`

Async-Funktionen sind eine syntaktische Vereinfachung für Promises. Mit dem `await`-Schlüsselwort kannst du asynchronen Code schreiben, der wie synchroner Code aussieht. Jede Funktion kann mit dem `async`-Schlüsselwort versehen werden:

```
async function myAsyncFunction() {
  // ...
}

const myAsyncArrowFunction = async () => {
  // ...
};
```

Innerhalb einer async-Funktion kannst du das `await`-Schlüsselwort verwenden, um auf die Auflösung eines Promises zu warten:

```
async function myAsyncFunction() {
  const value = await otherAsynchronousFunction();
  console.log(value);
}
```

> 💡 `async`-Funktionen geben immer ein Promise zurück. Wenn die Funktion einen Wert zurückgibt, wird das Promise mit diesem Wert aufgelöst. Selbst wenn du nicht das `return`-Schlüsselwort verwendest, wird die Funktion ein Promise zurückgeben, das mit `undefined` aufgelöst wird, wenn es das Ende ihres Gültigkeitsbereichs erreicht.

Was ist eine API?

Der Begriff *API* steht für *Application Programming Interface*.

Eine *API* bietet eine Möglichkeit, wie eine Software (die Anwendung) mit einer anderen Software interagieren kann. Daher kann eine Anwendung eine Reihe von Funktionen und Regeln definieren, wie andere Software mit ihr interagieren kann. Dies wird als *Schnittstelle* bezeichnet. Denke an einen Vertrag zwischen zwei Softwares, der erklärt, wie sie zusammenarbeiten können.

APIs können aus verschiedenen Perspektiven betrachtet werden und treten auf verschiedenen Ebenen auf.

Ein Browser bietet viele Web APIs. Jede Web-API definiert eine Möglichkeit, wie eine JavaScript-Anwendung eine vom Browser bereitgestellte Funktion nutzen kann, wie zum Beispiel:

- Web Animations API
- Battery API
- Fetch API

APIs, die in einer Serverumgebung ausgeführt werden, sind ein anderer Typ von API. Sie werden von einem *Server* bereitgestellt, im Gegensatz zu den vom Browser bereitgestellten APIs (die auch als *Client* bezeichnet werden). Ein häufiges Anwendungsfall für solche APIs ist das Lesen / Laden von Daten. Andere Operationen wie das Schreiben oder Löschen von Daten sind ebenfalls möglich. Es gibt gängige Ansätze für die Architektur einer serverseitigen API. Ein solcher Ansatz sind REST-APIs, die später in diesem Dokument erklärt werden.

Fetch API (mit async & await)

`fetch` ist eine web API, um Ressourcen asynchron aus dem Netzwerk zu laden, wie z.B. Textdokumente.

```
async function fetchData() {
  const response = await fetch("/url/to/something");
  const data = await response.json();
  return data;
}
```

Dies geschieht im obigen Beispiel:

1. Wir markieren die Funktion mit dem `async`-Schlüsselwort, weil wir `await` innerhalb der Funktion verwenden möchten.
2. Wir deklarieren eine Variable namens `response`. Sie speichert das Response-Objekt, das von `fetch` zurückgegeben wird.
3. Sobald dieses Promise aufgelöst ist (die Netzwerk-Anfrage ist abgeschlossen), rufen wir die `.json`-Methode auf der `response`-Variablen auf. Diese Funktion gibt ein weiteres Promise zurück.
4. Dieses zweite Promise wird mit den tatsächlichen Daten (Payload) aufgelöst, die von JSON (ein formatiertes Zeichenkette) in einen JavaScript-Wert oder -Objekt konvertiert wurden. Dieses Ergebnis wird in der Variable `data` gespeichert.
5. Die Funktion gibt den Wert zurück, der in der Variable `data` gespeichert ist.

> 💡 Die async und await-Syntax hilft dabei, mit den beiden aufeinander folgenden Promises umzugehen, die erforderlich sind, um die Antwortdaten zu erhalten (im Gegensatz zur verschachtelten Syntax mit .then()).

> 💡 Das Response-Objekt bietet weitere nützliche Methoden (alle geben Promises zurück), darunter die folgenden:

> response.json() gibt ein Promise zurück, das in die heruntergeladenen Daten JSON-geparst als JavaScript-Wert oder -Objekt aufgelöst wird response.text() gibt ein Promise zurück, das in die heruntergeladenen Daten als Rohtext aufgelöst wird response.formData() gibt ein Promise zurück, das in die heruntergeladenen Daten als FormData aufgelöst wird response.blob() gibt ein Promise zurück, das in die heruntergeladenen Daten als Roh-Blob (das ist ein maschinenlesbares Format mit Nullen und Einsen) aufgelöst wird

---

JSON

JavaScript Object Notation (JSON) ist ein standardisiertes textbasiertes Format zur Darstellung von strukturierten Daten basierend auf der JavaScript-Objektsyntax. Es wird häufig verwendet, um Daten zwischen einem Client (Browser) und einem Server in Webanwendungen zu übertragen.

JSON ist sehr nah dran, ein Subset der JavaScript-Syntax zu sein. Die meisten gültigen JSON-Daten sind auch gültiger JavaScript-Code. Das bedeutet, dass du JSON in jede `.js`-Datei kopieren kannst, ein `const myData =` davor setzen kannst und zuschauen kannst, wie Prettier es wie echtes JavaScript aussehen lässt. Auf der anderen Seite ist gültiges JavaScript kein gültiges JSON. Es sieht so aus:

```
{
  "groupName": "Students",
  "groupSize": 100,
  "students": [
    {
      "name": "John Doe",
      "age": 42,
      "location": "Pleasantville",
      "member": true,
      "groups": ["students", "citizens", "new"]
    },
    {
      "name": "Jane Doe",
      "age": 44,
      "location": "Pleasantville",
      "member": true,
      "groups": ["students", "citizens", "new"]
    },
    {
      "name": "Sam Doe",
      "age": 24,
      "location": "Pleasantville",
      "member": true,
      "groups": ["students", "citizens", "new"]
    }
  ]
}
```

Obwohl es stark der JavaScript-Objektsyntax ähnelt, kann es außerhalb von JavaScript verwendet werden. Andere Programmiersprachen bieten oft die Möglichkeit, JSON zu lesen (zu parsen).

JSON existiert als Zeichenkette, die in ein natives JavaScript-Objekt konvertiert werden muss, wenn du auf die Daten zugreifen möchtest. Kein Problem! - JavaScript bietet ein globales JSON-Objekt, das Konvertierungsmethoden in beide Richtungen enthält.

> 💡 JSON-Konvertierungsmethoden

> JSON.parse()

> Diese Methode parst eine JSON-Zeichenkette und konstruiert den durch die Zeichenkette beschriebenen JavaScript-Wert oder -Objekt.

> JSON.stringify()

> Diese Methode konvertiert einen JavaScript-Wert oder -Objekt in eine JSON-Zeichenkette.

HTTP Response Status Codes

HTTP-Antwortstatuscodes geben normalerweise an, ob eine bestimmte HTTP-Anfrage erfolgreich abgeschlossen wurde. Meistens wird *jede Art* von Antwort als erfolgreicher Abschluss der Anfrage angesehen.

Mit Ausnahme von benutzerdefinierter Serversoftware sind diese Antworten standardisiert. Eine Liste der HTTP-Antwortstatuscodes findest du hier.

Einer der bekanntesten HTTP-Antwortstatuscodes ist `404 not found`, den die meisten Webbenutzer wahrscheinlich schon einmal gesehen haben.

> 💡 Brauchst du eine niedliche Erklärung, was die HTTP-Antwortcodes bedeuten? Suche einfach nach 'HTTP status dogs' (oder Katzen, wenn du sie lieber magst).

Fehlerbehandlung

Es mag überraschen, dass `fetch()` keinen Fehler auslöst, wenn der Server einen schlechten HTTP-Status zurückgibt, z.B. Client- oder Serverfehler.

```
async function fetchSomething() {
  const response = await fetch("/bad/url/oops");
  const something = await response.json();
  return something;
}
```

Angenommen, dass im obigen Beispiel `'bad/url/oops` nicht zu einem vorhandenen Ort führt, würde der Server mit dem Statuscode `404` und dem Text `Page not found` antworten. Dies ist eine **abgeschlossene** HTTP-Anfrage.

Eine Anfrage wird nur als abgelehnt registriert, wenn keine Antwort abgerufen werden kann. Dies kann aufgrund von Netzwerkproblemen geschehen, wie z.B. keine Internetverbindung, der Host wurde nicht gefunden oder der Server antwortet nicht.

Wir können *gute* von *schlechten* HTTP-Antwortstatuscodes unterscheiden, indem wir uns auf die boolesche `response.ok`-Eigenschaft verlassen. Sie ist nur dann auf `true` gesetzt, wenn der HTTP-Antwortcode zwischen `200–299` liegt.

In unserem obigen Beispiel wäre `response.ok` auf `false` gesetzt und der Antwortcode wäre `404`.

Wenn etwas schiefgeht und wir keine Antwort erhalten, löst die Fetch API einen Fehler aus. Sobald ein Fehler ausgelöst wird, stoppt die Ausführung und JavaScript sucht nach dem nächsten `catch`-Block.

Jeder produktionsreife `fetch`-Aufruf sollte sich in einem `try...catch`-Block befinden:

```
async function fetchSomething() {
  try {
    const response = await fetch("/bad/url/oops");

    if (response.ok) {
      // Erfolg (Gute Antwort)
      const data = await response.json();
      return data;
    } else {
      // Fehler (Schlechte Antwort)
      console.error("Bad Response");
    }
  } catch (error) {
    // Fehler (Netzwerkfehler usw.)
    console.error("Ein Fehler ist aufgetreten");
  }
}
```

REST API

Wenn du eine API erstellst, musst du dir Gedanken darüber machen, wie du deine API strukturierst, die Funktionen, die du bereitstellst, und welche Regeln du anwendest. Dies wird als die Architektur deiner API bezeichnet.

Es gibt verschiedene gängige Ansätze, die im Web verwendet werden. Ein sehr beliebter ist die REST-API oder RESTful-API.

REST ist eine Sammlung von architektonischen Einschränkungen, kein Protokoll oder Standard. Du als API-Entwickler kannst REST auf verschiedene Weisen implementieren.

> 💡 REST steht für REpresentational State Transfer

Die Grundidee sieht so aus:

- Ein *Client* fordert eine *Ressource* von einem *Server* an (wie ein Dokument, das Informationen zu einem bestimmten Thema enthält)
- Der *Server* formuliert eine Antwort, die die Ressource in einem Format darstellt, das der *Client* versteht (wie JSON - andere Formate wie HTML, XML oder Klartext sind ebenfalls möglich)
- Dieser Datentransfer ändert den Zustand der Webanwendung.

Jede Ressource hat eine eindeutige Adresse. Die gesamte Kommunikation erfolgt über HTTP und verwendet verschiedene HTTP-Methoden (wie GET/POST/PUT/DELETE), um gewünschte Aktionen zu beschreiben.

> 💡 Dies ist eine sehr grundlegende und unvollständige Erklärung. Wenn du mehr darüber erfahren möchtest, was eine API RESTful macht, kannst du es hier nachlesen.

Ressourcen

- Thread on mdn
- Asynchronous on mdn
- Using Promises on mdn
- Async functions on mdn
- Promise.all() on mdn
- try...catch on mdn