JS Async Functions

Lernziele

- Verstehen, wie asynchroner Code funktioniert
- Arbeiten mit Promises
- Verwenden der Schlüsselwörter `async` und `await`

Asynchroner Code

Asynchroner Code ist Code, der im Hintergrund ausgeführt wird. 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 geschehen kann.

Das Blockieren des Hauptthreads ist schlecht, da 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, Dateisystemzugriff, Animationen und Timer.

Promises

Ein Promise ist ein Objekt, das den endgültigen Abschluss (oder das Scheitern) einer asynchronen Operation und dessen Ergebnis repräsentiert. Meistens wird es von einer Funktion zurückgegeben, die eine asynchrone Operation durchführt.

Das Promise-Objekt hat folgende Eigenschaften und Methoden:

Eigenschaft / Methode	Beschreibung
`state`	Der Status des Promise-Objekts, kann `"pending"`, `"resolved"` oder `"rejected"` sein
`result`	Das Ergebnis der asynchronen Operation (dies muss man fast nie direkt verwenden)
`then()`	Eine Methode, die eine Callback-Funktion annimmt, die aufgerufen wird, wenn die asynchrone Operation abgeschlossen ist
`catch()`	Eine Methode, die eine Callback-Funktion annimmt, die aufgerufen wird, wenn die asynchrone Operation fehlschlägt
`finally()`	Eine Methode, die eine Callback-Funktion annimmt, die aufgerufen wird, wenn die asynchrone Operation abgeschlossen ist, unabhängig davon, ob sie erfolgreich war oder nicht

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

```
});
```

> 💡 Promises werden fast immer von anderen asynchronen APIs für Sie erstellt, nur selten erstellt man sie selbst. Wenn Sie ein Promise selbst erstellen (`new Promise()`), wissen Sie entweder genau, was Sie tun, oder Sie machen wahrscheinlich etwas falsch.

Async Functions und das `await` keyword

Async Functions sind eine syntaktische Vereinfachung für Promises. Mit dem `await`-Schlüsselwort können Sie 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 Function können Sie das `await`-Schlüsselwort verwenden, um auf die Auflösung eines Promise zu warten:

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

Dies kann leichter zu lesen sein als die Promise-Syntax, insbesondere wenn Sie mehrere asynchrone Operationen haben, die voneinander abhängen.

```
async function myAsyncFunction() {
  const value1 = await functionThatReturnsAPromise1();
  const value2 = await functionThatReturnsAPromise2(value1);
  const value3 = await functionThatReturnsAPromise3(value2);
  console.log(value3);
}
```

Verglichen mit:

```
// vermeiden Sie folgendes:
function myFunction() {
  functionThatReturnsAPromise1()
```

```
    .then((value1) => {
      return functionThatReturnsAPromise2(value1);
    })
    .then((value2) => {
      return functionThatReturnsAPromise3(value2);
    })
    .then((value3) => {
      console.log(value3);
    });
}
```

> 💡 `async` Funktionen geben immer ein Promise zurück. Wenn die Funktion einen Wert zurückgibt, wird das Promise mit diesem Wert aufgelöst. Selbst wenn Sie das `return`-Schlüsselwort nicht verwenden, wird die Funktion ein Promise zurückgeben, das bei Erreichen des Endes ihres Bereichs mit `undefined` aufgelöst wird. Wenn die Funktion einen Fehler wirft, wird das Promise mit diesem Fehler abgelehnt.

Fehlerbehandlung

Beim Verwenden von Promises können Sie die `catch()`-Methode verwenden, um Fehler zu behandeln. Beim Verwenden von `async` Funktionen können Sie die `try`/`catch`-Syntax verwenden.

`try`/`catch`

```
async function myAsyncFunction() {
  try {
    const value = await functionThatReturnsAPromise();
    console.log(value);
  } catch (error) {
    console.error(error);
  }
}
```

Der `try`-Block wird ausgeführt, und wenn ein Fehler auftritt, wird der `catch`-Block ausgeführt. Der `catch`-Block hat Zugriff auf den geworfenen Fehler.

Wenn im `try`-Block ein Fehler auftritt, wird der `catch`-Block sofort ausgeführt und die Ausführung eines weiteren Codes im `try`-Block abgebrochen:

```
async function functionThatThrowsAnError() {
  throw new Error("ooops 😳");
}

async function myAsyncFunction() {
  try {
    const value = await functionThatThrowsAnError();
    // Der folgende Code wird nie ausgeführt, weil
```

```
    // `functionThatThrowsAnError()` einen Fehler wirft.
    // Die Ausführung springt zum `catch`-Block.
    console.log(value);
    const value2 = await functionThatReturnsAPromise2();
    console.log(value2);
  } catch (error) {
    console.error(error);
  }
}
```

`finally``

Sie können auch einen `finally`-Block nach jedem `try`-Block verwenden, um Code nach dem `try`-Block auszuführen, unabhängig davon, ob er erfolgreich war oder nicht:

```
async function myAsyncFunction() {
  try {
    const value = await functionThatReturnsAPromise();
    console.log(value);
  } catch (error) {
    console.error(error);
  } finally {
    console.log("done");
  }
}
```

`try`/`catch` ist nicht auf `async` Funktionen beschränkt

Die `try`/`catch`/(`finally`)-Syntax ist nicht auf `async` Funktionen beschränkt, Sie können sie mit jedem JavaScript-Code verwenden, der einen Fehler werfen könnte.

```
function functionThatThrowsAnError() {
  throw new Error("ooops 😳");
}

function myFunction() {
  try {
    functionThatThrowsAnError();
    console.log("this will never be executed");
  } catch (error) {
    console.error(error);
  } finally {
    console.log("done");
  }
}
```

Parallele Promises

`Promise.all()`

Wenn Sie mehrere asynchrone Operationen haben, die Sie parallel ausführen möchten, können Sie `Promise.all()` verwenden.

`Promise.all()` nimmt ein Array von Promises und gibt ein Promise zurück, das mit einem Array der Ergebnisse der Promises im gleichen Bestellungsreihenfolge aufgelöst wird.

```
async function myAsyncFunction() {
  try {
    const values = await Promise.all([
      functionThatReturnsAPromise1(),
      functionThatReturnsAPromise2(),
      functionThatReturnsAPromise3(),
    ]);
    console.log(values); // [value1, value2, value3]
  } catch (error) {
    console.error(error);
  }
}
```

Dies führt alle drei asynchronen Operationen parallel aus und wartet, bis alle abgeschlossen sind, bevor es fortfährt. Wenn eine der asynchronen Operationen fehlschlägt, wird `Promise.all()` ebenfalls fehlschlagen.

Steuerung, wann parallele Promises aufgelöst oder abgelehnt werden sollen

Es gibt fortgeschrittene Anwendungsfälle, in denen Sie steuern möchten, wann ein Promise aufgelöst oder abgelehnt wird. `Promise.allSettled()` (ist egal, ob Promises abgelehnt oder aufgelöst werden) und `Promise.any()` (löst auf, sobald das erste Promise aufgelöst ist) sind zwei Methoden, die Ihnen dies ermöglichen.

Asynchrone Browser-APIs

Hier sind einige Beispiele für asynchrone Browser-APIs, die ein Promise zurückgeben:

- `fetch()` — führt eine HTTP-Anfrage durch und gibt ein Promise zurück, das mit einem `Response`-Objekt aufgelöst wird
- `element.animate().finished` — animiert ein Element und gibt ein Promise zurück, das aufgelöst wird, wenn die Animation abgeschlossen ist
- `navigator.getBattery()` — ermittelt den aktuellen Batteriestand und gibt ein Promise zurück, das mit dem Batteriestand aufgelöst wird

---

Resources

- Thread on mdn
- Asynchronous on mdn
- Using Promises on mdn

- Async functions on mdn
- Promise.all() on mdn
- try...catch on mdn