4.3 - JSA: Asynchrone Programmierung

Inhaltsübersicht Abschnitt 4.3

- **Block A: Grundlagen der Asynchronität und Callback-Funktionen**
  - Synchron vs. Asynchron: Warum asynchrone Programmierung?
  - Callback-Funktionen als erstes Muster für Asynchronität
  - `XMLHttpRequest` für Serveranfragen mit Callbacks
  - Übungsaufgaben zu Block A
- **Block B: Promises für verbesserte Asynchronität**
  - Einführung in Promises: Lösung für "Callback Hell"
  - Erstellen und Konsumieren von Promises (`then`, `catch`, `finally`)
  - Promise Chaining und statische Promise-Methoden (`all`, `race`, `any`, `allSettled`)
  - Die `Workspace` API: Moderner, Promise-basierter HTTP-Client
  - Übungsaufgaben zu Block B
- **Block C: `async/await` für elegante Asynchronität**
  - `async` Funktionen und der `await` Operator
  - Fehlerbehandlung mit `try...catch` in `async` Funktionen
  - Parallele Ausführung mit `async/await`
  - Übungsaufgaben zu Block C
- **Micro-Projekt: Utility-Funktionen - Phase 4**

Vorbereitung: Ein einfacher Test-Server (Optional, für lokale Tests)

Für einige der folgenden Beispiele (insbesondere `XMLHttpRequest` und frühe `Workspace`-Beispiele, die auf den Edube-Texten basieren) wird ein lokal laufender Test-Server referenziert. Dieser Server kann wie folgt mit Node.js und Express aufgesetzt werden (basierend auf Edube 4.3.1):

`server.js` (Beispiel):

```
// server.js
const express = require('express');
const cors = require('cors'); // Cross-Origin Resource Sharing
const app = express();
const port = 3000;

// Middleware
app.use(cors()); // Erlaubt Anfragen von anderen Domains/Ports
app.use(express.urlencoded({ extended: true })); // für form-data
app.use(express.json()); // für application/json

// Hilfsfunktion für künstliche Verzögerung
const sleep = (waitTimeInMs) => new Promise(resolve => setTimeout(resolve, waitTimeInMs));

// Routen
app.get('/', (req, res) => {
  res.send(`
    <!DOCTYPE html>
    <html>
    <head><title>Test Server</title></head>
    <body><h1>Asynchroner Test Server läuft</h1><p>Verwende die /json Route zum Testen.</p>
</body>
    </html>
  `);
});

app.get('/json', async (req, res) => {
  const randomDelay = Math.floor(Math.random() * 1500) + 500; // 0.5 bis 2 Sekunden
  let value = req.query.value || 0;
  let square = parseFloat(value) * parseFloat(value);
```

```
  console.log(`Anfrage an /json mit value=${value}, Delay=${randomDelay}ms`);
  await sleep(randomDelay);

  if (isNaN(square)) {
    res.status(400).json({ error: 'Invalid value provided. Must be a number.' });
    return;
  }

  res.json({
    originalValue: value,
    square: square,
    delayTimeInMs: randomDelay,
    timestamp: new Date().toISOString()
  });
});

app.post('/data', async (req, res) => {
  const randomDelay = Math.floor(Math.random() * 1000) + 200;
  console.log('POST Anfrage an /data empfangen:', req.body);
  await sleep(randomDelay);

  if (!req.body || Object.keys(req.body).length === 0) {
    res.status(400).json({ message: 'No data received in POST request.', receivedData: req.body
});
    return;
  }

  res.status(201).json({
    message: 'Data received successfully!',
    receivedData: req.body,
    delayTimeInMs: randomDelay
  });
});

app.listen(port, () => {
  console.log(`Test-Server läuft auf http://localhost:${port}`);
});
```

Benötigte Pakete installieren: `npm init -y` (falls noch kein `package.json` vorhanden) `npm install express cors`

Server starten: `node server.js`

Wenn dieser Server läuft, können die Beispiele, die `http://localhost:3000/json` oder `http://localhost:3000/data` verwenden, lokal getestet werden. Für andere Beispiele verwenden wir öffentliche APIs.

Block A: Grundlagen der Asynchronität und Callback-Funktionen

A.1 Synchron vs. Asynchron: Warum asynchrone Programmierung? (Edube 4.3.2)

JavaScript ist in seiner grundlegenden Ausführung **single-threaded**, was bedeutet, dass es immer nur eine Anweisung (einen Task) zur gleichen Zeit ausführen kann.

- **Synchrone Ausführung:** Die Anweisungen werden strikt nacheinander abgearbeitet. Eine neue Anweisung beginnt erst, wenn die vorherige vollständig abgeschlossen ist. Wenn eine Operation lange dauert (z.B. das Laden einer großen Datei, eine komplexe Berechnung oder eine Netzwerkanfrage, die auf eine Antwort wartet), blockiert sie den gesamten Thread. Im Browser würde dies dazu führen, dass die Benutzeroberfläche einfriert und nicht mehr auf Benutzereingaben reagiert.

```
// Synchrone Beispiele
console.log("Task 1: Start");

// Simulierte langsame synchrone Operation
// (In echtem Code vermeiden, da es den Thread blockiert!)
```

```
const startTime = Date.now();
while (Date.now() - startTime < 2000) {
  // Warte 2 Sekunden
}
console.log("Task 2: Langsame Operation beendet");

console.log("Task 3: Ende");
// Ausgabe:
// Task 1: Start
// (2 Sekunden Pause)
// Task 2: Langsame Operation beendet
// Task 3: Ende
// Die gesamte Anwendung ist während der 2 Sekunden blockiert.
```

- **Asynchrone Ausführung:** Ermöglicht es, langwierige Operationen zu starten, ohne auf deren Beendigung warten zu müssen. Die JavaScript-Engine kann in der Zwischenzeit andere Aufgaben erledigen. Wenn die asynchrone Operation abgeschlossen ist, wird eine zuvor definierte Funktion (oft ein "Callback") ausgeführt, um das Ergebnis zu verarbeiten. Dies wird durch Mechanismen wie die Event Loop, die Callback Queue und Web APIs (im Browser) oder das C++ API von Node.js ermöglicht.

  Hauptvorteile der Asynchronität:

  - **Kein Blockieren:** Die Anwendung (insbesondere die UI im Browser) bleibt reaktionsfähig.
  - **Effizienz:** Die Wartezeit auf externe Ressourcen (z.B. Netzwerkantworten) kann für andere Aufgaben genutzt werden.

  ```
  // Asynchrones Beispiel mit setTimeout
  console.log("Async Task 1: Start");

  setTimeout(function handleTimeout() { // Dies ist eine Callback-Funktion
    console.log("Async Task 2: Timeout-Callback ausgeführt (nach 2 Sekunden)");
  }, 2000); // Starte den Timer, aber warte nicht

  console.log("Async Task 3: Nach setTimeout Aufruf - Ende");
  // Mögliche Ausgabe:
  // Async Task 1: Start
  // Async Task 3: Nach setTimeout Aufruf - Ende
  // (ca. 2 Sekunden später)
  // Async Task 2: Timeout-Callback ausgeführt (nach 2 Sekunden)
  // Task 3 wird ausgeführt, *bevor* der Timeout-Callback dran ist.
  ```

A.2 Callback-Funktionen als erstes Muster für Asynchronität (Edube 4.3.3)

Eine **Callback-Funktion** (kurz: Callback) ist eine Funktion, die einer anderen Funktion als Argument übergeben wird und zu einem späteren Zeitpunkt – oft nach Abschluss einer asynchronen Operation – von dieser anderen Funktion aufgerufen wird.

`setTimeout` als einfaches Beispiel

`setTimeout(callbackFunction, delayInMilliseconds, arg1, arg2, ...)` ist eine eingebaute Funktion, die `callbackFunction` nach Ablauf von `delayInMilliseconds` ausführt. Die Argumente `arg1, arg2, ...` werden an die `callbackFunction` übergeben.

```
function greet(name, timeOfDay) {
  console.log(`Good ${timeOfDay}, ${name}!`);
}

console.log("Scheduling greeting...");

setTimeout(greet, 3000, "Alice", "morning");
// 'greet' ist der Callback.
// Er wird nach 3000ms mit den Argumenten "Alice" und "morning" aufgerufen.

setTimeout(() => { // Anonyme Arrow Function als Callback
  console.log("This anonymous callback was executed after 1 second.");
```

```
}, 1000);

console.log("Greeting scheduled. Other code can run now.");
```

`XMLHttpRequest` für Serveranfragen mit Callbacks

Vor der Einführung von `Workspace` und Promises war das `XMLHttpRequest` (XHR) Objekt der Standardweg, um HTTP-Anfragen von JavaScript im Browser aus zu senden. Es kann auch in Node.js über Bibliotheken verwendet werden, ist aber primär ein Browser-Konzept. (Edube 4.3.3)

Wichtige Eigenschaften und Methoden von `XMLHttpRequest`:

- `new XMLHttpRequest()`: Erstellt ein neues XHR-Objekt.
- `open(method, url, async)`: Initialisiert eine Anfrage.
  - `method`: HTTP-Methode (z.B. "GET", "POST").
  - `url`: Die URL, an die die Anfrage gesendet wird.
  - `async`: Boolean, ob die Anfrage asynchron (`true`, Standard) oder synchron (`false`, veraltet und sollte vermieden werden!) sein soll.
- `send(body)`: Sendet die Anfrage. `body` ist optional und wird für POST-Anfragen verwendet.
- `onload` (Property) oder `addEventListener('load', callback)`: Event-Handler, der aufgerufen wird, wenn die Anfrage erfolgreich abgeschlossen wurde (Antwort vollständig empfangen).
- `onerror` (Property) oder `addEventListener('error', callback)`: Event-Handler für Netzwerkfehler.
- `onreadystatechange` (Property): Ein Handler, der bei jeder Änderung des `readyState` aufgerufen wird.
- `readyState`: Ein numerischer Wert, der den Zustand der Anfrage angibt (0: uninitialized, 1: open, 2: headers_received, 3: loading, 4: done).
- `status`: HTTP-Statuscode der Antwort (z.B. 200 für OK, 404 für Not Found). Verfügbar, wenn `readyState >= 2`.
- `statusText`: HTTP-Statusmeldung (z.B. "OK", "Not Found").
- `responseText`: Die Antwort des Servers als Text. Verfügbar, wenn `readyState >= 3` (teilweise) und `readyState === 4` (vollständig).
- `responseXML`: Die Antwort als XML-Dokument (wenn der Server XML sendet und der `Content-Type` korrekt ist).
- `setRequestHeader(headerName, headerValue)`: Setzt einen HTTP-Request-Header (muss nach `open()` und vor `send()` aufgerufen werden).

Beispiel 1: GET-Anfrage an den lokalen Test-Server (Stellen Sie sicher, dass der oben beschriebene `server.js` läuft: `node server.js`)

```
// Dieses Beispiel ist für den Browser gedacht (oder eine Node-Umgebung mit XHR-Polyfill)
function getSquareFromServer(value) {
  const xhr = new XMLHttpRequest();
  const requestUrl = `http://localhost:3000/json?value=${encodeURIComponent(value)}`;

  // Konfiguration des Event-Handlers für die Antwort
  xhr.onload = function() {
    // 'this' innerhalb von xhr.onload bezieht sich auf das xhr-Objekt
    if (this.status >= 200 && this.status < 300) { // Erfolgreicher HTTP-Status
      try {
        const serverResponse = JSON.parse(this.responseText);
        console.log(`Server Response (Value: ${value}):`, serverResponse);
        // Beispielhafte Weiterverarbeitung:
        // displayResult(serverResponse.square, serverResponse.delayTimeInMs);
      } catch (e) {
        console.error("Error parsing JSON response:", e);
        console.error("Raw response text:", this.responseText);
      }
    } else {
      // Fehler vom Server (z.B. 404, 500)
      console.error(`HTTP Error: ${this.status} - ${this.statusText}`);
      console.error("Server response:", this.responseText);
    }
  };
```

```
  // Konfiguration des Event-Handlers für Netzwerkfehler
  xhr.onerror = function() {
    console.error("Network error occurred while trying to connect to the server.");
  };

  // Anfrage initialisieren
  xhr.open("GET", requestUrl, true); // true für asynchron

  // Anfrage senden
  xhr.send();
  console.log(`Request sent to server for value: ${value}. Waiting for response...`);
}

getSquareFromServer(5);
getSquareFromServer(12);
getSquareFromServer("abc"); // Testet Fehlerbehandlung des Servers
```

In diesem Beispiel ist die Funktion, die `xhr.onload` zugewiesen wird, die Callback-Funktion. Sie wird asynchron aufgerufen, sobald der Server antwortet und die Daten geladen sind.

Beispiel 2: GET-Anfrage an eine öffentliche API (JSONPlaceholder)

```
function fetchUserDetails(userId, callback) {
  const xhr = new XMLHttpRequest();
  const apiUrl = `https://jsonplaceholder.typicode.com/users/${userId}`;

  xhr.open("GET", apiUrl, true);

  xhr.onload = function() {
    if (this.status === 200) {
      try {
        const userData = JSON.parse(this.responseText);
        callback(null, userData); // Erster Parameter 'null' für keinen Fehler
      } catch (e) {
        callback(new Error("Failed to parse user data: " + e.message), null);
      }
    } else {
      callback(new Error(`Failed to fetch user. Status: ${this.status} - ${this.statusText}`),
null);
    }
  };

  xhr.onerror = function() {
    callback(new Error("Network error during user data fetch."), null);
  };

  xhr.send();
  console.log(`Workspaceing user details for ID: ${userId}...`);
}

// Callback-Funktion zur Verarbeitung der Nutzerdaten
function displayUserData(error, data) {
  if (error) {
    console.error("Error fetching user:", error.message);
    return;
  }
  console.log("User Details Received:");
  console.log(`  Name: ${data.name}`);
  console.log(`  Email: ${data.email}`);
  console.log(`  Website: ${data.website}`);
}

fetchUserDetails(1, displayUserData);
fetchUserDetails(5, displayUserData);
fetchUserDetails(99, displayUserData); // Test für einen nicht existierenden Nutzer (404)
```

Das Callback-Muster (hier `displayUserData`) wird verwendet, um den Code auszuführen, sobald die asynchrone Operation (`WorkspaceUserDetails`) abgeschlossen ist, sei es erfolgreich oder mit einem Fehler.

Probleme mit tief verschachtelten Callbacks ("Callback Hell" / "Pyramid of Doom"): Wenn mehrere voneinander abhängige asynchrone Operationen ausgeführt werden müssen, kann dies zu tief verschachtelten Callback-Funktionen führen. Dieser Code wird schnell unübersichtlich, schwer zu lesen und zu warten.

```
// Hypothetisches Beispiel für Callback Hell
// operation1(function(result1) {
//   operation2(result1, function(result2) {
//     operation3(result2, function(result3) {
//       // ... und so weiter
//       console.log(result3);
//     }, handleError3);
//   }, handleError2);
// }, handleError1);
```

Promises (im nächsten Block) bieten eine Lösung für dieses Problem.

Übungsaufgaben zu Block A (Grundlagen und Callbacks)

Aufgabe A.1: Benutzerdefinierter Timer mit Callback Schreiben Sie eine Funktion `executeAfterDelay(delayInSeconds, callbackFunction, message)`. Diese Funktion soll die `callbackFunction` nach `delayInSeconds` Sekunden aufrufen und ihr das `message`-Argument übergeben. Die `callbackFunction` soll die empfangene Nachricht auf der Konsole ausgeben. Testen Sie dies mit verschiedenen Verzögerungen und Nachrichten.

Aufgabe A.2: Daten von JSONPlaceholder mit XHR abrufen Schreiben Sie eine Funktion `WorkspaceTodos(callback)`. Diese Funktion soll `XMLHttpRequest` verwenden, um die Liste der Todos von `https://jsonplaceholder.typicode.com/todos` abzurufen. Die `callback`-Funktion soll zwei Argumente erhalten: `error` und `data`.

- Wenn ein Fehler auftritt (Netzwerkfehler oder HTTP-Status nicht 2xx), soll `error` ein Error-Objekt sein und `data` `null`.
- Wenn die Anfrage erfolgreich ist, soll `error` `null` sein und `data` das geparste Array der Todo-Objekte. Implementieren Sie den `callback`, um entweder die Fehlermeldung oder die Anzahl der abgerufenen Todos und das erste Todo-Objekt auf der Konsole auszugeben.

Aufgabe A.3: XHR-Anfrage mit Parameter Erweitern Sie die Funktion aus Aufgabe A.2 zu `WorkspaceTodoById(todoId, callback)`. Diese Funktion soll ein spezifisches Todo anhand seiner ID von `https://jsonplaceholder.typicode.com/todos/{todoId}` abrufen. Passen Sie den Callback an, um die Details dieses einen Todos auszugeben oder einen Fehler, falls das Todo nicht gefunden wird (z.B. bei einer ungültigen ID, die zu einem 404-Fehler führt).

Aufgabe A.4: Simulation mehrerer abhängiger asynchroner Aufrufe mit `setTimeout` Simulieren Sie drei voneinander abhängige asynchrone Operationen mit `setTimeout` und Callbacks:

1. `getUserData(userId, callback)`: Simuliert das Abrufen von Benutzerdaten nach 1 Sekunde. Gibt ein Objekt `{ id: userId, name: "Benutzer " + userId }` an den Callback.
2. `getUserPosts(userData, callback)`: Simuliert das Abrufen von Posts für einen Benutzer nach 1,5 Sekunden. Erwartet `userData` von Schritt 1. Gibt ein Array von Post-Objekten (z.B. `[{ postId: 1, title: "Post 1 für " + userData.name }, { postId: 2, title: "Post 2 für " + userData.name }]`) an den Callback.
3. `getPostComments(postData, callback)`: Simuliert das Abrufen von Kommentaren für den ersten Post nach 0,5 Sekunden. Erwartet `postData` (das Array) von Schritt 2. Gibt ein Array von Kommentar-Strings (z.B. `["Kommentar A zu " + postData[0].title, "Kommentar B"]`) an den Callback.

Rufen Sie diese Funktionen so auf, dass `getUserData` gestartet wird, dessen Ergebnis an `getUserPosts` weitergegeben wird, und dessen Ergebnis (speziell der erste Post) an `getPostComments`. Geben Sie am Ende die Kommentare auf der Konsole aus. Achten Sie auf die Verschachtelung der Callbacks.

Block B: Promises für verbesserte Asynchronität

B.1 Einführung in Promises (Edube 4.3.4)

Nachdem wir Callbacks als ein grundlegendes Muster für asynchrone Operationen kennengelernt haben, stoßen wir bei komplexeren Szenarien schnell an deren Grenzen. Insbesondere das Problem der "Callback Hell" oder "Pyramid of Doom" – tief verschachtelte Callbacks – macht Code schwer lesbar, wartbar und fehleranfällig.

Promises, eingeführt in ECMAScript 2015 (ES6), bieten eine robustere und elegantere Lösung für die Verwaltung asynchroner Abläufe.

Was ist ein Promise?

Ein **Promise** ist ein Objekt, das das Ergebnis einer asynchronen Operation repräsentiert, die entweder bereits abgeschlossen ist oder irgendwann in der Zukunft abgeschlossen sein wird (oder fehlschlägt). [cite: 510] Man kann es sich als einen Platzhalter für einen zukünftigen Wert vorstellen.

Vorteile von Promises gegenüber Callbacks:

- **Bessere Lesbarkeit:** Vermeidung von tiefen Verschachtelungen durch Method chaining (`.then()`, `.catch()`).
- **Zentralisierte Fehlerbehandlung:** Fehler können effizienter mit `.catch()` am Ende einer Kette oder an spezifischen Stellen behandelt werden.
- **Bessere Komposition:** Promises lassen sich leichter zu komplexeren asynchronen Abläufen kombinieren (z.B. mit `Promise.all()`).
- **Klar definierte Zustände:** Ein Promise befindet sich immer in einem von drei klar definierten Zuständen.

Zustände eines Promises

Ein Promise kann sich in einem der folgenden drei Zustände befinden: [cite: 521]

1. `pending` **(ausstehend):** Der Initialzustand. Die asynchrone Operation wurde noch nicht abgeschlossen. [cite: 521]
2. `fulfilled` **(erfüllt):** Die asynchrone Operation wurde erfolgreich abgeschlossen. Das Promise hat nun einen Ergebniswert. [cite: 521]
3. `rejected` **(abgelehnt):** Die asynchrone Operation ist fehlgeschlagen. Das Promise hat nun einen Fehlergrund (Reason). [cite: 521]

Ein Promise, das nicht mehr `pending` ist (also entweder `fulfilled` oder `rejected`), wird als `settled` **(eingelöst oder erledigt)** bezeichnet. Sobald ein Promise `settled` ist, ändert sich sein Zustand und sein Ergebnis/Fehlergrund nicht mehr – es ist final. [cite: 531, 532]

B.2 Erstellen von Promises (`new Promise`)

Ein Promise wird mit dem `Promise`-Konstruktor erstellt. Dieser Konstruktor erwartet eine Funktion als Argument, die als **Executor-Funktion** bezeichnet wird. [cite: 514]

Syntax:

```
const myPromise = new Promise(function executor(resolve, reject) {
  // Asynchroner Code oder eine Operation hier...
  // Wenn erfolgreich:
  //   resolve(wertDesErfolgs);
  // Wenn fehlgeschlagen:
  //   reject(grundDesFehlers);
});
```

Die Executor-Funktion:

- Wird **sofort** beim Erstellen des Promises ausgeführt. [cite: 515]
- Erhält zwei Argumente: `resolve` und `reject`. Dies sind Funktionen, die von der JavaScript-Promise-Implementierung bereitgestellt werden. [cite: 516]
    - `resolve(value)`**:** Diese Funktion wird innerhalb des Executors aufgerufen, wenn die asynchrone Operation erfolgreich abgeschlossen wurde. Der `value` wird zum Ergebniswert des Promises, und das Promise geht in den Zustand `fulfilled` über. [cite: 517]

- `reject(reason)`: Diese Funktion wird aufgerufen, wenn während der asynchronen Operation ein Fehler auftritt. Der `reason` (üblicherweise ein `Error`-Objekt) wird zum Fehlergrund des Promises, und das Promise geht in den Zustand `rejected` über. [cite: 517]

Beispiel 1: Ein Promise, das sofort erfüllt wird [cite: 518]

```
const immediatelyFulfilledPromise = new Promise((resolve, reject) => {
  console.log("Executor function started (fulfilled).");
  resolve("Operation was successful!");
});

console.log(immediatelyFulfilledPromise); // Wird oft als Promise { <pending> } geloggt, da der
Callback noch nicht dran war.
                                          // Die tatsächliche Auflösung passiert sehr schnell.

// Um den Zustand nach der Auflösung zu sehen (demonstrativ):
setTimeout(() => {
  console.log("State of immediatelyFulfilledPromise after a tick:",
immediatelyFulfilledPromise);
  // Im Browser-DevTools oder Node.js REPL sieht man oft den finalen Zustand.
}, 0);
```

Beispiel 2: Ein Promise, das sofort abgelehnt wird [cite: 524]

```
const immediatelyRejectedPromise = new Promise((resolve, reject) => {
  console.log("Executor function started (rejected).");
  reject(new Error("Something went wrong!"));
});

console.log(immediatelyRejectedPromise); // Auch hier <pending>, aber wird schnell rejected.

setTimeout(() => {
  console.log("State of immediatelyRejectedPromise after a tick:", immediatelyRejectedPromise);
  // Wichtig: Ein nicht abgefangenes rejected Promise kann zu einem "Uncaught (in promise)"
Fehler führen.
  // Wir lernen gleich, wie man das abfängt.
}, 0);
```

Man muss `.catch()` verwenden, um den Fehler eines `rejected` Promise zu behandeln, sonst führt dies zu einem Fehler in der Konsole.

Beispiel 3: Promise mit simulierter asynchroner Operation (`setTimeout`) [cite: 527, 530]

```
function fetchDataFromServer(succeeds = true) {
  return new Promise((resolve, reject) => {
    console.log("Executor: Starting data fetch simulation...");
    setTimeout(() => {
      if (succeeds) {
        const data = { id: 1, content: "Some data from server" };
        console.log("Executor: Data fetch successful. Resolving promise.");
        resolve(data);
      } else {
        const error = new Error("Failed to fetch data from server.");
        console.log("Executor: Data fetch failed. Rejecting promise.");
        reject(error);
      }
    }, 2000); // Simuliert eine 2-Sekunden-Verzögerung
  });
}

const successfulFetchPromise = fetchDataFromServer(true);
```

```
console.log("Promise for successful fetch created:", successfulFetchPromise); // Status: pending

const failedFetchPromise = fetchDataFromServer(false);
console.log("Promise for failed fetch created:", failedFetchPromise);     // Status: pending

// Wie man diese Promises konsumiert, sehen wir im nächsten Abschnitt.
```

B.3 Konsumieren von Promises

Sobald ein Promise erstellt wurde, benötigt man Methoden, um auf seinen Zustand (`fulfilled` oder `rejected`) zu reagieren und den Ergebniswert oder Fehlergrund zu verarbeiten. Dafür gibt es primär `.then()`, `.catch()` und `.finally()`. [cite: 534]

B.3.1 `.then(onFulfilled, onRejected)`

Die `.then()`-Methode ist die primäre Methode zum Interagieren mit einem Promise. Sie registriert zwei optionale Callback-Funktionen:

- `onFulfilled(value)`: Wird aufgerufen, wenn das Promise in den Zustand `fulfilled` übergeht. Das Argument `value` ist der Wert, mit dem das Promise erfüllt wurde (der Wert, der an `resolve` übergeben wurde). [cite: 535, 546]
- `onRejected(reason)`: Wird aufgerufen, wenn das Promise in den Zustand `rejected` übergeht. Das Argument `reason` ist der Grund für die Ablehnung (der Wert, der an `reject` übergeben wurde). [cite: 535]

Beide Callback-Funktionen sind optional. Man kann auch nur einen `onFulfilled`-Handler angeben (`myPromise.then(handleSuccess)`) [cite: 549, 550] oder nur einen `onRejected`-Handler (`myPromise.then(null, handleError)`).

Wichtig: `.then()` gibt **immer ein neues Promise** zurück[cite: 581]. Dies ermöglicht das Verketten von Promises (Promise Chaining), wie wir später sehen werden.

Beispiel mit `WorkspaceDataFromServer`:

```
// (Fortsetzung von fetchDataFromServer Beispiel)

console.log("Consuming successfulFetchPromise:");
successfulFetchPromise.then(
  function handleSuccess(data) { // onFulfilled
    console.log("SUCCESS (.then): Data received:", data);
  },
  function handleError(error) { // onRejected
    console.error("ERROR (.then): Failed to fetch:", error.message);
  }
);

console.log("Consuming failedFetchPromise:");
failedFetchPromise.then(
  (data) => { // onFulfilled (wird hier nicht aufgerufen)
    console.log("SUCCESS (.then): Data received (from failed promise):", data);
  },
  (error) => { // onRejected (wird hier aufgerufen)
    console.error("ERROR (.then): Failed to fetch (from failed promise):", error.message);
  }
);
// Nach ca. 2 Sekunden werden die entsprechenden Callbacks ausgeführt.
```

B.3.2 `.catch(onRejected)`

Die `.catch()`-Methode ist ein syntaktischer Zucker für `.then(null, onRejected)`. [cite: 552] Sie dient speziell der Fehlerbehandlung und wird aufgerufen, wenn ein Promise in der Kette (oder das Promise selbst) abgelehnt wird.

```
// (Fortsetzung von fetchDataFromServer Beispiel)

console.log("Consuming successfulFetchPromise with .catch():");
```

```
fetchDataFromServer(true)
  .then(data => {
    console.log("SUCCESS (.catch example): Data received:", data);
    // return data; // um den Wert an die nächste .then-Kette weiterzugeben
  })
  .catch(error => {
    // Dieser .catch() wird hier nicht aufgerufen, da das Promise erfüllt wird.
    console.error("ERROR (.catch example): This should not be called for success:",
error.message);
  });

console.log("Consuming failedFetchPromise with .catch():");
fetchDataFromServer(false)
  .then(data => {
    // Dieser .then() Callback wird hier nicht aufgerufen.
    console.log("SUCCESS (.catch example): Data received (from failed):", data);
  })
  .catch(error => {
    // Dieser .catch() wird aufgerufen.
    console.error("ERROR (.catch example): Failed to fetch (from failed):", error.message);
  });
```

`.catch()` ist sehr nützlich, um Fehler an einer zentralen Stelle am Ende einer Promise-Kette abzufangen. Es gibt ebenfalls ein neues Promise zurück. [cite: 575, 577]

B.3.3 `.finally(onFinally)`

Die `.finally()`-Methode registriert einen Callback, der ausgeführt wird, wenn das Promise `settled` ist, also entweder `fulfilled` oder `rejected`. [cite: 553, 583] Sie ist nützlich für Aufräumarbeiten, die unabhängig vom Erfolg oder Misserfolg der Operation durchgeführt werden müssen (z.B. einen Ladeindikator ausblenden, Datenbankverbindungen schließen). [cite: 554]

- Der `onFinally`-Callback erhält **keine Argumente**, da nicht bekannt ist, ob das Promise erfüllt oder abgelehnt wurde.
- `.finally()` gibt das ursprüngliche Ergebnis (Wert oder Fehler) des Promises weiter. Das bedeutet, man kann `.then()` oder `.catch()` *nach* `.finally()` verketten, um das Ergebnis weiterzuverarbeiten. [cite: 554, 584]

```
// (Fortsetzung von fetchDataFromServer Beispiel)

console.log("Consuming with .finally() - success case:");
fetchDataFromServer(true)
  .then(data => {
    console.log("SUCCESS (.finally example): Data:", data);
    return data; // Wichtig: Wert weitergeben für evtl. nächste .then()
  })
  .catch(error => {
    console.error("ERROR (.finally example): Error:", error.message);
    throw error; // Wichtig: Fehler weiterwerfen für evtl. nächste .catch()
  })
  .finally(() => {
    console.log("FINALLY (.finally example): Operation settled (cleanup here).");
  });

console.log("Consuming with .finally() - failure case:");
fetchDataFromServer(false)
  .then(data => { // Wird nicht aufgerufen
    console.log("SUCCESS (.finally example): Data:", data);
  })
  .catch(error => { // Wird aufgerufen
    console.error("ERROR (.finally example): Error:", error.message);
    // Optional: throw error; um den Fehler an eine äußere Fehlerbehandlung weiterzugeben
  })
  .finally(() => {
    console.log("FINALLY (.finally example): Operation settled (cleanup here), regardless of
outcome.");
  });
```

B.4 Promise Chaining (Verkettung) [cite: 592, 593]

Da `.then()` (und `.catch()`) immer ein neues Promise zurückgeben, können sie miteinander verkettet werden. Dies ermöglicht es, eine Sequenz von asynchronen Operationen elegant und lesbar zu gestalten, wobei jede Operation erst startet, wenn die vorherige erfolgreich abgeschlossen wurde.

- Wenn ein `onFulfilled`-Handler in `.then()` einen **Wert zurückgibt**, wird das von diesem `.then()` zurückgegebene Promise mit diesem Wert erfüllt.
- Wenn ein `onFulfilled`-Handler ein **neues Promise zurückgibt**, wird das von `.then()` zurückgegebene (äußere) Promise den Zustand des inneren Promises "annehmen". Das äußere Promise wird also erst `settled`, wenn das innere Promise `settled` ist, und es wird mit demselben Wert/Grund erfüllt/abgelehnt. [cite: 596, 597]

Beispiel 1: Einfache Wert-Verkettung [cite: 585]

```
Promise.resolve(10) // Startet mit einem erfüllten Promise mit dem Wert 10
  .then(value => {
    console.log("Step 1, received:", value); // 10
    return value * 2; // Gibt einen neuen Wert zurück
  })
  .then(value => {
    console.log("Step 2, received:", value); // 20 (Ergebnis von value * 2)
    return value + 5;
  })
  .then(value => {
    console.log("Step 3, received:", value); // 25 (Ergebnis von value + 5)
    console.log("Final result:", value);
  })
  .catch(error => {
    console.error("Chaining error:", error);
  });
// Ausgabe:
// Step 1, received: 10
// Step 2, received: 20
// Step 3, received: 25
// Final result: 25
```

Beispiel 2: Verkettung von Promises, die neue Promises zurückgeben

```
function stepOne(initialValue) {
  return new Promise((resolve, reject) => {
    console.log("Starting Step 1...");
    setTimeout(() => {
      const result = initialValue + 10;
      console.log("Step 1 finished, result:", result);
      resolve(result);
    }, 1000);
  });
}

function stepTwo(valueFromStep1) {
  return new Promise((resolve, reject) => {
    console.log("Starting Step 2 with value:", valueFromStep1);
    setTimeout(() => {
      if (valueFromStep1 > 15) {
        const result = valueFromStep1 * 2;
        console.log("Step 2 finished, result:", result);
        resolve(result);
      } else {
        console.log("Step 2 failed: value too low.");
        reject(new Error("Value from Step 1 was not greater than 15."));
      }
    }, 1500);
```

```
  });
}

function stepThree(valueFromStep2) {
  return new Promise((resolve, reject) => {
    console.log("Starting Step 3 with value:", valueFromStep2);
    setTimeout(() => {
      const result = `Final processed value: ${valueFromStep2}`;
      console.log("Step 3 finished.");
      resolve(result);
    }, 500);
  });
}

// Erfolgreiche Kette
console.log("--- Starting successful promise chain ---");
stepOne(10) // Startwert für stepOne ist 10 -> result 20
  .then(resultFromStep1 => {
    // resultFromStep1 ist 20
    return stepTwo(resultFromStep1); // stepTwo wird mit 20 aufgerufen, returned ein Promise
  })
  .then(resultFromStep2 => {
    // resultFromStep2 ist 40 (20 * 2)
    return stepThree(resultFromStep2); // stepThree wird mit 40 aufgerufen, returned ein Promise
  })
  .then(finalResult => {
    console.log("Chain SUCCESS:", finalResult); // "Final processed value: 40"
  })
  .catch(error => {
    console.error("Chain ERROR:", error.message);
  })
  .finally(() => {
    console.log("--- Successful promise chain settled ---");
  });

// Kette mit Fehler in stepTwo
setTimeout(() => {
  console.log("\n--- Starting promise chain expected to fail ---");
  stepOne(3) // Startwert für stepOne ist 3 -> result 13
    .then(stepTwo) // Kurzschreibweise für: result => stepTwo(result)
    .then(stepThree)
    .then(finalResult => {
      console.log("Chain SUCCESS (failure case - should not happen):", finalResult);
    })
    .catch(error => {
      // Der Fehler von stepTwo ("Value from Step 1 was not greater than 15.") wird hier
abgefangen.
      console.error("Chain ERROR (failure case):", error.message);
    })
    .finally(() => {
      console.log("--- Failing promise chain settled ---");
    });
}, 4000); // Startet die zweite Kette nach der ersten, um die Logs zu trennen
```

Fehler in einer Kette "springen" zum nächsten geeigneten `.catch()`-Handler.

B.5 Statische Promise-Methoden

Die `Promise`-Klasse bietet einige statische Methoden, um mit Gruppen von Promises zu arbeiten.

`Promise.all(iterable)` [cite: 602]

Nimmt ein Iterable (z.B. ein Array) von Promises entgegen und gibt ein neues Promise zurück.

- Das zurückgegebene Promise wird **erfüllt**, wenn **alle** Promises im Iterable erfüllt sind. Der Erfüllungswert ist ein Array mit den Erfüllungswerten der einzelnen Promises, in derselben Reihenfolge wie im ursprünglichen Iterable. [cite: 603, 607, 610]

- Wenn **mindestens ein** Promise im Iterable **abgelehnt** wird, wird das von `Promise.all()` zurückgegebene Promise sofort mit dem Grund dieser ersten Ablehnung abgelehnt. Die anderen Promises werden zwar weiter ausgeführt, aber ihr Ergebnis wird ignoriert. [cite: 612]

```
const p1 = Promise.resolve(3);
const p2 = 42; // Wird automatisch zu Promise.resolve(42)
const p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});
const p4Rejected = new Promise((resolve, reject) => {
  setTimeout(reject, 50, new Error('P4 Rejected!'));
});

// Alle erfolgreich
Promise.all([p1, p2, p3])
  .then(values => {
    console.log("Promise.all SUCCESS:", values); // [3, 42, "foo"]
  })
  .catch(error => {
    console.error("Promise.all ERROR (should not happen here):", error.message);
  });

// Mit einem Fehler
Promise.all([p1, p3, p4Rejected]) // p4Rejected wird zuerst ablehnen
  .then(values => {
    console.log("Promise.all SUCCESS (with error - should not happen):", values);
  })
  .catch(error => {
    console.error("Promise.all ERROR (expected):", error.message); // "P4 Rejected!"
  });
```

Nützlich, wenn mehrere unabhängige asynchrone Operationen abgeschlossen sein müssen, bevor man fortfährt.

`Promise.race(iterable)` **[cite: 615]**

Nimmt ein Iterable von Promises entgegen.

- Gibt ein neues Promise zurück, das sich so einlöst (erfüllt oder ablehnt) wie das **erste** Promise im Iterable, das `settled` wird. [cite: 616]

```
const promiseA = new Promise(resolve => setTimeout(resolve, 500, 'A is faster'));
const promiseB = new Promise(resolve => setTimeout(resolve, 100, 'B is fastest'));
const promiseCRejectedFast = new Promise((resolve, reject) => setTimeout(reject, 50, new
Error('C rejected first')));

Promise.race([promiseA, promiseB])
  .then(winner => {
    console.log("Promise.race (A,B) winner:", winner); // "B is fastest"
  })
  .catch(err => console.error("Race error", err));

Promise.race([promiseA, promiseCRejectedFast])
  .then(winner => {
    console.log("Promise.race (A,C_reject) winner (should not happen):", winner);
  })
  .catch(error => {
    console.error("Promise.race (A,C_reject) caught error:", error.message); // "C rejected
first"
  });
```

Nützlich für Szenarien wie Timeouts oder wenn nur das schnellste Ergebnis zählt.

`Promise.any(iterable)` **[cite: 615]**

Nimmt ein Iterable von Promises entgegen.

- Gibt ein neues Promise zurück, das erfüllt wird, sobald **irgendein** Promise im Iterable **erfüllt** ist. Der Erfüllungswert ist der Wert des ersten erfüllten Promises. [cite: 616]
- Wenn **alle** Promises im Iterable **abgelehnt** werden, wird das von `Promise.any()` zurückgegebene Promise mit einem `AggregateError` abgelehnt. Ein `AggregateError` ist ein spezieller Fehlertyp, der ein `errors`-Property (ein Array der einzelnen Ablehnungsgründe) enthält. [cite: 621]

```
const promiseX = new Promise(resolve => setTimeout(resolve, 200, 'X fulfilled'));
const promiseYRejected = new Promise((resolve, reject) => setTimeout(reject, 100, new Error('Y
rejected')));
const promiseZRejected = new Promise((resolve, reject) => setTimeout(reject, 300, new Error('Z
rejected')));

// Ein Promise wird erfüllt
Promise.any([promiseX, promiseYRejected])
  .then(firstFulfilled => {
    console.log("Promise.any (X, Y_reject) first fulfilled:", firstFulfilled); // "X fulfilled"
  })
  .catch(aggregateError => { // Wird hier nicht aufgerufen
    console.error("Promise.any (X, Y_reject) AggregateError:", aggregateError);
  });

// Alle Promises werden abgelehnt
Promise.any([promiseYRejected, promiseZRejected])
  .then(firstFulfilled => { // Wird hier nicht aufgerufen
    console.log("Promise.any (Y_reject, Z_reject) first fulfilled:", firstFulfilled);
  })
  .catch(aggregateError => {
    console.error("Promise.any (Y_reject, Z_reject) AggregateError:", aggregateError.message);
    console.error("Individual errors:", aggregateError.errors.map(e => e.message)); // ["Y
rejected", "Z rejected"]
  });
```

Unterscheidet sich von `Promise.race`, da `Promise.any` auf das erste *erfüllte* Promise wartet und Ablehnungen ignoriert, es sei denn, alle werden abgelehnt. [cite: 620, 622]

`Promise.allSettled(iterable)` **(Nicht in Edube, aber sehr nützlich)**

Nimmt ein Iterable von Promises entgegen.

- Gibt ein neues Promise zurück, das **immer erfüllt** wird, sobald **alle** Promises im Iterable `settled` sind (egal ob erfüllt oder abgelehnt).
- Der Erfüllungswert ist ein Array von Objekten, die den Zustand und das Ergebnis/den Grund jedes einzelnen Promises beschreiben:
  - `{ status: 'fulfilled', value: ... }`
  - `{ status: 'rejected', reason: ... }`

```
const promiseS1 = Promise.resolve('Success 1');
const promiseS2 = new Promise(resolve => setTimeout(resolve, 100, 'Success 2'));
const promiseS3Rejected = Promise.reject(new Error('Failure S3'));

Promise.allSettled([promiseS1, promiseS2, promiseS3Rejected])
  .then(results => {
    console.log("Promise.allSettled results:");
    results.forEach(result => {
      if (result.status === 'fulfilled') {
        console.log(`  Fulfilled with value: ${result.value}`);
      } else {
        console.log(`  Rejected with reason: ${result.reason.message}`);
```

```
      }
    });
  });
// Ausgabe:
// Promise.allSettled results:
//   Fulfilled with value: Success 1
//   Rejected with reason: Failure S3
//   Fulfilled with value: Success 2 (Reihenfolge der Ergebnisse wie im Input-Array)
```

Nützlich, wenn man das Ergebnis jeder asynchronen Operation kennen möchte, auch wenn einige fehlschlagen, ohne dass die gesamte Operation abbricht.

B.6 Die `Workspace` API: Moderner, Promise-basierter HTTP-Client

Die `Workspace()` API ist die moderne, standardisierte Schnittstelle in Browsern (und auch in Node.js über Module oder nativ in neueren Versionen verfügbar), um HTTP-Anfragen durchzuführen. Sie ist von Grund auf Promise-basiert und ersetzt weitgehend das ältere `XMLHttpRequest`-Objekt. [cite: 643]

Grundlegende Syntax: `Workspace(resource, options)` [cite: 646]

- `resource`: Die URL der Ressource, die angefragt werden soll, oder ein `Request`-Objekt.
- `options` (optional): Ein Objekt zur Konfiguration der Anfrage (z.B. HTTP-Methode, Header, Body). Wenn nicht angegeben, wird standardmäßig eine GET-Anfrage gesendet.

`Workspace()` gibt ein **Promise** zurück. Dieses Promise wird mit einem **`Response`-Objekt** erfüllt, sobald der Server mit den HTTP-Headern antwortet. Es ist wichtig zu verstehen, dass dies *nicht* bedeutet, dass der gesamte Antwort-Body bereits geladen wurde. [cite: 647, 648]

Das `Response`-Objekt: Das `Response`-Objekt enthält Informationen über die Antwort und Methoden, um den Antwort-Body zu verarbeiten:

- `ok` (Boolean): `true`, wenn der HTTP-Statuscode im Erfolgsbereich liegt (200-299).
- `status` (Number): Der HTTP-Statuscode (z.B. `200`, `404`).
- `statusText` (String): Die HTTP-Statusmeldung (z.B. `"OK"`, `"Not Found"`).
- `headers` (Headers-Objekt): Ermöglicht den Zugriff auf die Antwort-Header.
- Methoden zum Lesen des Bodys (diese geben *ihrerseits Promises* zurück!):
  - `json()`: Parst den Antwort-Body als JSON und gibt ein Promise zurück, das mit dem resultierenden JavaScript-Objekt erfüllt wird. [cite: 650, 651]
  - `text()`: Gibt den Antwort-Body als Text (String) zurück.
  - `blob()`: Gibt den Antwort-Body als `Blob`-Objekt (für Binärdaten) zurück.
  - `formData()`: Gibt den Antwort-Body als `FormData`-Objekt zurück.
  - `arrayBuffer()`: Gibt den Antwort-Body als `ArrayBuffer` zurück.

Beispiel 1: Einfache GET-Anfrage an JSONPlaceholder [cite: 644]

```
const todoApiUrl = '[https://jsonplaceholder.typicode.com/todos/1]
(https://jsonplaceholder.typicode.com/todos/1)';

console.log(`Workspaceing data from: ${todoApiUrl}`);
fetch(todoApiUrl) // Gibt ein Promise zurück
  .then(response => {
    // Der erste .then() Callback erhält das Response-Objekt
    console.log("Initial response received. Status:", response.status, response.statusText);
    console.log("Response OK?", response.ok);

    if (!response.ok) {
      // Wenn der HTTP-Status ein Fehler ist (z.B. 404, 500),
      // wirft dies einen Fehler, der vom .catch() abgefangen wird.
      throw new Error(`HTTP error! Status: ${response.status} - ${response.statusText}`);
    }
    // response.json() gibt ebenfalls ein Promise zurück, das mit den geparsten JSON-Daten
erfüllt wird.
    return response.json(); [cite: 651]
```

```
  })
  .then(todoData => {
    // Dieser .then() Callback erhält die geparsten JSON-Daten
    console.log("Todo data successfully parsed:");
    console.log(`  User ID: ${todoData.userId}`);
    console.log(`  ID: ${todoData.id}`);
    console.log(`  Title: ${todoData.title}`);
    console.log(`  Completed: ${todoData.completed}`);
  })
  .catch(networkErrorOrHttpError => {
    // Fängt sowohl Netzwerkfehler (fetch selbst schlägt fehl)
    // als auch manuell geworfene Fehler (z.B. bei !response.ok) ab.
    console.error("Error during fetch operation:", networkErrorOrHttpError.message);
  })
  .finally(() => {
    console.log("Fetch attempt for todo/1 finished.");
  });

// Beispiel für einen nicht existierenden Endpunkt (404)
const nonExistentUrl = '[https://jsonplaceholder.typicode.com/nonexistent/123]
(https://jsonplaceholder.typicode.com/nonexistent/123)';
fetch(nonExistentUrl)
  .then(response => {
    console.log(`\nResponse for non-existent URL. Status: ${response.status}`);
    if (!response.ok) {
      throw new Error(`Resource not found. Status: ${response.status}`);
    }
    return response.json(); // Wird nicht erreicht, wenn response.ok false ist.
  })
  .catch(error => {
    console.error("Error for non-existent URL:", error.message); // "Resource not found. Status:
404"
  });
```

Fehlerbehandlung bei `Workspace`: Ein wichtiger Unterschied zu manchen älteren HTTP-Clients (wie jQuery.ajax): `Workspace()` lehnt sein initiales Promise **nur bei Netzwerkfehlern** ab (z.B. Server nicht erreichbar, DNS-Problem). HTTP-Fehlerstatuscodes wie `404 Not Found` oder `500 Internal Server Error` führen **nicht** dazu, dass das `Workspace`-Promise abgelehnt wird. Stattdessen wird es mit dem `Response`-Objekt erfüllt, aber das `ok`-Property des `Response`-Objekts ist `false` und `status` enthält den Fehlercode. Man muss also `response.ok` oder `response.status` explizit prüfen und ggf. selbst einen Fehler werfen, um ihn im `.catch()`-Block zu behandeln.

Beispiel 2: POST-Anfrage mit `Workspace` an ReqRes.in (öffentliche Test-API)

```
const createUserUrl = '[https://reqres.in/api/users](https://reqres.in/api/users)';
const newUser = {
  name: "Neo",
  job: "The One"
};

console.log(`\nSending POST request to: ${createUserUrl}`);
fetch(createUserUrl, {
  method: 'POST', // HTTP-Methode
  headers: {
    'Content-Type': 'application/json' // Wichtig, um dem Server mitzuteilen, dass wir JSON
senden
  },
  body: JSON.stringify(newUser) // Das zu sendende Objekt als JSON-String
})
  .then(response => {
    console.log("POST Response Status:", response.status, response.statusText);
    if (!response.ok) { // Hier auch Status 201 (Created) als ok
      throw new Error(`POST request failed! Status: ${response.status}`);
    }
    return response.json(); // ReqRes.in sendet die erstellten Daten und eine ID zurück
```

```
  })
  .then(createdUserData => {
    console.log("User successfully created (simulated):");
    console.log(createdUserData);
    // Erwartete Ausgabe enthält typischerweise die gesendeten Daten + id und createdAt
    // z.B.: { name: "Neo", job: "The One", id: "...", createdAt: "..." }
  })
  .catch(error => {
    console.error("Error during POST request:", error.message);
  });
```

Die `Workspace` API ist sehr mächtig und flexibel und bildet die Grundlage für moderne Webkommunikation.

Übungsaufgaben zu Block B (Promises)

Aufgabe B.1: Promise für eine Zufallszahl Schreiben Sie eine Funktion `generateRandomNumberAfterDelay(maxDelayInSeconds)`. Diese Funktion soll ein Promise zurückgeben. Das Promise soll nach einer zufälligen Verzögerung (zwischen 0 und `maxDelayInSeconds` Sekunden) mit einer Zufallszahl zwischen 1 und 100 erfüllt werden. Konsumieren Sie dieses Promise und geben Sie die Zufallszahl auf der Konsole aus. Fügen Sie auch einen `.catch()`-Handler für den unwahrscheinlichen Fall hinzu, dass beim Erstellen des Promises ein Fehler auftritt (obwohl es hier primär um `resolve` geht).

Aufgabe B.2: Verkettete Textmanipulation mit Promises Erstellen Sie drei Funktionen, die jeweils ein Promise zurückgeben:

1. `convertToUppercase(text)`: Erfüllt nach 0.5 Sekunden mit dem `text` in Großbuchstaben.
2. `addExclamation(text)`: Erfüllt nach 0.3 Sekunden mit dem `text` und einem "!" am Ende.
3. `logProcessedText(text)`: Erfüllt nach 0.1 Sekunden, nachdem der `text` auf der Konsole ausgegeben wurde. Verketten Sie diese Promises, beginnend mit einem Starttext (z.B. "hallo welt"). Der Output von einer Funktion soll der Input für die nächste sein.

Aufgabe B.3: Fehlerbehandlung in einer Promise-Kette Modifizieren Sie `convertToUppercase` aus Aufgabe B.2 so, dass es `reject`et, wenn der übergebene `text` kein String ist (z.B. eine Zahl). Fügen Sie einen `.catch()`-Handler am Ende der Kette hinzu, der diesen spezifischen Fehler abfängt und eine passende Meldung ausgibt. Testen Sie sowohl den Erfolgs- als auch den Fehlerfall.

Aufgabe B.4: `Promise.all()` für parallele API-Aufrufe Verwenden Sie `Workspace` und `Promise.all()`, um gleichzeitig Daten von drei verschiedenen Endpunkten von JSONPlaceholder abzurufen:

- `/users/1`
- `/todos/1`
- `/posts/1` Wenn alle Anfragen erfolgreich sind, geben Sie ein Objekt auf der Konsole aus, das die Daten von allen drei Quellen kombiniert (z.B. `{ user: ..., todo: ..., post: ... }`). Fügen Sie eine Fehlerbehandlung hinzu.

Aufgabe B.5: `Promise.race()` für den schnellsten Server Simulieren Sie Anfragen an zwei verschiedene Server (oder Endpunkte) mit `Workspace`. Einer soll schneller antworten als der andere (z.B. durch unterschiedliche Endpunkte von `https://deelay.me/` oder `httpstat.us` oder indem Sie unterschiedliche lokale Server-Routen mit verschiedenen `sleep`-Zeiten erstellen). Verwenden Sie `Promise.race()` um zu ermitteln, welcher "Server" zuerst antwortet, und geben Sie dessen Antwort (oder zumindest eine Kennung des Servers) aus.

Aufgabe B.6: `Workspace` mit Fehlerprüfung und `.finally()` Schreiben Sie eine Funktion `WorkspaceJsonData(url)`, die `Workspace` verwendet, um JSON-Daten von einer gegebenen `url` abzurufen.

- Die Funktion soll das Promise mit den geparsten JSON-Daten erfüllen.
- Sie soll HTTP-Fehler (wenn `response.ok` `false` ist) korrekt behandeln, indem sie das Promise ablehnt.
- Verwenden Sie `.finally()`, um eine Nachricht wie `"Datenabruf von ${url} beendet."` auszugeben, unabhängig vom Erfolg der Anfrage. Testen Sie die Funktion mit einer gültigen URL (z.B. von JSONPlaceholder) und einer ungültigen URL, die einen 404-Fehler erzeugt.

Aufgabe B.7 (Bonus): `Promise.allSettled()` für optionale Datenquellen Angenommen, Sie möchten Profildaten eines Benutzers aus drei verschiedenen (optionalen) Quellen laden (z.B. `/profile/1`, `/preferences/1`, `/activity/1` von JSONPlaceholder, auch wenn diese Endpunkte nicht alle existieren). Verwenden Sie `Promise.allSettled()`, um alle Anfragen zu starten. Geben Sie nach Abschluss aller Anfragen für jede Quelle aus, ob sie erfolgreich war (und die Daten) oder fehlgeschlagen ist (und den Fehlergrund). So können Sie so viele Daten wie möglich anzeigen, auch wenn einzelne Quellen nicht verfügbar sind.

Block C: `async/await` für elegante Asynchronität

Obwohl Promises eine erhebliche Verbesserung gegenüber reinen Callbacks darstellen, kann das Verketten vieler `.then()`-Aufrufe bei komplexen asynchronen Abläufen immer noch zu Code führen, der nicht sofort intuitiv lesbar ist. ECMAScript 2017 (ES8) führte die Schlüsselwörter `async` und `await` ein, die eine noch klarere und oft einfacher zu verstehende Syntax für die Arbeit mit Promises bieten. `async/await` ist im Wesentlichen "syntaktischer Zucker" über Promises, d.h., es ändert nichts an der zugrundeliegenden Funktionsweise von Promises, macht aber deren Verwendung oft geradliniger. (Edube 4.3.5)

C.1 Die `async` Funktion

Das Schlüsselwort `async` wird verwendet, um eine Funktion als **asynchrone Funktion** zu deklarieren.

Syntax:

```
// Async function declaration
async function myFunctionDeclaration(param1, param2) {
  // ... code
  return 'some value'; // oder ein Promise
}

// Async function expression
const myFunctionExpression = async function(param1) {
  // ... code
};

// Async arrow function
const myArrowFunction = async (param1) => {
  // ... code
};

// Async method in an object literal
const myObject = {
  async myAsyncMethod() {
    // ... code
  }
};

// Async method in a class
class MyClass {
  async myClassAsyncMethod() {
    // ... code
  }
}
```

Wichtige Eigenschaften von `async` Funktionen:

1. **Implizite Rückgabe eines Promises:** Eine `async` Funktion gibt **immer** ein Promise zurück.
    - Wenn die `async` Funktion einen Wert mit `return wert;` zurückgibt, wird das von der `async` Funktion zurückgegebene Promise mit diesem `wert` **erfüllt** (`fulfilled`).
    - Wenn die `async` Funktion einen Fehler wirft (entweder explizit mit `throw new Error(...)` oder weil ein `await`-Ausdruck einen Fehler wirft, der nicht abgefangen wird), wird das von der `async` Funktion zurückgegebene Promise **abgelehnt** (`rejected`) mit dem geworfenen Fehler als Grund.
    - Wenn die `async` Funktion ein Promise zurückgibt (z.B. `return someOtherPromise;`), wird das implizit zurückgegebene Promise den Zustand dieses explizit zurückgegebenen Promises annehmen.

Beispiel 1: Einfache `async` Funktion

```
async function getGreetingMessage() {
  console.log("Async function: getGreetingMessage started.");
  // Simuliert eine kleine Verzögerung oder Operation
  const a = await Promise.resolve(1); // Minimales await Beispiel
```

```
  return "Hello from async function!"; // Dieser Wert wird der Erfüllungswert des Promises
}

const greetingPromise = getGreetingMessage();
console.log("Promise returned by async function:", greetingPromise); // Promise { <pending> }
(zunächst)

greetingPromise
  .then(message => {
    console.log("Async function SUCCESS:", message); // "Hello from async function!"
  })
  .catch(error => {
    console.error("Async function ERROR:", error);
  });

async function getFailedPromise() {
  console.log("Async function: getFailedPromise started.");
  throw new Error("Something went intentionally wrong in async function.");
}

getFailedPromise()
  .then(value => { // Wird nicht aufgerufen
    console.log("Async failure SUCCESS (should not happen):", value);
  })
  .catch(error => {
    console.error("Async failure ERROR (expected):", error.message); // "Something went
intentionally wrong..."
  });
```

C.2 Der `await` Operator

Das Schlüsselwort `await` kann **nur innerhalb einer `async` Funktion** verwendet werden (mit der Ausnahme von Top-Level `await` in ES-Modulen, was über den Rahmen dieses Grundkurses hinausgeht). `await` pausiert die Ausführung der `async` Funktion, bis das Promise, auf das es angewendet wird, `settled` ist (erfüllt oder abgelehnt).

Syntax:

```
// Innerhalb einer async function:
async function someAsyncOperation() {
  // ...
  const result = await somePromiseOrFunctionThatReturnsAPromise();
  // Code hier wird erst ausgeführt, nachdem somePromiseOrFunctionThatReturnsAPromise() erfüllt
ist.
  // 'result' enthält dann den Erfüllungswert des Promises.
  // Wenn das Promise abgelehnt wird, wirft 'await' einen Fehler.
  // ...
}
```

Verhalten von `await`:

- Wenn das Promise, auf das `await` angewendet wird, **erfüllt** (`fulfilled`) wird, gibt der `await`-Ausdruck den Erfüllungswert dieses Promises zurück. Die Ausführung der `async` Funktion wird dann fortgesetzt.
- Wenn das Promise **abgelehnt** (`rejected`) wird, wirft der `await`-Ausdruck den Ablehnungsgrund (den Fehler). Dieser geworfene Fehler kann dann mit einem `try...catch`-Block innerhalb der `async` Funktion abgefangen werden.

Beispiel 2: Verwendung von `await` mit der `WorkspaceDataFromServer`-Funktion aus Block B

```
// Annahme: fetchDataFromServer(succeeds) ist definiert wie in Block B
// function fetchDataFromServer(succeeds = true) {
//   return new Promise((resolve, reject) => { /* ... */ });
// }
```

```
async function processServerData() {
  console.log("processServerData: Initiating data fetch...");
  try {
    // Warte, bis fetchDataFromServer(true) erfüllt ist.
    // 'data' wird den Erfüllungswert des Promises enthalten.
    const data = await fetchDataFromServer(true);
    console.log("processServerData: Data successfully fetched and processed:", data);
    // Weitere Verarbeitung mit 'data'
    const processedContent = data.content.toUpperCase();
    console.log("processServerData: Processed content:", processedContent);
    return processedContent; // Das von processServerData zurückgegebene Promise wird hiermit
erfüllt.

  } catch (error) {
    console.error("processServerData: An error occurred during data fetching:", error.message);
    // Das von processServerData zurückgegebene Promise wird mit diesem Fehler abgelehnt.
    throw error; // Fehler weiterwerfen, wenn er auch außerhalb behandelt werden soll
  }
}

async function processFaultyServerData() {
  console.log("processFaultyServerData: Initiating faulty data fetch...");
  try {
    const data = await fetchDataFromServer(false); // Dieses Promise wird ablehnen
    console.log("processFaultyServerData: Data fetched (should not happen):", data);
  } catch (error) {
    console.error("processFaultyServerData: Caught expected error:", error.message);
    // Hier könnte man den Fehler spezifisch behandeln oder einen Default-Wert zurückgeben
    return "Default value due to error";
  }
}

// Aufruf der async Funktionen
processServerData()
  .then(result => console.log("Result from processServerData:", result))
  .catch(err => console.error("Error from processServerData (outer):", err));

setTimeout(() => { // Um die Ausgaben zu trennen
  processFaultyServerData()
    .then(result => console.log("Result from processFaultyServerData:", result)) // "Default
value due to error"
    .catch(err => console.error("Error from processFaultyServerData (outer):", err));
}, 5000); // Genug Zeit für die erste Operation
```

Der Code innerhalb von `processServerData` liest sich fast wie synchroner Code, obwohl `WorkspaceDataFromServer` asynchron ist.

C.3 Fehlerbehandlung mit `try...catch` in `async` Funktionen

Da `await` bei einem abgelehnten Promise einen Fehler wirft, ist der Standardmechanismus zur Fehlerbehandlung in `async` Funktionen der `try...catch`-Block, genau wie bei synchronem Code.

```
async function fetchDataWithTryCatch(url) {
  console.log(`\n[fetchDataWithTryCatch] Attempting to fetch: ${url}`);
  try {
    const response = await fetch(url); // await für das Promise von fetch()
    console.log(`[fetchDataWithTryCatch] Initial response status: ${response.status}`);

    if (!response.ok) {
      // Manuell einen Fehler werfen für HTTP-Fehlerstatus
      throw new Error(`HTTP error! Status: ${response.status} for ${url}`);
    }

    const data = await response.json(); // await für das Promise von response.json()
    console.log(`[fetchDataWithTryCatch] Data successfully fetched from ${url}:`, data.title ||
```

```
data.name || "No title/name");
    return data;

  } catch (error) {
    // Fängt sowohl Netzwerkfehler (wenn fetch selbst fehlschlägt)
    // als auch den oben manuell geworfenen HTTP-Fehler ab.
    console.error(`[fetchDataWithTryCatch] Error fetching ${url}: ${error.message}`);
    // Optional: Fehler weiterwerfen, wenn die aufrufende Funktion ihn behandeln soll
    // throw error;
    return null; // Oder einen Default-Wert/Fehlerindikator zurückgeben
  } finally {
    console.log(`[fetchDataWithTryCatch] Fetch attempt for ${url} finished.`);
  }
}

// Erfolgreicher Aufruf
fetchDataWithTryCatch('[https://jsonplaceholder.typicode.com/todos/1]
(https://jsonplaceholder.typicode.com/todos/1)');

// Aufruf, der einen HTTP-Fehler (404) erzeugt
setTimeout(() =>
fetchDataWithTryCatch('[https://jsonplaceholder.typicode.com/nonexistent/todos/999]
(https://jsonplaceholder.typicode.com/nonexistent/todos/999)'), 3000);

// Aufruf, der einen Netzwerkfehler erzeugen könnte (z.B. falsche Domain, kein Internet)
// setTimeout(() => fetchDataWithTryCatch('[https://nonexistentdomain123abc.com/api/data]
(https://nonexistentdomain123abc.com/api/data)'), 6000);
```

C.4 Parallele Ausführung mit `async/await` und `Promise.all()`

Wenn man `await` mehrmals hintereinander für unabhängige asynchrone Operationen verwendet, werden diese **sequenziell** ausgeführt – die nächste Operation startet erst, wenn die vorherige abgeschlossen ist.

```
async function fetchSequentially() {
  console.time("fetchSequentially");
  console.log("Fetching sequentially...");

  const user = await fetch('[https://jsonplaceholder.typicode.com/users/1').then(res]
(https://jsonplaceholder.typicode.com/users/1').then(res) => res.json());
  console.log("User fetched");

  const posts = await fetch('[https://jsonplaceholder.typicode.com/posts?userId=1').then(res]
(https://jsonplaceholder.typicode.com/posts?userId=1').then(res) => res.json());
  console.log("Posts fetched");

  const comments = await fetch('[https://jsonplaceholder.typicode.com/comments?
postId=1').then(res](https://jsonplaceholder.typicode.com/comments?postId=1').then(res) =>
res.json());
  console.log("Comments fetched");

  console.timeEnd("fetchSequentially"); // Zeigt die Gesamtzeit
  return { user: user.name, posts: posts.length, comments: comments.length };
}
// fetchSequentially().then(console.log);
```

Wenn die Operationen voneinander unabhängig sind, ist dies ineffizient. Um sie parallel auszuführen und auf den Abschluss aller zu warten, kombiniert man `async/await` mit `Promise.all()`.

```
async function fetchInParallel() {
  console.time("fetchInParallel");
  console.log("Fetching in parallel...");
```

```
  const userPromise = fetch('[https://jsonplaceholder.typicode.com/users/2').then(res]
(https://jsonplaceholder.typicode.com/users/2').then(res) => res.json());
  const postsPromise = fetch('[https://jsonplaceholder.typicode.com/posts?userId=2').then(res]
(https://jsonplaceholder.typicode.com/posts?userId=2').then(res) => res.json());
  const todosPromise = fetch('[https://jsonplaceholder.typicode.com/todos?
userId=2&completed=false').then(res](https://jsonplaceholder.typicode.com/todos?
userId=2&completed=false').then(res) => res.json());

  // Starte alle Anfragen (Promises werden erstellt), ohne sofort auf jede einzeln zu warten.
  // Dann warte auf alle mit Promise.all().
  try {
    const [userData, postsData, todosData] = await Promise.all([userPromise, postsPromise,
todosPromise]);

    console.log("All data fetched in parallel.");
    console.timeEnd("fetchInParallel");
    return { user: userData.name, posts: postsData.length, openTodos: todosData.length };
  } catch (error) {
    console.error("Error during parallel fetch:", error);
    console.timeEnd("fetchInParallel"); // Auch im Fehlerfall Zeit stoppen
    throw error;
  }
}

// fetchInParallel()
//  .then(results => console.log("Parallel results:", results))
//  .catch(err => console.error("Outer catch for parallel:", err));
```

In `WorkspaceInParallel` werden die `Workspace`-Aufrufe (die Promises zurückgeben) zuerst gestartet und ihre Promises in Variablen gespeichert. Dann wartet `await Promise.all([...])` darauf, dass alle diese gestarteten Operationen abgeschlossen sind. Dies ist deutlich schneller, wenn die Operationen unabhängig voneinander sind.

C.5 `async/await` mit der `Workspace` API – Vertiefende Beispiele (Edube 4.3.5)

Hier sind die Beispiele für GET und POST mit `Workspace` unter Verwendung von `async/await`, was den Code oft übersichtlicher macht als reine Promise-Ketten.

Beispiel: GET-Anfrage mit `async/await` (Vergleich Edube 4.3.5, `getSquare` mit `Workspace`)

```
async function getTodoDetails(todoId) {
  const apiUrl = `https://jsonplaceholder.typicode.com/todos/${todoId}`;
  console.log(`[getTodoDetails] Fetching todo with ID: ${todoId} from ${apiUrl}`);

  try {
    const response = await fetch(apiUrl); // Pausiert, bis die Header-Antwort da ist

    console.log(`[getTodoDetails] Response status: ${response.status}`);
    if (!response.ok) {
      throw new Error(`HTTP Error! Status: ${response.status}, URL: ${apiUrl}`);
    }

    const todoData = await response.json(); // Pausiert, bis der JSON-Body geparst ist
    console.log("[getTodoDetails] Successfully fetched and parsed todo:", todoData);
    return todoData;

  } catch (error) {
    console.error(`[getTodoDetails] Failed to fetch todo ${todoId}:`, error.message);
    throw error; // Damit der Aufrufer den Fehler auch behandeln kann
  }
}

// Aufruf
async function displayTodo() {
  try {
    const todo = await getTodoDetails(5);
```

```
    console.log(`Displayed Todo: ${todo.title} (Completed: ${todo.completed})`);

    const nonExistentTodo = await getTodoDetails(5000); // Sollte einen 404-Fehler auslösen
    if (nonExistentTodo) { // Wird nicht erreicht bei Fehler
        console.log(`Displayed Non-existent Todo: ${nonExistentTodo.title}`);
    }
  } catch(e) {
    console.error("Error in displayTodo flow:", e.message);
  }
}
// displayTodo();
```

Beispiel: POST-Anfrage mit `async/await`

```
async function createUserOnServer(userData) {
  const apiUrl = '[https://reqres.in/api/users](https://reqres.in/api/users)'; // Öffentliche
Test-API für POST
  console.log(`[createUserOnServer] Attempting to create user:`, userData);

  try {
    const response = await fetch(apiUrl, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(userData)
    });

    console.log(`[createUserOnServer] POST Response status: ${response.status}`);
    if (!response.ok) { // reqres.in gibt 201 für erfolgreiches Erstellen zurück
      throw new Error(`HTTP Error! Status: ${response.status} while creating user.`);
    }

    const createdUser = await response.json();
    console.log("[createUserOnServer] User successfully created on server (simulated):",
createdUser);
    return createdUser;

  } catch (error) {
    console.error("[createUserOnServer] Failed to create user:", error.message);
    throw error;
  }
}

// Aufruf
async function demonstrateUserCreation() {
  const newUser = { name: "Morpheus", job: "Leader" };
  try {
    const serverResponse = await createUserOnServer(newUser);
    console.log(`Server response for user creation: ID ${serverResponse.id}, CreatedAt
${serverResponse.createdAt}`);
  } catch (e) {
    console.error("Error in demonstrateUserCreation flow:", e.message);
  }
}
// demonstrateUserCreation();
```

Der Code mit `async/await` ist oft linearer und ähnelt synchronem Code, was die Lesbarkeit und das Verständnis asynchroner Abläufe erleichtert.

Übungsaufgaben zu Block C (`async/await`)

Aufgabe C.1: `async` Funktion für Benutzerdaten Schreiben Sie eine `async` Funktion `getUserProfile(userId)`. Diese Funktion soll `Workspace` verwenden, um Benutzerdaten von `https://jsonplaceholder.typicode.com/users/{userId}` abzurufen.

- Verwenden Sie `await` für den `Workspace`-Aufruf und für `response.json()`.
- Implementieren Sie eine Fehlerbehandlung mit `try...catch` für Netzwerkfehler und HTTP-Fehler.
- Die Funktion soll das Benutzerobjekt zurückgeben oder `null` im Fehlerfall. Testen Sie die Funktion, indem Sie das Ergebnis auf der Konsole ausgeben (nutzen Sie `.then()` und `.catch()` auf dem von `getUserProfile` zurückgegebenen Promise).

Aufgabe C.2: Mehrere API-Aufrufe mit `async/await` (sequenziell) Schreiben Sie eine `async` Funktion `getPostAndAuthor(postId)`.

1. Rufen Sie zuerst `https://jsonplaceholder.typicode.com/posts/{postId}` ab, um die Post-Daten zu erhalten.
2. Extrahieren Sie die `userId` aus den Post-Daten.
3. Rufen Sie dann `https://jsonplaceholder.typicode.com/users/{userId}` ab, um die Daten des Autors zu erhalten.
4. Die Funktion soll ein Objekt zurückgeben, das sowohl die Post-Daten als auch die Autor-Daten enthält (z.B. `{ post: ..., author: ... }`). Stellen Sie sicher, dass Sie `await` korrekt verwenden, um die sequenzielle Abhängigkeit zu handhaben, und implementieren Sie eine Fehlerbehandlung.

Aufgabe C.3: Parallele Datenbeschaffung mit `async/await` und `Promise.all()` Schreiben Sie eine `async` Funktion `getDashboardData(userId)`. Diese Funktion soll gleichzeitig (parallel) folgende Daten für einen Benutzer abrufen:

- Seine Todos: `https://jsonplaceholder.typicode.com/users/{userId}/todos`
- Seine Alben: `https://jsonplaceholder.typicode.com/users/{userId}/albums` Verwenden Sie `Promise.all()` zusammen mit `await`, um auf beide Ergebnisse zu warten. Die Funktion soll ein Objekt `{ todos: [...], albums: [...] }` zurückgeben. Fehlerbehandlung nicht vergessen.

Aufgabe C.4: Fehlerbehandlung und Rückgabewerte Erstellen Sie eine `async` Funktion `WorkspaceWithTimeout(url, timeoutMs)`. Diese Funktion soll versuchen, Daten von der `url` abzurufen.

- Wenn der Abruf länger als `timeoutMs` Millisekunden dauert, soll die Funktion das Promise mit einem `TimeoutError` ablehnen (simulieren Sie dies, indem Sie `Promise.race` mit `Workspace` und einem `setTimeout`-Promise, das nach `timeoutMs` ablehnt, verwenden).
- Wenn `Workspace` einen HTTP-Fehler zurückgibt (nicht `response.ok`), soll dieser Fehler weitergegeben werden.
- Implementieren Sie dies innerhalb einer `async` Funktion mit `try...catch`. Testen Sie die Funktion mit einer URL, die schnell antwortet, einer, die langsam antwortet (z.B. `https://deelay.me/` oder `httpstat.us`), und einer, die einen 404-Fehler erzeugt.

Aufgabe C.5: Refactoring von Promise-Ketten zu `async/await` Nehmen Sie das verkettete Promise-Beispiel aus Aufgabe B.2 (`convertToUppercase`, `addExclamation`, `logProcessedText`) und schreiben Sie es als eine einzige `async` Funktion um, die `await` verwendet, um die sequenziellen Schritte auszuführen.

Micro-Projekt: Utility-Funktionen - Phase 4 (Asynchrone Utility)

Wir erweitern unsere `textUtils.js` (oder erstellen eine neue `apiUtils.js`) um eine asynchrone Hilfsfunktion.

Aufgabe für das Micro-Projekt (Phase 4):

Entwickeln Sie eine `async` Funktion `WorkspaceJsonWithRetry(url, maxRetries = 3, delayMs = 1000)` in einer Datei `apiUtils.js`. Diese Funktion soll versuchen, JSON-Daten von der gegebenen `url` mit `Workspace` abzurufen.

- **Erfolgsfall:** Wenn `Workspace` erfolgreich ist (`response.ok` ist `true`) und die Antwort erfolgreich als JSON geparst werden kann, soll das Promise mit den geparsten Daten erfüllt werden.
- **Fehlerfall/Retry:**
  - Wenn `Workspace` einen Netzwerkfehler erzeugt oder `response.ok` `false` ist, soll die Funktion die Anfrage nach `delayMs` Millisekunden erneut versuchen.
  - Dies soll bis zu `maxRetries` Mal wiederholt werden.
  - Wenn nach `maxRetries` Versuchen immer noch ein Fehler auftritt, soll das von `WorkspaceJsonWithRetry` zurückgegebene Promise mit dem letzten aufgetretenen Fehler abgelehnt werden.
- Verwenden Sie `async/await` und `try...catch` für die Implementierung.
- Loggen Sie jeden Versuch und jeden Fehler auf der Konsole.

Beispielaufruf:

```
// In einer Testdatei:
// import { fetchJsonWithRetry } from './apiUtils.js';

async function testFetchWithRetry() {
  const validUrl = '[https://jsonplaceholder.typicode.com/todos/1]
(https://jsonplaceholder.typicode.com/todos/1)';
  const invalidUrl = '[https://jsonplaceholder.typicode.com/nonexistent/resource]
(https://jsonplaceholder.typicode.com/nonexistent/resource)';
  const temporarilyUnavailableUrl = 'http://localhost:3000/flaky-endpoint'; // ( hypothetical -
needs server setup to simulate)

  try {
    console.log("--- Testing valid URL ---");
    const data1 = await fetchJsonWithRetry(validUrl);
    console.log("Data from valid URL:", data1.title);
  } catch (e) {
    console.error("Error fetching valid URL (should not happen often):", e.message);
  }

  try {
    console.log("\n--- Testing invalid URL (expecting failure after retries) ---");
    const data2 = await fetchJsonWithRetry(invalidUrl, 2, 500); // 2 retries, 0.5s delay
    // Dieser Teil wird bei Fehler nicht erreicht
    if(data2) console.log("Data from invalid URL:", data2);
  } catch (e) {
    console.error("Expected error after retries for invalid URL:", e.message);
  }
}

// testFetchWithRetry();
```

Für `temporarilyUnavailableUrl` müssten Sie Ihren lokalen Testserver so anpassen, dass er z.B. die ersten paar Male mit einem 500er Fehler antwortet und dann mit 200. Das ist optional für die Übung, konzentrieren Sie sich auf die `Workspace` und Retry-Logik.