Promises, Async/Await, and Axios Requests in Node.js and Express

Estimated time needed: **20 minutes**,

Objectives

In this reading, you will learn about:

- **Promises**: The core building block for handling asynchronous operations in JavaScript
- **Async/Await**: A modern and more readable syntax for working with Promises
- **Axios**: A widely-used HTTP client for making network requests

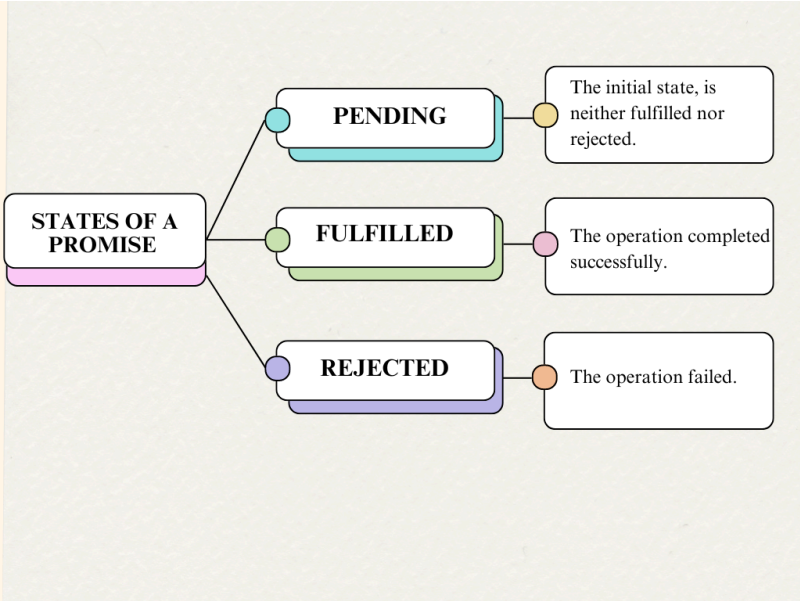
Introduction

JavaScript's asynchronous programming model is foundational for developing responsive and performant web applications. In Node.js and Express, handling asynchronous operations effectively is essential for tasks such as making API requests, reading files, or querying databases.

1. Promises

A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to chain operations in a more readable and manageable way.

1.1 States of a Promise



1.2 Creating a Promise

To create a Promise, you use the new Promise constructor, which takes a function with two parameters: resolve and reject. The resolve function is called when the asynchronous operation completes successfully, and the reject function is called when it fails.

```
// Creating a new Promise object and assigning it to the variable myPromise
const myPromise = new Promise((resolve, reject) => {
  // Simulating a condition with a boolean variable 'success'
  let success = true;
  // If the condition is true, call resolve to mark the promise as fulfilled
  if (success) {
    resolve("The operation was successful!");
  } else {
    // If the condition is false, call reject to mark the promise as rejected
    reject("The operation failed!");
  }
});
```

1.3 Using Promises with .then() and .catch()

You can handle the resolved value or the error using `.then()` and `.catch()` methods. These methods also return Promises, allowing you to chain multiple asynchronous operations in sequence.

```
// Execute the promise and handle the fulfilled and rejected states
```

```
myPromise
  // Handle the resolved state of the promise
  .then((message) => {
    // This block will execute if the promise is resolved
    console.log(message); // "The operation was successful!"
  })
  // Handle the rejected state of the promise
  .catch((error) => {
    // This block will execute if the promise is rejected
    console.error(error); // "The operation failed!"
  });
```

1.4 Example: Reading a file

Here's an example using the fs.promises module to read a file. The fs.promises module provides Promise-based methods for file system operations.

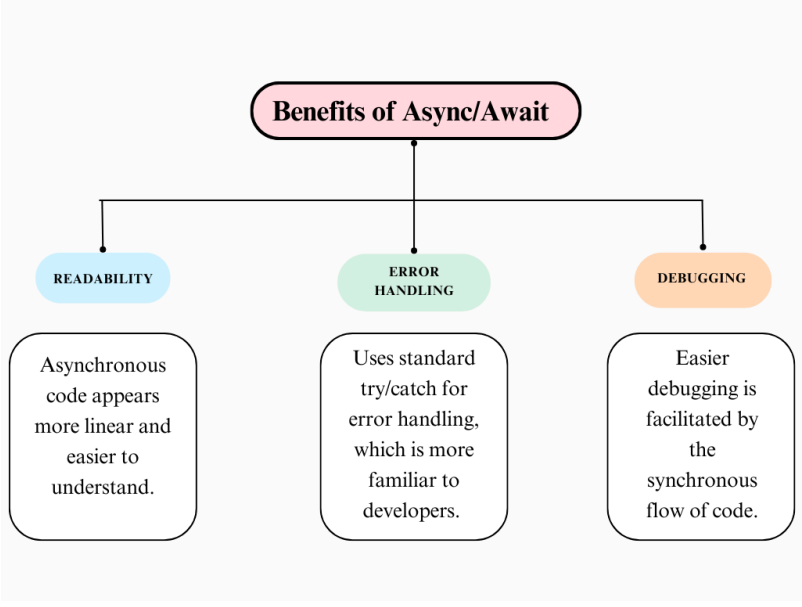
```
// Import the 'fs' module and use its promise-based methods
const fs = require('fs').promises;
// Read the content of the file 'example.txt' with 'utf8' encoding
fs.readFile('example.txt', 'utf8')
  // Handle the resolved state of the promise
  .then((data) => {
    // This block will execute if the file is read successfully
    console.log(data); // Print the file content to the console
  })
  // Handle the rejected state of the promise
  .catch((err) => {
    // This block will execute if there is an error reading the file
    console.error('Error reading file:', err); // Print the error message to the console
  });
```

2. Async/Await

As you might have already learnt, Java Script is a single-threaded scripting language. It means the process can happen only sequentially and no two processes can happen simultaneously. This is a big deterrent to any language and JS solved this by introducing asynchronous programming through Promises. While Promises solved the issues with synchronous programming, nested `then` can compilcate the structure and readability of the code.

Async and Await are syntactic sugar over Promises, making asynchronous code look more like synchronous code, which is easier to read and write. An async function returns a Promise, and you can use await inside an async function to pause execution until a Promise is resolved or rejected.

2.1 Benefits of Async/Await:



2.2 Using async and await

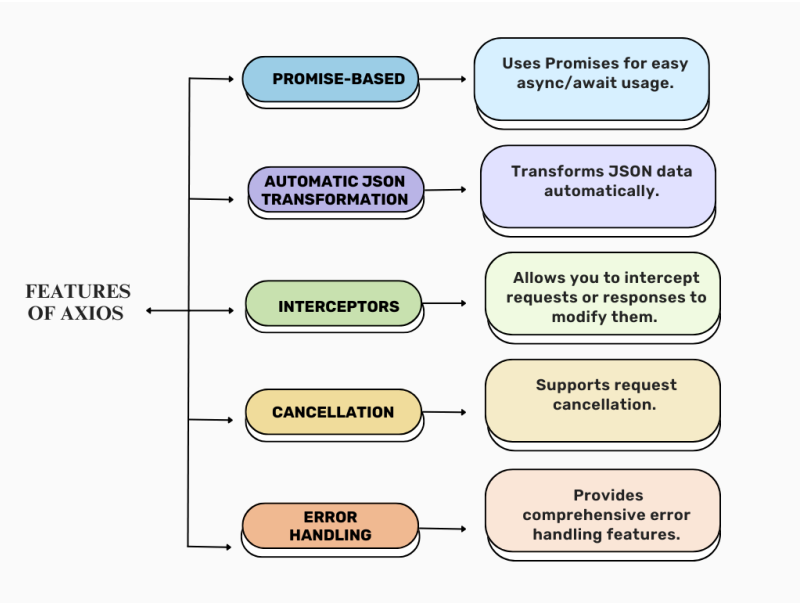
An async function always returns a Promise. Inside the async function, you can use the await keyword to pause execution until a Promise is resolved or rejected.

```
// Async function that wraps the operation
async function myAsyncFunction() {
  // Simulating a condition with a boolean variable 'success'
  let success = true;
  // If the condition is true, resolve with a success message
  if (success) {
    return "The operation was successful!";
  } else {
    // If the condition is false, throw an error to simulate rejection
    throw new Error("The operation failed!");
  }
}
// Using async function to handle Promise
async function executeAsyncFunction() {
  try {
    // Await the async function call to get the result
    const result = await myAsyncFunction();
    console.log(result); // Output the result if successful
  } catch (error) {
    console.error(error.message); // Handle and output any errors
  }
}
// Call the async function to execute
executeAsyncFunction();
```

This example shows how async and await can simplify asynchronous programming in JavaScript. The async function `myAsyncFunction` simulates a conditional operation, returning a success message if the condition is met and throwing an error otherwise. The `executeAsyncFunction` uses await to call `myAsyncFunction` and handles the result or any errors using try and catch. This approach makes the code easier to read and understand compared to handling promises with `.then()` and `.catch()` chains.

3. Axios Requests

Axios is a promise-based HTTP client for the browser and Node.js. It makes it easy to send asynchronous HTTP requests to REST endpoints and perform CRUD operations. Axios automatically transforms JSON data and provides a clean and simple API.



3.1 How Axios is Used

1. You can install Axios using the following command:

```
npm install axios
```

2. Making a GET Request

- Axios simplifies making HTTP GET requests. Here's how to fetch data from a public API:

```
// Import the axios library
const axios = require('axios');
```

```
// Using the axios.get method to make a GET request to the specified URL.
axios.get('https://api.example.com/data')
  // If the request is successful, the `.then` block is executed.
  .then(response => {
    // The response object contains the data returned from the server.
    // We log the `data` property of the response to the console.
    console.log(response.data);
  })
  // If there is an error during the request, the `.catch` block is executed.
  .catch(error => {

    // We log an error message to the console along with the error object.
    // This helps in debugging and understanding what went wrong with the request.

    console.error('Error fetching data:', error);
  });
```

- The axios.get method returns a Promise that resolves with the response object, allowing you to access the data with response.data.

3. Making a POST Request:

- To send data to a server, use a POST request:

```
// Import the axios library.
const axios = require('axios');
// Data to be sent in the POST request. This is a JavaScript object containing the user information.
const data = {
  name: 'John Doe',
  age: 30
};
// Using the axios.post method to make a POST request to the specified URL with the data object.
axios.post('https://api.example.com/users', data)

// If the request is successful, the `.then` block is executed.
  .then(response => {

// The response object contains the data returned from the server.
// We log a message along with the `data` property of the response to the console.

    console.log('User created:', response.data);
  })
  // If there is an error during the request, the `.catch` block is executed.

    .catch(error => {
    // We log an error message to the console along with the error object.
    // This helps in debugging and understanding what went wrong with the request.

    console.error('Error creating user:', error);
  });
```

- This code snippet demonstrates the basic usage of axios for making HTTP POST requests and handling responses and errors.

4. **Example: Using Async/Await with Axios**

Combining async/await with Axios provides a clean approach to handle HTTP requests:

```
const axios = require('axios'); // For Node.js, or include via CDN for browser
// Asynchronous function to post data to an API
async function postData() {
  try {
    // Await the response from the Axios POST request
    let response = await axios.post('https://jsonplaceholder.typicode.com/posts', {
      title: 'foo', // The title of the post
      body: 'bar',  // The body/content of the post
      userId: 1     // The user ID associated with the post
    });
    // Log the response data to the console
    console.log(response.data);
  } catch (error) {
    // If there is an error, log the error message to the console
    console.error('Error posting data:', error);
  }
}
// Call the async function to execute the request
postData();
```

Here, await pauses the function execution until the POST request completes, and the result is handled inside the try block.

Summary

In this reading, you learned about the fundamental concepts of asynchronous programming in Node.js and Express, focusing on Promises, async/await syntax, and the Axios HTTP client. Promises provide a way to handle asynchronous operations, making code more readable and easier to manage by allowing chaining of operations. The async/await syntax builds on Promises, enabling you to write asynchronous code that resembles synchronous code, thus simplifying the handling of asynchronous tasks. Finally, you explored how to use Axios to make HTTP requests, which is essential for interacting with external APIs and services. By mastering these concepts, you can write more efficient and maintainable asynchronous code, which is crucial for building robust web applications.

Author(s)

Rajashree Patil
Sapthashree K S