Promises

HyperionDev

Muhammad Zahir Junejo

Lecture - Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
  - Please review Code of Conduct (in Student Undertaking Agreement) if unsure
- No question is daft or silly - **ask them!**
- Q&A session at the end of the lesson, should you wish to ask any follow-up questions.
- Should you have any questions after the lecture, please schedule a mentor session.
- For all non-academic questions, please submit a query: www.hyperiondev.com/support

Lecture Objectives

1. What are Promises?
2. Creating Promises
3. Resolving and Rejecting Promises
4. Using the async-await syntax

Introduction

- Welcome to the world of asynchronous programming!
- In web development, many tasks take time to complete, like fetching data from a server. Asynchronous programming allows us to execute code while waiting for these tasks to finish.
- We'll explore Promises, a powerful tool for managing asynchronous operations in JavaScript.

What are Promises?

- Promises are JavaScript objects representing the eventual completion or failure of an asynchronous operation.
- They help us work with asynchronous code more effectively, making it easier to manage complexity.
- Think of Promises as a contract: a promise that a result will be available in the future.

Benefits of Promises

- Promises offer several benefits:
  - Improved Code Readability: They make asynchronous code look more like synchronous code, making it easier to understand.
  - Better Error Handling: Promises have built-in mechanisms for handling errors, reducing the risk of unhandled exceptions.
  - Avoid Callback Hell: Promises help eliminate callback nesting, making your code cleaner and more maintainable.

Anatomy of a Promise

- Promises have three states:
  - Pending: Initial state, neither resolved nor rejected.
  - Resolved: The asynchronous operation completed successfully, and a result is available.
  - Rejected: An error occurred during the operation.
- Promises transition from pending to either resolved or rejected.

Creating Promises

- To create a Promise, use the new Promise() constructor, which takes a function as its argument.
- This function, known as the executor, contains the asynchronous operation.
- The executor receives two functions as parameters: resolve and reject, which are used to signal the Promise's outcome.

```
const myPromise = new Promise((resolve, reject) => {
  // Simulate an async task
  setTimeout(() => {
    const result = 42;
    resolve(result); // Promise resolved with the result
  }, 1000);
});
```

Resolving Promises

- Resolving a Promise means that the asynchronous operation succeeded.
- We handle resolved Promises using the .then() method, which takes a callback function.
- The callback function receives the result of the resolved Promise.

```
myPromise.then((result) => {
  console.log(`Promise resolved with result: ${result}`);
});
```

Rejecting Promises

- Rejecting a Promise indicates that an error occurred during the asynchronous operation.
- We handle rejected Promises using the .catch() method, which also takes a callback function.
- The callback function receives the error information.

```
const errorPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    const error = new Error("Something went wrong");
    reject(error); // Promise rejected with an error
  }, 1000);
});
errorPromise.catch((error) => {
  console.error(`Promise rejected with error: ${error.message}`);
});
```

Chaining Promises

- Promises can be chained together for sequential asynchronous operations.
- Chaining helps avoid callback hell.
- Each .then() returns a new Promise, allowing you to chain more operations.

```
fetchData()
  .then(processData)
  .then(displayData)
  .catch(handleError);
```

Async-Await Syntax

- Async-await is a more recent way to work with Promises, providing a cleaner and more readable syntax.
- It's built on top of Promises, making asynchronous code look synchronous.

```
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(`Error fetching data: ${error.message}`);
  }
}
```

Using async-await

```
async function example() {
  try {
    const result = await someAsyncFunction();
    console.log(`Async result: ${result}`);
  } catch (error) {
    console.error(`Error: ${error.message}`);
  }
}
```

Promises in the Real-world

- Promises and async-await are used for:
  - Making API requests
  - Handling user interactions
  - Loading resources in a web application

Questions and Answers

Thank You!