JavaScript-Promises

Promises in JavaScript: A Summary

In JavaScript, `Promise` is a mechanism for handling asynchronous operations. It allows you to write code that can be executed when the operation is complete, without blocking the execution of other parts of your code.

Why Promises?

Before promises, developers used callbacks to handle asynchronous operations. However, this approach had several limitations:

- Callback hell: nested callbacks made the code hard to read and maintain
- Error handling: errors were not propagated properly, making it difficult to debug

Promises address these issues by providing a more elegant way to handle asynchronous operations.

Basic Promise API

The basic promise API consists of three methods:

1. `Promise`: The constructor for creating a new promise.
2. `then()`: A method that takes two arguments: `onFulfilled` and `onRejected`. It's called when the promise is resolved or rejected, respectively.
3. `catch()`: A method that handles rejection by providing an alternative value.

Example

Here's a simple example of creating a promise:

```
 // Create a new promise
const promise = new Promise((resolve, reject) => {
  // Simulate an asynchronous operation (e.g., fetching data from API)
  setTimeout(() => {
    resolve("Data received"); // Resolve the promise with data
  }, 2000); // Delay by 2 seconds

  // Reject the promise if something goes wrong
  reject("Error occurred");
});

// Handle resolution and rejection
promise.then((data) => {
  console.log(data); // Log the received data
}).catch((error) => {
  console.error(error); // Log any errors that occur
});
```

In this example:

- We create a new promise using `new Promise`.
- Inside the constructor, we simulate an asynchronous operation by calling `setTimeout` to resolve the promise with some data after 2 seconds.
- If anything goes wrong, we reject the promise with an error message.
- We then use `then()` and `catch()` to handle resolution and rejection, respectively.

Chaining Promises

Promises can be chained together using the `.then()` method. This allows you to perform multiple asynchronous operations in sequence:

```
 const promise = new Promise((resolve) => {
  setTimeout(() => resolve("Data received"), 2000);
});

promise.then((data) => {
  console.log(data); // Log the received data

  const nestedPromise = new Promise((resolve) => {
    setTimeout(() => resolve("Nested data received"), 3000);
  });

  return nestedPromise;
})
.then((nestedData) => {
  console.log(nestedData); // Log the nested data
})
.catch((error) => {
  console.error(error); // Log any errors that occur
});
```

In this example, we chain two promises together. The first promise resolves with some data after 2 seconds, and then we return a new promise that resolves with more data after 3 seconds.

Conclusion

Promises provide a powerful way to handle asynchronous operations in JavaScript. By using the `Promise` constructor, you can create promises that resolve or reject when an operation is complete. You can chain multiple promises together to perform complex asynchronous workflows. With this knowledge, you'll be able to write more efficient and maintainable code for handling async operations in your applications.

Curated by Brajesh Kumar