Power of Promises: A Deep Dive into Asynchronous JavaScript

Are you new to JavaScript ? Wondering how Promises work in JavaScript ? This presentation will provide all the answers you need!

by Gourav Roy

What is Promise in JavaScript

In JavaScript, a Promise is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value.

```
// Example: Creating a Promise for a simulated asynchronous
const fetchData = (isSuccess) => {
  return new Promise((resolve, reject) => {
    // Simulating an asynchronous operation (e.g., fetching
    setTimeout(() => {
      if (isSuccess) {
        // Resolve the promise with a value if the operatior
        resolve("Data fetched successfully!");
      } else {
        // Reject the promise with an error message if the c
        reject("Error: Unable to fetch data");
      }
    }, 2000); // Simulating a 2-second delay
  });
};
```

How to Create a Promise in JavaScript

In JavaScript, creating a Promise involves wrapping an asynchronous operation inside the `Promise` constructor. This constructor takes a function with two parameters: `resolve` and `reject`. The `resolve` function is used to fulfill the promise with a successful result, while the `reject` function is used to reject the promise with an error.

```
// Example: Creating a Promise for a simulated asynchronous
const fetchData = (isSuccess) => {
  return new Promise((resolve, reject) => {
    // Simulating an asynchronous operation (e.g., fetching
    setTimeout(() => {
      if (isSuccess) {
        // Resolve the promise with a value if the operatior
        resolve("Data fetched successfully!");
      } else {
        // Reject the promise with an error message if the c
        reject("Error: Unable to fetch data");
      }
    }, 2000); // Simulating a 2-second delay
  });
};
```

Async/Await in JavaScript

In JavaScript, `async/await` is a syntactic sugar on top of Promises, offering a more concise and readable way to work with asynchronous code. It allows developers to write asynchronous code that looks and behaves more like synchronous code, making it easier to reason about.

```
// Example function using async/await
async function fetchData() {
  try {
    // Simulating an asynchronous operation (e.g., fetching data)
    const data = await fetchDataFromAPI(); // Assuming fetchDataFr

    // Process the data once the Promise is resolved
    console.log("Data fetched successfully:", data);

    // Further synchronous operations
    const processedData = processData(data);
    console.log("Processed data:", processedData);

    return processedData; // The value will be wrapped in a resolv
  } catch (error) {
    // Handle errors if the Promise is rejected
    console.error("Error fetching data:", error);

    // Optionally, rethrow the error or handle it gracefully
    throw error;
```

Error Handling With Async/Await

Handling errors in `async/await` involves using a combination of `try`, `catch`, and optionally, `finally` blocks. The `try` block contains the code that might throw an error, and the `catch` block handles any errors that occur.

```
// Example function using async/await with error handling
async function fetchData() {
  try {
    // Simulating an asynchronous operation (e.g., fetching data)
    const data = await fetchDataFromAPI(); // Assuming fetchDataF

    // Process the data once the Promise is resolved
    console.log("Data fetched successfully:", data);

    // Simulating an error during further processing
    throw new Error("Simulated error during processing");
  } catch (error) {
    // Handle errors if the Promise is rejected or if an error oc
    console.error("Error during fetchData:", error);

    // Optionally, rethrow the error or handle it gracefully
    throw error;
  } finally {
    // Code in this block will be executed regardless of whether
    console.log("Cleanup or finalization code");
```

Promise Chaining in JavaScript

Promise chaining in JavaScript is a technique that allows you to execute multiple asynchronous operations sequentially, ensuring that one operation completes before the next one starts.

```
// Chaining Promises
fetchData()
  .then((result) => {
    // This block executes when the first Promise resolves successf
    console.log(result);
    // Returning a new Promise for the next asynchronous operation
    return result.toUpperCase();
  })
  .then((processedData) => {
    // This block executes when the second Promise resolves success
    console.log("Processed Data:", processedData);
    // Returning a new Promise or a value for the next step
    return `${processedData} - Processed Again`;
  })
  .then((finalResult) => {
    // This block executes when the third Promise resolves successf
    console.log("Final Result:", finalResult);
  })
  .catch((error) => {
    // This block executes if any of the Promises in the chain reje
    console.error("Error:", error);
```

Promise.all() in JavaScript

`Promise.all` is a powerful utility in JavaScript that allows you to efficiently handle multiple asynchronous operations concurrently.

It takes an array of Promises as input and returns a new Promise that resolves with an array of the resolved values when all input Promises are resolved.If any of the input Promises reject, the entire `Promise.all` is rejected.

```
// Using Promise.all to fetch data from multiple APIs concurrently
Promise.all([
  fetchDataFromAPI1(),
  fetchDataFromAPI2(),
  fetchDataFromAPI3(),
])
  .then((results) => {
    // This block executes when all Promises resolve successfully
    console.log("Data from APIs:", results);
  })
  .catch((error) => {
    // This block executes if any of the Promises in Promise.all reject
    console.error("Error fetching data:", error);
  });
```

Promise.race() in JavaScript

`Promise.race` is a method in JavaScript that takes an iterable of Promises and returns a new Promise that is settled (fulfilled or rejected) as soon as one of the input Promises is settled.

```
const promise1 = fetchDataWithDelay("Data from Source 1", 2000);
const promise2 = fetchDataWithDelay("Data from Source 2", 1000);
const promise3 = fetchDataWithDelay("Data from Source 3", 3000);

// Using Promise.race to respond to the first settled Promise
Promise.race([promise1, promise2, promise3])
  .then((result) => {
    console.log("First Promise Resolved:", result);
    // Further processing or actions based on the fastest result
  })
  .catch((error) => {
    console.error("First Promise Rejected:", error);
    // Handling errors if the first Promise is rejected
  });
```

Implement Retry Logic Using Promises

Implementing a retry logic using Promises involves creating a function that attempts an asynchronous operation and, in case of failure, retries it after a specified delay. This can be achieved with a concise code snippet:

```
function retryOperation(operation, maxAttempts, delay) {
  return new Promise((resolve, reject) => {
    const attempt = (currentAttempt) => {
      operation()
        .then(resolve)
        .catch((error) => {
          if (currentAttempt < maxAttempts) {
            console.log(`Attempt ${currentAttempt} failed. Retrying in
            setTimeout(() => attempt(currentAttempt + 1), delay);
          } else {
            console.log(`Max attempts reached. Operation failed.`);
            reject(error);
          }
        });
    };

    attempt(1); // Start the initial attempt
  });
}
```