JavaScript Promise finally()

Summary: in this tutorial, you will learn how to use the JavaScript Promise `finally()` method to execute the code once the promise is settled, regardless of its outcome.

Introduction to the JavaScript Promise finally() method

The `finally()` method of a Promise instance allows you to schedule a function to be executed when the promise is settled.

Here's the syntax for calling the `finally()` method:

```
promise.finally(onFinally)
```

In this syntax:

- `onFinally` is a function that executes asynchronously when the promise becomes settled.

The finally() method returns a Promise object, allowing you to conveniently chain calls to other methods of the Promise instance.

The `finally()` method was introduced in ES2018. With the `finally()` method, you can place the code that cleans up resources when the promise is settled, regardless of its outcome.

By using the `finally()` method, you can avoid duplicate code in the `then()` and `catch()` methods like this:

```
promise
    .then(result => {
        // process the result
        // clean up the resources
    })
    .catch(error => {
```

```
        // handle the error
        // clean up the resources
    });
```

Now, you can move the clean up the resources part to the `finally()` method as follows:

```
promise
    .then(result => {
        // process the result
    })
    .catch(error => {
        // handle the error
    })
    .finally(() => {
        // clean up the resources
    });
```

The `finally()` method is similar to the `finally` block in the `try...catch...finally` statement.

In synchronous code, you use the `finally` block to clean up the resources. In asynchronous code, you use the `finally()` method instead.

The JavaScript Promise finally() method examples

Let's take some examples of using the Promise `finally()` method.

1) Using the finally() method to clean up resources

The following defines a `Connection` class:

```
class Connection {
    execute(query) {
        if (query != 'Insert' && query != 'Update' && query != 'Delete') {
            throw new Error(`The ${query} is not supported`);
        }
        console.log(`Execute the ${query}`);
        return this;
    }
```

```
    close() {
        console.log('Close the connection')
    }
}
```

The `Connection` class has two methods: `execute()` and `close()`:

- The `execute()` method will only execute the insert, update, or delete query. It will issue an error if you pass into another query that is not in the list.
- The `close()` method closes the connection and cleans up the resource.

The following `connect()` function returns a promise that resolves to a new `Connection` if the success flag is set to true:

```
const success = true;

function connect() {
    return new Promise((resolve, reject) => {
        if (success)
            resolve(new Connection());
        else
            reject('Could not open the database connection');
    });
}
```

The following example uses the `finally()` method to close the connection:

```
let globalConnection;

connect()
    .then((connection) => {
        globalConnection = connection;
        return globalConnection.execute('Insert');
    })
    .then((connection) => {
        globalConnection = connection;
        return connection.execute('Select');
```

```
    })
    .catch(console.log)
    .finally(() => {
        if (globalConnection) {
            globalConnection.`close()`;
        }
    });
```

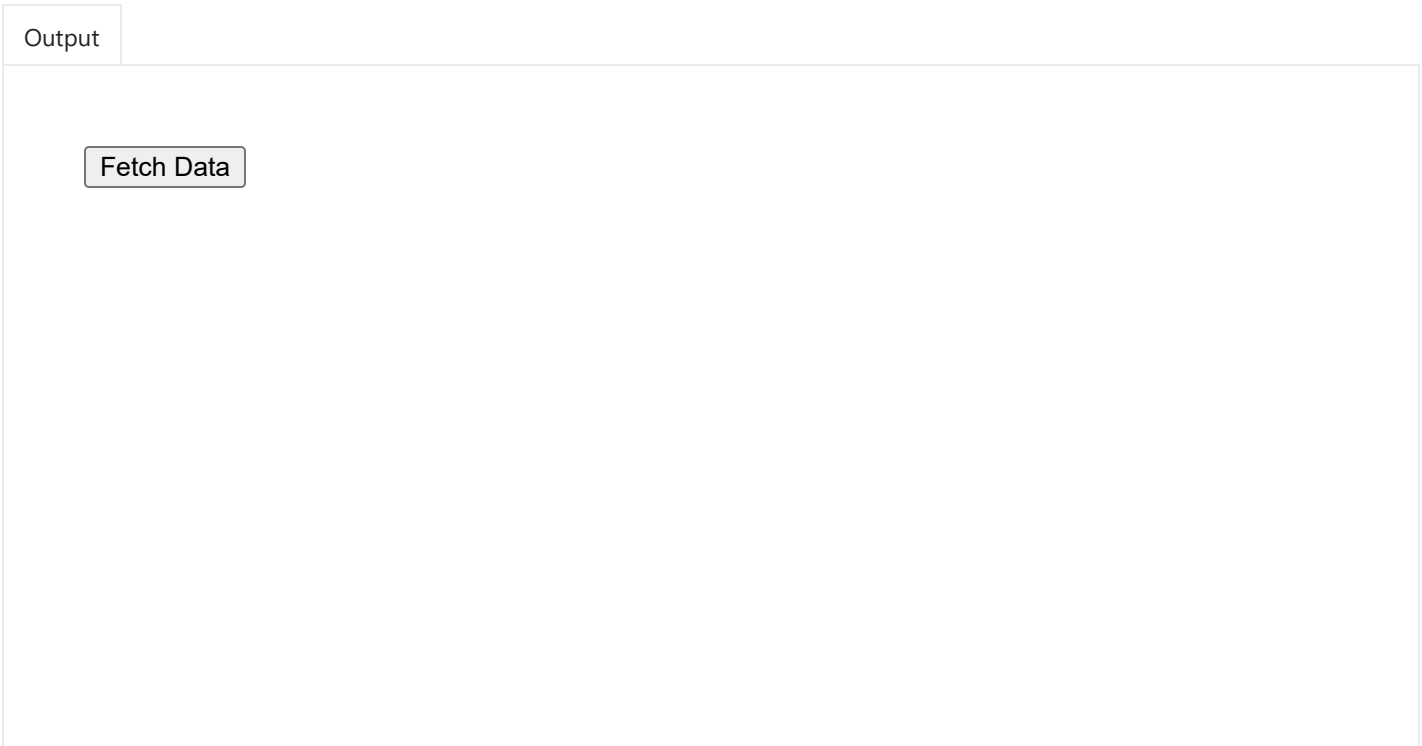
In this example:

- The `connect()` function resolves to a new `Connection` object because the `success` flag is set to `true`.
- The first `then()` method executes the `Insert` query and returns a `Connection` object. The `globalConnection` is used to save the connection.
- The second `then()` method executes the `Select` query and issues an error. The `catch()` method shows the error message and the `finally()` method closes the connection.

2) Using the Promsie finally() method show a loading status

The following example shows how to use the `finally()` method to hide the loading element after calling the public API https://jsonplaceholder.typicode.com/posts.

Output



index.html

```
<!DOCTYPE html>
<html lang="en">

    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>JavaScript finally - API Call with Loading State</title>
        <script src="app.js" defer></script>
    </head>

    <body>
        <button id="fetchButton">Fetch Data</button>
        <div id="loading" style="display: none">Loading...</div>
        <div id="content" style="display: none"></div>
    </body>

</html>
```

app.js

```
document.getElementById('fetchButton').addEventListener('click', () => {
  const loadingElement = document.getElementById('loading');
  const contentElement = document.getElementById('content');

  // Show loading and hide content
  loadingElement.style.display = 'block';
  contentElement.style.display = 'none';

  // Make the API call to get posts
  fetch('https://jsonplaceholder.typicode.com/posts')
    .then((response) => response.json())
    .then((posts) => {
      // Render the posts
      const renderedPosts = posts
        .map((post) => {
          return `
```

```
            <h1>${post.title}</h1>
            <p>${post.body}</p>
            `;
        })
        .join('');

      // Show the posts
      contentElement.innerHTML = renderedPosts;
    })
    .catch((error) => {
      // Handle any errors
      contentElement.innerHTML = `<p>Failed to load data</p>`;
    })
    .finally(() => {
      // Hide loading and show content
      loadingElement.style.display = 'none';
      contentElement.style.display = 'block';
    });
});
```

How it works.

First, add a click event handler to the button:

```
document.getElementById('fetchButton').addEventListener('click', () => {
   // ...
});
```

Second, show the loading element and hide the content element:

```
const loadingElement = document.getElementById('loading');
const contentElement = document.getElementById('content');

 // Show loading and hide content
loadingElement.style.display = 'block';
contentElement.style.display = 'none';
```

Third, call an API using the Fetch API and render the posts:

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then((response) => response.json())
  .then((posts) => {
    // Render the posts
    const reenderedPosts = posts
      .map((post) => {
        return `
            <h1>${post.title}</h1>
            <p>${post.body}</p>
            `;
      })
      .join('');

    // Show the posts
    contentElement.innerHTML = reenderedPosts;
  })
  .catch((error) => {
    // Handle any errors
    contentElement.innerHTML = `<p>Failed to load data</p>`;
  })
  .finally(() => {
    // Hide loading and show content
    loadingElement.style.display = 'none';
    contentElement.style.display = 'block';
  });
```

In the `finally()` method, hide the loading element and show the content element.

Summary

- The `finally()` method schedule a function to execute when the promise is settled, either fulfilled or rejected.
- It's good practice to place the code that cleans up the resources in the `finally()` method once the promise is settled, regardless of its outcome.

Quiz