W02 Learning Activity: JavaScript async/await

Overview

Imagine you're ordering food online. You click "order," and then... you wait. You can't really do much else related to that order until the food arrives, right? That's kind of how traditional synchronous JavaScript code can feel when dealing with things that take time, like fetching data. You send a request, and then you just sit there waiting for a response. It's like waiting for your food to be delivered.

But what if you could keep doing other things while waiting for that food? That's where async/await comes in. It lets you write code that looks like it's synchronous (like you're waiting for your food), but under the hood, it's actually asynchronous (like you're multitasking while waiting).

In this learning activity you will learn about async and await, then in a separate learning activity on the Fetch API you will practice using them.

Prepare

`Async/await` is the modern way of handling asynchronous operations. It allows you to write asynchronous code that looks and behaves like synchronous code, which makes it easy to read and maintain.

> In the past developers used callback functions and promises to handle asynchronous code, but the async/await approach is easier to maintain and is the preferred method, and it is the approach you should use in this course.

Promises

Async/await is built on top of promises, so you need to understand how promises work before diving into async/await. A promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. When you create a promise, you can attach callbacks to handle the success or failure of that operation.

Here's a simple example of a promise:

```
const myPromise = new Promise((resolve, reject) => {
  const success = true; // Simulate success or failure
  if (success) {
    resolve("Operation was successful!");
  } else {
    reject("Operation failed.");
  }
});
```

In this example, a new promise is created that simulates an asynchronous operation. If the operation is successful, the `resolve` function is called with a success message. If it fails, the `reject` function is called with an error message.

To handle the result of a promise, you can use the `.then()` and `.catch()` methods:

```
myPromise
  .then((result) => {
    console.log(result); // Output: "Operation was successful!"
  })
  .catch((error) => {
    console.error(error); // Output: "Operation failed."
  });
```

In this example, if the promise resolves successfully, the success message is logged to the console. If it fails, the error message is logged instead.

Async/Await

Now, let's see how async/await can simplify this process. The `async` keyword is used to define an asynchronous function, and the `await` keyword is used to pause the execution of that function until a promise is resolved or rejected.

Here's how you can rewrite the previous example using async/await:

```
const myAsyncFunction = async () => {
  try {
    const result = await myPromise; // Wait for the promise to resolve
    console.log(result); // Output: "Operation was successful!"
  } catch (error) {
    console.error(error); // Output: "Operation failed."
  }
};
```

In this example, the `myAsyncFunction` is defined as an asynchronous function using the `async` keyword. Inside the function, the `await` keyword is used to pause execution until the `myPromise` is resolved or rejected. If the promise resolves successfully, the result is logged to the console. If it fails, the error is caught and logged.

Async/await makes your code look more like synchronous code, which can be easier to read and understand. It also helps you avoid "callback infierno," where you have nested callbacks that can make your code messy and hard to follow.

Here's a simple example of using async/await to fetch data from an API:

```
const fetchData = async () => {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/posts,
    const data = await response.json(); // Wait for the response to be conve
    console.log(data); // Output the fetched data
  } catch (error) {
    console.error("Error fetching data:", error); // Handle any errors
  }
};
```

In this example, the **`fetchData`** function is defined as an asynchronous function. Inside the function, the **`await`** keyword is used to pause execution until the fetch request is completed and the response is converted to JSON. If any errors occur during this process, they are caught and logged.

Async/await is a powerful tool for handling asynchronous operations in JavaScript. It allows you to write cleaner and more readable code while still taking advantage of the benefits of asynchronous programming.

Learning Activity

You will have a chance to apply **`async/await`** in the next learning activity, **Fetch API** where you will **`fetch`** latter-day prophet data from a JSON file and process that data to build a page when it becomes available.

Optional Resources

- Asynchronous JavaScript – MDN
- async function – MDN
- await – MDN
- try...catch block – MDN

Back