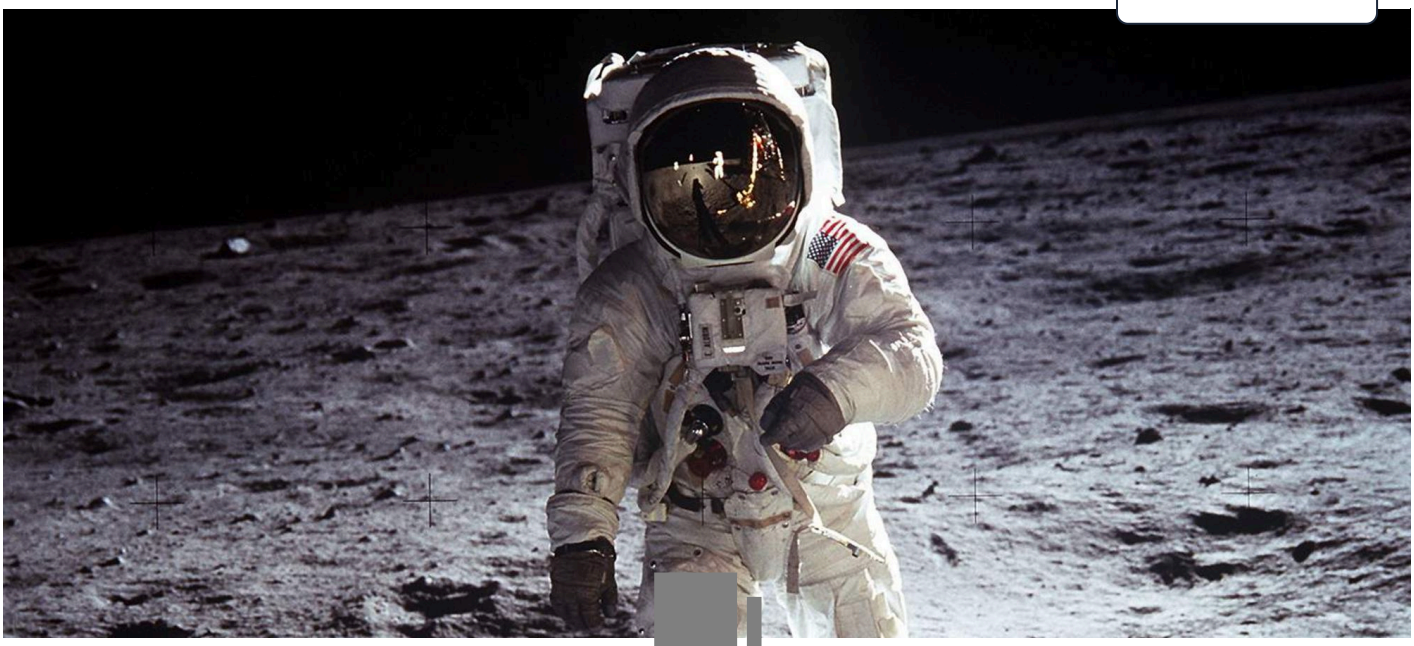
Cohort 2.0 FullStack Open Source / Week 8 / Week 8.3

Download



Week 8.3

Axios vs Fetch (Offline)

In this quick offline tutorial, Harkirat covers the `Fetch method` and `Axios` for data fetching in web development. He explains the differences between them and provides straightforward code examples for both, making it easy to grasp the essentials of these commonly used techniques.

fetch() Method

The `fetch()` method in JavaScript is a modern API that allows you to make network requests, typically to retrieve data from a server. It is commonly used to interact with web APIs and fetch data asynchronously. Here's a breakdown of what the `fetch()` method is and why it's used:

What is the `fetch()` Method?

The `fetch()` method is a built-in JavaScript function that simplifies making HTTP requests. It returns a Promise that resolves to the `Response` to that request, whether it is successful or not.

Why is it Used?

1. **Asynchronous Data Retrieval:**
    - The primary use of the `fetch()` method is to asynchronously retrieve data from a server. Asynchronous means that the code doesn't wait for the data to arrive before moving on. This is crucial for creating responsive and dynamic web applications.
2. **Web API Interaction:**
    - Many web applications interact with external services or APIs to fetch data. The `fetch()` method simplifies the process of making HTTP requests to these APIs.
3. **Promise-Based:**
    - The `fetch()` method returns a Promise, making it easy to work with asynchronous operations using the `.then()` and `.catch()` methods. This promotes cleaner and more readable code.
4. **Flexible and Powerful:**
    - `fetch()` is more flexible and powerful compared to older methods like `XMLHttpRequest`. It supports a wide range of options, including headers, request

methods, and handling different types of responses (JSON, text, etc.).

Basic Example:

Here's a basic example of using the `fetch()` method to retrieve data from a server:

```
fetch('<https://api.example.com/data>')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json();
  })
  .then(data => {
    console.log('Data from server:', data);
  })
  .catch(error => {
    console.error('Fetch error:', error);
  });
```

In this example, we use `fetch()` to make a GET request to 'https://api.example.com/data', handle the response, and then parse the JSON data. The `.then()` and `.catch()` methods allow us to handle the asynchronous flow and potential errors.

fetch() vs axios()

Fetch API

1. **Native Browser API:**
    - `fetch` is a native JavaScript function built into modern browsers for making HTTP requests.
2. **Promise-Based:**
    - It returns a Promise, allowing for a more modern asynchronous coding style with `async/await` or using `.then()`.
3. **Lightweight:**
    - `fetch` is lightweight and comes bundled with browsers, reducing the need for external dependencies.

Example Usage:

```
fetch('<https://api.example.com/data>')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Axios

1. **External Library:**
    - Axios is a standalone JavaScript library designed to work in both browsers and Node.js environments.
2. **Promise-Based:**
    - Similar to `fetch`, Axios also returns a Promise, providing a consistent interface for handling asynchronous operations.
3. **HTTP Request and Response Interceptors:**
    - Axios allows the use of interceptors, enabling the modification of requests or responses globally before they are handled by `then` or `catch`.
4. **Automatic JSON Parsing:**
    - Axios automatically parses JSON responses, simplifying the process compared to `fetch`.

Example Usage:

```
import axios from 'axios';

axios.get('<https://api.example.com/data>')
  .then(response => console.log(response.data))
  .catch(error => console.error('Error:', error));
```

Comparison Points

1. **Syntax:**
    - `fetch` uses a chain of `.then()` to handle responses, which might lead to a more verbose syntax. Axios, on the other hand, provides a concise syntax with `.then()` directly on the Axios instance.
2. **Handling HTTP Errors:**
    - Both `fetch` and Axios allow error handling using `.catch()` or `.finally()`, but Axios may provide more detailed error information by default.
3. **Interceptors:**

- Axios provides a powerful feature with interceptors for both requests and responses, allowing global modifications. `fetch` lacks built-in support for interceptors.

4. **Request Configuration:**

- Axios allows detailed configuration of requests through a variety of options. `fetch` may require more manual setup for headers, methods, and other configurations.

5. **JSON Parsing:**

- Axios automatically parses JSON responses, while with `fetch`, you need to manually call `.json()` on the response.

6. **Browser Support:**

- `fetch` is natively supported in modern browsers, but if you need to support older browsers, you might need a polyfill. Axios has consistent behavior across various browsers and does not rely on native implementations.

7. **Size:**

- `fetch` is generally considered lightweight, being a part of the browser. Axios, being a separate library, introduces an additional file size to your project.

Conclusion

- **Use `fetch` when:**
  - Working on a modern project without the need for additional features.
  - Prefer a lightweight solution and have no concerns about polyfills.
- **Use Axios when:**
  - Dealing with more complex scenarios such as interceptors.
  - Needing consistent behavior across different browsers.
  - Desiring a library with built-in features like automatic JSON parsing.

> In summary, both `fetch` and Axios have their strengths, and the choice depends on the specific requirements and preferences of the project. `fetch` is excellent for simplicity and lightweight projects, while Axios provides additional features and consistent behavior across different environments.

Comparing Code Implementation

Below are code snippets for making GET and POST requests using `async/await` with Fetch and Axios: Good Resource to Inspect Incoming HTTP Requests

Fetch (GET Request)

```
async function fetchData() {
  try {
    const response = await fetch('<https://api.example.com/data>');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}

fetchData();
```

Axios (GET Request)

```
// Install Axios using npm or yarn: npm install axios
import axios from 'axios';

async function fetchData() {
  try {
    const response = await axios.get('<https://api.example.com/data>');
    console.log(response.data);
  } catch (error) {
    console.error('Error:', error);
  }
}

fetchData();
```

Fetch (POST Request)

```
async function postData() {
  const url = '<https://api.example.com/postData>';
  const dataToSend = {
    key1: 'value1',
    key2: 'value2',
  };
```

```
  try {
    const response = await fetch(url, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(dataToSend),
    });
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}

postData();
```

Axios (POST Request)

```
// Install Axios using npm or yarn: npm install axios
import axios from 'axios';

async function postData() {
  const url = '<https://api.example.com/postData>';
  const dataToSend = {
    key1: 'value1',
    key2: 'value2',
  };

  try {
    const response = await axios.post(url, dataToSend);
    console.log(response.data);
  } catch (error) {
    console.error('Error:', error);
  }
}

postData();
```

The above examples demonstrate how to use `async/await` with Fetch and Axios to make asynchronous HTTP requests. In GET Requests although you can send HEADERS but you cannot send BODY while in case of POST, DELETE, PUT Requests you can send both HEADERS as well as BODY.