JavaScript: Fetch API

3rd semester @ Erhvervsakademi København

GitHub

Goal

Learn how to make HTTP requests using the Fetch API in JavaScript.

What is Fetch API?

> A JavaScript API that allows you to make network requests to retrieve resources from a server.

The fetch API takes in two main arguments:

1. The URL of the resource you want to fetch.
2. An optional configuration object (options) that specifies details like method, headers, body, etc.

```
fetch(url, options);
```

Return value of fetch

> The fetch function returns a Promise that resolves to the Response object representing the response to the request.

A promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value:

Promises

> A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

A promise can be in one of three states:

- **Pending**: The initial state, neither fulfilled nor rejected.
- **Fulfilled**: The operation completed successfully, and the promise has a value.
- **Rejected**: The operation failed, and the promise has a reason for the failure.

Why do we need promises?

> Promises allow us to handle asynchronous operations.

Examples of asynchronous operations:

- Fetching data from an API
- Reading files
- Timers (setTimeout, setInterval)
- User interactions (click events, form submissions)

> **Asynchronous operations** are operations that do not block the execution of the program. They allow the program to continue running while waiting for the operation to complete.

How do we work with promises?

> We can use the `.then()` and `.catch()` methods to handle the resolved value or the error of a promise.

Example:

```
someAsyncOperation() // Returns a promise
  .then(value => console.log("Promise resolved:", value))
  .catch(error => console.log("Promise rejected:", error));
```

Promises and Fetch

> When you call `fetch()`, it returns a Promise that resolves to the Response object.

```
// Returns a Promise that resolves to the Response object
fetch("https://jsonplaceholder.typicode.com/posts/1")
    // .then() is when the promise is resolved
    .then(resp => resp.json())
    .then(data => console.log(data))
    // .catch() is when the promise is rejected
    .catch(error => console.error("Error fetching data:", error));
```

Handling responses

Fetch does not `reject` on HTTP error status (like 404 or 500). So you must check `response.ok` and `response.status` to handle HTTP errors.

```
fetch("https://someapi.com/api/users")
  .then(response => {
    if (!response.ok) {
      throw new Error('HTTP error!: ' + response.status);
    }
    // Response is OK here
  });
```

Error handling

Use `.catch()` to handle network errors and Always check `response.ok` for HTTP errors

```
fetch("https://someapi.com/api/users")
  .then(response => {
    if (!response.ok) {
      throw new Error('HTTP error!: ' + response.status);
    }
    // Response is OK here
  })
  // Handle network errors
  .catch(error => {
    console.error('Fetch error:', error);
  });
```

Working with JSON

> Most APIs return JSON data so we need to convert to a JavaScript object using `response.json()` which also returns a promise.

```
fetch("https://someapi.com/api/users")
  .then(response => {
    if (!response.ok) {
      throw new Error('HTTP error!: ' + response.status);
    }
    return response.json();
  })

  // Do something with the JSON data
  .then(data => { console.log(data);})

  // Handle network errors
  .catch(error => {console.error('Fetch error:', error);});
```

Sending data with Fetch API

> To send data (`POST` , `PUT` , `PATCH`), we can use the `body` and `method` options in the fetch configuration object.

```
fetch("https://someapi.com/api/users",{
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ name: 'John', age: 30 })})
  .then(response => {
    if (!response.ok) {
      throw new Error('HTTP error!: ' + response.status);
    }
    return response.json();
  })
  .then(data => { console.log(data);})
  .catch(error => {console.error('Fetch error:', error);});
```

Fetch API and CORS

> **When making cross-origin requests (requests to a different domain), the server must allow it through CORS.**

- CORS = Cross-Origin Resource Sharing
- Controls which domains can access resources on a server
- If CORS is not enabled on the server, fetch requests from other origins will fail
- **Example on error you could encounter:**

```
Access to fetch at 'https://someapi.com/api/users' from origin 'http://localhost:5500' has been blocked by CORS policy
```

> **We will look more into CORS when we build a full-stack app.**

Using Async/Await with Fetch

> **Async/Await** are keywords in JavaScript, which make working with promises easier. They are syntactic sugar for working with promises and make async code look like sync code!

We use `async` to declare an asynchronous function and `await` to wait for a promise to resolve.

> **Notice** that `await` can only be used inside an `async` function!

Async/Await with Fetch Example

```
async function fetchData() {
  try {
    const response = await fetch("https://someapi.com/api/users");
    if (!response.ok) {
      throw new Error('HTTP error!');
    }
    return await response.json();
  } catch (error) {
    console.error('Fetch error:', error);
  }
}

// Usage: Still returns a promise so we can use
// .then() or await it in another async function
fetchData();
```