JavaScript
TUTORIAL

JavaScript Fetch API

Summary: in this tutorial, you'll learn about the JavaScript Fetch API and how to use it to make asynchronous HTTP requests.

Introduction to JavaScript Fetch API

The Fetch API is a powerful and modern tool that simplifies making HTTP requests directly from web browsers.

> If you've used `XMLHttpRequest` object before, you'll find that the Fetch API can handle all the same tasks, but with much more elegance and ease.

Fetch API leverages Promise, providing a cleaner and more flexible way to interact with servers. It helps handle asynchronous requests and responses more intuitively.

The `fetch()` is a method of the global window object, which allows you to send an HTTP request to a URL with a single command. Whether you're retrieving data, submitting a form, or interacting with APIs, Fetch API helps streamline the entire process, making your code much readable.

Sending a request

The `fetch()` requires only one parameter which is the URL of the resource that you want to fetch:

```
fetch(url);
```

The `fetch()` method returns a `Promise` so you can use the `then()` and `catch()` methods to handle it:

```
fetch(url)
    .then(response => {
        // handle the response
    })
    .catch(error => {
        // handle the error
    });
```

Once the request is completed, the resource becomes available, and the `Promise` resolves into a `Response` object.

The `Response` object serves as an API wrapper around the fetched resource.

Reading the response

If the response contains JSON data, you can use the `json()` method of the `Response` object to parse it.

The `json()` method returns a `Promise` that resolves with the full contents of the fetched resource, allowing to access the JSON data:

```
fetch(url)
    .then(response => response.json())
    .then(data => console.log(data));
```

In practice, you often use the `async` / `await` with the `fetch()` method to make the code more obvious:

```
const response = await fetch(url);
const data = await response.json();
console.log(data); // json data
```

Besides the `json()` method, the `Response` object has other methods such as `text()` , `blob()` , `formData()` and `arrayBuffer()` for handling the respective data types.

Handling HTTP status code

The `Response` object has a `status` property that gives you the HTTP status code of the response:

```
response.status
```

The HTTP status code allows you to determine whether the request was successful or not:

```
const response = await fetch(url);
if (response.status === 200) {
   // success
}
```

In practice, you'll use a convenient property `ok` that checks if the status code is in the range of 200-299. If it is `false`, the request was not successful:

```
const response = await fetch(url);
if (!response.ok) {
   throw new Error(`HTTP error! Status: ${response.status}`);
}
```

JavaScript Fetch API example

We'll make a GET request to the following API endpoint that returns a list of users:

```
https://jsonplaceholder.typicode.com/users
```

Step 1. Create a new directory such as fetch to store the project files.

Step 2. Create an `index.html` file within the project directory:

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

    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Fetch API Demo</title>
        <script src="js/app.js" defer></script>
    </head>

    <body>
        <div id="root">
            <div id="content"></div>
            <p id="message"></p>
            <p id="loading"></p>
        </div>
    </body>

</html>
```

The HTML file has the root element that contains three elements:

- `#content` for rendering the user list.
- `#message` for showing any error message.
- `#loading` for displaying a loading message.

Step 3. Create an `app.js` inside the `js` directory:

```
const getUsers = async () => {
  const url = 'https://jsonplaceholder.typicode.com/users';
  const response = await fetch(url);
  return await response.json();
};

const render = (users) => {
  return users.map(({ name, email }) => `<li>${name} (${email})</li>`).join('');
};

(async () => {
  const users = await getUsers();
  document.querySelector('#content').innerHTML = `<ul>${render(users)}</ul>`;
})();
```

How it works.

First, define a function `getUsers` that uses the `fetch()` method to fetch data from the API endpoint: https://jsonplaceholder.typicode.com/users

```
const getUsers = async () => {
  const url = 'https://jsonplaceholder.typicode.com/users';
  const response = await fetch(url);
  return await response.json();
};
```

Second, create a new function `render` that returns an HTML snippet for showing a list of users on the screen:

```
const render = (users) => {
  return users.map(({ name, email }) => `<li>${name} (${email})</li>`).join('');
};
```

Third, define an IIFE function that calls the `getUsers()` function and displays the user list on the screen:

```
(async () => {
  const users = await getUsers();
  document.querySelector('#content').innerHTML = `<ul>${render(users)}</ul>`;
})();
```

Handling errors

When making a web request, various errors can occur, such as network outages, server downtime, or other connection issues.

To handle the errors, you can use the `try...catch` statement:

```
(async () => {
  try {
    // fetch the users
    const users = await getUsers();
    // show the user list
    document.querySelector('#content').innerHTML = `<ul>${render(users)}</ul>`;
  } catch (err) {
    // show the error message
    document.querySelector('#message').textContent = err.message;
  }
})();
```

In this example, if an error occurs, the `catch` block will execute, logging the error message in the console and displaying a user-friendly error message to the user.

To test this, you can change the API endpoint to an invalid URL, such as:

```
https://jsonplaceholder.typicode.net/users
```

(change `.com` to `.net`), and then open the index.html file. The page will display the following error message:

```
Error getting users
```

And the console window will have the following error:

```
GET https://jsonplaceholder.typicode.net/users net::ERR_NAME_NOT_RESOLVED
```

Displaying a loading indicator

If the network is slow, you will see a blank page for a brief period, which leads to a poor user experience.

To enhance this, you can display a loading message while the data is being fetched.

To do that, you can modify the IIFE function as follows:

```
(async () => {
  // show the loading element
  const loadingElem = document.querySelector('#loading');
  loadingElem.innerHTML = 'Loading...';
  try {
    // fetch the users
    const users = await getUsers();

    // show the user list
    document.querySelector('#content').innerHTML = `<ul>${render(users)}</ul>`;
  } catch (err) {
    // show the error message
    document.querySelector('#message').textContent = err.message;
  } finally {
    loadingElem.innerHTML = '';
  }
})();
```

Before making a request, we set the loading message to `'Loading...'`:

```
const loadingElem = document.querySelector('#loading');
loadingElem.innerHTML = 'Loading...';
```

After the request is completed, whether successful or not, you clear the loading message by setting it to blank in the `finally` block:

```
// ...
finally {
    loadingElem.innerHTML = '';
}
```

Keep in mind that if the network speed is fast enough, the loading message might not be visible. To test the loading message, you can simulate a network delay as follows:

```
(async () => {
  // show the loading element
  const loadingElem = document.querySelector('#loading');
  loadingElem.innerHTML = 'Loading...';

  // simulate network delay
  const delay = (ms) => new Promise((resolve) => setTimeout(resolve, ms));
  await delay(2000); // delay 2 seconds

  try {
    // fetch the users
    const users = await getUsers();

    // show the user list
    document.querySelector('#content').innerHTML = `<ul>${render(users)}</ul>`;
  } catch (err) {
    // show the error message
    document.querySelector('#message').textContent = err.message;
  } finally {
    loadingElem.innerHTML = '';
  }
})();
```

In this code, we include the following snippet to introduce a 2-second delay before making the request:

```
// simulate network delay
const delay = (ms) => new Promise((resolve) => setTimeout(resolve, ms));
await delay(2000); // delay 2 seconds
```

Downloading the project source code

Click here to download the project source code

Making an HTTP POST request

We show you how to make an HTTP POST request to the following API endpoint that creates a new blog post with the `title`, `body`, and `userId`:

```
<code>https://jsonplaceholder.typicode.com/posts</code>
```

Step 1. Create a new directory to store project files.

Step 2. Create a new `index.html` file in the project directory.

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

    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Fetch API Demo - HTTP POST</title>
        <meta name="robots" content="noindex">
        <script src="js/app.js" defer></script>
    </head>
    <body>
    </body>

</html>
```

The `index.html` file includes the `js/app.js` file in its header.

Step 3. Create a new `js/app.js` file with the following code:

```javascript
async function create(blogPost) {
  try {
    // Create the URL
    const url = 'https://jsonplaceholder.typicode.com/posts';

    // Create the headers
    const headers = {
      'Content-Type': 'application/json',
    };

    // Create the POST body
    const body = JSON.stringify({
      title: blogPost.title,
      body: blogPost.body,
      userId: blogPost.userId,
    });

    // Send the POST request
    const response = await fetch(url, { method: 'POST', headers, body });
    // Check the response status
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }

    // Parse the JSON response
    const data = await response.json();
    console.log('Success:', data);
  } catch (error) {
    // Handle any errors
    console.error('Error:', error);
  }
}

create({
  title: 'Test Post',
  body: 'This is a test post',
  userId: 1,
});
```

How it works.

First, define a function that creates a new blog post that accepts a post object:

```
async function create(post)
```

Second, construct a URL endpoint where the POST request will be sent:

```
const url = 'https://jsonplaceholder.typicode.com/posts';
```

Third, create an HTTP header to be sent with the request:

```
const headers = {
  'Content-Type': 'application/json',
};
```

Set the `'Content-Type'` header to `'application/json'` to instruct that the request's body is in JSON format.

Fourth, create a body of the POST request by serializing the post object into a JSON string using `JSON.stringify()` method:

```
const body = JSON.stringify({
  title: post.title,
  body: post.body,
  userId: post.userId,
});
```

This assumes that the `blogPost` object has a `title` , `body` , and `userId` properties.

Fifth, send the POST request using the `fetch()` method:

```
const response = await fetch(url, {
    method: 'POST',
    headers,
    body
});
```

In this syntax:

- `url` : The API endpoint where you send the POST request.
- `method: 'POST'` : Specifies that this is an HTTP POST request.
- `headers` : Includes the header in the request.
- `body` : Contains the JSON string included in the body of the HTTP POST request.

The `fetch()` method returns a `Response` object.

Sixth, check the response status and throw an error if the request was not successful:

```
if (!response.ok) {
  throw new Error(`HTTP error! Status: ${response.status}`);
}
```

Seventh, parse the JSON response and log the data into the console window:

```
const data = await response.json();
console.log('Success:', data);
```

Eighth, catch any error that occurs during the request and log it to the console:

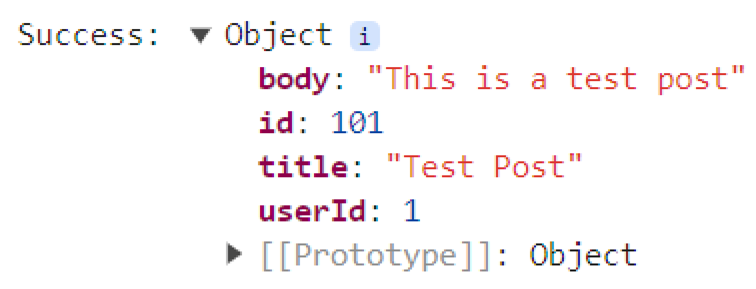
```
catch (error) {
  console.error('Error:', error);
}
```

Finally, call the `create()` function with a blog post object containing `title` , `body` , and `userId` properties:

```
create({
  title: 'Test Post',
  body: 'This is a test post',
  userId: 1,
});
```

Step 4. Open the index.html in the web browser. It'll execute the app.js file that makes an HTTP POST request.

Step 5. Open the console window, you'll see the following message if the request was successful:



```
Success:  ▼ Object i
              body: "This is a test post"
              id: 101
              title: "Test Post"
              userId: 1
            ▶ [[Prototype]]: Object
```

Downloading the project source code

Click here to download the project source code

Summary

- Fetch API provides a simpler and more flexible way to make HTTP requests compared to `XMLHttpRequest` object.
- Use `fetch()` method to make an asynchronous web request to a URL.
- The `fetch()` returns a `Promise` that resolves into a `Response` object.
- Use the `status` or `ok` property of the `Response` object to check whether the request was successful.
- Use the `json()` method of the `Response` object to parse the contents into JSON data.
- Use the `try...catch` statement to handle errors when making HTTP requests.