

Fetch API

- The Fetch API is a modern, promise-based mechanism that allows you to make network requests similar to XMLHttpRequest (XHR) but in a more powerful and flexible way.
- It is used to interact with web servers to retrieve resources such as JSON data, HTML, text files, etc.
- Fetch API is part of the browser's JavaScript environment and works asynchronously.
- It is built on promises, which means it uses then() and catch() methods for handling the results of the request instead of callbacks.
- The Fetch API is a modern interface that allows us to make HTTP requests in web browsers.
- The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses.
- The Fetch API includes a fetch() method, defined in the window object (global scope), which we can use to make requests. This method returns a Promise that can be used to retrieve the response.
- The fetch() method is used to send requests to the server without refreshing the page.
- The Fetch API provides an interface for fetching resources from the server.
- fetch() follows a two-step process when handling JSON data. First, it makes the actual request, and then the .json() method is called on the response to process the data.
- Fetch offers a better alternative that can easily integrate with other technologies such as Service Workers.
- Fetch also provides a single, logical place to define other HTTP-related concepts like CORS and extensions to HTTP.

Basic code structure while using Fetch API

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

Why is Fetch API Used?

- **Simplicity:** Fetch API simplifies the syntax of making asynchronous HTTP requests. It replaces older methods like XMLHttpRequest, which required more complex coding and handling of responses.
- **Promise-based:** The Fetch API is based on JavaScript Promises, making it easier to chain operations like parsing and error handling.
- **Improved Code Readability:** The structure of the Fetch API leads to more readable and maintainable code.
- **Wide Support:** Fetch API is supported by almost all modern browsers.

XMLHttpRequest

XMLHttpRequest (XHR) is a legacy API for making HTTP requests with the goal of retrieving data from a URL without refreshing the page (also known as AJAX). It allowed web pages to be updated asynchronously by exchanging small amounts of data with the server.

Basic code structure while using XMLHttpRequest

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data', true);

xhr.onload = function() {
  if (xhr.status === 200) {
    console.log(JSON.parse(xhr.responseText));
  }
};

xhr.onerror = function() {
  console.error('Error occurred.');
```

```
};

xhr.send();
```

Difference between FetchAPI and XMLHttpRequest

Feature	Fetch API	XMLHttpRequest
Promise-based	Yes, uses Promises, making it easier to handle responses, errors, and asynchronous operations.	No, uses callbacks, making the code more complex and harder to follow.
Simplicity	Fetch API is simpler with cleaner syntax and easier to read and maintain.	XHR syntax is more complex and requires more boilerplate code.
Response Handling	Fetch API's response object provides a body stream and various methods like <code>.json()</code> , <code>.text()</code> , and <code>.blob()</code> .	XHR provides the data in the <code>responseText</code> property and has limited support for streaming responses.
Error Handling	Fetch API treats HTTP errors (like 404 or 500) as resolved promises. You have to check <code>response.ok</code> to detect failures.	XHR throws errors for network issues, but HTTP errors must be handled manually by checking status.
CORS (Cross-Origin Requests)	Fetch API requires more explicit configurations (like <code>mode</code> , <code>credentials</code> , etc.) for CORS requests.	XHR can handle CORS but typically requires more setup and configuration.
Timeouts	No built-in support for request timeouts. You need to implement it manually (e.g., using <code>Promise.race()</code>).	XHR has the <code>timeout</code> property, which allows you to specify a request timeout.
Request Cancellation	Not natively supported in Fetch, though you can use <code>AbortController</code> .	XHR supports request cancellation using the <code>abort()</code> method.
Progress Monitoring	Fetch API doesn't natively support progress monitoring of requests.	XHR supports progress events, such as <code>onprogress</code> , which can be used to track the progress of the request.
Older Browser Support	Fetch API is not supported in Internet Explorer and some older browsers. Polyfills are required for backward compatibility.	XHR is supported in all modern browsers and older versions, including IE.

Fetch API Advantages Over XMLHttpRequest

- **Modern Syntax:** Fetch is much cleaner to write and more intuitive. It avoids the callback hell often seen in XHR, especially with deeply nested operations.
- **Promise-Based Handling:** Fetch makes use of Promises, allowing you to write more modern, asynchronous code with `.then()` and `.catch()` blocks. It's easier to chain operations with Fetch.
- **Streaming Responses:** Fetch supports streaming responses, making it more flexible, especially for handling large files or continuous data.
- **Code Simplicity and Readability:** Fetch simplifies the code, making it easier to work with for basic and complex HTTP requests.
- **Same-origin by Default:** Fetch API applies stricter security policies by default. For example, it applies CORS settings for cross-origin requests more explicitly than XHR, which is beneficial for security-conscious applications.

Limitations of Fetch API

- **No Progress Events:** Unlike XMLHttpRequest, Fetch does not have built-in progress events like `onprogress`, making it less suitable for applications that need to track the upload or download progress.
- **No Timeouts:** The Fetch API does not support setting a timeout for requests natively. You need to manually implement a timeout using additional code or an `AbortController`.

Http Requests and Responses

- **HTTP Requests:** HTTP requests are messages sent by a client (like a web browser or a JavaScript function) to a server to perform certain actions like retrieving data or submitting data. The client initiates the request, and the server responds.

Components of an HTTP Request

- **Request Method:** The request method indicates the type of operation the client wants to perform. Common methods include:
 - **GET:** Retrieves data from the server. (No payload is sent)
 - **POST:** Submits data to the server. (Payload can be sent in the body)
 - **PUT:** Updates existing data on the server.
 - **DELETE:** Removes data from the server.
- **Request URL:** The URL (Uniform Resource Locator) is the address of the server resource the client wants to interact with. It consists of:
 - **Scheme:** Protocol (http, https).
 - **Domain:** The server address (e.g., example.com).
 - **Path:** The resource on the server (/api/products).
 - **Query parameters (optional):** Data sent to the server in a GET request (e.g., ?page=2&size=10).
- **Request Headers:** Headers provide additional metadata for the server. Examples include:
 - **Content-Type:** Indicates the type of data being sent (e.g., application/json for JSON payload).
 - **Authorization:** Provides credentials for authentication.
 - **Accept:** Indicates the data format expected from the server (e.g., application/json).
- **Request Body:** The body contains data sent in POST, PUT, or PATCH requests. It is often in JSON format but can also be text, form-data, or other formats.

- **HTTP Responses:** An HTTP response is the server's reply to the client's request. It contains information about the status of the request and, typically, the data requested by the client.

Components of an HTTP Response

- **Status Code:** The status code is a 3-digit number that indicates the outcome of the request. Common status codes:
 - **200 OK:** The request was successful.
 - **201 Created:** A resource was successfully created (e.g., after a POST).
 - **400 Bad Request:** There's something wrong with the request from the client.
 - **401 Unauthorized:** Authentication is required.
 - **404 Not Found:** The requested resource doesn't exist.
 - **500 Internal Server Error:** The server encountered an error.
- **Response Headers:** Headers in the response contain metadata such as:
 - **Content-Type:** Describes the type of data (e.g., application/json).
 - **Cache-Control:** Instructs the browser on how to handle caching.
 - **Set-Cookie:** Sends cookies to the client.
- **Response Body:** The body contains the actual data sent by the server, typically in JSON, HTML, or plain text. If the request is successful, this is where the fetched data will be.

• HTTP Request Lifecycle (In Fetch API)

The Fetch API follows these steps:

- Client sends a request using the `fetch()` function, specifying method, headers, and optionally body data.
- Server receives the request and processes it (this could involve reading data from a database, performing an action, etc.).
- Server sends a response back to the client with the status code, headers, and possibly a body.
- Client processes the response using JavaScript (usually with `then()`, `async/await`, etc.).

HTTP Request & Response Examples

Example: Making the GET request

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => response.json()) // Parse the response body as JSON
  .then(data => console.log(data))   // Handle the fetched data
  .catch(error => console.error('Error:', error));
```

Example: POST Request with Body

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({ id: 45, title: 'My Post Title' }),
})
  .then(response => response.json())
  .then(data => console.log('Post Created:', data))
  .catch(error => console.error('Error:', error));
```

Example: Handling Different Status Codes

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => {
    if (!response.ok) {
      // Handle different error codes
      if (response.status === 404) {
        throw new Error('Product not found');
      } else {
        throw new Error('Something went wrong');
      }
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Example: GET Request with Async/Await

```
async function fetchProduct() {  
  try {  
    const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');  
    if (!response.ok) {  
      throw new Error('Product not found');  
    }  
    const data = await response.json(); // Parse the JSON data  
    console.log(data); // Use the data  
  } catch (error) {  
    console.error('Error:', error); // Handle any errors  
  }  
}  
fetchProduct();
```



Example: Fetch data from an API and display it in the browser

index.html

```
<h1>Fetching the API data</h1>
<div id="data-container"></div>
<div id="error-message" style="color: red;"></div>
<script src="index.js"></script>
```

index.js

```
function fetchData() {
  const apiUrl = 'https://jsonplaceholder.typicode.com/posts';
  const dataContainer = document.getElementById('data-container');
  const errorMessage = document.getElementById('error-message');
  fetch(apiUrl)
    .then(response => {
      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }
      return response.json(); // Parse response as JSON
    })
    .then(data => {
      // To clear any previous text content or error message
      errorMessage.textContent = "";
      // To display the fetched data in the browser
      data.forEach(item => {
        const postElement = document.createElement('div');
        postElement.innerHTML = `<h3>${item.title}</h3><p>${item.body}</p>`;
        dataContainer.appendChild(postElement);
      });
    })
    .catch(error => {
      // Handle errors (network errors or response errors)
      errorMessage.textContent = `Failed to fetch data: ${error.message}`;
    });
}

// Call the fetchData function when the page loads
window.onload = fetchData;
```

Chaining Fetch Requests

- Chaining multiple fetch requests using `.then()` is a common pattern in JavaScript for handling sequential asynchronous operations.
- Each fetch call can depend on the result of the previous one, allowing for complex interactions with APIs.
- When you chain `.then()` methods, each method receives the result of the previous one as its argument. This makes it easy to pass data from one request to the next.

Example: Fetching User Data and Then Fetching Posts

```
const userId = 1;
```

```
fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
  .then(response => {
    if (!response.ok) throw new Error('Network response was not ok');
    return response.json();
  })
  .then(user => {
    console.log('User:', user);
    return fetch(`https://jsonplaceholder.typicode.com/posts?userId=${user.id}`);
  })
  .then(response => {
    if (!response.ok) throw new Error('Network response was not ok');
    return response.json();
  })
  .then(posts => {
    console.log('User Posts:', posts);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Example: Fetching Data from Multiple APIs

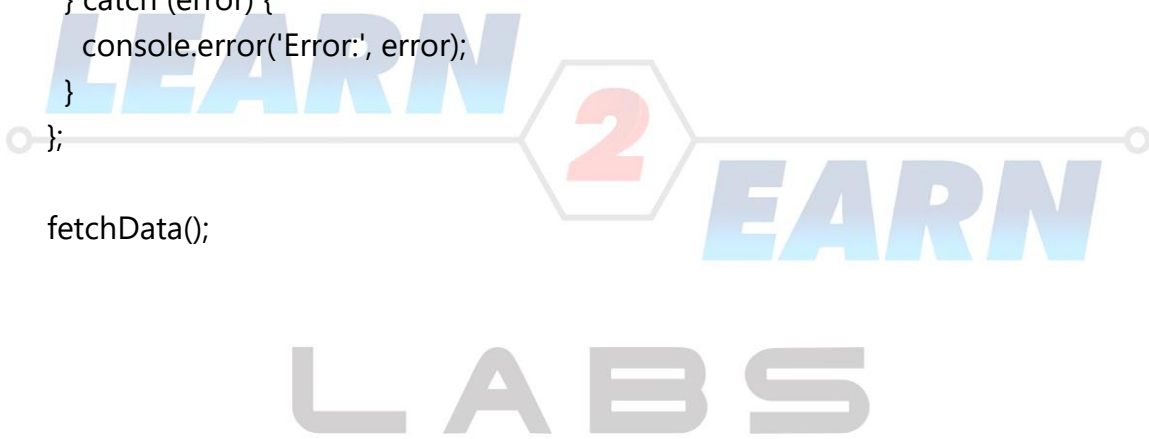
```
const api1 = 'https://jsonplaceholder.typicode.com/posts';
const api2 = 'https://jsonplaceholder.typicode.com/users';
fetch(api1)
  .then(response => {
    if (!response.ok) throw new Error('Network response was not ok');
    return response.json(); })
  .then(posts => {
    console.log('Posts:', posts);
    return fetch(api2); })
  .then(response => {
    if (!response.ok) throw new Error('Network response was not ok');
    return response.json(); })
  .then(users => {
    console.log('Users:', users); })
  .catch(error => {
    console.error('Error:', error); });
```

Example: Fetching User Comments Based on Post ID

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => {
    if (!response.ok)
      throw new Error('Network response was not ok');
    return response.json(); })
  .then(posts => {
    console.log('Posts:', posts);
    const firstPostId = posts[0].id; // Get the ID of the first post
    return
    fetch(`https://jsonplaceholder.typicode.com/comments?postId=${firstPostId}`);
  })
  .then(response => {
    if (!response.ok)
      throw new Error('Network response was not ok');
    return response.json(); })
  .then(comments => {
    console.log('Comments for Post 1:', comments); })
  .catch(error => { console.error('Error:', error); });
```

Example: Using Async/Await for Cleaner Chaining

```
const fetchData = async () => {  
  try {  
    const userResponse = await  
    fetch('https://jsonplaceholder.typicode.com/users/1');  
    if (!userResponse.ok) throw new Error('Network response was not ok');  
    const user = await userResponse.json();  
    console.log('User:', user);  
  
    const postsResponse = await  
    fetch(`https://jsonplaceholder.typicode.com/posts?userId=${user.id}`);  
    if (!postsResponse.ok) throw new Error('Network response was not ok');  
    const posts = await postsResponse.json();  
    console.log('User Posts:', posts);  
  } catch (error) {  
    console.error('Error:', error);  
  }  
};  
  
fetchData();
```

The logo for Learn 2 Earn Labs is positioned in the background. It features the word "LEARN" in blue, a large red number "2" inside a hexagon, the word "EARN" in blue, and the word "LABS" in grey below it.