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.

The Fetch API is a modern interface that allows you to make HTTP requests to servers from web browsers.

If you have worked with `XMLHttpRequest` (`XHR`) object, the Fetch API can perform all the tasks as the `XHR` object does.

In addition, the Fetch API is much simpler and cleaner. It uses the `Promise` `(https://www.javascripttutorial.net/es6/javascript-promises/)` to deliver more flexible features to make requests to servers from the web browsers.

The `fetch()` method is available in the global scope that instructs the web browsers to send a request to a URL.

Sending a Request

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

```
let response = 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
    });
```

When the request completes, the resource is available. At this time, the promise will resolve into a `Response` object.

The `Response` object is the API wrapper for the fetched resource. The `Response` object has a number of useful properties and methods to inspect the response.

Reading the Response

If the contents of the response are in the raw text format, you can use the `text()` method. The `text()` method returns a `Promise` that resolves with the complete contents of the fetched resource:

```
fetch('/readme.txt')
    .then(response => response.text())
    .then(data => console.log(data));
```

In practice, you often use the `async` / `await` (https://www.javascripttutorial.net/es-next/javascript-async-await/) with the `fetch()` method like this:

```
async function fetchText() {
    let response = await fetch('/readme.txt');
    let data = await response.text();
    console.log(data);
}
```

Besides the `text()` method, the `Response` object has other methods such as `json()`, `blob()`, `formData()` and `arrayBuffer()` to handle the respective type of data.

Handling the status codes of the Response

The `Response` object provides the status code and status text via the `status` and `statusText` properties. When a request is successful, the status code is `200` and status text is `OK`:

```
async function fetchText() {
    let response = await fetch('/readme.txt');

    console.log(response.status); // 200
    console.log(response.statusText); // OK

    if (response.status === 200) {
        let data = await response.text();
        // handle data
    }
}

fetchText();
```

Output:

```
200
OK
```

If the requested resource doesn't exist, the response code is `404`:

```
let response = await fetch('/non-existence.txt');

console.log(response.status); // 400
console.log(response.statusText); // Not Found
```

Output:

```
400
Not Found
```

If the requested URL throws a server error, the response code will be `500`.

If the requested URL is redirected to the new one with the response `300-309`, the `status` of the `Response` object is set to `200`. In addition the `redirected` property is set to `true`.

The `fetch()` returns a promise that rejects when a real failure occurs such as a web browser timeout, a loss of network connection, and a CORS violation.

JavaScript Fetch API example

Suppose that you have a json file that locates on the webserver with the following contents:

```
[{
        "username": "john",
        "firstName": "John",
        "lastName": "Doe",
        "gender": "Male",
        "profileURL": "img/male.png",
        "email": "john.doe@example.com"
    },
    {
        "username": "jane",
        "firstName": "Jane",
        "lastName": "Doe",
        "gender": "Female",
        "profileURL": "img/female.png",
        "email": "jane.doe@example.com"
    }
]
```

The following shows the HTML page:

```
<!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>
    <link rel="stylesheet" href="css/style.css">
</head>
<body>
    <div class="container"></div>
    <script src="js/app.js"></script>
</body>
</html>
```

In the `app.js`, we'll use the `fetch()` method to get the user data and render the data inside the `<div>` element with the class `container`.

First, declare the `getUsers()` function that fetches `users.json` from the server.

```
async function getUsers() {
    let url = 'users.json';
    try {
        let res = await fetch(url);
        return await res.json();
    } catch (error) {
        console.log(error);
    }
}
```

Then, develop the `renderUsers()` function that renders user data:

```
async function renderUsers() {
    let users = await getUsers();
    let html = '';
    users.forEach(user => {
        let htmlSegment = `<div class="user">
                        <img src="${user.profileURL}" >
                        <h2>${user.firstName} ${user.lastName}</h2>
                        <div class="email"><a href="email:${user.email}">${user
```

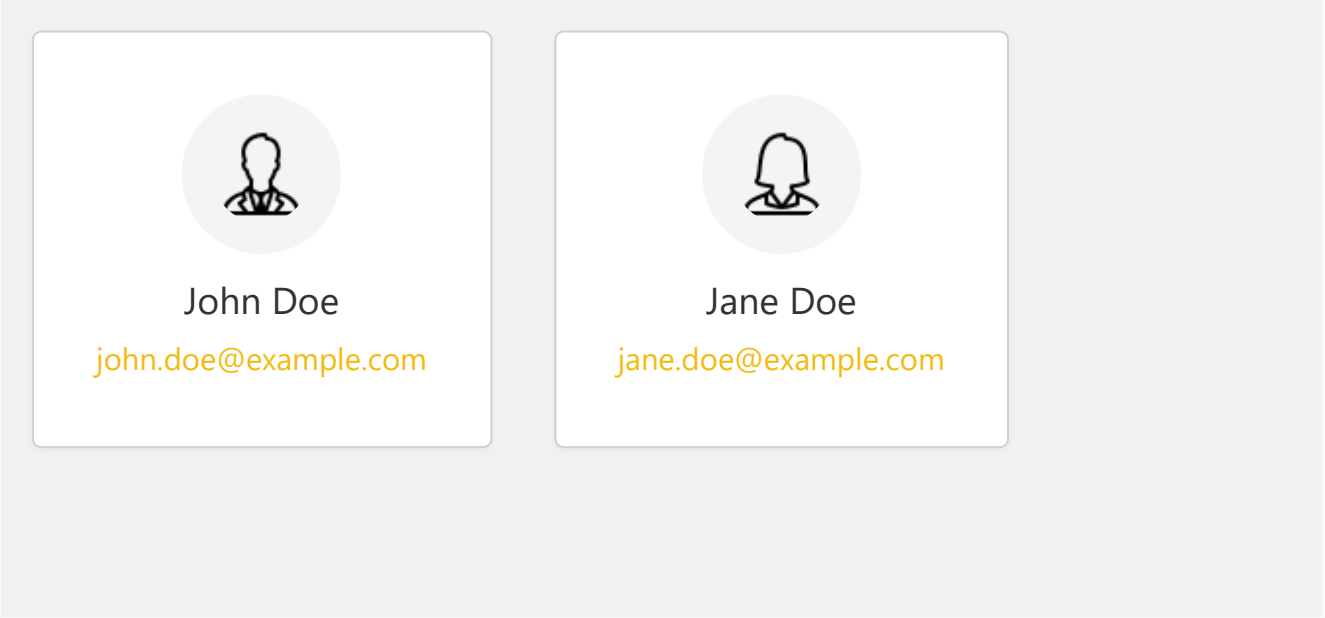
```
                        </div>`;

            html += htmlSegment;
        });

        let container = document.querySelector('.container');
        container.innerHTML = html;
}

renderUsers();
```

Output



Check out the Fetch API demo (https://www.javascripttutorial.net/sample/api/fetch/index.html) .

Summary

- The Fetch API allows you to asynchronously request for a resource.
- Use the `fetch()` method to return a promise that resolves into a `Response` object. To get the actual data, you call one of the methods of the Response object e.g., `text()` or `json()` . These methods resolve into the actual data.

- Use the `status` and `statusText` properties of the `Response` object to get the status and status text of the response.
- use the `catch()` method or `try...catch` statement to handle a failure request.