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*

});

Code language: JavaScript (javascript)

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);

}

Code language: JavaScript (javascript)

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();

Code language: JavaScript (javascript)

Output:

200

OK

Code language: JavaScript (javascript)

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); *// OK*

Output:

400

Not Found

Code language: JavaScript (javascript)

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>**

n 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);

}

}

Code language: JavaScript (javascript)

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.email}</a></div>

</div>`;

html += htmlSegment;

});

let container = document.querySelector('.container');

container.innerHTML = html;

}

renderUsers();

## Understanding JavaScript Promises

In JavaScript, a promise is an object that returns a value which you hope to receive in the future, but not now.

Because the value will be returned by the promise in the future, the promise is very well-suited for handling asynchronous operations.

It’ll be easier to understand the concept of JavaScript promises through an analogy.

Suppose that you promise to complete learning JavaScript by next month.

You don’t know if you will spend your time and effort to learn JavaScript until next month. You can either be completing learning JavaScript or not.

A promise has three states:

* Pending: you don’t know if you will complete learning JavaScript by the next month.
* Fulfilled: you complete learning JavaScript by the next month.
* Rejected: you don’t learn JavaScript at all.

A promise starts in the pending state which indicates that the promise hasn’t completed. It ends with either fulfilled (successful) or rejected (failed) state.

## Creating a promise: the Promise constructor

To create a promise in JavaScript, you use the Promise constructor:

let completed = true;

let learnJS = new Promise(function (resolve, reject) {

if (completed) {

resolve("I have completed learning JS.");

} else {

reject("I haven't completed learning JS yet.");

}

});

Code language: JavaScript (javascript)

The Promise constructor accepts a function as an argument. This function is called the executor.

The executor accepts two functions with the names, by convention, resolve() and reject().

When you call the new Promise(executor), the executor is called automatically.

Inside the executor, you manually call the resolve() function if the executor is completed successfully and invoke the reject() function in case of an error occurs.

If you embed the above JavaScript code in an HTML document and check the console window, you will see that the promise is resolved because the completed variable is set to true.

To see the pending state of the promise, we wrap the code of the executor in the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function:

let completed = true;

let learnJS = new Promise(function (resolve, reject) {

setTimeout(() => {

if (completed) {

resolve("I have completed learning JS.");

} else {

reject("I haven't completed learning JS yet.");

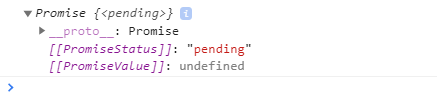
}

}, 3 \* 1000);

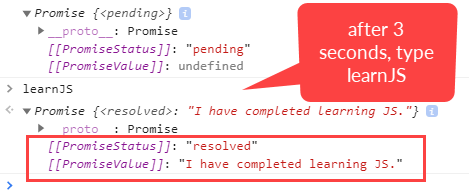
});

Code language: JavaScript (javascript)

Now, you see that the promise starts with the pending state with the value is undefined. The promise value will be returned later once the promise is completed.



After about 3 seconds, type the learnJS in the console window, you will see that the state of the promise becomes resolved and the promise value is the string that we passed to the resolve() function.

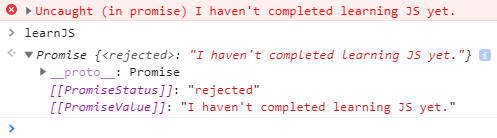


So calling the resolve() function moves the promise object to the fulfilled state. If you change the value of the completed variable to false and run the script again:

let completed = false;

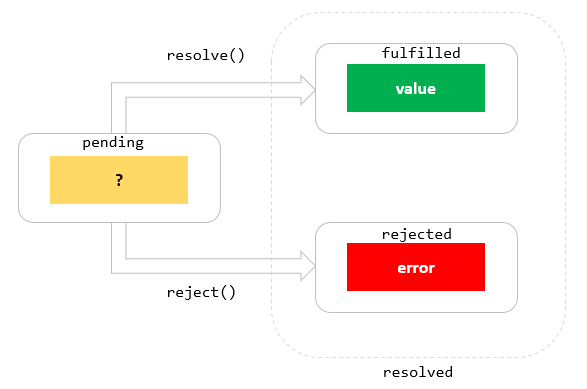
Code language: JavaScript (javascript)

You will see an error message and the state of the promise becomes rejected after 3 seconds:



In other words, calling the reject() method moves the promise object to the rejected state.

The following picture illustrates the states of a promise and the effect of calling the resolve() and reject() functions:



Once the promise reaches either fulfilled state or rejected state, it stays in that state and can’t switch.

In other words, a promise cannot go from the fulfilled state to the rejected state and vice versa. It also cannot go back from the fulfilled state or rejected state to the pending state.

If the promise reaches fulfilled state or rejected state, the promise is resolved.

Once a new Promise object is created, it is in the pending state until it is resolved. To schedule a callback when the promise is either resolved or rejected, you call the methods of the Promise object: then(), catch(), and finally().

## Consuming a Promise: then, catch, finally

### 1) The then() method

The then() method is used to schedule a callback to be executed when the promise is successfully resolved.

The then() method takes two [callback functions](https://www.javascripttutorial.net/javascript-callback/):

promiseObject.then(onFulfilled, onRejected);

Code language: CSS (css)

The onFulfilled callback is called if the promise is fulfilled. The onRejected callback is called when the promise is rejected.

The following function returns a Promise object:

function makePromise(completed) {

return new Promise(function (resolve, reject) {

setTimeout(() => {

if (completed) {

resolve("I have completed learning JS.");

} else {

reject("I haven't completed learning JS yet.");

}

}, 3 \* 1000);

});

}

Code language: JavaScript (javascript)

And the following calls the makePromise() function and invokes the then() method:

let learnJS = makePromise(true);

learnJS.then(

success => console.log(success),

reason => console.log(reason)

);

Code language: JavaScript (javascript)

It is possible to schedule a callback to handle the fulfilled or rejected case only. The following runs the fulfilled case:

learnJS.then(

value => console.log(value)

);

Code language: JavaScript (javascript)

And the following schedules a callback to handle the rejected case:

let masterJS = makePromise(false);

masterJS.then(

undefined,

reason => console.log(reason)

);

Code language: JavaScript (javascript)

Note that you must pass undefined to the then() method as the first argument.

### 2) The catch() method

If you want to schedule a callback to be executed when the promise is rejected, you can use the catch() method of the Promise object:

learnJS.catch(

reason => console.log(reason)

);

Code language: JavaScript (javascript)

Internally, the catch() method invokes the then(undefined, onRejected) method.

### 3) The finally() method

Sometimes, you want to execute the same piece of code whether the promise is fulfilled or rejected. For example:

function createApp() {

*// ...*

}

learnJS

.then(

(success) => {

console.log(success);

createApp();

}

).catch(

(reason) => {

console.log(reason);

createApp();

}

);

Code language: JavaScript (javascript)

As you can see, the createApp() function call is duplicated in both then() and catch() methods.

To remove this duplicate and execute the createApp() whether the promise is fulfilled or rejected, you use the finally() method, like this:

learnJS

.then(success => console.log(success))

.catch(reason => console.log(reason))

.finally(() => createApp());

Code language: 1

JavaSc

[**https://api.github.com/users**](https://api.github.com/users)

console.log("api called");

let x = document.getElementById("btnBtn");

let y = document.getElementById("container");

function GetData() {

console.log("Started GetData");

url = "records.json";

fetch(url).then((response) => {

console.log("Inside First then");

return response.text();

}).then((data) => {

console.log("Inside Second then");

console.log(data);

});

}

console.log("Before Get Data");

GetData();

console.log("After Get Data");

With both resolve & reject

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title></title>

<script>

console.log("api called");

function GetData() {

console.log("GetData Method is called");

url = "ajax\_info.txt";

fetch(url).then(function (response, reject) {

console.log("Inside 1st block");

return response.text();

}).then(function (data) {

console.log("Inside 2nd block");

console.log(data);

document.getElementById("container").innerHTML = data;

}).catch(function () {

console.log("There was some error");

});

}

</script>

</head>

<body>

<input type="button" id="btnGetData" onclick="GetData()" />

<div id="container"></div>

</body>

</html>

function GetData() {

console.log("Started GetData");

url = "https://api.github.com/users";

fetch(url).then((response) => {

console.log("Inside First then");

return response.json();

}).then((data) => {

console.log("Inside Second then");

console.log(data);

});

}

**For post**

**fake post request api**

[**http://dummy.restapiexample.com/**](http://dummy.restapiexample.com/)

function PostData() {

console.log("Started PostData");

data = {

"status": "success",

"data": {

"name": "test",

"salary": "123",

"age": "23",

"id": 25

}

}

params =

{

method: 'post',

headers: {

'Content-Type':'application/json'

},

body:JSON.stringify(data)

}

url = "http://dummy.restapiexample.com/api/v1/create";

fetch(url,params).then((response) => {

console.log("Inside First then");

return response.json();

}).then((data) => {

console.log("Inside Second then");

console.log(data);

});

}

**Using Arrow function**

function PostData() {

console.log("Started PostData");

data = {

"name": "Anamika Sawhney",

"job": "Consultant"

},

params =

{

method: 'post',

headers: {

'Content-Type': 'application/json',

"Access-Control-Allow-Headers": "Content-Type",

"Access-Control-Allow-Origin": "\*",

"Access-Control-Allow-Methods": "OPTIONS,POST,GET"

},

body:JSON.stringify(data)

}

url = "https://reqres.in/api/users";

fetch(url,params).then(response =>

response.json()

).then(data =>

console.log(data)

);

}

PostData();