The Fetch API

The XMLHttpRequest object was finally standardized by the WHATWG and W3C as part of the HTML5 specification, despite it originally being implemented by Microsoft many years earlier, and already available in most browsers.

It has since been superseded by the*Fetch API*, which is currently a living standard for requesting and sending data asynchronously across a network. The Fetch API uses promises to avoid callback hell, and also streamlines a number of concepts that had become cumbersome when using the XMLHttpRequest object.

We're going to start by taking a look at how the Fetch API works and the different interfaces that it uses. After this we'll build a page that demonstrates the ideas we've looked at.

Basic Usage

The Fetch API provides a global fetch() method that only has one mandatory argument, which is the URL of the resource you wish to fetch. A very basic example would look something like the following piece of code:

fetch('https://example.com/data')

.then( // code that handles the response )

.catch( // code that runs if the server returns an error )

Response Interface - Very cool stuff “ return a promise”

The Fetch API introduced the Response interface that deals with the object that’s returned when the promise is fulfilled. Response objects have a number of properties and methods that allow us to process the response effectively.

const url = 'https:example.com/data';

fetch(url)

.then((response) => {

if(response.ok) {

return response;

}

throw Error(response.statusText);

})

.then( response => // do something with response )

.catch( error => console.log('There was an error!')

#### **File Responses**

The blob() method is used to read a file of raw data, such as an image or a spreadsheet. Once it has read the whole file, it returns a promise that resolves with a blob object.

Here is an example of how a file response promise would be resolved:

fetch(url)

.then( response => response.blob() ); // transforms the data into a blob object

.then( blob => console.log(blob.type) )

.catch( error => console.log('There was an error: ', error))

#### **JSON Responses**

JSON is probably the most common format for AJAX responses. The json() method is used to deal with these by transforming a stream of JSON data into a promise that resolves to a JavaScript object.

Here is an example of how a JSON response promise would be resolved:

fetch(url)

.then( response => response.json() ); // transforms the JSON data into a JavaScript object

.then( data => console.log(Object.entries(data)) )

.catch( error => console.log('There was an error: ', error))

## Concepts and usage

Fetch provides a generic definition of [Request](https://developer.mozilla.org/en-US/docs/Web/API/Request) and [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response) objects (and other things involved with network requests). This will allow them to be used wherever they are needed in the future, whether it’s for service workers, Cache API and other similar things that handle or modify requests and responses, or any kind of use case that might require

The fetch() method takes one mandatory argument, the path to the resource you want to fetch. It returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/API/Promise) that resolves to the [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response) to that request, whether it is successful or not. You can also optionally pass in an init options object as the second argument (see [Request](https://developer.mozilla.org/en-US/docs/Web/API/Request)).

Once a [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response) is retrieved, there are a number of methods available to define what the body content is and how it should be handled (see [Body](https://developer.mozilla.org/en-US/docs/Web/API/Body)).

You can create a request and response directly using the [Request()](https://developer.mozilla.org/en-US/docs/Web/API/Request/Request) and [Response()](https://developer.mozilla.org/en-US/docs/Web/API/Response/Response) constructors, but you are unlikely to do this directly. Instead, these are more likely to be created as results of other API actions (for example, [FetchEvent.respondWith()](https://developer.mozilla.org/en-US/docs/Web/API/FetchEvent/respondWith" \o "The respondWith() method of FetchEvent prevents the browser's default fetch handling, and allows you to provide a promise for a Response yourself.) from service workers).

### Aborting a fetch

Browsers have started to add experimental support for the [AbortController](https://developer.mozilla.org/en-US/docs/Web/API/AbortController" \o "The AbortController interface represents a controller object that allows you to abort one or more DOM requests as and when desired.) and [AbortSignal](https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal" \o "The AbortSignal interface represents a signal object that allows you to communicate with a DOM request (such as a Fetch) and abort it if required via an AbortController object.) interfaces (aka The Abort API), which allow operations like Fetch and XHR to be aborted if they have not already completed. See the interface pages for more details.

## Sending Information

We can also use Ajax to send information. This can be a variety of formats, but is usually a JSON string.

To illustrate this, we're going to create a very simple To Do list application that sends information about a task to a server in JSON format, then receives a response to confirm that the task has been saved on a server.

Unfortunately, we don't have a database to save our tasks to, so we're going to have to use a dummy site called[JSONPlaceholder](https://jsonplaceholder.typicode.com/). This spoofs the process of sending JSON data to a server, then receiving JSON data in response. It has a number of fake APIs that can be used to create fake examples of posts, comments, albums, photos, todos and users. We'll be using the fake todo API.

To get started, create an HTML document called 'todo.html' that contains the following code:

<!doctype html>

<html lang='en'>

<head>

<meta charset='utf-8'>

<title>To Do List</title>

</head>

<body>

<form id='todo' action='https://jsonplaceholder.typicode.com/todos' method='POST'>

<input type='text' name='task' placeholder='Add Task' autofocus required>

<button type='submit'>Add Task</button>

</form>

<script src='main.js'></script>

</body>

</html>

### Putting It All Together

We can use the Headers, Request and Response objects to put together a typical example that sets up the URL, Request and Headers before calling the fetch() method:

const url = 'https:example.com/data';

const headers = new Headers({ 'Content-Type': 'text/plain', 'Accept-Charset' : 'utf-8', 'Accept-Encoding':'gzip,deflate' })

const request = (url,{

headers: headers

})

fetch(request)

.then( function(response) {

if(response.ok) {

return response;

}

throw Error(response.statusText);

})

.then( response => // do something with response )

.catch( error => console.log('There was an error!') )

### Call and Apply Methods

The call() method can be used to set the value of this inside a function to an object that is provided as the first argument.

In the following example, the sayHello() function refers to an unspecific object called this that has a property called name :

function sayHello(){

return `Hello, my name is ${ this.name }`;

}

We can create some objects that have a name property, then use the call() method to invoke the sayHello() function, providing each object as an argument. This will then take the value of this in the function:

const clark = { name: 'Clark' };

const bruce = { name: 'Bruce' };

sayHello.call(clark);

<< 'Hello, my name is Clarke'

sayHello.call(bruce);

<< 'Hello, my name is Bruce'

If the function that’s called requires any parameters, these need to be provided as arguments after the first argument, which is always the value of this . For example, let's update the sayHello() function to give a more generalized greeting that’s provided as an argument:

function sayHello(greeting='Hello'){

return `${ greeting }, my name is ${ this.name }`;

}

sayHello.call(clark, 'How do you do');

<< 'How do you do, my name is Clark'

sayHello.call(bruce);

<< 'Hello, my name is Bruce'

If a function doesn’t refer to an object as this in its body, it can still be called using the call() method, but you need provide null as its first argument. For example, we could call the square() function using the call() method, like so:

square.call(null, 4)

<< 16

The apply() method works in the same way, except the arguments of the function are provided as an array, even if there is only one argument:

square.apply(null, [4])

<< 16

## Immediately Invoked Function Expressions

An**Immediately Invoked Function Expression**– or IIFE – (pronounced 'iffy') is an anonymous function that, as the name suggests, is invoked as soon as it’s defined. This is easily achieved by placing parentheses at the end of the function definition (remember we use parentheses to invoke a function). The function also has to be made into an expression, which is done by placing the whole declaration inside parentheses, as in this example:

(function(){

const temp = 'World';

console.log(`Hello ${temp}`);

})();

<< 'Hello World'

### Creating Self-contained Code Blocks

An IIFE can be used to enclose a block of code inside its own private scope so it doesn’t interfere with any other part of the program. Using IIFEs in this way means code can be added or removed separately. The example shows two blocks, A and B, that are able to run code independently of each other:

(function() {

// block A

const name = 'Block A';

console.log(`Hello from ${name}`);

}());

(function() {

// block B

const name = 'Block B';

console.log(`Hello from ${name}`);

}());

<< Hello from Block A

Hello from Block B

## Recursive Functions

A recursive function is one that invokes itself until a certain condition is met. It’s a useful tool to use when iterative processes are involved. A common example is a function that calculates the[factorial](http://en.wikipedia.org/wiki/Factorial)of a number:

function factorial(n) {

if (n === 0) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

This function will return 1 if 0 is provided as an argument (0 factorial is 1), otherwise it will multiply the argument by the result of invoking itself with an argument of one less. The function will continue to invoke itself until finally the argument is 0 and 1 is returned. This will result in a multiplication of 1, 2, 3 and all the numbers up to the original argument.

#### **Error-first Callbacks**

The code example above uses theerror-firstcallback style popularized by Node.js. In this coding pattern, callbacks have two arguments. The first is the error argument, which is an error object provided if something goes wrong when completing the operation. The second argument is any data returned by the operation that can be used in the body of the callback.

#### **Creating A Promise**

A promise is created using a constructor function. This takes a function called an**executor**as an argument. The executor initializes the promise and starts the asynchronous operation. It also accepts two functions as arguments: the resolve() function is called if the operation is successful, and the reject() function is called if the operation fails. The general layout of a promise can be seen in the code below:

const promise = new Promise( (resolve, reject) => {

// initialization code goes here

if (success) {

resolve(value);

} else {

reject(error);

}

});

### Async Functions

Async functions were added to the ES2017 specification. These functions are preceded by the async keyword and allow you to write asynchronous code as if it was synchronous. This is achieved by using the await operator before an asynchronous function. This will wrap the return value of the function in a promise that can then be assigned to a variable. The next line of code is not executed until the promise is resolved.

The example below shows how the loadGame() function can be written an async function:

async function loadGame(userName) {

try {

const user = await login(userName);

const info = await getPlayerInfo (user.id);

// load the game using the returned info

}

catch (error){

throw error;

}

}

### A Counter Example

Closures not only haveaccessto variables declared in a parent function's scope, they can also change the value of these variables. This allows us to do things like create a counter() function like the one in the example below:

function counter(start){

let i = start;

return function() {

return i++;

}

}