Creating Asynchronous HTTP Requests in JavaScript

This chapter will teach you how to retrieve data from a web server through HTTP requests.

We'll cover the following

In the previous chapter, we discussed synchronous vs asynchronous requests. Since synchronous requests block the calling process until their result is received, only asynchronous HTTP requests should be used when building a web application. However, asynchronous code can be tricky to write and to understand, since statements won't be executed in a linear and sequential fashion like with synchronous operations.

The `fetch()` Method

The best way to send asynchronous HTTP requests in JavaScript is to use the `fetch()` method. Here is its general usage form.

```
// Sends an asynchronous HTTP request to the target url
fetch(url)
  .then(() => {
    // Code called in the future when the request ends successfully
  })
  .catch(() => {
    // Code called in the future when an errors occurs during the request
  });
```

> You might encounter JavaScript code that uses an object called `XMLHttpRequest` to perform HTTP operations. This is a more ancient

technique now replaced by `fetch()`.

Under the Hood: Promises

When the `fetch()` method is executed, it immediately returns a promise, which is a wrapper for an operation whose result might be available in the future. A *promise* is in one of these states:

- *pending*: initial state, not *fulfilled* or *rejected*
- *fulfilled*: meaning that the operation completed successfully
- *rejected*: meaning that the operation failed

A JavaScript promise is an object with `then()` and `catch()` methods. `then()` is called when the promise is *fulfilled*. It takes the operation result as a parameter. On the contrary, `catch()` is called when the promise is *rejected*.

What's great about promises is that they can be chained together. Here's how you could perform a series of asynchronous operations in JavaScript.

```
getData()
  .then(a => filterData(a)) // Called asynchronously when getData() returns
  .then(b => processData(b)) // Called asynchronously when filterData() returns
  .then(c => displayData(c)) // Called asynchronously when processData() returns
  // ...
```

Example: Retrieving a Text File

Let's start with a very basic example: displaying the content of a text file located on a web server. The file URL is https://raw.githubusercontent.com/bpesquet/thejsway/master/resources/languages.txt and it has the following content.

> C++;Java;C#;PHP

Here's how to do this in JavaScript using `fetch()`.

```
fetch(
  "https://raw.githubusercontent.com/bpesquet/thejsway/master/resources/languages.txt"
)
```

```
  .then(response => response.text()) // Access and return response's text content
  .then(text => {


    console.log(text); // Display file content in the console
  });
```

The result of the asynchronous HTTP request created by `fetch()` comes under the the form of a `Response` object. This object has several methods to deal with the response of the HTTP call. The `text()` method used in this example reads the response's text content and returns another promise. Its result is managed by the second `then()` method, which simply displays the file's textual content in the console.

To learn more about the `Response` object, consult, as usual, the Mozilla Developer Network.

Dealing with Errors

By nature, external HTTP requests are subject to errors: network failure, missing resource, etc. Handling these errors is done by adding a `catch()` method to the `fetch()` call. A basic level of error handling is to log the error message in the console.

```
fetch("http://non-existent-resource")
  .catch(err => {
    console.error(err.message);
  });
```