# AJAX, XHR AND FETCH

## INTRODUCTION

In this lab we are going to demonstrate a range of techniques for making AJAX calls to load data into a webpage.

The techniques used are based around XHR (XMLHttpRequest). In Javascript XHR objects are used to interact with servers. The technique is also commonly known as AJAX (Asynchronous Javascript And XML). XML is not the only data format that can be called with XHR/AJAX. Many modern web applications make use of JSON (Javascript Object Notation) as their data format of choice.

With the ES6 upgrade to Javascript the new Fetch API can be used to replace traditional XHR calls. It uses modern techniques such as Promises to manage XHR calls.

## THE BENEFITS OF AJAX/XHR

The core benefit of AJAX/XHR is the ability of Javascript to make calls to data resources without the need for the user to manually refresh the page. Javascript can, either through user interactions or discretely through page logic, make calls to data resources and update the DOM based on the data received.

A traditional http call and response is managed by the browser:


With XHR the http call and response if managed by Javascript:


The concepts behind AJAX/XHR have led to the development of Single Page Applications that have 'one' HTML page that uses XHR calls to update the 'view' (the DOM) based on user interaction and application logic.

## KEY TERMINOLOGY

There is a lot of terminology and acronyms around these technologies. They are provided here for reference:

| | |
|---|---|
| **AJAX** | Asynchronous Javascript And XML<br>A clever 'backronym' for the early use of XHR that was focused on XML data transfer. Largely inter-changeable with the less frequently used but technically more accurate label XHR. |
| **API** | Application Programming Interface<br>An interface with an application and a term not exclusive to AJAX.<br>In the AJAX world an API comes in the form of an URL than sends and/or receives data. These can also be described as Endpoints.<br>The API may return data in many different formats including plain text and XML. The most common format for web applications is JSON. |
| **Asynchronous** | Not happening at the same time. Javascript is single threaded. That is only one action can be processed at a time. AJAX is an asynchronous technology as the application will have to wait for data to be returned and how long that will take is unknown. Asynchronous technologies such as callbacks, Promises and async/await allow other code to run whilst the call is resolved. |
| **Callback** | A function set up to be executed once an AJAX call responds. |
| **Endpoint** | An alternative term for a URL called by the AJAX. |
| **JSON** | Javascript Object Notation<br>Data stored in the style of Javascript Objects. Very widely used data format. |
| **jQuery** | The ubiquitous Javascript library. It has AJAX specific methods such as $.getJSON(). |
| **ES6** | ECMAScript 6, also known ECMAScript 2015, was the second major revision to Javascript. ES6 introduced the Fetch API, Promises and Arrow => functions. |
| **Fetch** | The Fetch API is a modern incarnation of XHR introduced in the ES6 version of Javascript. It makes use of Promises. |
| **Promise** | An ES6 feature for use with asynchronous calls. A Promise represents a proxy for the data as not yet received. |
| **XHR** | XMLHttpRequest<br>The XMLHttpRequest object is the original technology. Initially developed by Microsoft and quickly adopted as a standard. |
| **XML** | Extensible Mark-up Language<br>A text-based data format based around nodes in a similar vein to HTML. HTML and XML are both derivatives of SMGL (Standard Generalised Markup).<br>XML is still widely used but has become less popular in Web Applications in comparison JSON. |

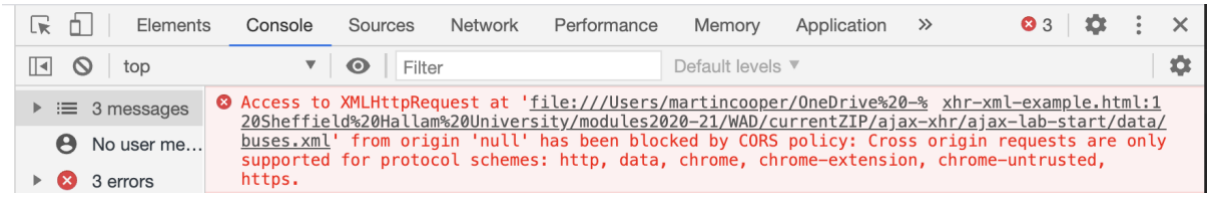
## LAB FILES

The lab consists of examples of the core techniques in AJAX/XHR. We shall build:

- A XMLRequest call to an XML file.
- A XMLRequest call to a JSON file.
- Use jQuery's `$.getJSON` to call a JSON file.
- Use the Fetch API to call a JSON file.
- Use the Fetch API to send and receive data from an API.

## CORS (CROSS ORIGIN REQUESTS)

Google Chrome will not allow locally run pages to make XHR calls unless they are using the http or https protocol. This is because of the CORS (Cross Origin Requests) policy. If you see an error as below it is because you are attempting to run the pages locally rather than via http/https.




## VIEWING YOUR PAGES

Given the requirement for http/https the pages needed to be tested on a webserver. This can be a "localhost" locally run webserver if available. A localhost server can be set up via the likes of XAMPP or IIS through Visual Studio.

## VIEWING YOUR PAGES THROUGH GITHUB PAGES

Setting up a localhost webserver can be a little complex. If you are using GitHub there is a straight-forward way to place pages on a http connection. When simple HTML files are placed in a GitHub Repository, they can be made available through a webserver. This feature is known as GitHub Pages and can be accessed via Settings. It is a handy way to run the examples. For more information see https://pages.github.com/.

The finished examples can be seen running on GitHub pages at https://mustbebuilt.github.io/ajax-lab-completed/index.html.

## QUICK START EXAMPLE

Open the file *fetch-example-no-arrow.html* and view it via http.

Locate the `<script>` at the bottom of the file. Note there is a call to an endpoint of *data/myData.json*.

This is managed by the Fetch API.

```
(function() {
    // IIFE
    let api = "data/myData.json";
    // let api = "//www.ywonline.co.uk/web/newincid.nsf/incidentsjson";
    fetch(api)
     .then(function(response){
            response.json()
    })
     .then(function(myJson){
            console.dir(myJson)
     })
     .catch(function(){
            console.log("damn that Jason")
   })

  })()
```

This basic example receives JSON data from the endpoint/URL and the `then()` methods are used to capture the response and output the data to the console.

In the following examples we'll see how techniques evolved to this Fetch API approach and see different techniques for adding the data to the DOM.

## USING THIRD PARTY DATA

In the code there is a link to a more complex set of data from a third-party API. Try switching the API value and view the new data output in the console.

```
var api = "//www.ywonline.co.uk/web/newincid.nsf/incidentsjson";
```

Note the URL above does not work on https. To try one of the https valid API below:

| |
|---|
| //got-quotes.herokuapp.com/quotes |
| //api.coronavirus.data.gov.uk/v1/data?filters=areaType=nation;areaName=england&structure={%22date%22:%22date%22,%22newCases%22:%22newCasesByPublishDate%22} |

## XML AND XMLREQUEST

The previous example used the Fetch API. This next example will demonstrate the earlier XHR approach and how data from an XML file can be loaded into a HTML file to be displayed. XHR was originally designed to work with XML – thus the name XHR (XMLHttpRequest).

Open the file *data/buses.xml*. This is an Extensible Markup language file. With XML the names of the nodes used are up to the developer, but the file must be well formed – that is it must follow XML rules of:

- XML documents must have a root element.
- XML elements must have a closing tag.
- XML tags are case sensitive.
- XML elements must be properly nested.
- XML attribute values must be quoted.

To load the XML we will need a XMLHttpRequest object. These are created with the `new XMLHttpRequest()` method.

A XMLHttpRequest object has a range of properties and methods a full list of which is available at https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

The core methods to make a XHR call are:

| | |
|---|---|
| open() | Initializes a request with a http method, target URL and asynchronous flag. |
| send() | Sends the request. |

Prior to the `send()` method an `addEventListener()` method should be set to 'listen' for the XHR related events. A key event is `readystatechange`.

The event handler function when called as the result of a `readystatechange` can check the XHR call status via the `readyState` property. A value of 4 indicates a completed call. This represents a 'callback'.

The `responseXML` property represents the returned XML. If not working with XML (the most likely alternative been JSON) then the `responseText` property is used.

Open the file *xhr-xml-example.html*.

Add a `<script>` tag before the closing `</body>` tag and add the following:

```
(function() {
    var api = "data/buses.xml";
    var myXHR = new XMLHttpRequest();
    var busList = document.querySelector(".busList");
    var adList = "";
    myXHR.open("GET", api, true);
    myXHR.addEventListener("readystatechange", showData);
    myXHR.send();
    function showData() {
      if (myXHR.readyState == 4) {
        console.dir(myXHR.responseXML);
        }
    }
  })();
```

The code is all included in an IIFE (Immediately Invoked Function Expression). A number of a variables are declared including a XHR object using the `new XMLHttpRequest()` method.

Save and test the page. The XHR call should be visible in the Google Chrome Console's Network tab. If the call is successful, the returned data is also visible through the console call. Amend the code in the `if` block to extract the data from the XML and add it to the DOM of the file.

```
if (myXHR.readyState == 4) {
        console.dir(myXHR.responseXML);
        var buses = myXHR.responseXML.getElementsByTagName("bus");
        // console.dir(buses)
        for(var bus of buses){
         // console.dir(bus)
         var start = bus.getElementsByTagName("start");
         console.dir(start);
         adList += "<li>Start: "
         adList += start[0].innerHTML
         adList += " Destination: "
         adList += bus.getElementsByTagName("destination")[0].innerHTML
         adList += "</li>"
        }
        busList.innerHTML = adList;
      }
```

It can be difficult traversing complex XML files so increasing data is in the JSON format.

## CALLING JSON DATA VIA XMLHTTPREQUEST

Open the file *xhr-json-example.html*. This will load the JSON data located at *data/myData.json*.

JSON is Javascript Object Notation. With JSON files data is stored in "name" : "value" pairs. Notice that the name and value are in double quotes. Data can get complex as a valid value can be another "name" : "value" pair. The "name" : "value" pairs can also be stored in arrays.

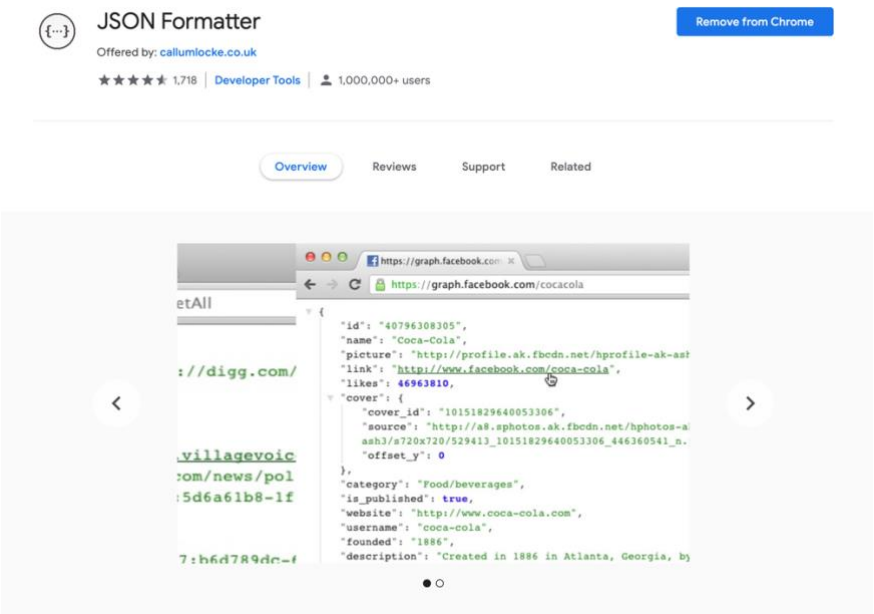
The data file consists of an array of JSON "name":"value" pairs.

```
[
 {
  "name": "Bob",
  "email": "bob@business.com",
  "age": 23
 },
 {
  "name": "Jane",
  "email": "jane@business.com",
  "age": 34
 },
 {
  "name": "Ian",
  "email": "ian@business.com",
  "age": 45
 },
 {
  "name": "Joe",
  "email": "joe@business.com",
  "age": 60
 }
]
```

Although based on Javascript Object, JSON differs in a number of subtle ways.

- Javascript Objects can have methods. JSON as pure data does not.
- Javascript Objects properties need not be quoted. JSON 'names' are always placed in double quotes.

You can use tools such as https://jsonlint.com/ to check your file is valid JSON. There are also plugins for Google Chrome to improve the viewing of JSON files such as JSON Formatter.




## MANIPULATING JSON DATA

To load the *data/myData.json* data into the HTML page with XMLHttpRequest the code is very similar to the previous example except that the data is retrieved with the property `responseText`.

With JSON data called via XHR it is initially a string of text. To convert it to an object (in this case an array) we use `JSON.parse()`. Once the string of text is parsed an Array is returned.

```
▼Array(4) i                                                    xhr-json-example.html:53
  ▶0: {name: "Bob", email: "bob@business.com", age: 23}
  ▶1: {name: "Jane", email: "jane@business.com", age: 34}
  ▶2: {name: "Ian", email: "ian@business.com", age: 45}
  ▶3: {name: "Joe", email: "joe@business.com", age: 60}
    length: 4
  ▶__proto__: Array(0)
```

The reverse of `JSON.parse()` is `JSON.stringify()` that is used to convert JSON into a string. This is mainly done to allow the data to be sent via HTTP methods.

```
[{"name":"Bob","email":"bob@business.com","age":23},{"nam      xhr-json-example.html:66
e":"Jane","email":"jane@business.com","age":34},{"name":"Ian","email":"ian@business.co
m","age":45},{"name":"Joe","email":"joe@business.com","age":60}]
```

The following code snippet illustrates the concepts of `JSON.parse()` and `JSON.stringify()` through messages sent to the console.

Add a `<script>` tag before the closing `</body>` tag and add the following:

```
(function() {
    var api = "data/myData.json";
    var myXHR = new XMLHttpRequest();
    myXHR.open("GET", api, true);
    myXHR.addEventListener("readystatechange", showData);
    myXHR.send();
    var staffList = [];
    var staffStr = ""
    var tableBody = document.querySelector("tbody");
    function showData() {
     if (myXHR.readyState == 4) {
       console.dir(myXHR.responseText);
       // data from server is a string make into an object
       // parse to make an object
       staffList = JSON.parse(myXHR.responseText);
       console.dir(staffList);
       staffList.forEach(function(staffMember){
         staffStr +=
         `<tr>
         <td>${staffMember.name}</td>
         <td>${staffMember.email}</td>
         <td>${staffMember.age}</td>
         </tr>`
       })
       tableBody.innerHTML = staffStr
       // JSON.stringify does the opposite
       // Javascript object is turned into a string
       var myStr = JSON.stringify(myJson);
       console.dir(myStr);
     }
    }
  })();
```

Note the use of a `forEach()` to loop the array of objects in the JSON file. A `forEach()` loop calls a function for each element in the array.

This file also uses the ES6 backtick `` ` `` syntax to make the concatenation of the HTML to the DOM a little easier. The backticks `` ` `` denote a template literal where variables can be place in `${…}` syntax.

## USING THIRD PARTY DATA

Change the above example to use one of the external endpoints used previously.

Use the console to investigate the data return and amended the code adding data to the DOM.

## XHR CALLS WITH JQUERY

The popular jQuery library applies it 'write less, do more' maxim to XHR calls. The library has a range of XHR call methods `$.get()`, `$.post()`, `$.getJSON()` and `$.ajax()`.

Open the file *jquery-example.html*.

Notice that the jQuery library has been called in the `<head>` of the file.

Add a `<script>` tag before the closing `</body>` tag and add the following:

```
(function() {
    var api = "data/myData.json";
    var staffStr = ""
    var tableBody = document.querySelector("tbody");
    $.getJSON(api, function(myData) {
      console.dir(myData);
        myData.forEach(function (staffMember) {
        staffStr +=
          `<tr>
          <td>${staffMember.name}</td>
          <td>${staffMember.email}</td>
          <td>${staffMember.age}</td>
          </tr>`
      })
      tableBody.innerHTML = staffStr
    });
  })();
```

The `$.getJSON` jQuery method allows for more streamlined code.

Notice the `$.getJSON()` method's parameters are an endpoint/URL parameter and a callback function. In the above example the callback is an inline function and receives the data in the form of the parameter `staffMember`.

Try changing the above to use the external Endpoint from jwonline as we previously did with the XMLHttpRequest example.

## THE FETCH API

The jQuery methods streamline the code but are reliant on a Library. Modern browsers that support new ES6 features can make use of the fetch API. The fetch API is designed to update XMLHttpRequest with a flexible feature set.

One of the key differences is that the fetch API uses Javascript Promises to handle callbacks.

A Promise is a technique for working with asynchronous calls. A Promise is a way to set a proxy for a value that you have not yet received. An asynchronous method can 'promise' to return the called value at some point. A Promise object has a `then()` method that is called after a promise is resolved.

A Promise has three states of pending, fulfilled and rejected. When any of these options happen a `then()` method can be called.

```
▼Promise i                                                    fetch-demo1.html:20
  ▶__proto__: Promise
   [[PromiseState]]: "fulfilled"
   [[PromiseResult]]: undefined
```

The `fetch()` method has one mandatory parameter, the URL of an endpoint. The `fetch()` method then returns a Promise. Once fulfilled/resolved a chained `then()` method is called.

In the `then()` method a Promise is resolved in the form of the Response Object.

This has various methods for accessing the response.

```
▼Response i                                                   fetch-demo1.ht
    body: (...)
    bodyUsed: false
  ▶headers: Headers {}
    ok: true
    redirected: false
    status: 200
    statusText: "OK"
    type: "basic"
    url: "http://mustbebuilt.co.uk/SHU/WAD/demo/es6/myData.json"
  ▶__proto__: Response
>
```

The methods of the Response Object all return Promises so a second `then()` is chained to receive the data.

As such a basic fetch call could be summarised as:

```
fetch(CALL THIS URL)
.then(WAIT FOR PROMISE TO BE RESOLVED INTO A RESPONSE OBJECT)
.then(WAIT FOR PROMISE TO RESOLVE RESPONSE OBJECT INTO DATA)
```

The quick start fetch example at the start of the lab made use of traditional functions:

```
let api = "data/myData.json"
 fetch(api)
  .then(function(response){
                    return response.json()
  })
         .then(function(myJson){
         console.dir(myJson)
  })
  .catch(function(){
                    console.log("damn that Jason")
 })
```

Arrow functions allow the replacement of the function keyword as follows:

```
        let api = "data/myData.json";
  fetch(api)
    .then((response) => response.json())
    .then((myJson) => console.dir(myJson))
    .catch(() => console.log("damn that Jason"))
```

Features of arrow functions are:

- the `function` keyword is replaced with `=>`;
- single line functions do not need curly braces `{…}`;
- single line functions that return a value do not need a `return` keyword;
- if only one parameter is sent there is no need for parenthesis `(…)`;

**Tip:** Beware the use of `this` with arrow functions it has different binding behaviour. See https://medium.com/@josephcardillo/arrow-functions-and-this-in-es6-4f1d350a85cf.

Open the file *fetch-example.html*.

Add a `<script>` tag before the closing `</body>` tag and add the following:

```
// IIFE
let api = "data/myData.json";
fetch(api)
  .then(response => response.json())
  .then(myJson => {
    console.dir(myJson);
    document.getElementById("myData").innerHTML = myJson.map(obj => {
        let myStr = `<tr>
          <td>${obj.name}</td>
          <td>${obj.email}</td>
          <td>${obj.age}</td>
        </tr>`
        return myStr
      }).join("")
      // use join with template literals to remove extra comma
  })
  .catch(() => console.log("damn that Jason"))

})()
```

Notice the use of the Javascript `map()` method. The `map()` method returns a new array containing the results of the provided function for each element in the original array. In the above example every object sent the to the array produces a string representing a table row. As such the `map()` will produce an array of HTML table rows. The `join()` method converts the array into a string to allow it to be added to the DOM.

Change the above example to use one of the external endpoints used previously.

Use the console to investigate the data return and amended the code adding data to the DOM.

## AWAIT ASYNC

Even with the improvment of Promises, the callbacks can become difficult to read.

The keywords `await` and `async` can be used to set a function to be asynchronous.

The `async` keyword is placed before a function and ensures a function always returns a Promise.

The `await` keyword **can only be used** in an `async` function and it makes the Javascript wait until that Promise is resolved before returning the result. Properties set as `await`, 'wait' for the asynchronous call to finish.

Open the file *await-example.html*.

Add a `<script>` tag before the closing `</body>` tag and add the following:

```
(function () {
  var api = "data/myData.json";
  async function getData(api) {
    const fetchResult = fetch(api);
    const response = await fetchResult;
    const jsonData = await response.json();
    document.getElementById("myData").innerHTML = jsonData
      .map((obj) => {
        let myStr = `<tr>
            <td>${obj.name}</td>
            <td>${obj.email}</td>
            <td>${obj.age}</td>
          </tr>`;
        return myStr;
      })
      .join("");
  }
  getData(api);
})();
```

## SENDING DATA TO AN ENDPOINT

All the above technologies support sending data as well as requesting it.

Open the file *fetch-post.html*.

Add a `<script>` tag before the closing `</body>` tag and add the following:

The Fetch API calls an endpoint set to calculate age from a date of birth.

```
(function() {
    var api = "https://mustbebuilt.co.uk/SHU/WAD/demo/xhr/posted-api.php";
    var myForm = document.getElementById("myForm");
    myForm.addEventListener("submit", ev => {
      ev.preventDefault();
      var formData = new FormData(myForm);
      var sendObject = {};
      console.dir(formData);
      formData.forEach(function(value, key) {
        sendObject[key] = value;
      });
      console.dir(sendObject);
      var sendStr = JSON.stringify(sendObject);
      console.dir(sendStr);

      //
      fetch(api, {
        method: "post",
        body: sendStr,
        headers: {
          "Content-Type": "application/json" }
        }).then(response => response.json())
        .then(myJson => {
          console.dir(myJson);
          document.querySelector(".age").innerText = myJson.age
          })
    });
  })();
```

Notice how `fetch()` has a second parameter that outlines the options for the http request. In this case it is a `POST` request, the payload is the `sendStr` var and the `headers` indicate the data is of type JSON.

Try amending the above to use `async` and `await`.