





OVERVIEW



- What is Asynchronous JavaScript?
- Asynchronous JavaScript-enabling technologies
- Client and Server architecture
- JSON
- Promises
- The Fetch API
- Async/Await
- Appendix - XMLHttpRequest

QA What is asynchronous JavaScript?

- A methodology for creating rich Internet applications
- Used to create highly-responsive applications
- Rich content and interactions
- A client-focused model
 - Uses client-side technologies – JavaScript, CSS, HTML
- A user-focused model
 - Asynchronous behaviour based on user interactions
 - User-first' development model
- An asynchronous model
 - Communications with the server are made asynchronously
 - User activity is not interrupted

Users of web applications increasingly expect a user experience that provides a high-level of interactivity on as close a level to a desktop application as possible.

The typical server-bound web application cannot easily provide this because it has to defer to the server for any significant updates to the user interface.

Ajax enables us to create a highly-responsive, rich user interface. It does this by enabling several design options through standard-based technologies.

Key to this are the client-side technologies: HTML with JavaScript and CSS for the styling of our pages. Due to developments and standardisation of these technologies, we can change the focus of our application from server-centric to client-centric.

We focus our development efforts on user interactions with our application and have a user-driven, event-based model, so that we can concentrate as much as possible on the user experience.

Through asynchronous communication with the server, we are able to retrieve data or HTML fragments that we can then insert into the DOM programmatically. The main benefits of asynchronous communication in this scenario are that we are only transmitting a small amount of information and are semi-coupled with the server; more importantly, user activity and interactions with our application are not interrupted during this request process.

QA Four principles of asynchronous JavaScript

- The browser hosts an application
- A richer document is sent to the browser
- JavaScript manages the client-side interaction with the user
- The server delivers data
 - Requests for data - not content - are sent to the server
 - Less network traffic and greater responsiveness
- User interaction can be continuous and fluid
 - The client is able to process simple user requests
 - Near instantaneous response to the user

When we work in an Asynchronous environment, we have to think differently to how we may usually think with a server-bound application. We need to change our idea of where the application is running from the server to the client. When we add Asynchronous functionality to a web application, we will be adding a certain amount of code that we would not add normally to a typical server-bound web application. This means that the browser will essentially be hosting an application and not simply content. We will be sending a richer document to the browser, including JavaScript files and then we will manage interaction from the client by making requests for data (not content) and using this to update the DOM within the browser. This mechanism creates less network traffic and, therefore, allows us to show greater responsiveness within our application.

User interaction is simplified in a similar way to event-driven desktop applications in that the client browser using our JavaScript code is able to process the simple (or even relatively complex) requests and make an asynchronous request for data (if necessary), providing a near-instantaneous response to the user due to the decreased traffic and potentially lower processing overhead on the server.

However, we will be writing a lot more code and constructing an application rather than just a series of effects on the client, so we are in the realm of real coding and we need to take a disciplined approach to this.

Client-centric development model

- Primarily implemented on the client
- Presentation layer driven from client script
- Uses HTML, CSS and JavaScript
- This means:
 - First request
 - A smarter, more interactive application is delivered from the server
- Subsequently:
 - Less interaction between the browser and the server
- Which:
 - Encourages greater interaction with the user
 - Provides a richer, more intuitive experience

As mentioned, there are two main development models that we can consider within Asynchronous applications.

The first is the client-centric development model.

This model has the application mostly implemented on the client with the user interface drive from client script with JavaScript driving the DOM.

At first request, we will have a "smarter" application delivered from the server. By this, we mean that more code will be downloaded so that the client browser can do more of the processing of the application itself. In turn, this reduces the amount of interaction that must happen during the course of the applications life – this encourages a much greater interaction with the user and provides a richer, more intuitive experience.

This is the model we would use for a pure Asynchronous application, where we are communicating with data and manipulating the DOM programmatically.

QA Server-centric development model

- Primarily implemented on the server
- Application logic and most UI decisions remain on the server
- This means:
 - First request
 - A regular page is retrieved from the server
- Subsequently:
 - Incremental page updates are sent to the client
- Which:
 - Reduces latency and increases interactivity
 - Gives the opportunity to keep core UI and application logic on the server

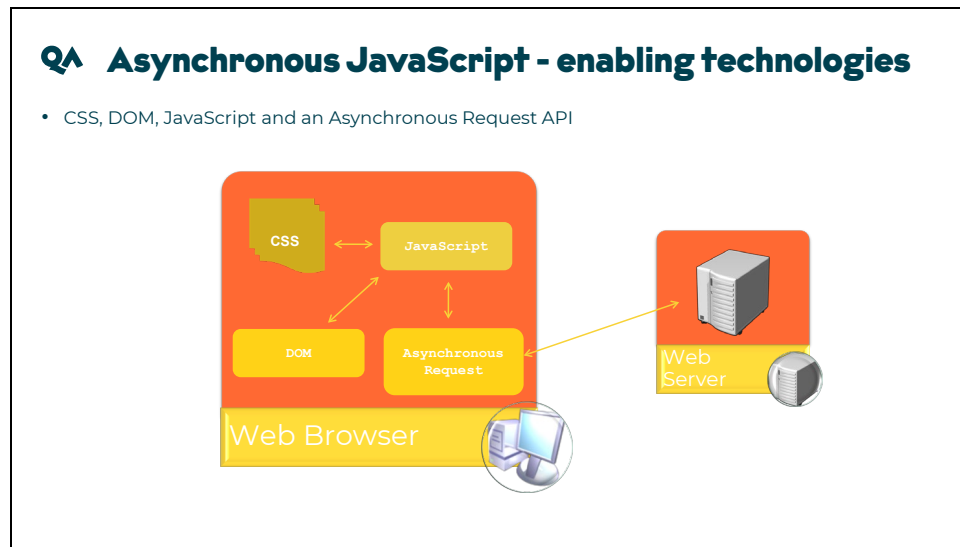
The second model is the server-centric development model.

This is also known as, the partial page update model.

Most of the application logic and UI decisions are still made by the server. On application startup, a normal page is retrieved from the server. After that page has been retrieved, the client interacts with the page and requests are sent to the server for fragments of HTML with which to update the page.

This happens incrementally, so reducing latency and increasing the level of interactivity possible without a full-page refresh. It also gives the opportunity to keep the core logic of the Application and UI on the server which may be a required decision for some sensitive applications.

Within any one application, it is possible to have the two models working side-by-side on different pages with one page more suited to the server-centric model and the other more suited to the client-centric model. It would be very unusual and indeed very confusing to mix the two approaches on the same page.

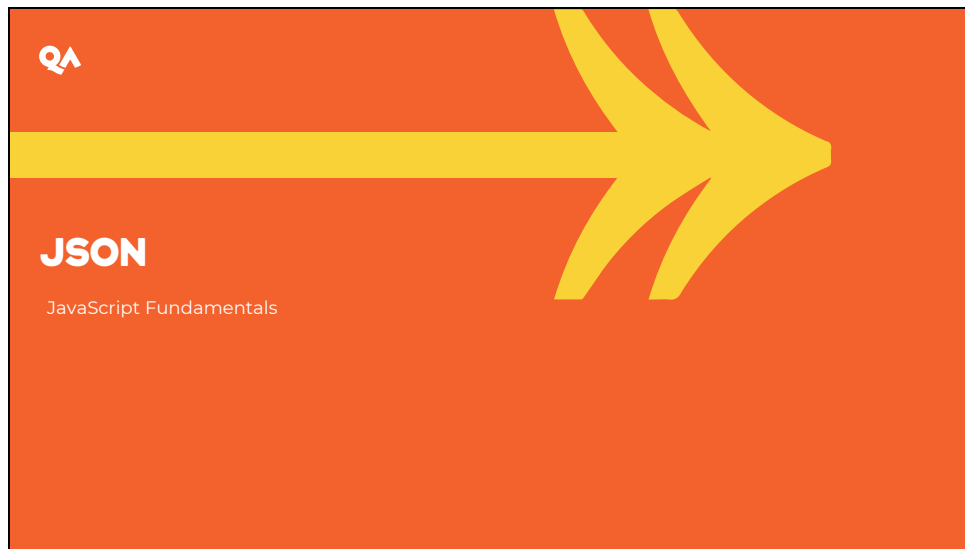



The diagram above shows the main enabling technologies of Asynchronous JavaScript.

At the client end, we have the web browser, which will host the document that is requested. This document is made available via a Document Object Model (DOM), so that we can programmatically manipulate the document using JavaScript and also hook into the events of the browser, document and constituent elements.


In addition, we can use an Asynchronous Request API to initiate requests to a server in order to retrieve data, then use that data through JavaScript to update the document through the DOM.

Slide 8





JavaScript Object Notation (JSON)



- Lightweight data-interchange format
 - Compared to XML
- Simple format
 - Easy for humans to read and write
 - Easy for machines to parse and generate
- JSON is a text format
 - Programming language independent
 - Conventions familiar to programmers of the C-family of languages, including C# and JavaScript

Transferring data can be a cumbersome task. XML is all well and good; however, it requires a DOM parser in order to read/write and is not easily realised into object format.

JavaScript Object Notation (or JSON) is a lightweight data interchange format that is easy to read and write and, more importantly, easy for machines to parse and to generate.

JSON is a text format that is programming-language-independent and uses conventions familiar to C family programmers. To JavaScript, JSON looks and behaves as an associative array and so can be parsed (using eval) and turned into a fully functioning object, which is very easily consumed.

QA JSON structures

- Universal data structures supported by most modern programming languages
- A collection of name/value pairs
 - Realised as an object (associative array)
- An ordered list of values
 - Realised as an array
- JSON object
 - Unordered set of name/value pairs
 - Begins with { (left brace) and ends with } (right brace)
 - Each name followed by a : (colon)
 - Name/Value pairs separated by a , (comma)

```
{
  "results": [
    {
      "home": "React Rangers",
      "homeScore": 3,
      "away": "Angular Athletic",
      "awayScore": 0
    },
    {
      "home": "Ember Town",
      "homeScore": 2,
      "away": "React Rangers",
      "awayScore": 2
    }
  ]
}
```

JSON consists of structures that are supported by most modern programming languages and so is immediately accessible to most.

It is an associative array (name/value pairs) and can contain an ordered list of values as an array.

The overall JSON object consists of an unordered list of name/value pairs contained within curly braces with each name and value pair separated by a colon and the name/value pairs separated by a comma.

QA JSON and JavaScript

JSON is a subset of the object literal notation of JavaScript.

- Can be used in the JavaScript language with no problems

```
let myJSONObject = {  
  "searchResults": [  
    {  
      "productName": "Aniseed Syrup",  
      "unitPrice": 10  
    },  
    {  
      "productName": "Alice Mutton",  
      "unitPrice":  
        39  
    }  
  ]  
};
```

JSON is a subset of the object-literal notation of JavaScript and so can be used (as shown above) in JavaScript with no problems.

The object realised in the above example can be accessed using either dot or subscript operators as shown in the second example.

QA The JSON object

- The JSON object is globally available
- The **parse** method takes a string and parses it into JavaScript objects
- The **stringify** method takes JavaScript objects and returns a string
- Makes working with JSON data a trivial affair

```
let obj = JSON.parse('{ "name": "Adrian" }');  
console.log(obj.name); //returns Adrian
```

```
let str = JSON.stringify({ name: "John" });
```

There are a series of overloaded methods for the type:

JSON.parse(text) – Converts a serialised JSON string into a JavaScript object.

JSON.parse(text, translate) – Uses a translation function to convert values or remove them entirely.

JSON.stringify(obj) – Converts an object into a serialised JSON string.

JSON.stringify(obj, ["white", "list"]) – Serialises only a specific white list of properties.

JSON.stringify(obj, translate) – Serialises the object using a translation function.

JSON.stringify(obj, null, 2) – Adds the specified number of spaces to the output, printing it evenly.

QA RESTful services

RESTful services are commonly used to supply data to web applications.

- **RE**presentational **S**tate **T**ransfer
 - Essentially they are a server, possibly attached to a Database that returns the requested data:
- Make a request to a URL – can CRUD
 - Create
 - Read
 - Update
 - Delete
- Response will be in the form of JSON

QA Mocking a RESTful service

- json-server is an npm package that allows you to:

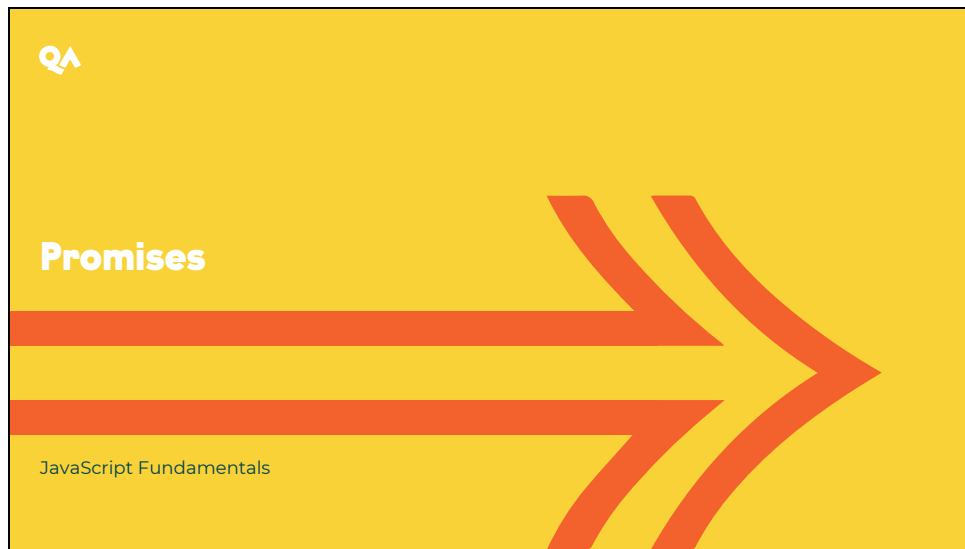
“Get a full fake REST API with zero coding in less than 30 seconds”


- Need to install the package (globally if it will be used frequently)
- Need to supply it with a properly-formed .json file
- Runs on http://localhost:3000 by default (can be changed when spinning up)
- Allows full CRUD requests and saves changes to .json file

<https://www.npmjs.com/package/json-server>


QA QuickLab 16a – create some JSON

- Generate a small JSON file to use with json-server
- Install and run json-server





WHAT IS A PROMISE?



A placeholder for some data that will be available: immediately, some time in the future or possibly not at all.

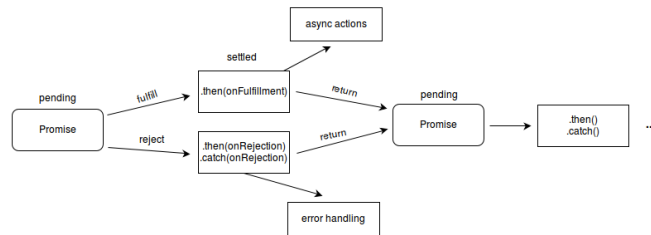
- JavaScript is executed from the top down
 - Each line of code evaluated and executed in turn

What happens if needed data is potentially not available immediately?

- Most commonly we may be waiting for some data to come from a remote endpoint
- Need some way to be able to execute code when the data is available or deal with the fact that it will never be available
- This is the job of a promise

QA Promises

- A promise is the representation of an operation that will complete at some unknown point in the future
- We can associate handlers to the operation's eventual success (or failure)
- Exposes `.then` and `.catch` methods to handle resolution or rejection



QA Promises

Construct a new promise passing in an 'executor' function which will be immediately evaluated and is passed both resolve and reject functions as arguments.

```
let newPromise = new Promise((resolve, reject) => { });
```

The Promise is in one of three states:

- Pending
- Fulfilled - Operation completed successfully
- Rejected - Operation failed

Which we can attach associated handlers too:

- **.then(onFulfilled, onRejected)** appends handlers to the original promise, returning a promise resolving to the return of the called handler or the original settled value if the called handler is undefined
- **.catch(onRejected)** same as then but only handles the rejected condition

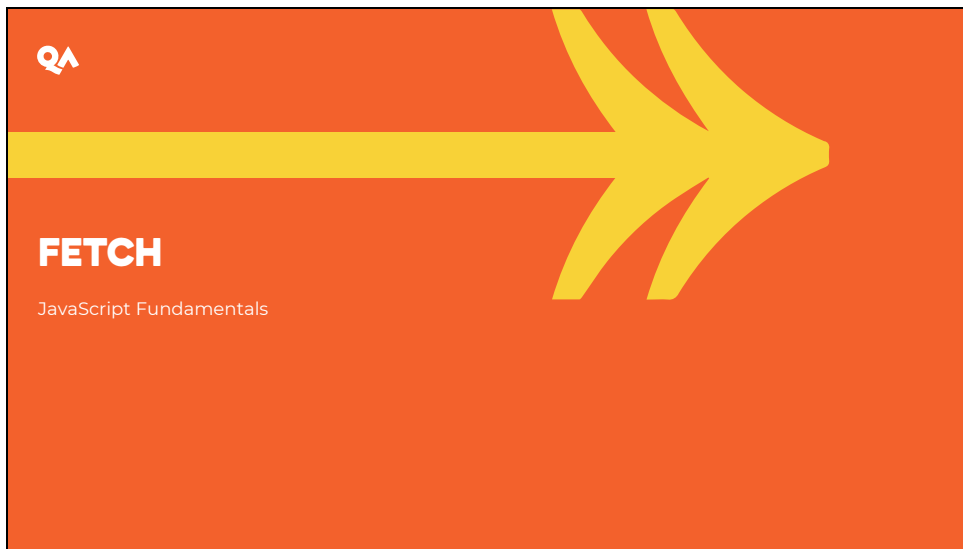
QA Promises: example


```
let aPromise = new Promise((resolve, reject) => {
  let delayedFunc = setTimeout(() => {
    //whether it resolves or rejects is unknown
    (Math.random() < 0.5) ? resolve("resolved") : reject("rejected");
  }, Math.random() * 5000); //function will return sometime: 0-5s
});

aPromise
  .then(
    //resolved
    data => {
      console.log(v);
    },
    //rejected
    error => {
      console.log(v);
    }
  );
```


QA QuickLab 16b – promises

- Experiment with promises





Fetch



- “The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global **fetch()** method that provides an easy, logical way to fetch resources asynchronously across the network”
- In short, **Fetch** provides the functionality hitherto provided by **XMLHttpRequest**
- It greatly simplifies making requests and dealing with responses
- **Fetch** requests return **Promises**
- **Fetch** is supported from Chrome 42, Edge 14, Firefox 39, Safari 10.1, Opera 29

XMLHttpRequest is an older technology, generally used to implement AJAX (Asynchronous JavaScript And XML). See the appendix for information on how Asynchronous requests using the XMLHttpRequest object are made.

QA Fetch

- Making a **fetch** request can be as simple as passing a URL and chaining appropriate `.then` and `.catch` methods onto the return

```
fetch('https://www.qa.com/courses.json')
  .then(response => response.json())
  .then(myJson => console.log(myJson))
  .catch(err=> console.error(err))
```

- Note how we don't have to use `JSON.parse` as response objects have a `.json()` method which returns a **Promise** that resolves to with the result of parsing the body text of the response as JSON
- By default, a **fetch** request is of type **GET**

QA Fetch – full example

- We can make more complex requests using the second argument, an init object that allows us to control a number of aspects of the request – including any data we wish to include with it

```
fetch(url, {  
  body: JSON.stringify(data),  
  // must match 'Content-Type' header  
  cache: 'no-cache',  
  // *default, no-cache, reload, force-cache, only-if-cached  
  credentials: 'same-origin', // include, same-origin, *omit  
  headers: {  
    'content-type': 'application/json'  
  },  
  method: 'POST',           // *GET, POST, PUT, DELETE, etc  
  mode: 'cors',             // no-cors, cors, *same-origin  
  redirect: 'follow',       // manual, *follow, error  
  referrer: 'no-referrer',  // *client, no-referrer  
})  
.then(response => response.json())  
.then(myJSON => console.log(myJSON))  
.catch(err => console.log(err));
```

QA Fetch

- A **fetch** promise does not **reject** on receiving an error code from the server (such as 404) instead it **resolves** and will have a property **response.ok = false**.
- To correctly handle **fetch** requests, we would need to also check whether the server responded with a **response.ok === true**

```
fetch(url)
  .then(response => {
    if (response.ok) {
      //do things
    }
    else {
      //handle error
    }
  });
```

QuickLab 16c – fetch

- Use the Fetch API to send and receive data



Async functions

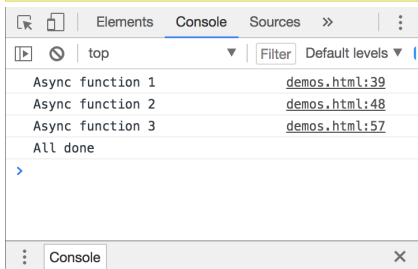
- An **async** function will return a **Promise** which **resolves** with the value returned by the function, or **rejected** with any uncaught exceptions
- An **async** function can contain an **await** expression which **pauses** the execution of the **async** function until completion of the **Promise** and then resumes



QA Async Functions

```
async function doThings() {  
  await asyncFunc1();  
  await asyncFunc2();  
  await asyncFunc3();  
  return "All done";  
}
```


```
doThings().then(console.log);
```



```
async function asyncFunc1() {  
  return new Promise((resolve, reject)=>{  
    setTimeout(()=>{  
      console.log('Async function 3');  
      resolve();  
    }, 3000);  
  });  
}  
  
async function asyncFunc2() {  
  return new Promise((resolve, reject)=>{  
    setTimeout(()=>{  
      console.log('Async function 3');  
      resolve();  
    }, 2000);  
  });  
}  
  
async function asyncFunc3() {  
  return new Promise((resolve, reject)=>{  
    setTimeout(()=>{  
      console.log('Async function 3');  
      resolve();  
    }, 1000);  
  });  
}
```

QA QuickLab 16d – async/await


- Use async/await to be able to send and receive data



REVIEW

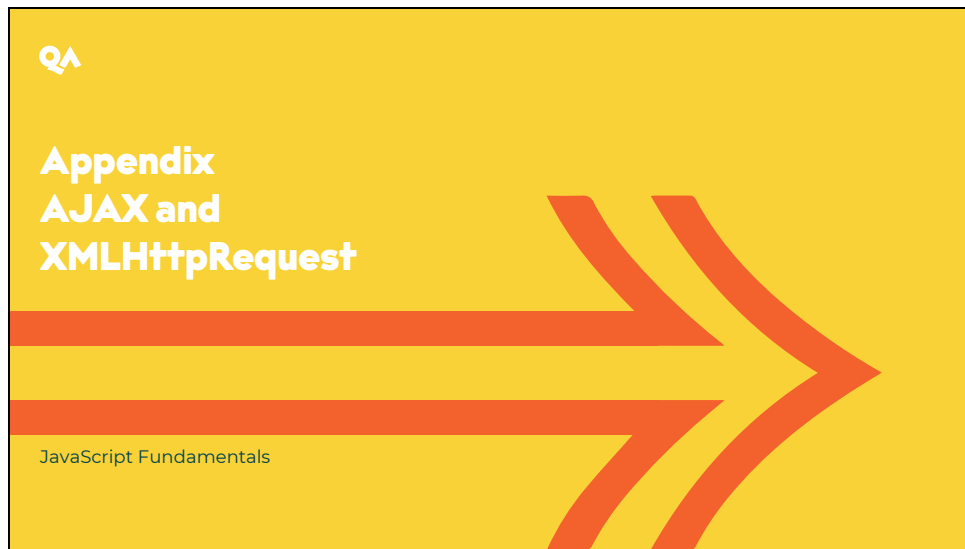
Asynchronous JavaScript is...


- A methodology for creating rich Internet applications
- A client and user-focused model
- A methodology that enables asynchronous requests
- Fetch API
- **async** functions and the **await** declaration




QA Hackathon Part 2

- In this part Hackathon, you will build on a partially developed solution (whether that be your previous iteration or the provided starting point) for QA Cinemas' website by allowing submission of the user data from the form to a remote backend. This should be simulated by using json-server. All the necessary tools, knowledge and techniques have been covered in the course so far
- This part of the Hackathon is intended to help you develop your skills and knowledge to be able to use JavaScript to submit data from a 'Sign-Up' form for users of the QA Cinemas website





**AJAX
AND
XMLHTTPREQUEST**



- Asynchronous JavaScript And XML
- Still used commonly to describe asynchronous calls
- XMLHttpRequest
- Object required to make asynchronous calls in ES5 and below

XMLHttpRequest – overview

Handles the request process

- w3c specification
- See <http://www.w3.org/TR/XMLHttpRequest/>
- Defines an API that provides scripted client functionality for transferring data between a client and a server

Benefits

- Simple to use
- Can be used for any request type, e.g. GET, POST
- Can be used synchronously or asynchronously

- Request headers can be added
- Response headers can be read
- Support in all modern browsers

The w3c document referenced above defines an interface that is implemented by the XMLHttpRequest object within conformant browsers. This includes all modern browsers.

The object contains a simple API for creating most types of request (GET, POST, HEAD, PUT, DELETE, OPTIONS) and can be used over HTTP or HTTPS. It can be used for making synchronous or asynchronous requests.

Like any typical HTTP request, we can manipulate the headers by adding extra entries to the request and we can read the response headers.

QA XMLHttpRequest – requests

- **open** method
- Sets up the **XMLHttpRequest** object for communications

```
request.open(sendMethod, sendUrl[, booleanAsync, stringUser, stringPwd]);
```

- **send** method
- Initiates the request
- **abort** method
- Cancels a request currently in process
- **setRequestHeader** method
- Adds custom HTTP headers to the request
- Used mainly to set content type

```
request.send([varData]);
```

```
request.setRequestHeader(sName, sValue);
```

There are four named request methods; open, send, abort and setRequestHeader.

The open method takes a string indicating the method (GET, POST etc...). It also requires the Url as a string. The other parameters are optional; however to make an asynchronous call you will need to pass true as the bAsync parameter. You can also, optionally, provide a username and password to use for authentication.

The send method initiates the request and has three ways of invocation. You can send the request without any data (e.g. when invoking for a GET request). In addition, the varData parameter can contain either a string containing name/value pairs as would be the body of the request, or it can contain an XML Document.

The abort method allows us to cancel a request that is currently processing.

The setRequestHeader method is used to add headers to the request and is used mainly to set the content type when we want to POST data. Both parameters are strings.

QA XMLHttpRequest – responses

- **readystatechange** event
- Fires for each stage in the request cycle
- **readyState** property – Progress indicator (0 to 4)
 - Most important is 4 (Loaded); you can access the data
- **responseXXX** property - retrieves the response
- **responseText** – as a string
- **responseBody** – as an array of unsigned bytes
- **status** property, **statusText** property
- Return the HTTP response code or friendly text respectively
- **load** event
- You can listen to this event in IE9 and above rather than check **readyState** on every **readystatechange** event

The key to retrieving or handling the response is to hook into the **readystatechange** event of the **XMLHttpRequest** object.

This event fires for each stage in the request lifecycle and can be used to retrieve the content of the response. The **readyState** property indicates what stage the request/response is at.

You can check the HTTP status code using the **status** property and obtain a friendly text description from the **statusText** property. To retrieve the data, you use the **responseText**, **responseXML** methods. You can interrogate the response headers by using **getResponseHeader** or **getAllResponseHeaders**.

QA XMLHttpRequest – example

- Using **XMLHttpRequest**
 - Create a new **XMLHttpRequest** object
 - Set the request details using the open method
 - Hook-up the load event to a callback function
 - Easiest way is to use an anonymous function
- Send the request

```
let request = new XMLHttpRequest();
request.open(
  "GET",
  "SomeHandler.ashx", true);

request.onload = () => {
  if (request.status == 200) {
    // Do something with
    // request.responseText
  }
}

request.send();
```

In the example above, we are making a request to a simple handler called `SomeHandler.ashx`.

First, we instantiate an `XMLHttpRequest` object by calling its constructor.

We then invoke `open` with our `GET` method and the `Url` and state that the request will be asynchronous.

Next, we attach a handler to the load event. You can provide a function name here but, in the example, we have used an arrow function.

We then check to see that the request completed successfully (HTTP response code 200 – OK).

We can then do something with the `responseText/responseXML`.

Finally, we initiate the request by invoking the `send` method.