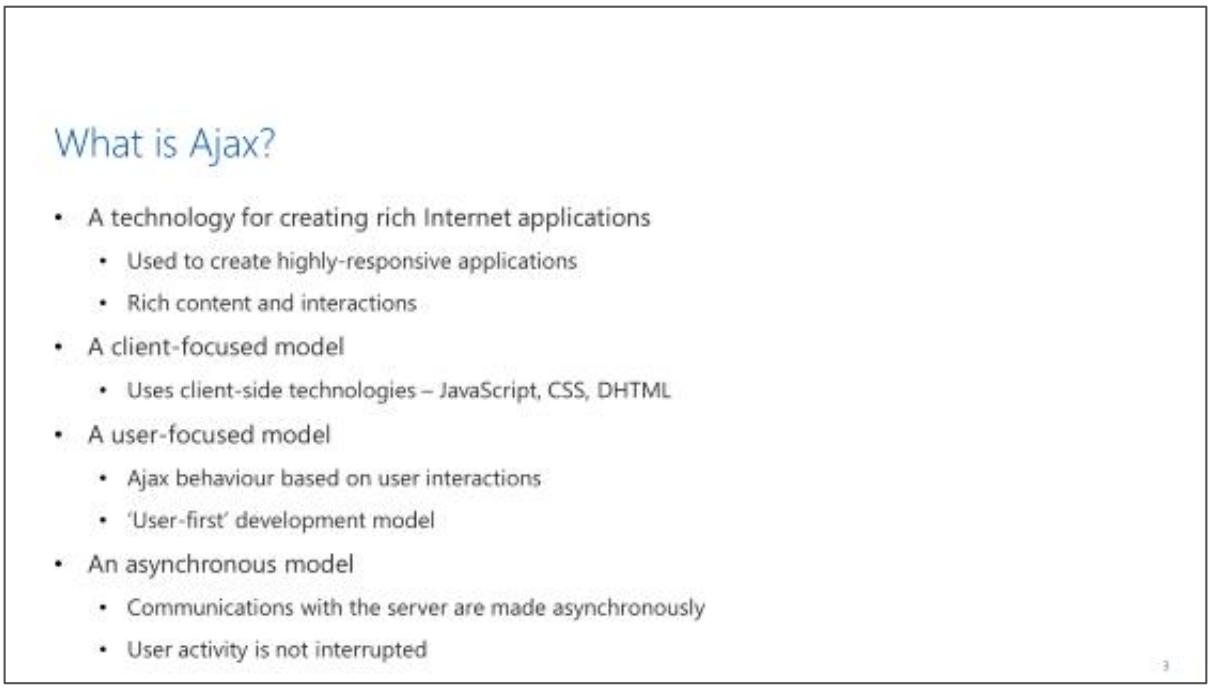
AJAX and JSON

PROGRAMMING WITH JAVASCRIPT

Overview

- What is Ajax?
- Ajax-enabling technologies
- XMLHttpRequest object
- Client and Server architecture
- XMLHttpRequest – Requests
- XMLHttpRequest – Responses
- The Fetch API
- JavaScript Object Notation (JSON)
- JSON Structures
- ECMAScript5 JSON object

2



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.

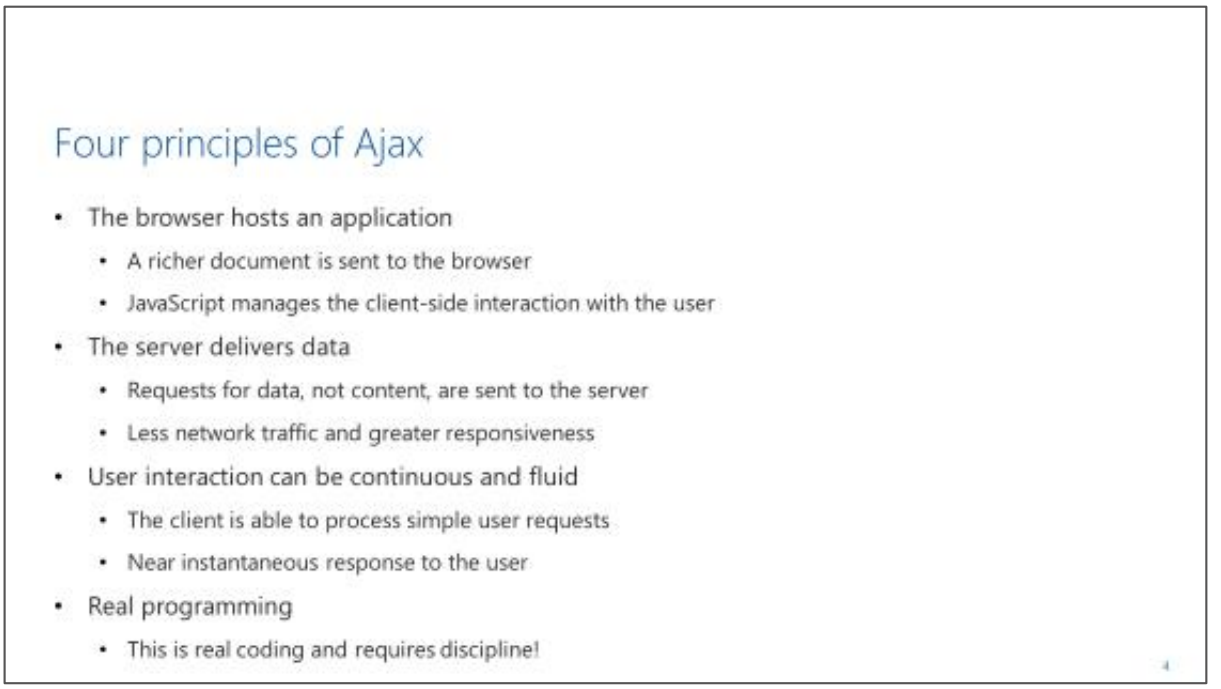
The typical server-bound web application cannot easily provide this because it has to defer to the server for any significant updates to the user interface.

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: DHTML 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.

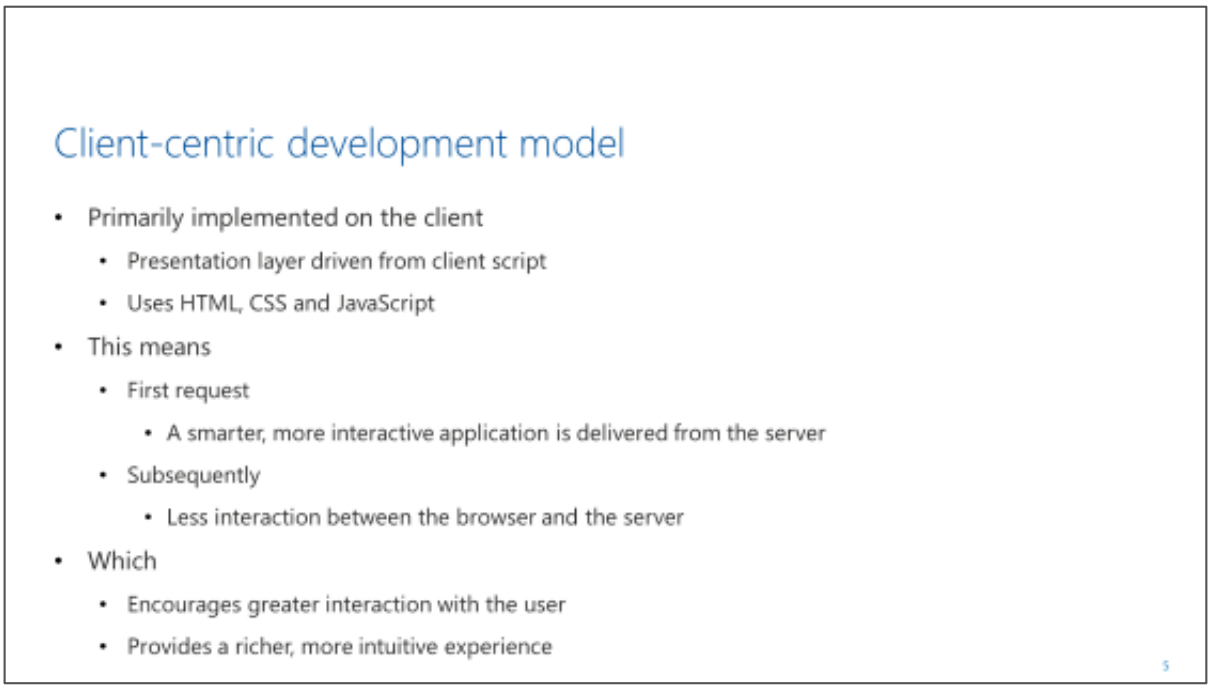
Through asynchronous communication with the server, we are able to retrieve data or HTML fragments that we can then insert into the DOM programmatically. The main benefits of asynchronous communication in this scenario are that we are only transmitting a small amount of information and are semi-coupled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



When we work in an Ajax 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 Ajax 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.



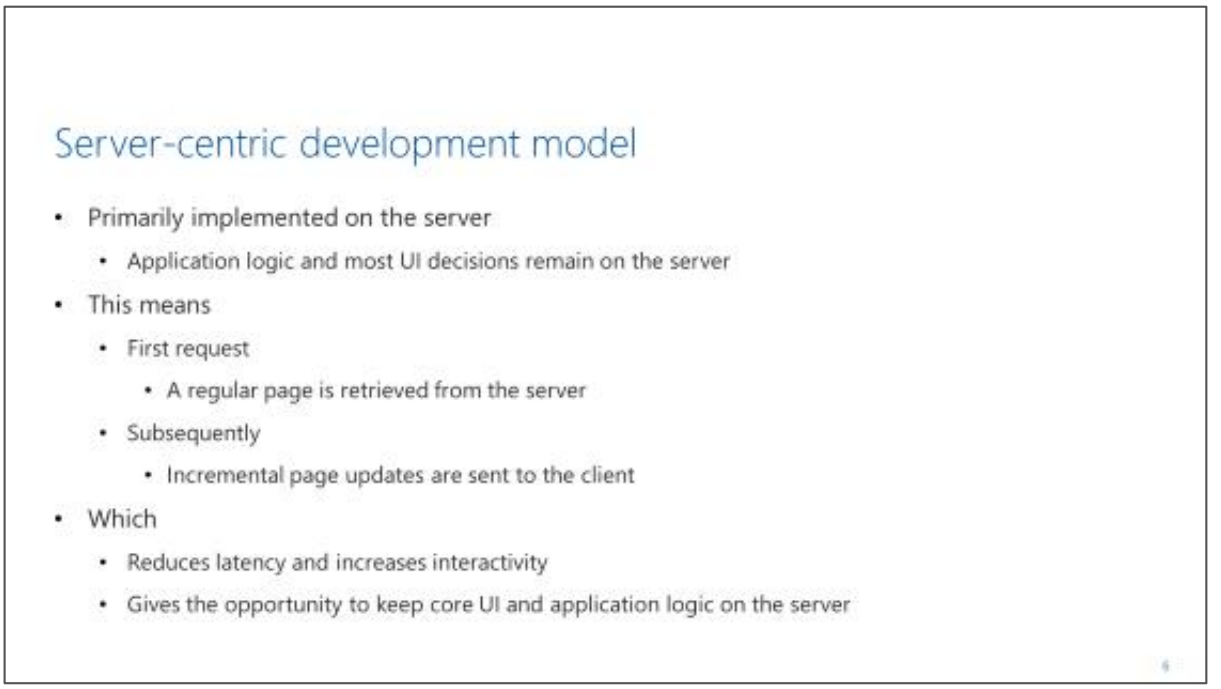
As mentioned, there are two main development models that we can consider within Ajax.

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 Ajax application, where we are communicating with data and manipulating the DOM programmatically.



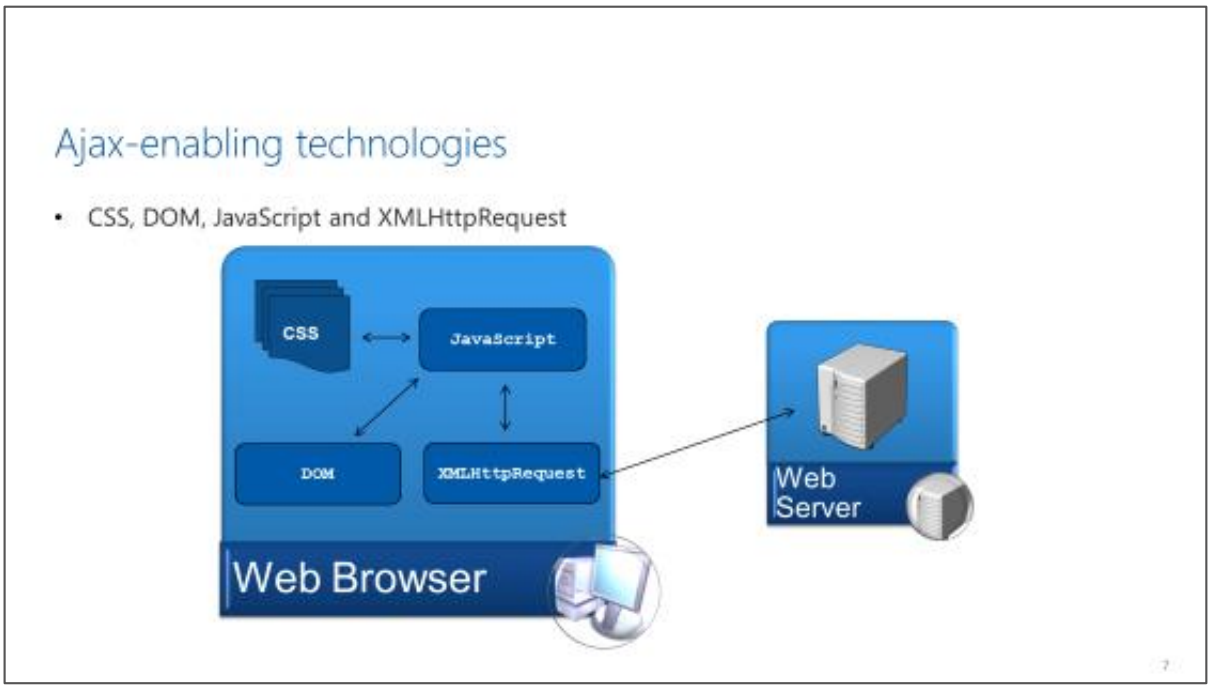
The second model is the server-centric development model.

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 login of the Application and UI on the server which may be a required decision for some sensitive applications.

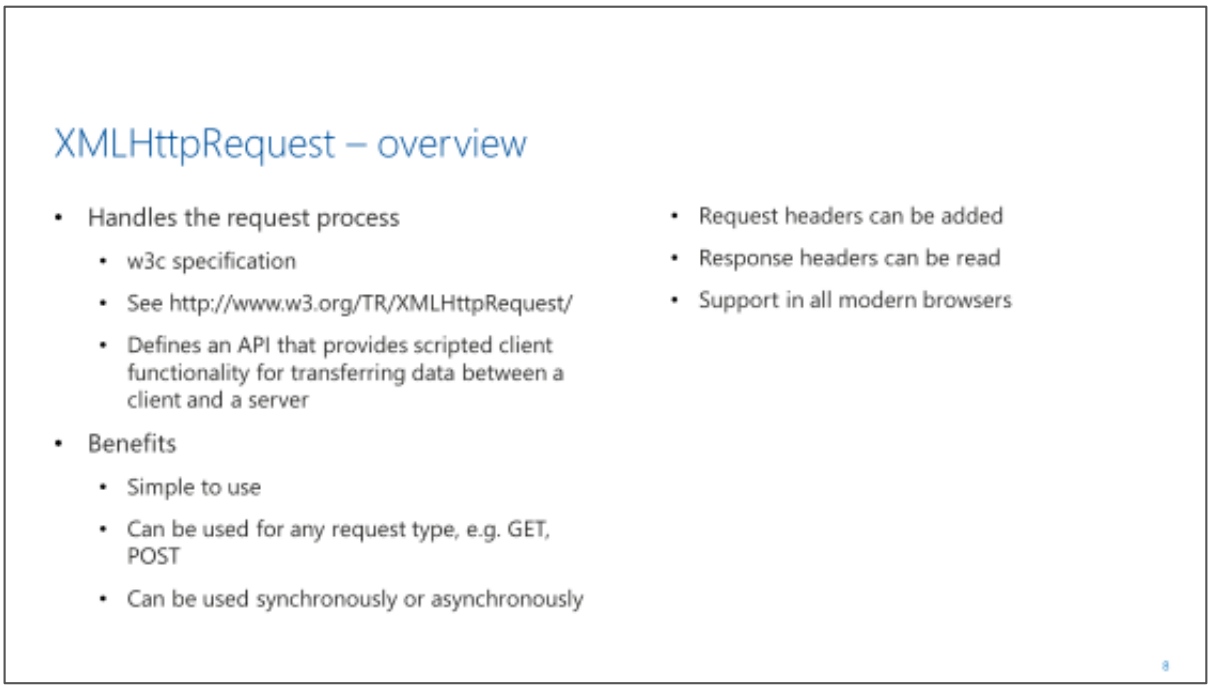
Within any one application, it is possible to have the two models working side-by-side on different pages with one page more suited to the server-centric model and the other more suited to the client-centric model. It would be very unusual and indeed very confusing to mix the two approaches on the same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



The diagram above shows the main enabling technologies of Ajax.

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.

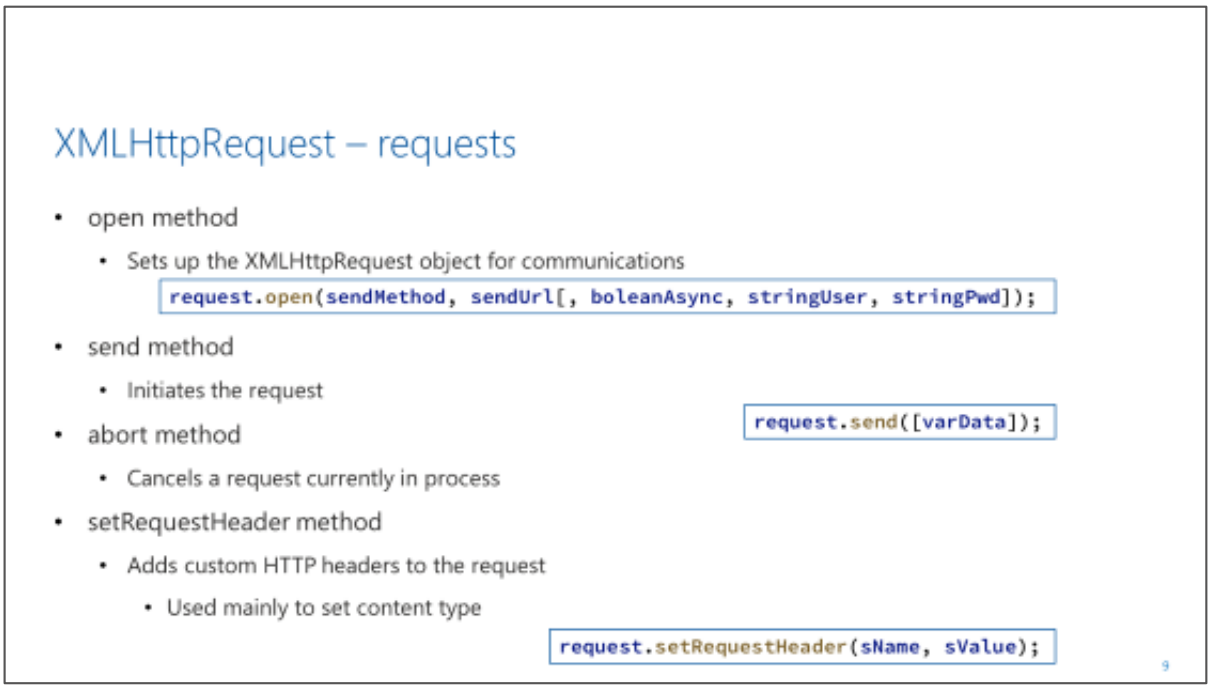
In addition, we can use the XMLHttpRequest object to initiate requests to a server in order to retrieve data, then use that data through JavaScript to update the document through the DOM.



The w3c working draft 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.



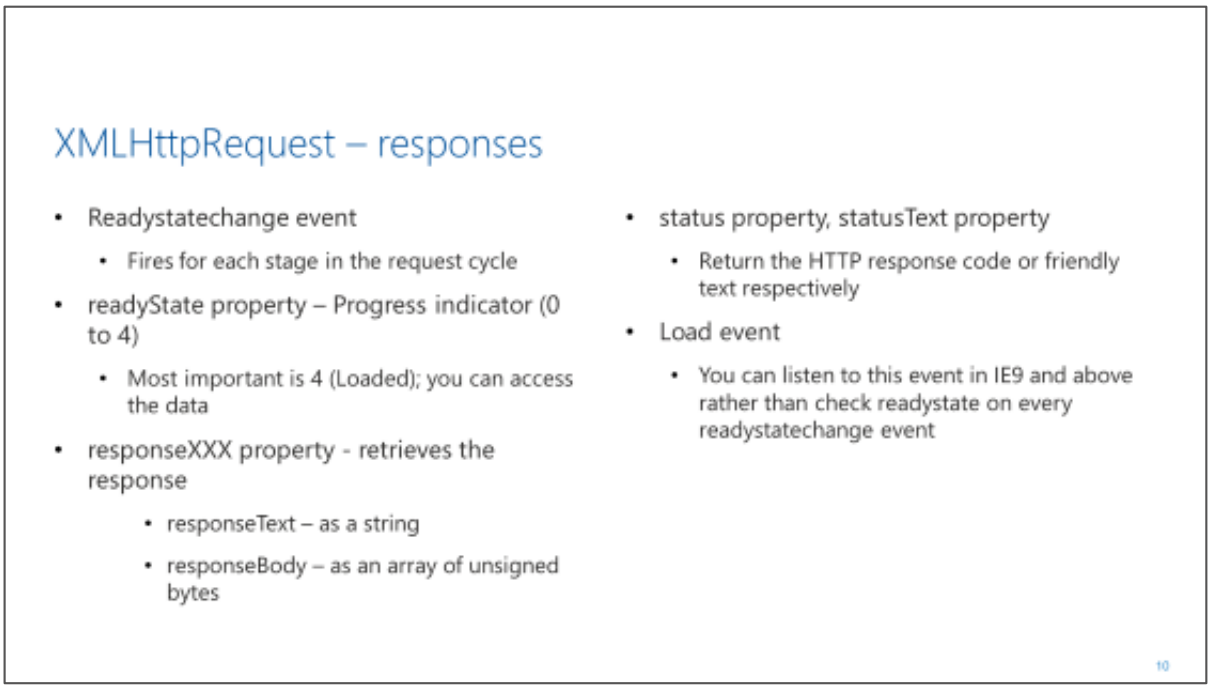
There are four named request methods; open, send, abort and setRequestHeader.

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.

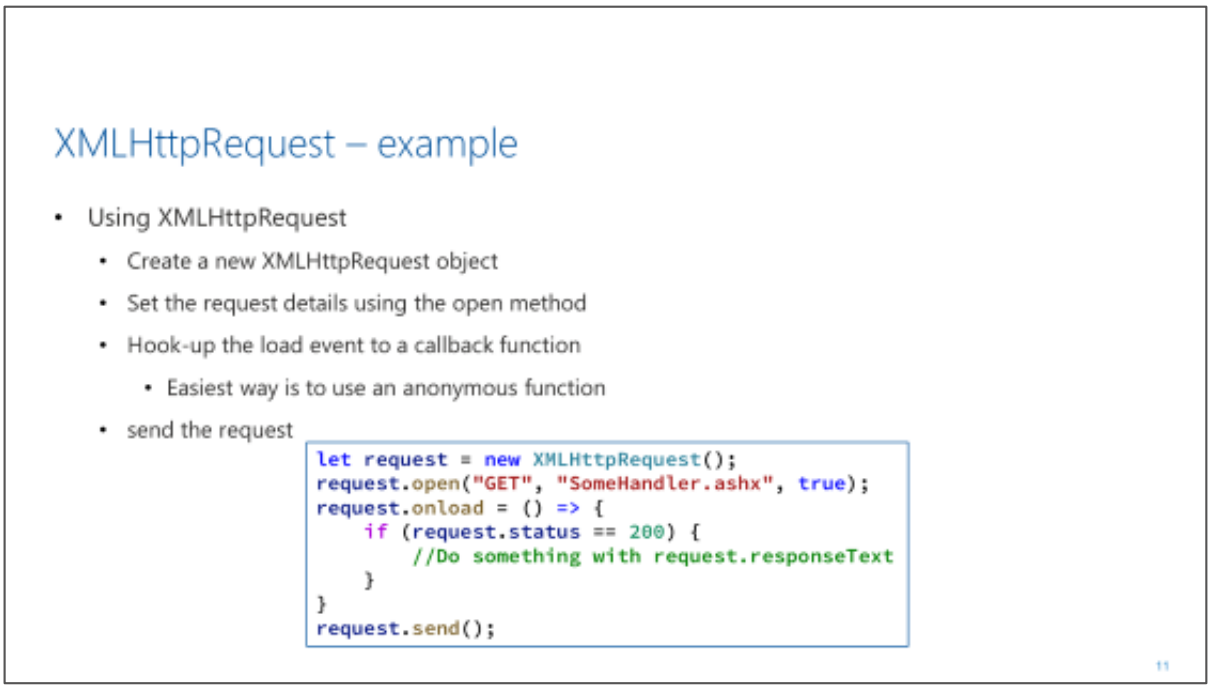


The key to retrieving or handling the response is to hook into the readystatechange event of the XMLHttpRequest object.

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:

readyState value	Description
0	Unsent
1	Opened
2	Headers Received
3	Loading
4	Loaded (Done) – data is fully loaded

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.



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.

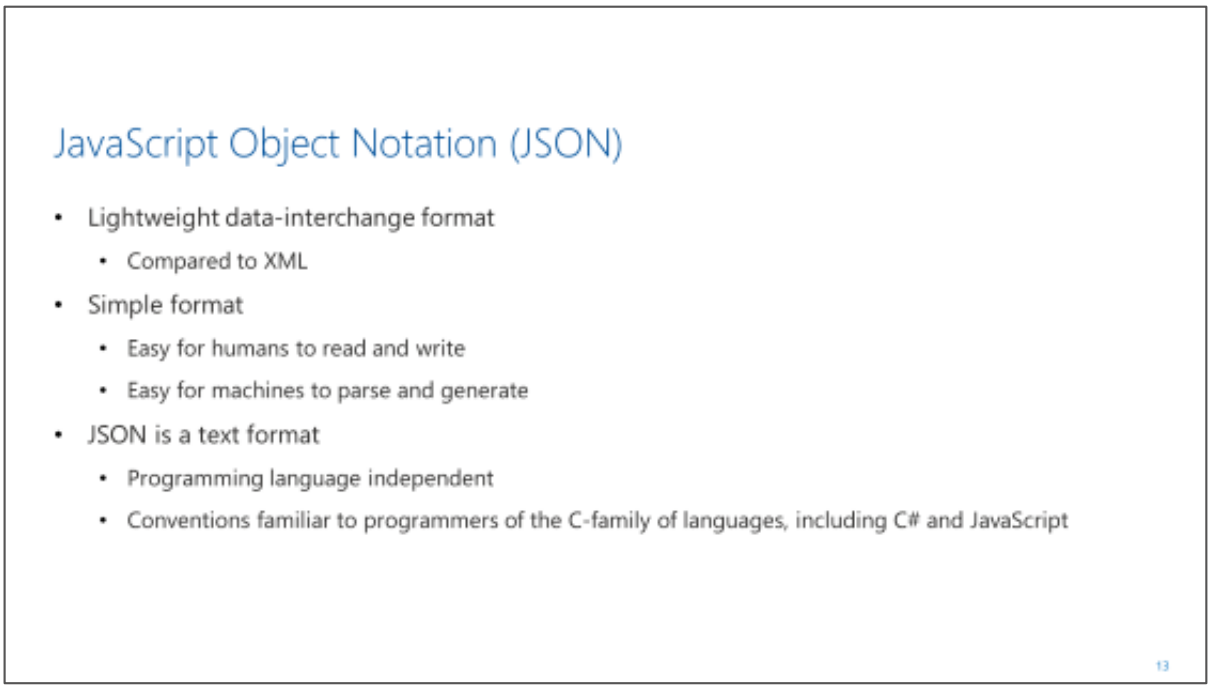
We then check to see that the request completed successfully (HTTP response code 200 – OK).

We can then do something with the responseText/responseXML.

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

JSON

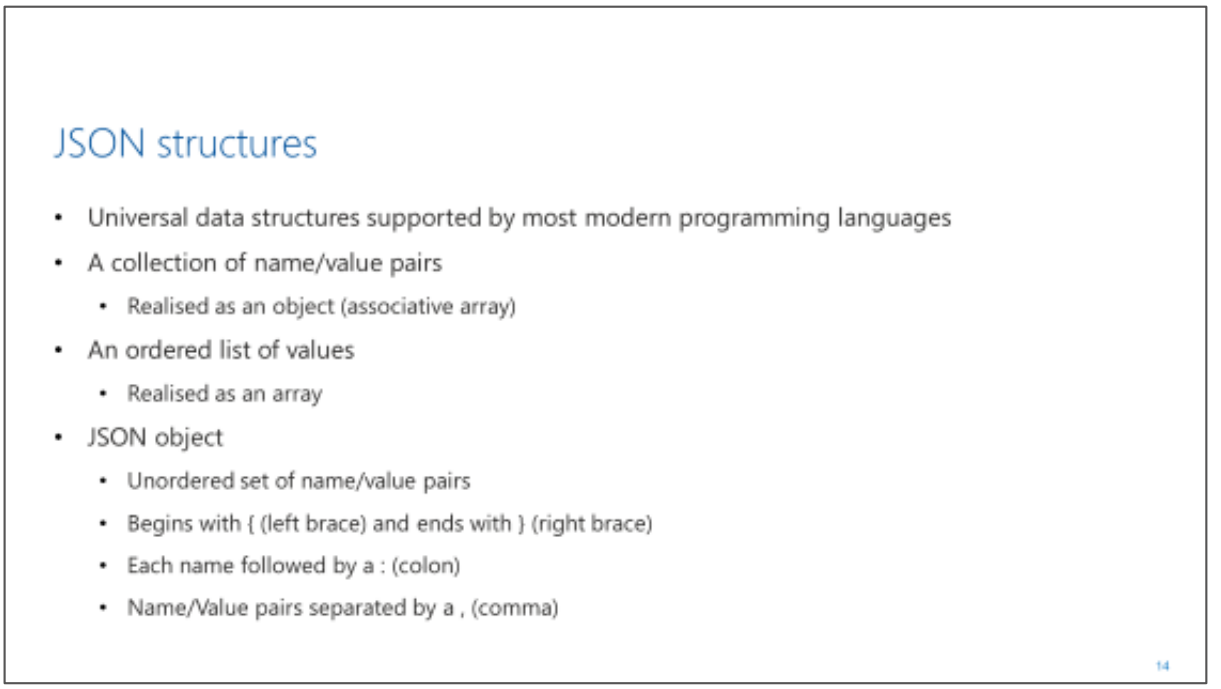
PROGRAMMING WITH JAVASCRIPT



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.



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.

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
        }
    ]
};
```

15

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.

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.

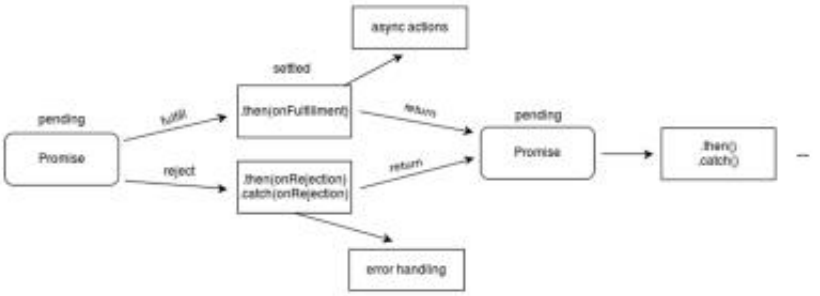
JSON.stringify(obj, null, 2) – Adds the specified number of spaces to the output, printing it evenly.

Promises

PROGRAMMING WITH JAVASCRIPT

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



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

19

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);
        }
    );
```

20

Fetch

PROGRAMMING WITH JAVASCRIPT

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

22

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))
    .cach(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

23

Fetch

- 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));
```

24

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 OK

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

25

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

Async Functions

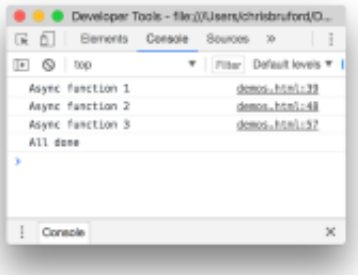
```
async function asyncFunc1() {
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log('Async function 3');
            resolve();
        },3000);
    });
}

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);
    });
}
```

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

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



27

Review

- AJAX is...
  - A technology for creating rich Internet applications
  - A client and user-focused model
  - A technology that enables asynchronous requests
- XMLHttpRequest is used for data requests from the client to the server
- JSON.parse and .stringify allow us to translate to JSON and back
- Fetch
- Async functions and the await declaration

Exercise

- Turning the Slideshow into an Ajax-driven resource

29