Advanced JavaScript for Web Sites and Web Applications

AJAX

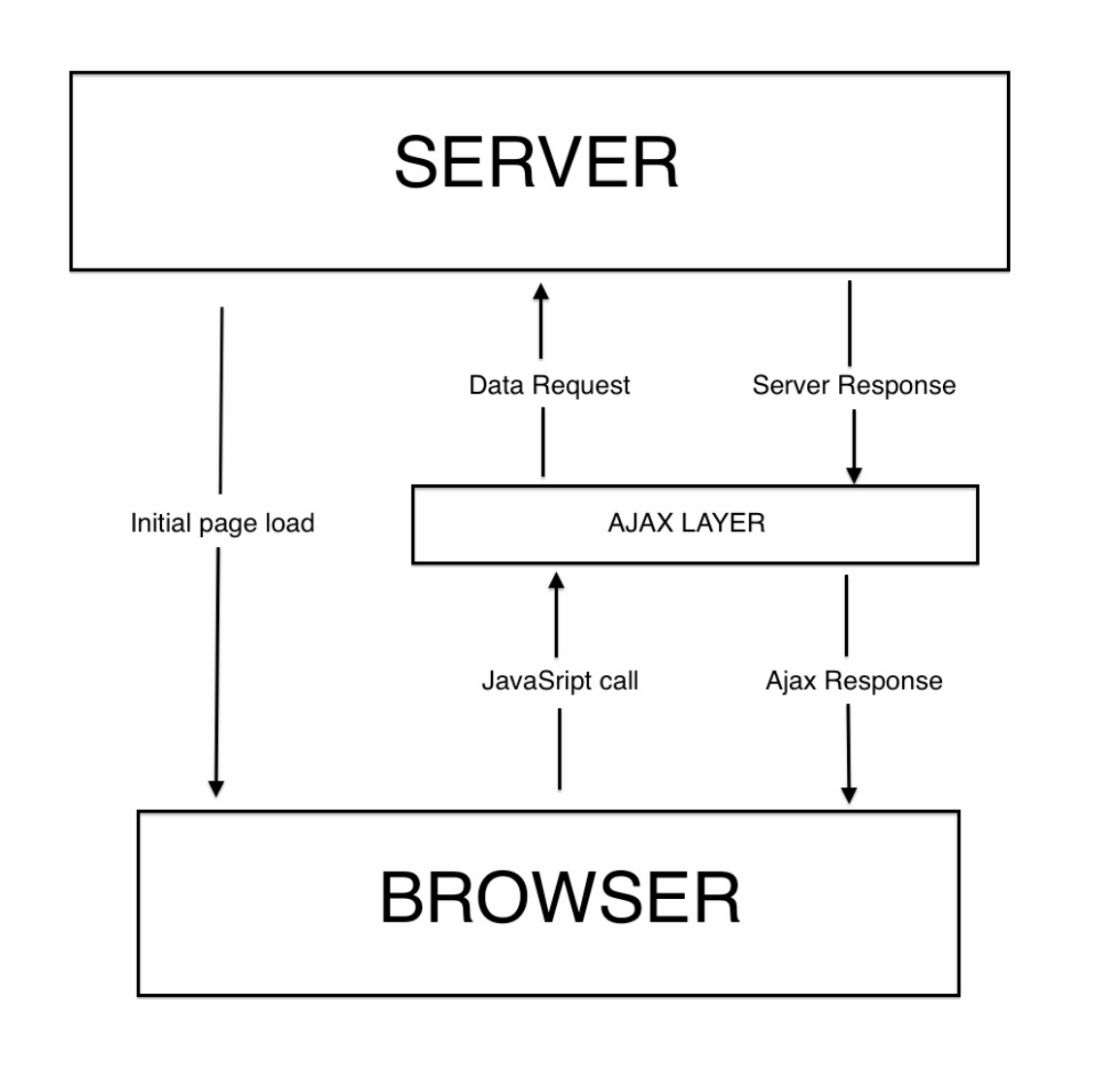
What is AJAX?

- Ajax lets you send HTTP requests to a web server from within your script
- The server can send data back to you, which you can then use/display
- This all happens without the page being reloaded

What is it good for?

- You can use it to make quick requests to the server and then display the results
  - E.g. a form's auto-complete input field.
- You can use it to load the main page content after the initial page has loaded
  - E.g. lazy-load/infinite scroll
- You can use it for pretty much anything you like!

AJAX Diagram



Basic AJAX - How it works

- Ajax utilises the browser's built-in `XMLHttpRequest` object.
- It can retrieve data through the *http*, *ftp* or *file* protocols
- The data can be in many different formats (xml, html, json, text etc.).

The `XMLHttpRequest` object

- This is how we use the object:
  - create a new *instance* of the object
  - pass this new object the *URL* to send the request to and the *method* to use (GET, POST etc.)
  - tell the object to *send* the request
  - do something with the returned data

The XMLHttpRequest object - example

```
 // create a new instance of the object
var myRequest = new XMLHttpRequest();

// pass the URL and the method (GET, POST etc.)
// to the object's open() method
myRequest.open("get", "data.txt", false);

// make the ajax request
// for a GET request, no data needs to be sent
myRequest.send();

// Data returned from URL is available
// via the object's responseText property
console.log(myRequest.responseText);
```

`XMLHttpRequest` - Some problems...

- In the previous example, `false` was passed to `open()` as its third argument.
  - This tells the `open` method to wait for a response before returning.
- This is not good... the JavaScript will freeze while we wait for a response from the URL!

`XMLHttpRequest` - Some problems...

- This type of request is called a synchronous request
  - It is deprecated in recent versions of the language
- So we are not going to be using it...

Asynchronous requests

- Instead of waiting for a response, we can tell `open()` to perform the request *asynchronously*.
  - We do this by passing `true` as the third argument to `open()`
- This will cause `open()` to add the request to the browser's queue, and then return, allowing the rest of the program to continue

Asynchronous requests - the response

- However, we now have another problem…
- We can't use *responseText* until the request is complete…
- So we need a way of knowing when it has completed…

The `load` event

- Solution: Use an event listener, attached to the request object's *load event*
- The *load event* is triggered when a response has been received from the requested URL

The `load` event

- Within the event handling function, we can use `this` to access the request object...
- ... and we have already seen that the *responseText* is a property of the request object...

Listening for the `load` event

```
// With an anonymous function:
myRequest.addEventListener("load", function () {
    console.log(this.responseText);
});

//OR, with a named function:
myRequest.addEventListener("load", myFunction);
```

```
/* Asynchronous request example */

// Function to run when request completes
function reqListener () {
    console.log(this.responseText);
}

// create a new instance of the object
var myRequest = new XMLHttpRequest();
// Attach function to request's "load" event
myRequest.addEventListener("load", reqListener);
// pass "true" as 3rd argument (asynchronous)
myRequest.open("get", "data.txt", true);
// make the ajax request
myRequest.send();
```

Introducing JSON

- Commonly the data we receive via AJAX will be formatted as *JSON*.
- Why?
  - *Simple syntax = small files = fast requests*
- It also integrates nicely with JavaScript... and many other languages

JSON

- JSON is a data storage format, which uses the JavaScript object notation syntax.
- It is:
  - Lightweight
  - transferable as text (frequently used with AJAX)
  - easy to use with JavaScript as it uses the same syntax.

Simple JSON object example

```
{
    "firstname": "John",
    "lastname": "Smith",
    "age": 25
}
```

JSON object with nested objects/arrays

```
{
    "name": "John Smith",
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York"
    },
    "phoneNumbers": [{
        "type": "home",
        "number": "646 555-1234"
    }, {
        "type": "office",
        "number": "646 555-4567"
    }]
}
```

JSON syntax

- To be valid, all the property names of a JSON object need to be enclosed in double quotes.
- JSON objects are enclosed in `{ }` (just like JS)
- JSON arrays are enclosed in `[ ]` (just like JS)
- Everything in a JSON data structure must be stored in an object or an array.
- The official JSON language specification:
  `http://www.json.org/`

JSON and JavaScript

- In JavaScript, you can parse a JSON string to a JavaScript entity using `JSON.parse()`.

```
var myData = JSON.parse("JSON string");
```

- JSON objects become JavaScript objects
- JSON arrays become JavaScript arrays

JSON and JavaScript - example

- Assuming `json_str` contains:

```
{
    "firstname": "John",
    "lastname": "Smith",
    "age": 25
}
```

```
var jData = JSON.parse(json_str);
console.log(jData.firstName); // "John"
```

JSON and JavaScript - example

- Assuming `json_str` contains:

```
["apples", "bananas", "pears"]
```

```
var jData = JSON.parse(json_str);
console.log(jData[0]); // "apples"
console.log(jData[2]); // "pears"
```

JSON and JavaScript - example

- Assuming `json_str` contains:

```
{
    "name": "John",
    "children": ["Tom", "Mary"]
}
```

```
var jData = JSON.parse(json_str);
console.log(jData.name); // "John"
console.log(jData.children[0]); // "Tom"
```

Now, where were we…

- Getting back to AJAX…

JSON response

- If the URL we send our request to returns data formatted as JSON…
- *responseText* will contain the JSON data as a string.
- We need to transform this string into a JavaScript object before we can use it

AJAX and JSON example : The JSON data

```
{
    "firstname": "John",
    "surname": "Smith",
    "married": true,
    "age": 25
}
```

AJAX and JSON example : The Javascript

```
// Function to run when request completes
function reqListener () {
    // Transform JSON to JS object
    var data = JSON.parse(this.responseText);
    // Get "firstname" property of object
    console.log(data.firstname);
}

var myRequest = new XMLHttpRequest();
myRequest.addEventListener("load", reqListener);
myRequest.open("get", "data.json", true);
myRequest.send();
```

AJAX... the bad news

- Unfortunately, browser quirks and differences in their implementations of `XMLHttpRequest` mean the examples we have looked at will not work in all browsers.

AJAX and Browsers

- Writing code to deal with the various issues and inconsistencies is not a trivial task.
- For this reason, most people will use a 3rd party library/module/function when working with AJAX.
- We will look at how **jQuery** does it...

AJAX with jQuery: ajax()

- Ajax calls can be made with jQuery via it's `ajax()` function
  - Other functions exist (`.get`, `.post`), but they are simply wrappers around `ajax()`
- The `ajax()` function returns a *jQuery XMLHttpRequest* object (`jqXHR`)
  - similar to the native `XMLHttpRequest` object, with some extra stuff

jQuery `.ajax()` example

```
// myJqxhr will be a jqXHR object
var myJqxhr = $.ajax({
    url: "data.json",
    dataType: "json",
    type: "get"
});
```

`.ajax()` - arguments

- We can pass many arguments to `ajax()`
  - usually as an object where the properties are the settings we want configure
- Some of the commonly used settings are:
  - `url` : The URL to send request to
  - `dataType`: The type of data type the server will return (JSON, HTML, Etc.).
  - `type`: The *method* to use (get, post, etc.)

`.ajax()` - callbacks

- In jQuery, we don’t need to add event listeners to capture the response text
- the `jqXHR` object returned by `ajax()` has several methods that we can use to register *callback* functions with
- These callback functions will be executed at various stages of the request

jqXHR - `.done()`

- A function passed to `done()` will be executed if the request completes successfully (i.e returns HTTP status 200 or similar)

```
myJqxhr.done(function (data, text, jqXHR) {
    // Do stuff
});
```

- The function will receive 3 arguments
  - The `data` returned by the server, a status message (`text`), the `jqXHR` object

jqXHR - `.fail()`

- A function passed to `fail()` will be executed if the request fails (i.e. returns HTTP status 404, time out etc.)

```
myJqxhr.fail(function (jqXHR, text, err) {
    // Handle error
});
```

- Again, the function receives 3 arguments
  - The `jqXHR` object, a status message (`text`), the error message returned by server/browser (`err`)

jqXHR - `.always()`

- A function passed to `.always()` will be executed when the request is complete

```
myJqxhr.always(function (arg1, arg2, arg3) {
    // I always run!
});
```

- For a successful request, this function will receive the same arguments as the `done` callback
- If the request failed, it receives the same arguments as the `fail` callback

Registering callbacks - example

```
var myJqxhr = $.ajax({
    url: "data.json",
    dataType: "json"
});

myJqxhr.done(function (data, text, jqXHR) {
    console.log('It worked');
});
myJqxhr.fail(function (jqXHR, text, err) {
    console.log('Oops!');
});
```

`.ajax()` - callbacks

- Remember, the `done`, `fail` and `always` methods belong to the `jqXHR` object returned by `ajax()`
- `done`, `fail` and `always` all return the `jqXHR` object that they belong to.
- In other words, them, and the initial call to `ajax()` are *chainable*

Registering callbacks - chained example

```
var myJqxhr = $.ajax({
    url: "data.json",
    dataType: "json"
}).done(function (data, text, jqXHR) {
    console.log('It worked');
}).fail(function (jqXHR, text, err) {
    console.log('Oops!');
});
```

`.ajax()` - Callbacks via settings

- As an alternative to `done`/`fail`/`always`, we can also assign callback functions via the settings object we pass to `.ajax()`
- They work the same way as `done`/`fail`/`always` but are less fexible

`.ajax()` - The callback settings

- `success`: runs if the request is successful (i.e returns HTTP status 200 or similar)
- `error`: if the request returns an error (i.e. returns HTTP status 404, time out etc.)
- `complete`: runs once the request is done, regardless of whether it was successful or not. This always runs *after* the success/error functions.

.ajax() - Callback examples

```
var myJqxhr = $.ajax({
    url: "data.json",
    dataType: "json",
    success: function (data, textStatus, jqXHR) {
        console.log('It worked!);
    },
    error: function (jqXHR, textStatus, err) {
        console.log('Oops!');
    },
    complete: function (jqXHR, textStatus) {
        console.log('I always run!');
    }
});
```

ajax() - data types

- When we specify a *dataType* in our `ajax()` call, jQuery will perform some transfomations to the returned data, *before* passing it to our success function.

ajax() - data types

- If we specify JSON as the *dataType*, jQuery will convert the returned JSON string into a JavaScript object for us
  - So we don't need to use `JSON.parse`!
- If we want the success function to receive the raw JSON data, we can set *dataType* to *text*

Some notes about AJAX...

- It's great!, but...
- We shouldn't abuse/over-use it
  - performance will suffer
- Browsers implement a *same-origin* policy
  - by default, we can only make AJAX requests to URLs on the same domain as our script

The same-origin policy

- The *same-origin policy* is a security feature implemented by browsers.
- It restricts the way that scripts loaded from different domains can interact with each other
- By default, it prevents us making Ajax requests to scripts stored on external URLs

The same-origin policy

- To work around the same-origin policy problem, we can use *jsonp* as the *dataType* argument with $.ajax
- If we use *json* as the *dataType* with a URL on an external domain, jQuery will change the dataType to *jsonp* behind the scenes.
- We will use *jsonp* exclusively

jsonp

- Note, from our code's perspective, there is no difference between handling *JSONP* and *JSON*
  - We still receive a JavaScript object containing the JSON data
- The differences all occur on the server we make the requests to, and within the browser (before the data is passed to us)

Exercises

- Download the exercises document from Moodle and do *Exercise 1* and *Exercise 2*