POWERING
POTENTIAL

Asynchronous JavaScript

JavaScript Fundamentals

OVERVIEW

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

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

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

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

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:

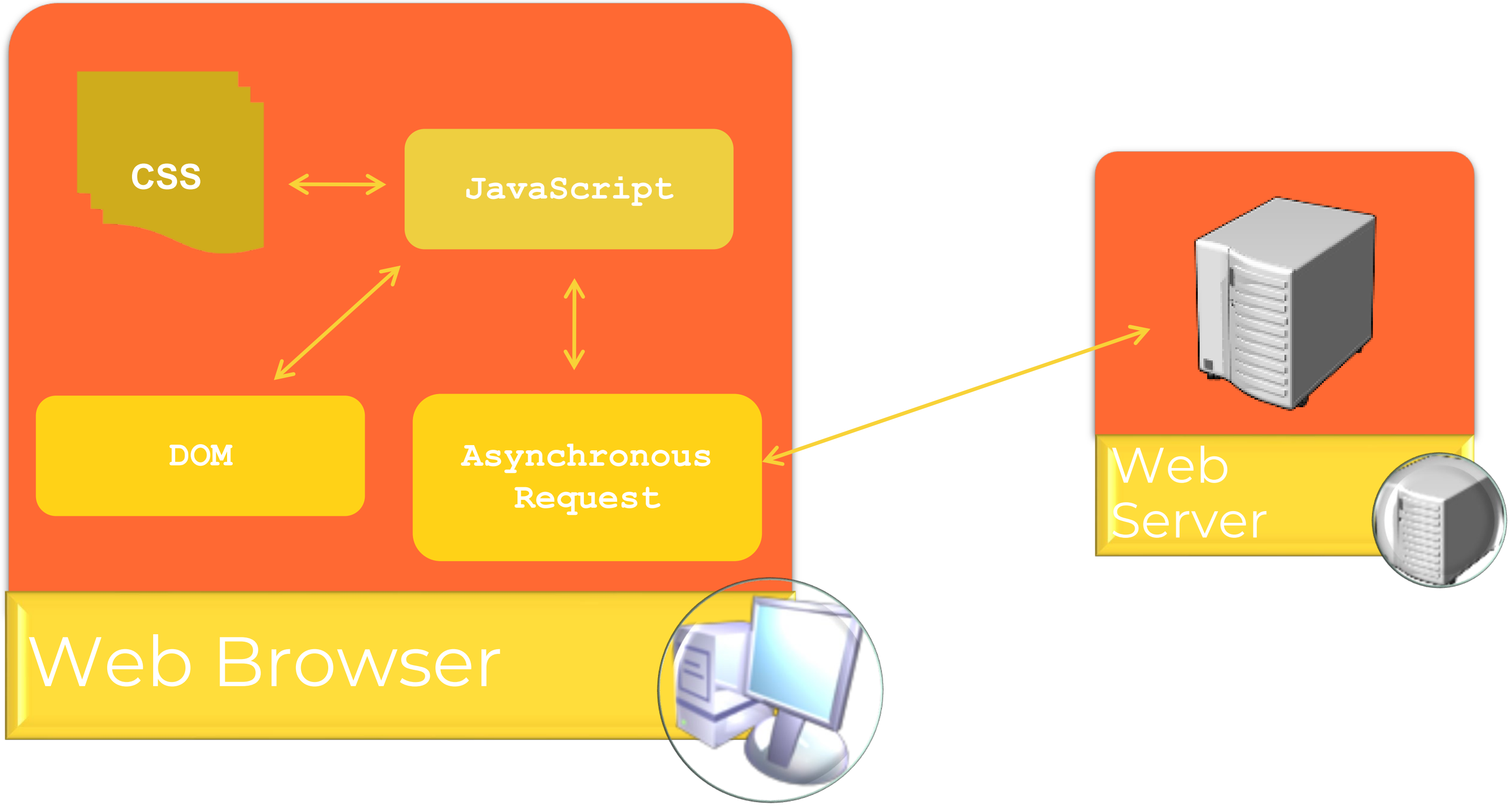
- Incremental page updates are sent to the client

Which:

- Reduces latency and increases interactivity
- Gives the opportunity to keep core UI and application logic on the server

Asynchronous JavaScript - enabling technologies

- CSS, DOM, JavaScript, and an Asynchronous Request API



JSON

JavaScript Fundamentals

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

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)

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

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
        }
    ]
};
```

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" });
```

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

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

QuickLab 16a – create some JSON

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

POWERING
POTENTIAL

Promises

JavaScript Fundamentals

WHAT IS A PROMISE?

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

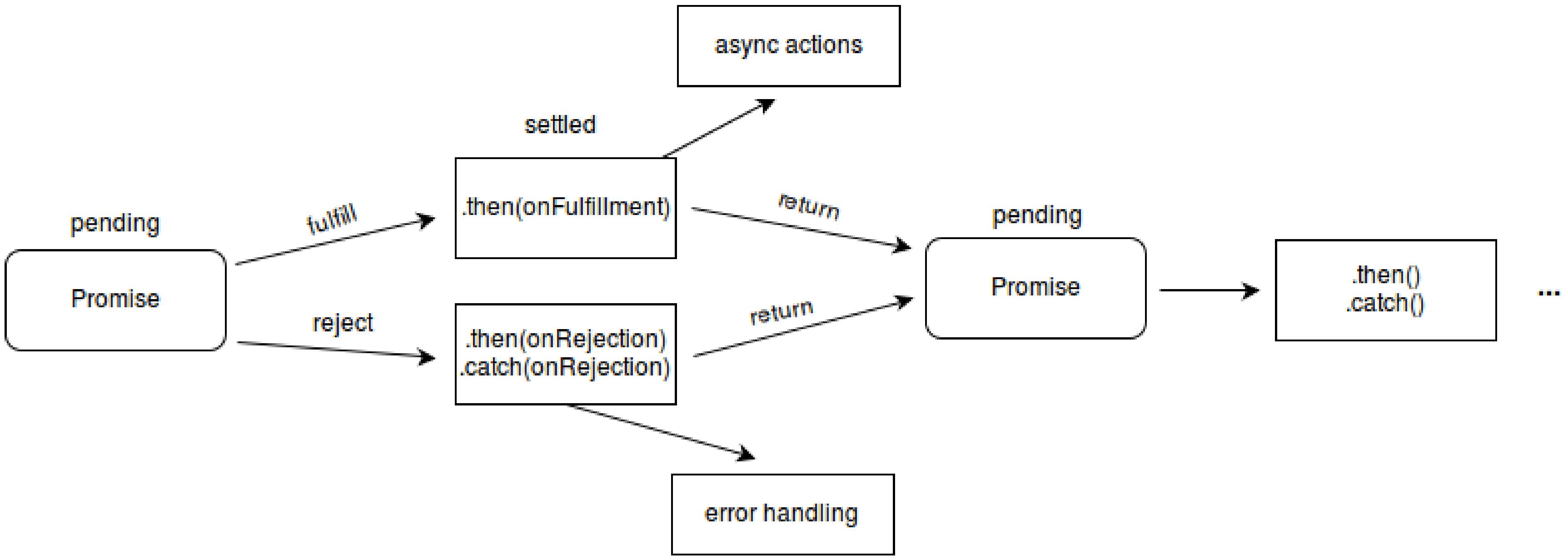
- JavaScript is executed from the top down
  - Each line of code evaluated and executed in turn

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

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

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);
        }
    );
```

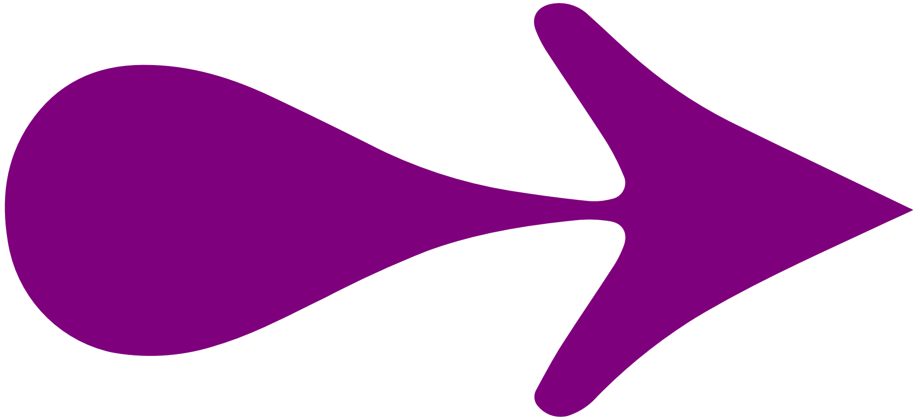
QuickLab 16b – promises

- Experiment with promises

Fetch

JavaScript Fundamentals

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

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`

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));
```

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

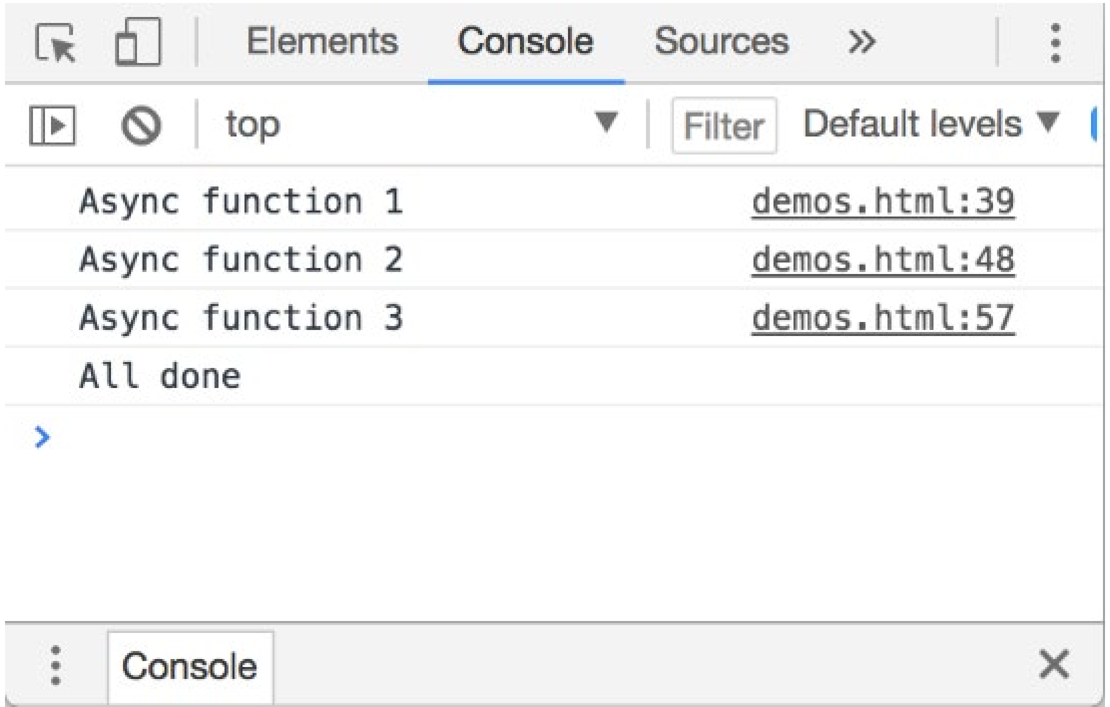
Async functions

- 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

Async Functions

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

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



```
Async function 1        demos.html:39
Async function 2        demos.html:48
Async function 3        demos.html:57
All done
>
```

```
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);
    });
}
```

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

Hackathon Part 2

- In this part of the 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.

POWERING
POTENTIAL

Appendix AJAX and XMLHttpRequest

JavaScript Fundamentals

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

XMLHttpRequest – requests

`open` method

- Sets up the **`XMLHttpRequest`** object for communications

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

`send` method

- Initiates the request

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

`abort` method

- Cancels a request currently in process

`setRequestHeader` method

- Adds custom HTTP headers to the request
- Used mainly to set content type

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

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

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();
```