Mobile Applications Development

Week 1 – Background

Practical Exercises

# 1. Introduction

**Note: this week’s worksheet is long because there is a fair amount of reading. There is a large overlap between the content of the worksheet and the lecture video and the worksheets will not always be so large!**

# 2. REstful APIs and the OPen API specification

#### Exercise 1: RESTful APIs

1. Familiarise yourself with how RESTful APIs work. Start by watching the lecture video, and then do some research until you fully understand the concepts. There are many tutorials and explanations on the web. For example:

* <https://restfulapi.net/>
* <https://www.redhat.com/en/topics/api/what-is-a-rest-api>
* <https://youtu.be/7YcW25PHnAA>

2. Download and run the movieDB server from Moodle

* Unzip the directory to somewhere on your machine
* Navigate to the directory in your terminal and run ‘npm install’ (a directory called ‘node\_modules’ should appear with all of the dependencies.
* Run ‘npm start’ to run the server. The terminal should print ‘Listening on port 3000…’

3. Download Postman and familiarise yourself with it by interacting with the MovieDB server.

* https://www.postman.com/
* Use the lecture video to help you navigate Postman
* Hit each of the end points and get used to sending requests through Postman
* Can you break the server with bad data? Bad headers? Bad routes? (i.e. test the server to destruction)

4. (Advanced) Write your own OpenAPI specification.

* Read through the Idratherbewriting tutorial discussed in the lecture video: <https://idratherbewriting.com/learnapidoc/pubapis_openapi_step1_openapi_object.html>
* Write an API specification using Swagger (<https://editor.swagger.io>) for interacting with a ‘basket’ on an e-commerce site. The API should implement the following end-points:
  + GET /basket – Returns all items in the basket
  + POST /basket – Add a new item to the basket. The item object should include an ‘item\_name’ and ‘quantity’ field
  + GET /basket/:item\_name – Returns one item with the same name as the parameter
  + PATCH /basket/:item\_name – Updates the item with the same name as the parameter
  + DELETE /basket/:item\_name – Deletes the item with the item name form the basket
* The MovieDB specification has been made available on Moodle for you to use as a guide.
* Some questions to consider:
  + What are the limitations of this API design?
  + What happens if you add an item that already exists to your basket?
  + How would your API handle the baskets of multiple users?

# 3. JavaScript and ES6

## The ECMAScript standard

**Taken from:** <https://www.tutorialspoint.com/What-is-the-difference-between-JavaScript-and-ECMAScript>

European Computer Manufacturer's Association (ECMA) Script is a Standard for scripting languages such as JavaScript, JScript, etc. It is a trademark scripting language specification. JavaScript is a language based on ECMAScript. A standard for scripting languages like JavaScript is ECMAScript. JavaScript is considered as one of the most popular implementations of ECMAScript.

## ES6/ECMAScript 2015

Mozilla maintain a clear list of past and current versions of the ECMAScript standard (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources>). For this course, we will be using ECMAScript 2015 (also called ES6). The specification for ES6 can be found online (<https://www.ecma-international.org/ecma-262/6.0/index.html>). You are not expected to familiarise yourself with the entire specification for this course. However, we will cover some popular elements.

### Browser compatibility

All modern browsers now support ES6. CanIUse (<http://caniuse.com>) maintains details of features that are supported by different browsers. When creating applications for the web and mobile (React Native uses JavaScriptCore as its virtual machine. The same JavaScript engine that powers Safari) it is good often helpful to check any components/features with CanIUse to ensure that your application will work on all intended browser versions. For example, we can check ES6 support here: <https://caniuse.com/#search=ECMAScript%202015>

### Variable declarations: var, let and const

#### Exercise 2: Strict mode

ES6 requires you to declare your variables as either a ‘var’, ‘let’ or ‘const.’ We will look at the differences between these three in the exercises that follow but to begin, create a file called lab2.js and enter the following:

hello = "world";

console.log(hello);

In the terminal, run the script by navigating to the directory where the file is saved and running ‘node lab2.’

The script runs successfully without us declaring the hello variable using ‘var’, ‘let’ or ‘const.’ This is because the script isn’t running in ‘strict’ mode.

**Below from:** <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode>

“JavaScript's strict mode, introduced in ECMAScript 5, is a way to opt in to a restricted variant of JavaScript, thereby implicitly opting-out of "sloppy mode". Strict mode is not just a subset: it intentionally has different semantics from normal code. Browsers not supporting strict mode will run strict mode code with different behaviour from browsers that do, so do not rely on strict mode without feature-testing for support for the relevant aspects of strict mode. Strict mode code and non-strict mode code can coexist, so scripts can opt into strict mode incrementally.

Strict mode makes several changes to normal JavaScript semantics:

1. Eliminates some JavaScript silent errors by changing them to throw errors.
2. Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
3. Prohibits some syntax likely to be defined in future versions of ECMAScript.”

To enable strict mode, add the following to the beginning (top) of your file:

'use strict';

Running the script now gives us a ReferenceError because the hello variable has not been defined properly. Fix this by declaring the hello variable as a var:

'use strict';

var hello = "world";

console.log(hello);

Running the script will now again work successfully, but in strict mode.

#### Exercise 3: Variable scoping

‘var’ is function scoped whereas ‘let’ and ‘const’ are block scoped. To understand what the differences are, run the following:

'use strict';

function incrementAllValues(values){

var incremented\_values = [];

for(var i = 0; i < values.length; i++){

var incremented\_value = values[i] += 1;

incremented\_values.push(incremented\_value);

}

console.log(incremented\_values);

console.log(i);

console.log(incremented\_value);

return incremented\_values;

}

var results = incrementAllValues([3,4,5,5]);

With all the variables being ‘var’, you will notice that we are able to log the values outside of the for loop. This is because ‘var’ is function scoped. Rarely would we ever want this to happen (perhaps in nested functions) and it may yield significant performance and security issues. As well as complications when it comes to testing.

‘let’ and ‘const’ on the other hand are block scoped. What this means is that they are only scoped to the current block (surrounded by curly braces, like a loop or if statement). Change all of the ‘var’ variables to ‘let’ variables and run the script again. You will no longer be able to log the contents of the for loop outside of the loop itself.

#### Exercise 4: Variable reassignment

‘let’ and ‘const’ are almost exactly the same. Only, they differ in that variables declared with a ‘let’ can be reassigned whereas variables declared with a ‘const’ stay constant.

Run the code below:

'use strict';

let first\_name = "Ashley";

first\_name = "Nick";

console.log(first\_name);

What happens when you change the variable to a ‘const’?

We cannot however call JavaScript ‘const’ variables immutable. Although the value of the variable cannot be reassigned, we can still, for example, have an object where the structure stays constant but the values change, or a function where the specification is constant but the output depends on the parameters. For example, try the code below:

'use strict';

const person = {

first\_name: "Ashley"

};

person.first\_name = "Nick";

console.log(person);

And:

'use strict';

const person = {

first\_name: "Ashley"

};

const set\_name = function(name){

person.first\_name = name;

}

set\_name("Nick");

console.log(person);

**Read more:** <https://tylermcginnis.com/var-let-const/>

### Arrow functions

Arrow functions are essentially a short-hand, cleaner way for writing functions. Let’s take the previous exercise as an example. Writing as an arrow function, the set\_name function now becomes:

const set\_name = (name) => {

person.first\_name = name;

}

Note the following when using arrow functions:

1. Arrow functions must be declared before they are used.
2. Always declare functions using ‘const’ as the function definition does not change
3. Arrow functions do not use ‘this’ and are therefore not well suited for writing functions within objects.

**Read more:** <https://www.sitepoint.com/es6-arrow-functions-new-fat-concise-syntax-javascript/>

### Classes

ES6 also introduces JavaScript classes. This does not introduce a new OO inheritance model for JavaScript, but does serve as a cleaner way to interface with JavaScript’s existing inheritance.

**Read more:** <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

#### Exercise 5: Classes

Consider the following:

'use strict';

class Person{

constructor(id\_number, first\_name, last\_name){

this.id\_number = id\_number;

this.first\_name = first\_name;

this.last\_name = last\_name;

}

static full\_name(){

return this.first\_name + " " + this.last\_name;

}

get id(){ //getter

return this.id\_number;

}

}

**Below from:** <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

**Static**: The static keyword defines a static method for a class. Static methods are called without instantiating their class and cannot be called through a class instance. Static methods are often used to create utility functions for an application.

We can now write classes that inherit from the Person class:

'use strict';

class Staff extends Person{

constructor(id\_number, first\_name, last\_name, job\_title, salary, pension\_contribution){

super(id\_number, first\_name, last\_name);

this.job\_title = job\_title;

this.salary = salary;

this.pension\_percent = pension\_contribution;

}

get job\_details(){

return this.job\_title + ", " + this.salary;

}

get pension\_calculation(){

return this.calculate\_pension\_contribution();

}

calculate\_pension\_contribution(){ //object method

return this.salary \* this.pension\_percent;

}

}

Let’s create a Staff object:

let s1 = new Staff(123456, "Mark", "Johnson", "CEO", 1000000, 0.07);

console.log(s1.pension\_calculation);

* Now write a class called ‘Customer’ which also inherits the Person class.
* Customers should have an id\_number, first\_name, last\_name, email\_address and customer\_start\_date.
* Create a getter method that calculates and returns the number of years that the customer has been a customer.

### Arrays: find() and findIndex()

#### Exercise 6: find()

**Taken from:** <https://www.w3schools.com/js/js_es6.asp>

The find() method returns the value of the first array element that passes a test function.

Consider the following:

'use strict';

let people = [

{first\_name: "Ash"},

{first\_name: "Megan"},

{first\_name: "Euan"},

{first\_name: "Manuel"},

{first\_name: "Hannah"},

{first\_name: "Kate"}

];

const findNamesWithN = (value, index, array) => {

return value.first\_name.includes("n");

}

let first\_with\_n = people.find(findNamesWithN);

console.log(first\_with\_n);

The find method returns the first person in the list whose name contains an ‘n’. Note the three parameters: the item value, the item index and the array to operate on.

Alter the code so that it instead returns the first palindrome (a word that reads the same backwards as forwards).

#### Exercise 7: findIndex()

The findIndex() method works the same as the find(), only instead of returning the object itself, the objects index is returned. Alter your solution to Exercise 5 to instead return the index of the first palindrome.

### Callbacks, promises, async and await

**Callbacks**

**Taken from:** <https://codeburst.io/javascript-what-the-heck-is-a-callback-aba4da2deced>

A callback is a function that is to be executed after another function has finished executed.

JavaScript is an event driven language, meaning that instead of waiting for a response before moving on, JavaScript will continue executing while listening for other events.

A common example of a callback is the setTimeout function. The setTimeout function takes in two parameters, a callback function to execute after the timeout, and the timeout period in milliseconds.

setTimeout(function(){

console.log("Executed");

}, 1000);

The above code executes the function in the first parameter after 1000 milliseconds. To view callbacks in all their glory, let us put the setTimeout inside another function.

'use strict';

function first(){

setTimeout(function(){

console.log("First");

}, 1000);

}

function second(){

console.log("Second");

}

first();

second();

Run the above code. Note that although the first function is called first, the code is executed after 1000 milliseconds while the script carries on executing meaning that it appears second in the terminal.

Watch this video on how JavaScript handles events (though the talk is specific to Chromes V8 engine, which is different to JavaScript Core, mentioned earlier): <https://www.youtube.com/watch?v=8aGhZQkoFbQ>

**Promises**

**Taken from:** <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>

The promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A Promise is in one of these states:

* **pending**: initial state, neither fulfilled nor rejected.
* **fulfilled**: meaning that the operation completed successfully.
* **rejected**: meaning that the operation failed.

A pending promise can either be fulfilled with a value, or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's ‘then’ method are called. (If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached).

As the Promise.prototype.then() and Promise.prototype.catch() methods return promises, they can be chained.

For example, let us implement the previous callback example, but this time in a promise:

'use strict';

let timeoutPromise = new Promise((resolve, reject) => {

setTimeout(function(){

resolve("Timeout complete");

}, 1000);

});

timeoutPromise.then((successMessage) => {

console.log(successMessage);

});

Note that the promise is first declared and a value (“Timeout complete”) is passed when the promise is resolved. When the promise is called, the success message is passed as a parameter for completion.

Here is another explanation of promises: <https://www.youtube.com/watch?v=IGYxfTTpoFg>

**Async & Await**

**Taken from:** <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await>

Async and await are newer than ES6, being included in ECMAScript 2017. However, they are good features to have in your JavaScript toolkit. Both async and await can be used as an abstract way to write promises, or for making any function into an asynchronous promise. This makes promises easier to write and easier for others to read.

The below simply defines an asynchronous function that prints hello:

'use strict';

let sayHello = async () => {

return("Hello");

};

sayHello().then(console.log);

Changing the last line to ‘console.log(sayHello());’ shows that a promise is returned.

The ‘await’ keyword can be placed in front of any promise based function to pause the code on that line until the promise fulfils before returning the resulting value. While waiting, any other waiting code gets the opportunity to execute.

The code below merges the previous two examples to print hello after 1000 milliseconds.

'use strict';

let timeoutPromise = new Promise((resolve, reject) => {

setTimeout(function(){

resolve("Timeout complete");

}, 1000);

});

let sayHelloIn1Second = async () => {

let timeout = await timeoutPromise;

console.log(timeout);

return("Hello");

};

sayHelloIn1Second().then(console.log);

#### Exercise 8: Read an article

Take some time to read and understand this article. It provides a great overview on the differences between callbacks, promises and async functions: <https://scotch.io/courses/10-need-to-know-javascript-concepts/callbacks-promises-and-async>

# 4. HelloWorld in React native

#### Exercise 9: Hello!

Follow the lecture video and helper guide to generate and run your first React Native application. You don’t need to do anything extra to it for now. This exercise is just about making sure that everything is set up and installed ready to go.

# 5. Weekly Assignment Checklist

The assignment checklist for this week is pretty short. That’s because the assignment will not be released until week two. The items below are just intended to ensure that you are ready to start the assignment.

* Understand the concepts covered in the RESTful APIs lecture video
* Complete the HelloWorld in React Native activity (exercise 9)

# 6. Additional resources

1. <https://medium.com/the-react-native-log/a-brief-overview-of-es6-for-react-native-developers-15e7c68315da>