Session 9

Recap

In the previous session we covered the following:

- DOM manipulation

Timers

In JavaScript we may have actions that we want to happen after a certain time. There are two ways of achieving this.

SetTimeout

The first one is `setTimeout` that allows us to run a set of actions after a period of time (once).

`setTimeout` takes a two arguments. The first is a function you want to run/execute. The second is the time duration after which you want the function to execute (in milliseconds).

For example:

```
  setTimeout(function() {
    console.log('Using standard function');
  }, 2000); // here 2000 is 2 seconds
```

We can also call a function directly inside the `setTimeout`:

```
const getCurrentRates = () => {
  return {
    "USD": 123.00,
    "GDP": 43.44
  }
}

setTimeout(getCurrentRates, 2000); // retrieve rates after 2 seconds
```

So what about tasks we want to run after a given period?

SetInterval

Similar to the `setTimeout`, there is a `setInterval` which allows you to run a set of code repeatedly after a period of time:

```
const getFlightDetails = () => {
  return {
    flights: [
      {
        "id": 1,
        "airline": "British Airways",
        "aircraft": "737"
      },
      {
        "id": 2,
        "airline": "Emirates",
        "aircraft": "888"
      }
    ]
  }
}

// this would repeatedly run every 3 seconds while the window is open!!
setInterval(getFlightDetails, 3000);
```

⚠️ We have seen we can create a timer that runs every 3 seconds here, **repeatedly**. What if this was an expensive operation (i.e. we are calling a back end service, which may have limits or may think it's being attacked).

We need to also a way of stopping the timer (one that is created using `setInterval`). To do so, we can use the useful `clearInterval` which takes the interval object as an argument (so that it can be stopped when called).

So if we take our previous example, we can stop this timer once we have received the flights:

```
const getFlightDetails = () => {
  return {
    flights: [
      {
        "id": 1,
        "airline": "British Airways",
        "aircraft": "737"
      },
      {
        "id": 2,
        "airline": "Emirates",
        "aircraft": "888"
      }
    ]
  }
}

const getFlights = () => {
  // this would repeatedly run every 3 seconds while the window is open!!
  const flightsInterval = setInterval(getFlightDetails, 3000);

  clearInterval(flightsInterval);
}
```

Which one we you use and when is based on the type of application you need to write.

For example, if you want to run code once, use `setTimeout`. If you want to repeatedly run code, use `setInterval` (you may want to clear the interval, but again this is down to your application requirements).

When you may not need a clear interval? Imagine having the following:

status-example

We may want to keep polling our back end server to tell us whether the back end services are active or not (say every 5 minutes). This is a pattern used in the industry.

Another example may be as you have seen, in an airport, you see the flight details appearing on those monitors or in the train station showing you the up to date train times. This is an example of where you may repeatedly retrieve data to display to the end user.

flight-times

Synchronisation

JavaScript is a single threaded language in that all processes if you want to think of it, run on one road.

When we load a website page, JavaScript interprets the page from top to bottom meaning it needs to know what is on the page and how to render it.

Can you imagine trying to load a web page and it loading gradually and in stages?

Imagine Instagram and the pictures all showing one by one rather than all at once. I don't think it would have as much popularity!

When a page loads and items on that page render sequentially, this is known as synchronous. We request something, we then wait for it on the page while it loads (meaning we cannot do anything else other than wait - how frustrating/poor would this experience be). This is how it used to be a long time ago but things have changed ...

In order to allow for items on a website to be loaded at any given time without **blocking** the user's experience, we have to adopt a way that let's us achieve this. In JavaScript, this is referred to as **asynchronous**.

If you imagine a road, a single lane is how you can visualise synchronous. No car can overtake another despite the cars behind being substantially faster than the one at the front:

You may have come across (or may not have) something called `AJAX`.

What is it exactly?

> Asynchronous JavaScript and XML - We don't really care about the XML part because let's be honest no one really uses it now and if they do, they need to move on.

> XML is a lightweight format of transferring data. We won't be using XML but instead JSON.

Now sending and receiving data from a web browser we need to find a way of not *blocking* the user's experience (i.e. letting the user continue to do their actions without stopping the browser from rendering further).

Synchronous Vs Asynchronous

Let's look at how synchronous would work:

```
function otherFunction() {
    console.log("We are in another function");
    console.log("Do something");
}

console.log("Start");

otherFunction();

console.log("End");
```

When we run this code, JavaScript adds these steps to a call stack. Each command is added onto of the call stack as we continue processing from top to bottom. Each item gets added, then removed.

So how can we see the same but in an asynchronous manner? We can immitate it using the `setTimeout` handler.

```
console.log("Start");

setTimeout(() => {
    console.log("Waiting ...");
}, 2000); // wait 2 seconds

console.log("End");
```

What do you expect to see? `Start, Waiting ..., End`? Wrong - what actually gets printed is: `Start, End, Waiting...`.

No code was blocked but if we look at the call stack for this process, it doesn't make sense really. JavaScript is very bad at doing multiple things at once.

Well what JavaScript does is make use of the browsers' in built Web APIs to push the `setTimeout` into that space and on completion it gets pushed back onto the call stack.

In real life operations we cannot determine how long something will take to process. We can't guarantee for example when searching for an item on a website, how long that process will take for it to return results.

> JavaScript is really poor at doing multiple things at once by default so we need to do something to allow us to 'multi-task'.

There are a number of ways thankfully that JavaScript has to help alleviate this asynchronous way of working.

In order to illustrate how we can best handle asynchronous operations we'll look at different ways using a few use case scenario: `Netflix`.

Use Case: Netflix

Imagine we are working at Netflix and we want to be able to retrieve user details when the user provides a username/email and password. On the back of this task we want to obtain their liked programme list. How would we do this in an asynchronous manner?

Given the following we can start to orchestrate our scenario:

```
console.log("Start");

const loginUser = (username, password) => {
  // simulate a server checking the username and password:
  setTimeout(() => {
    console.log('Now we have the data username!');
    return { account: username };
  }, 3000);
};

const loggedInUser = loginUser("johndoe@foo.com", 12324);
console.log("loggedInUser", loggedInUser);

console.log("Finish");
```

When we run this code, we expect to see the `loggedInUser` is set but instead we get `undefined`. Why? well because the process of retrieving the user hasn't completed in the time we opened the browser. We just got the following:

```
Start
undefined
Finish
Now we have the data username!
```

Callbacks: Netflix

fonejacker

To get around the issue of receiving the data in an asynchronous manner, we can use something called a `callback`. A callback is simply a function that you pass to act as the receiver for when your asynchronous code/logic is complete. We can pass a callback function to another function (i.e. to the one that is performing the asynchronous logic).

To illustrate actions being performed in a random sequence, the `setTimeout` function.

For example, we can now do this:

```
console.log("Start");

const loginUser = (username, password, callback) => {
  // simulate a server checking the username and password using setTimeout:
  setTimeout(() => {
    console.log('Now we have the data username!');
    callback({ account: username });
  }, 3000);
};

const loggedInUser = loginUser("johndoe@foo.com", 12324, (userDataReceived) => {
    console.log("userDataReceived", userDataReceived);
});
```

```
console.log("Finish");
```

The `callback` parameter name can be anything but is used as a `function`.

So now you see we have the user account after 3 seconds. What if we now wanted to retrieve the liked programmes?

```
console.log("Start");

const loginUser = (username, password, callback) => {
  // simulate a server checking the username and password using setTimeout:
  setTimeout(() => {
    console.log('Now we have the data username!');
    callback({ account: username });
  }, 3000);
};

const retrieveUserLikedProgrammes = (username, callback) => {
  // simulating the retrieval of liked programmes using setTimeout:
  setTimeout(() => {
    callback(["Breaking Bad", "24", "Friends"]);
  }, 1000);
};

const loggedInUser = loginUser("johndoe@foo.com", 12324, (userDataReceived) => {
    console.log("userDataReceived", userDataReceived);

    retrieveUserLikedProgrammes(userDataReceived.account, (likedProgrammes) => {
        console.log("likedProgrammes", likedProgrammes);
    })
});

console.log("Finish");
```

Ok now what if we wanted to retrieve the details for a liked programme? Well we know the programmes now so we can perform another check (imitate another server request):

```
console.log("Start");

const loginUser = (username, password, callback) => { //another callback
  // simulate a server checking the username and password using setTimeout:
  setTimeout(() => {
    console.log('Now we have the data username!');
    callback({ account: username });
  }, 3000);
};

const retrieveUserLikedProgrammes = (username, callback) => { //another callback
  // simulating the retrieval of liked programmes using setTimeout:
  setTimeout(() => {
    callback(["Breaking Bad", "24", "Friends"]);
  }, 1000);
};

const retrieveProgrammeDetail = (programmeName, callback) => { //another callback
    // simulating the retrieval of programme details using setTimeout:
```

```
    setTimeout(() => {
        callback({
            duration: "180m",
            supportedLanguages: ["en", "fr"]
        })
    }, 2000);
}

const loggedInUser = loginUser("johndoe@foo.com", 12324, (userDataReceived) => {
    console.log("userDataReceived", userDataReceived);

    retrieveUserLikedProgrammes(userDataReceived.account, (likedProgrammes) => {
        console.log("likedProgrammes", likedProgrammes);

        retrieveProgrammeDetail(likedProgrammes[0], (details) => {
            console.log("Details for programme", details);
        })
    })
});

console.log("Finish");
```

Great! we are getting all the data we want. However what you will see is that you end up with this weird *nested* structure:

```
retrieveUserLikedProgrammes(userDataReceived.account, (likedProgrammes) => {
        console.log("likedProgrammes", likedProgrammes);

        retrieveProgrammeDetail(likedProgrammes[0], (details) => {
            console.log("Details for programme", details);

            retrieveBlahA(..., () => {
                // A ...
                retrieveBlahB(..., () => {
                    /// B ...
                    retrieveBlahC(..., () => {
                        // C ...
                        retrieveBlahD(..., () => {
                            /// D ..
                        })
                    })
                })
            })
        })
    })
```

This is what we call `callback hell`. So how do we get around improving this? Well thankfully we can start doing this by using `Promise` objects.

One thing we haven't looked at is how do we handle scenarios where we may or may not get the data back due to say a failure in the process? Well we could further expand our callback way to do this:

```
console.log("Start");

const actualPasswordFromDB = 123456; // for example purposes

const loginUser = (username, password, onSuccess, onFailure) => { //another callback
```

```
    // simulate a server checking the username and password using setTimeout:
    setTimeout(() => {
        console.log('Now we have the data username!');

        if (password !== actualPasswordFromDB) {
            onFailure('Not found');
        }
        else {
          onSuccess({ account: username });
        }
    }, 3000);
  };

  // other code not shown
```

Now there is nothing wrong with this approach, but what we now have to do is provide **two** functions as callbacks. One to handle successes and the other, failures. This may seem fine for one process but can you imagine doing this on a multi-step process? It becomes super complex!

Thankfully the JavaScript folk introduced a more sophisticated approach in the form of a `Promise` object.

Promises: Netflix

A `Promise` is an object that simply returns a result of an asynchronous operation of a failure.

So how do we use a `Promise`? Well a `Promise` has two functions that we can use (think of them as two special callback functions).

Let's start with a simple example:

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve({message: 'Hello!!!!'});
    }, 3000);
})

promise.then(message => {
    console.log(message);
});
```

Run this code snippet. You'll see it runs the code after 3 seconds but looks much simpler when invoked 🆒

So how do we handle scenarios when the coe doesnt return a result, but instead an error?

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject(new Error('Failed to find data')); // good practice to return an Error
object
    }, 3000);
})

promise.then(message => {
    console.log(message);
})
```

```
.catch(err => console.error(err)); // here we catch any Error thrown and simply log it
to the console window
```

To access the `Error` object message only, you can do `err.message`.

So how do we use Promises in our Netlfix example:

```
console.log("Start");

const loginUser = (username, password) => {
    return new Promise((resolve, reject) => {
        // simulate a server checking the username and password using setTimeout:
        setTimeout(() => {
            console.log("Now we have the data username!");
            resolve({ account: username });
        }, 3000);
    });
};

const retrieveUserLikedProgrammes = (username) => {
  return new Promise((resolve, reject) => {
    // simulating the retrieval of liked programmes using setTimeout:
    setTimeout(() => {
        resolve(["Breaking Bad", "24", "Friends"]);
    }, 1000);
  });
};

const retrieveProgrammeDetail = (programmeName) => {
  return new Promise((resolve, reject) => {
    // simulating the retrieval of programme details using setTimeout:
    setTimeout(() => {
        resolve({
            duration: "180m",
            supportedLanguages: ["en", "fr"],
        });
    }, 2000);
  });
};

// now let's perform our steps:
loginUser("johndoe@foo.com", 12324)
.then(userDataReceived => retrieveUserLikedProgrammes(userDataReceived.account))
.then(likedProgrammes => retrieveProgrammeDetail(likedProgrammes[0]))
.then(details => console.log("Details for programme", details));


/*
const loggedInUser = loginUser("johndoe@foo.com", 12324, (userDataReceived) => {
  console.log("userDataReceived", userDataReceived);

  retrieveUserLikedProgrammes(userDataReceived.account, (likedProgrammes) => {
    console.log("likedProgrammes", likedProgrammes);

    retrieveProgrammeDetail(likedProgrammes[0], (details) => {
      console.log("Details for programme", details);
    });
  });
});
*/
```

```
console.log("Finish");
```

How clear and concise does this now look? 👌🏼 😁

Promises: Give me everything!

What if we faced a situation where we needed to retrieve multiple operation results at the same time in order to continue? Well thankfully using the `Promise` object we can by using the `.all` function.

Let's replicate another scenario: Obtaining various user details to access the Government website (disclaimer - this is not how they do things but presents the idea).

```
const checkRoadTax = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log('Checked road tax');
            resolve({roadTaxDue: false});
        }, 5000);
    });
}

const checkCouncilTax = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log('Checked council tax');
            resolve({councilTaxDue: false});
        }, 3000);
    });
}

Promise.all([checkCouncilTax, checkRoadTax]) // we can run both promises and get all
responses
.then(results => console.log(results));
```

Although the example for `Promise.all` was shown it used functions that did not take any arguments. Therefore the `Promise.all` should be used only when necessary based on the types of promises we are trying to resolve.

We've done callbacks and Promises but what would you say if I said there is an even simpler approach introduced in late ES5/early ES6..... `Async/Await`.

Async/Await (ES6): Netflix

Let's refactor our example to use `Async/Await` then:

```
console.log("Start");

const loginUser = (username, password) => {
    return new Promise((resolve, reject) => {
        // simulate a server checking the username and password using setTimeout:
        setTimeout(() => {
            console.log("Now we have the data username!");
            resolve({ account: username });
        }, 3000);
    });
};

const retrieveUserLikedProgrammes = (username) => {
  return new Promise((resolve, reject) => {
    // simulating the retrieval of liked programmes using setTimeout:
    setTimeout(() => {
        resolve(["Breaking Bad", "24", "Friends"]);
    }, 1000);
  });
};

const retrieveProgrammeDetail = (programmeName) => {
  return new Promise((resolve, reject) => {
    // simulating the retrieval of programme details using setTimeout:
    setTimeout(() => {
        resolve({
            duration: "180m",
            supportedLanguages: ["en", "fr"],
        });
    }, 2000);
  });
};

const performUserDetailsRetrieval = async () => {
    const loggedInUser = await loginUser("johndoe@foo.com", 12324);
    const likedProgrammes = await retrieveUserLikedProgrammes(loggedInUser.account);
    const programmeDetails = await retrieveProgrammeDetail(likedProgrammes[0]);

    console.log(loggedInUser, likedProgrammes, programmeDetails);
}

performUserDetailsRetrieval();

// Using Promises we had:
/*
loginUser("johndoe@foo.com", 12324)
.then(userDataReceived => retrieveUserLikedProgrammes(userDataReceived.account))
.then(likedProgrammes => retrieveProgrammeDetail(likedProgrammes[0]))
.then(details => console.log("Details for programme", details));
*/

/*
const loggedInUser = loginUser("johndoe@foo.com", 12324, (userDataReceived) => {
  console.log("userDataReceived", userDataReceived);

  retrieveUserLikedProgrammes(userDataReceived.account, (likedProgrammes) => {
    console.log("likedProgrammes", likedProgrammes);

    retrieveProgrammeDetail(likedProgrammes[0], (details) => {
      console.log("Details for programme", details);
    });
  });
});
*/

console.log("Finish");
```

When using `Async/Await` you have to make sure the function that calls the logic is marked using the keyword `async` so that the interpreter knows to expect some asynchronous code.

So how do we handle errors in `Async/Await`? Simple, we just wrap our logic in an a `try/catch`:

```javascript
// other code not shown...

const performUserDetailsRetrieval = async () => {
    try {
        const loggedInUser = await loginUser("johndoe@foo.com", 12324);
        const likedProgrammes = await retrieveUserLikedProgrammes(loggedInUser.account);
        const programmeDetails = await retrieveProgrammeDetail(likedProgrammes[0]);

        console.log(loggedInUser, likedProgrammes, programmeDetails);
    } catch (err) {
        console.error('Unable to retrieve user details', err);
    }
}

// ...
```

Making HTTP Requests

Before we delve into the practical aspects of this section, its best to first understand what HTTP requests are, the types of requests we can make and the type of data we can send/receive.

Overview

Hypertext Transfer Protocol is an application-layer protocol for transmitting data. It was designed for communication between web browsers and web servers but it can be used for other purposes.

It was invented alongside HTML to create the first interactive, text-based web browser: the original World Wide Web. There is also a more secure protocol in HTTPS which allows for data to be transferred but in a more encrypted/secure manner.

HTTP Request Methods

- `GET` requests a specific resource in its entirety
- `HEAD` requests a specific resource without the body content
- `POST` adds content, messages, or data to a new page under an existing web resource
- `PUT` directly modifies an existing web resource or creates a new URI if need be
- `DELETE` gets rid of a specified resource
- `TRACE` shows users any changes or additions made to a web resource
- `OPTIONS` shows users which HTTP methods are available for a specific URL
- `CONNECT` converts the request connection to a transparent TCP/IP tunnel
- `PATCH` partially modifies a web resource

All HTTP servers use the `GET` and `HEAD` methods but not all support the rest of the methods listed.

So how does the front end communicate with the back end?

Figure: Overview of a web request response cycle: web-request-response

Use Case Scenario: Shopping online on ASOS

As illustrated in the diagram above, we'll cover the process involved in following a customer journey to how steps may be carried out in *front end* and *back end* aspects.

1. User opens a browser and types in ASOS.
2. The search engine provides a hit for ASOS and the user clicks to request the HTML (homepage) for that domain.
3. The webserver, serves the user with that homepage.
4. When a user selects a product and decides to purchase it, he/she adds it to their shopping basket (which at this point in time is stored in the browser session - if the user closes the browser, the basket will be empty).
5. The user at this point has not logged in and decides to checkout.
6. When the user tries to checkout, he/she is prompted to either login or register (for the sake of simplicity, we'll assume this is a user who has already registered).
7. The user clicks on the 'login' button and is redirected to the login form.
8. He/she then enters their credentials and clicks on submit.
9. These details are sent to the *back end* using a HTTPS `POST` request with information relating to the username and password used, for verification against a database which the server-side code will perform checks against.
10. The *back end* will then send a HTTP response containing details linked to that user so they can be stored in a session (cookie).
11. Now the *frontend* knows the user is genuine so the user is redirected (usually this is done using a POST/GET sequence) back to their shopping basket.
12. As the basket was previously stored in the session (maybe as a cookie), the user then fills in their address and bank details.
13. Again the details are submitted after the user presses 'Purchase'.
14. The *back end* receives the details that include the items sought, the address of the user as well the bank details.
15. The *back end* will then send HTTPS requests to the bank merchant in order to verify the purchase (i.e. does the person have enough funds). Assuming here they do.
16. Once bank funds are verified, the *back end* then processes the order and sends a request off to the *distribution team* in order to fulfil the order. Subsequently, the order number is generated and returned back to the user in the browser (with the usual 'Thank you for your purchase ...' message).

Please note this is just a very simplified process overview. However it does give you an idea as to the communication that takes place between the *front end* and *back end* aspects of a system.

In summary:

The *front end* should:

- Be responsible to displaying the correct page on request.
- Be responsible for allowing for details to obtained and submitted in a secure manner.
- Be responsible for handling errors to feed back to the user if something does not go correctly.
- For the general look and feel of a website/mobile app/interface etc.

The *back end* should:

- Be responsible for obtaining and handling HTTP(S) requests. This depends on which methods the server supports, i.e. `GET`, `POST` etc.

- Be responsible for the business logic
- Be responsible for communicating with other external entities, this includes the database.
- Be responsible for managing interactions and response from external entities and sending them to the *front end* in a common/agreed format.

These points are further illustrated in the following diagram that shows the different layers in a `pre-Single Page Application (SPA)` web application:

web-application-layers

For Single Page Applications (i.e. written in say React, Vue, Angular etc), the following architecture is seen:

web-application-layers-spa

Again there are caveats to this so it comes down to each organisation's interpretation of these layers and where they best see fit for that area to reside. For example, the data access layer may live in on the server for most organisations and the front end (client) merely pulls that data using APIs (Application Programming Interfaces).

Making HTTP(s) Requests

We have seen that there are multiple HTTP methods we can use in order to send or receive data. Let's look at how we can achieve this in the code.

To make HTTP(s) requests in JavaScript, we can use the `Fetch` API.

The Fetch API provides an interface for fetching resources and provides a generic definition of how we send data and how we receive it.

Previously we touched upon the need to use JSON as our data transfer type. Other formats of data transfer are possible such as XML, files, plain text, streamed content, however to keep things simple, we will be working with and using JSON as it's one of if not the most popular forms of data transfer.

To use the Fetch API, we simply make use of the `fetch` function/method. The `fetch` function returns a promise (which if you recall can either be a result or an error).

```
fetch('some URL')
    .then(response => response.json())
    .then(data => console.log(data));
```

The gotcha in using the `fetch` method is that if successful, it returns a promise response which in turn returns a promise (hence the double `then`). The first resolved promise is function we need to call in order to get the response as JSON. The last `then` is then used to receive the JSON (here `data` is just a variable name, it could be anything you wanted).

As mentioned, we need to ensure we make use of the `fetch` API in an asynchronous manner. Out of the box it will run like you're using Promises.

So let's make use of the `fetch` method but using the `Async/Await` approach (cleaner and preferred):

```
const exchangeRatesByCountryURL = `https://open.er-api.com/v6/latest/GBP`;

// remember you need an async function that triggers the first call:
const retrieveExchangeRates = async () => {
    const jsonResponse = await fetch(exchangeRatesByCountryURL);
    const data = await jsonResponse.json();

    console.log('Rates against GBP are: ', JSON.stringify(data, null, 2));
}
```

By default the fetch API makes a `GET` request (which is what the example above uses)

> To view returned JSON in a user-friendly manner you can use this site. For personal use this is fine but try not to use it within your organisation unless you're allowed to. Alternatively, you can just use `console.log(JSON.stringify(responseObj, null, 2))`

Challenge: Custom Rates

Given that you should by now know how to manipulate the DOM and have just seen how to perform a HTTP request,

Create an application using HTML and JavaScript that allows the user to enter their currency and render the results of the rates that match that currency.

Inside the session-5 folder, create a folder `custom-rates` and add your solution inside.

Hint: I have already given you a head start in the JavaScript in the previous section

For those CSS enthusiasts, feel free to jazz up your solutions.

Paired Challenge - Weather Application

Create a simple weather application that requires a location and then calls a Weather API to render the forecast. Display the forecast in an appropriate output on the screen.

Also cater for error cases where the user enters a place that does not exist (i.e. 'StarWars').

API: https://openweathermap.org/api

You will need to register.

An example of the API is:

http://api.openweathermap.org/data/2.5/forecast?id=524901&appid={API key}

The API key will be the one you get once you register.

Demo: Json Server for faking back end data: json-server