Chapter 8: Asynchronous JavaScript: Promises, AsyncAwait and AJAX:

1. Section Intro:

Asynchronous JavaScript is a fancy term for some code that keeps running in the background. while our main code is still executing. Typically, that is for stuff like requesting some data from a remote server, like an API

2. An example of Asynchronous JavaScript:

index.html:

```
...
<body>
    <h1> Asynchronous JavaScript </h1>
    <script>
        Const second = () => {
            setTimeout(() => {
                console.log('Async Hey there');
            }, 2000);
```

```
}

const first = () => {
    console.log('Hey there');
    second();
    console.log('The end');
}

first();

</script>
</body>
```

Output: Hey there

The end

Async Hey there ← This comes after 2 seconds.

setTimeout function is a function that we can pass a callback and also the time. First is the callback. and the second argument we pass into the setTimeout function is for how long we want the timer to run in

milliseconds. Here we put 2000 milliseconds which means 2 seconds. After this 2 seconds have passed, the callback function that we passed will run.

3. Understanding Asynchronous JavaScript : The Event Loop:

Synchronous vs. Asynchronous:

Synchronous:

```
const second = () => {
  console.log('How are you doing?');
};

const first = () => {
  Console.log ('Hey there!');
  second ();
  console.log ('The end');
};

first ();
```

Asynchronous:

```
const image = document.
  getElementById ('img').src;

processLargeImage (image, () => {
  console.log ('Image processed!');
});
```

Suppose we select an image from our DOM and pass it into a process Large Image function. We know that this function is gonna take some time to process the image. Just like before, we don't want the code to have to wait. We don't want it to stop while the image is processing. What we do here is to also pass in a callback function that we want to be called as soon as the function is done processing. Just like that, we have created asynchronous code.

Asynchronous:

→ Allow asynchronous functions to run in the background.

→ We pass in callbacks that run once the function has finished its work.

→ Move on immediately: Non-blocking.

How does all this work behind the scenes of JavaScript. That's where the Event Loop comes in.

Consider this code:

```
const second = () => {
    setTimeout(() => {
        console.log('Async Hey there!');
    }, 2000);
};

const first = () => {
    console.log('Hey there!');
    second();
    console.log('The end');
};

first();
```

The Event Loop: The Event Loop is part of the bigger picture of what happens behind the scenes of JavaScript. When we

call functions and handle events like DOM events. We already talked about the Execution stack and Message Queue when we first talked about events. What is here is the Event Loop as well as the Web APIs which together with the Execution Stack and the Message Queue make up our JavaScript runtime. This runtime is responsible for how JavaScript works behind the scenes as it executes our code. It's extremely important to understand how all these pieces fit together in order to execute Asynchronous JavaScript.

Let's now take a look at how the code from the last lecture is executed inside our JavaScript engine, line by line.

→ It starts by calling the first function and and execution context for that function is put on top of the execution stack, which can also be called the call stack.

→ In the next line of code,.

```
Console.log('Hey there!');
```

console.log function is called and a new execution context is created and the text is logged to the console. Then the function returns and the execution context pops off the stack.

→ Moving on to the second function, a new execution context is created and in the next line, the setTimeout function is called. This causes yet an another execution context to be created.

→ But where does this setTimeout function actually come from? It's part of something

called the Web APIs, which actually live outside the JavaScript engine itself. Stuff like DOM manipulation methods, setTimeout, HTTP requests for AJAX, geoLocation, local storage and tons of other things actually live outside of the JavaScript engine. We just have access to them because they are also in a JavaScript runtime. This is exact where the timer will keep running for two seconds, asynchronously of course, so that our code can keep running without being blocke

→ When we call the setTimeout function, the timer is created together with our callback function right inside the Web APIs environment. There it keeps sitting until it finishes its work all in an asynchronous way.

→ The callback function is not called right now but instead it stays attached to the

timer until it finishes. Since the timer keeps working in the background, we don't have to wait and can keep executing our code.

→ Next, the setTimeout function returns, pops off the stack and so does the execution context of the second function which returns as well. We're now back to the initial first function. Now we just log the end to the .console and we give ourselves a new execution context, print the text to the console and pop the context off again. Next, the function returns and we're back to our original state.

→ Rightnow, we have executed all our code in a synchronous way and have the timer running asynchronously in the background.

→ Let's suppose our two seconds have passed

and the timer disappears. But what happens to our callback function now? It simply moves to the message queue where it waits to be executed as soon as the execution stack is empty. This is exactly what happens with DOM events as well. That's because it actually works the exact same way. In the case of DOM events our listeners sit in the web APIs environment waiting for a certain event to happen. As soon as that event happens, then the callback function is placed on a message queue ready to be executed. But how are these callback functions in the message queue executed?

→ That's where, finally, the Event Loop comes in. The job of the event loop is to constantly monitor the message queue and

the execution stack and to push the first callback function in line onto the execution stack as soon as the stack is empty.

→ In our example here, right now the stack is empty. And we have one callback function waiting to be executed. So the event loop takes the callback and pushes it onto the stack where a new execution context is created for that function. That's what the event loop does. Inside that callback function now, we simply run this:

```
console.log('Async Hey There!');
```

this logs 'Async Hey There!' to the console. Then the context pops off the stack and we're done.

→ If there were some more callbacks waiting right now, like data coming back

from an AJAX request, or the handler of a DOM event, then the event loop would continue pushing them onto the stack until all of them were processed. That's how the event loop works.

Please note that the setTimeout function that we used here as an example is to simulate a more real-world asynchronous function like doing an AJAX request to fetch some data from an external API, just like we're gonna do in one of the next lectures.

4. The Old Asynchronous JavaScript with Callbacks:

Asynchronous JavaScript with callbacks is a traditional way of dealing with asynchronous code. But there is a drawback called callback hell.

In this code, we are simulating loding a data from a remote web server using

setTimeout.

script.js:

```
function getRecipe(){
    setTimeout(() => {
        const recipeID = [523, 883, 432, 974];
        console.log(recipeID);

        setTimeout(id => {
            const recipe = {title: 'Fresh tomato
              pasta', publisher: 'Jonas'};
            console.log(`${id}: ${recipe.title}`);

            setTimeout(publisher => {
                const recipe2 = {title: 'Italian
                  Pizza', publisher: 'Jonas'};
                console.log(recipe);
            }, 1500, recipe.publisher);

        }, 1500, recipeID[2]);

    }, 1500);
}
getRecipe();
```

Output: ▶ (4) [523, 883, 432, 974] ← This comes after 1.5 seconds

432: Fresh tomato pasta ← This comes after 1.5 seconds after the above one

▶ {title: "Fresh tomato pasta", publisher: "Jonas"}

← This comes 1.5 seconds after the above one.

This code is like having three chained AJAX calls to get some data from the server. This is getting a bit out of hand. Imagine that we had more and more chaining like 10 levels, then we would have all of the callbacks inside of one another. That is the so-called "Callback Hell" in JavaScript. The triangular shape in the code is a sign of callback hell.

For this in ES6, "Promises" were introduced. With promises, we can avoid all of this.

5. From Callback Hell to Promises:

What are Promises?

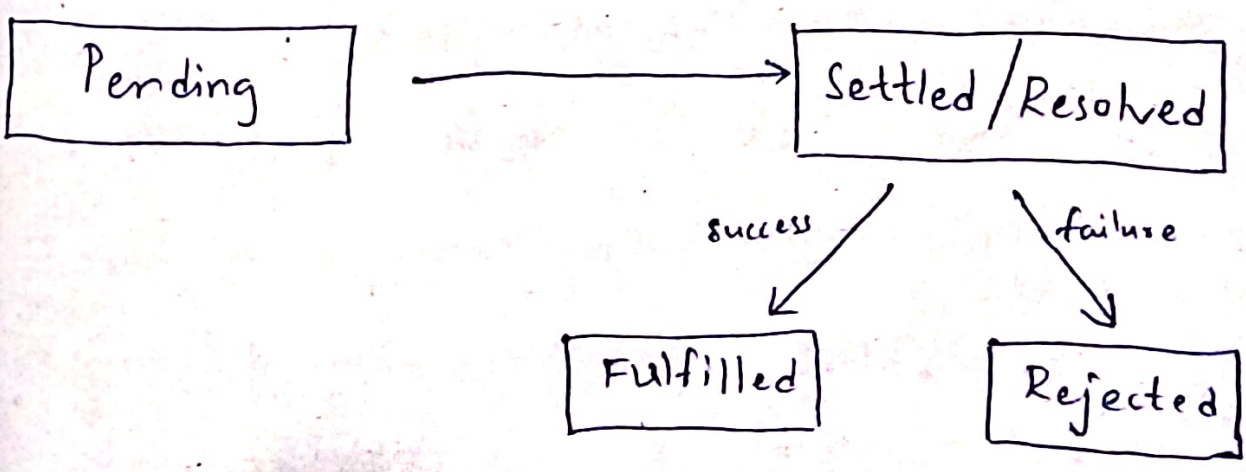
→ Object that keeps track about whether a certain event has happened already or not.

→ Determines what happens after the event has happened.

→ Implements the concept of a future value that we are expecting.

It's like us saying: Hey, get me some data from the server in the background and the promise then promises us to get that data so we can handle it in the future.

Promise States:



A promise can have different states.

→ Before the event has happened, the promise is pending.

→ After the event has happened, the promise is called settled or resolved.

→ When the promise is successful, which means that a result is available, then the promise is fulfilled. But if there was an error then the promise is rejected.

This is important to know because we will then be able to handle these two different situations in our code.

In more practical terms, we can produce and consume promises. When we produce a promise, we create a new promise and send a result using that promise. Then when we consume it, we can use callback functions for fulfillment and for rejection of our promise.

script.js: (view this coding video again)

```
const getIDs = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve([523, 883, 432, 974]);
    }, 1500);
});

const getRecipe = recID => {
    return new Promise((resolve, reject) => {
        setTimeout(ID => {
            const recipe = { title: 'Fresh tomato
                pasta', publisher: 'Jonas' };
            resolve(`${ID}: ${recipe.title}`);
        }, 1500, recID);
    });
};

const getRelated = publisher => {
    return new Promise((resolve, reject) => {
```

```
        setTimeout(pub => {
            const recipe = {title: 'Italian Pizza',
                publisher: 'Jonas'};
            resolve(`${pub}: ${recipe.title}`);
        }, 1500, publisher);
    });
};

getIDs
.then(IDs => {
    console.log(IDs);
    return getRecipe(IDs[2]);
})
.then(recipe => {
    console.log(recipe);
    return getRelated('Jonas Schmedtmann');
})
.then(recipe => {
    console.log(recipe);
```

```
})
.catch(error => {
    console.log('Error!!');
});
```

Output: ▶(4) [523, 883, 432, 974] ← after 1.5 secs

432: Fresh tomato pasta ← after 3 secs

Jonas Schmedtmann: Italian Pizza.

↑ after 4.5 secs.

In this code, we have produced all of our promises and then in the end we simply consumed them with the chain of "then"s and finally then a catch to catch if there is any error.

6. From Promises to Async Await:

The syntax to consume promises is still confusing and difficult to manage. In ES8,

something called Async/Await was introduced to make promises easier to consume.

Async/Await was designed for us to consume promises, and not to produce them. To produce, we can just do it in the way we learned in the last lecture.

script.js: use the same code from the last lecture to produce promises:

```
Const getIDs ....
    :

};

async function getRecipesAW() {
    const IDs = await getIDs;
    console.log(IDs);

    const recipe = await getRecipe(IDs[2]);
    console.log(recipe);
```

```
    const related = await getRelated('Jonas Schmedtmann');
    console.log(related);

    return recipe;
}

getRecipesAW().then(result => console.log(`${result} is the best ever!`));
```

Summary of the code:

→ We created an async function using the async keyword.

→ Inside an async function, we can have one or more await expressions.

→ To consume our first promise, we wrote this

```
const IDs = await getIDs;
```

getIDs is our first promise. What happens here is that the await expression will stop the code from executing at this point

until the promise is fulfilled. In this case it it getIDs promise. If the promise is resolved, then the value of the await expression is the resolved value of the promise, which is then assigned to the IDs varinble.

→ Then we console logged IDs.

→ Again, what happens here is that when we call the getRecipesAW function, it keeps running asynchronously in the background. With await, we wait for this promise to resolve, assign a value to IDs variable and log it to the console.

→ We did the same for the other two promises.

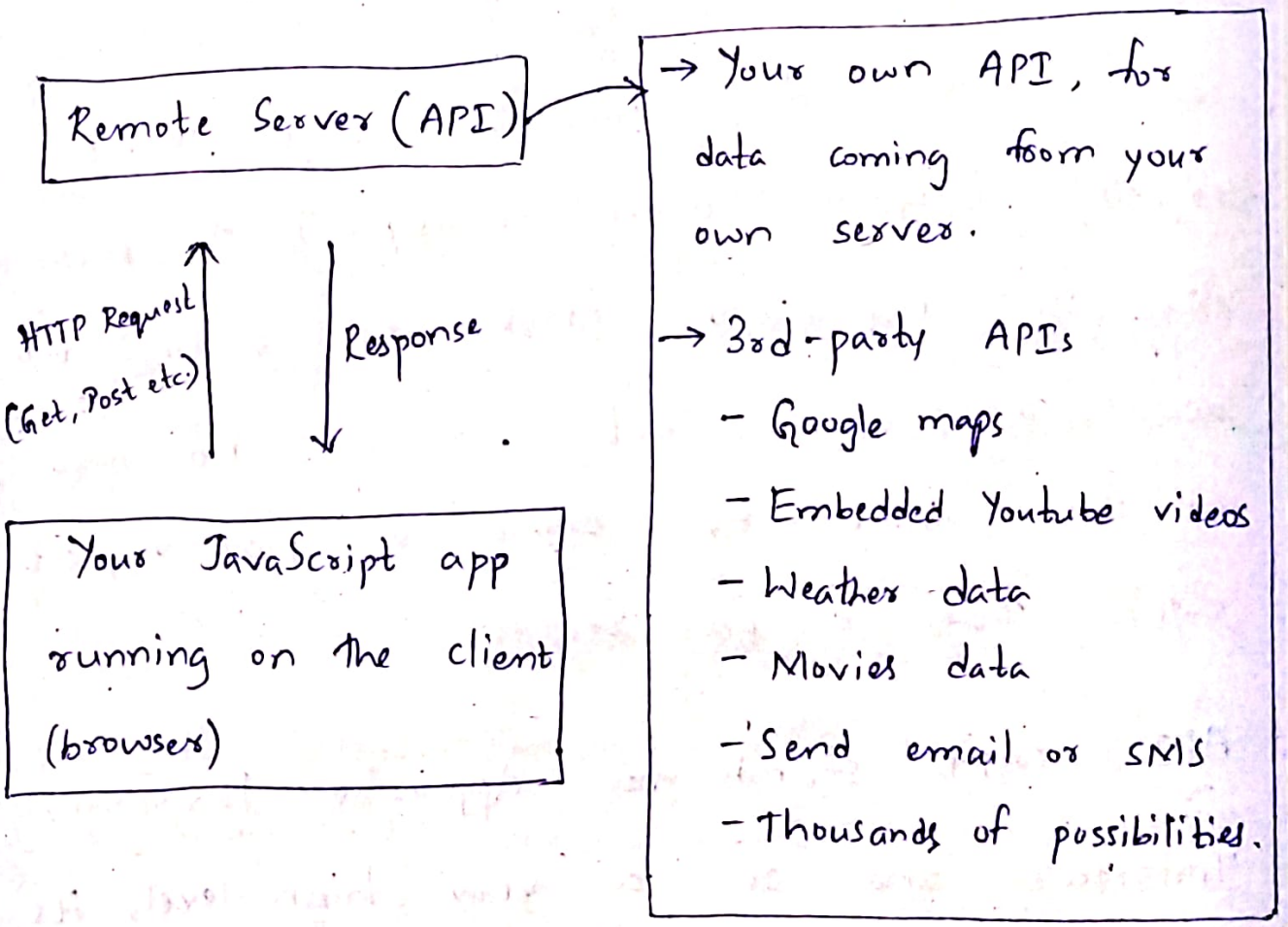
→ In the last line, we used 'then' which is to consume the promise which is possible simply because of the fact that an async

function always returns a promise that we can then consume.

7. AJAX & APIs:

AJAX (Asynchronous JavaScript And XML)

API (Application Programming Interface)



AJAX : It stands for Asynchronous JavaScript And XML. It allows us to asynchronously communicate with remote servers.

Let's say we have our JavaScript app running in the browser, which is called the client. We want the app to get some data

from our server without having to reload the entire page. With AJAX, we can do a simple get HTTP request to our server which will then send back a response containing the data that we requested. All of this happens asynchronously in the backgrour Also, this not only works for getting data from the server, but also to send data to the server by doing a post request rather than a get request.

API: API stands for Application Programming Interface. and on a very high level, it's basically a piece of software that can be used by another piece of software in order to allow applications to tak to each other.

In reference to web development and AJAX and servers, the API is not the server itself but its like a part of the server. Like an application that receives requests

and sends back responses.

We have two types of APIs in JavaScript.

→ Our own APIs.

→ 3rd party APIs like Google Maps, Weather data etc.

8. Making AJAX Calls with Fetch & Promises:

script.js:

```
function getWeather(woeid) {
    fetch(`https://crossorigin.me/https://www.
        metaweather.com/api/location/${woeid}/`)
    .then(result => {
        console.log(result);
        return result.json();
    })
    .then(data => {
        const today = data.consolidated_weather[0];
        console.log(`Temperatures in ${data.title}
```

```
    stay between ${today.min_temp} and
    ${today.max_temp}.`);
  })
  .catch(error => console.log(error));
}

getWeather(2487956);
getWeather(44418);
```

Summary of the code:

→ We have used metaweather API to fetch weather data.

→ Install JSON Viewer Chrome extension.

→ To avoid the error of same origin policy in JavaScript, prefix the url with this:

https://crossorigin.me/

→ Look at this code:

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

The callback function here has one argument and that is the resolved value of the promise.

What we need to know is that the Fetch API gets our data and returns a promise. So we can use the 'then' and 'catch' methods on this promise.

We called our callback function's argument as result. Those lines means that the data that comes back from this fetch AJAX request will be called result in the callback function. Then we logged it to the console.

→ This line:

```
return result.json();
```

will return a promise because this happens asynchronously. What this does is it converts the result into JSON. It might take some time. So it happens asynchronously.

→ To handle what comes from this:

return result.json();

we have used another then method.

9. Making AJAX Calls with Fetch & AsyncAwait:

We did an AJAX call with fetch and ES6 promises, let's now use async/await to consume the promises in an easier way.

script.js:

```
async function getWeatherAW(woeid) {
    try {
        const result = await fetch(`https://...
            ...);
        const data = await result.json();
        Const tomorrow = data.consolidated_weather[1
        console.log(`Temperatures tomorrow in
          ${data.title} stay between ${tomorrow.
            min_temp} and ${tomorrow.max_temp}`);
        return data;
```

```
  } catch (error) {
      alert (error);
  }
}

getWeatherAW (2487956);

let dataLondon;
getWeatherAW(44418).then (data => {
    dataLondon = data
    console.log(dataLondon);
});
```

Summary of the code:

→ Look at this line:

```
const result = await fetch(....);
```

We have created a new variable called result and then awaited the fetch. So the fetch will get the data from the server while the execution in the function stops.

Once it's done, once the promise is fulfilled, it will then assign a result of the promise to the 'result' variable.