

ES6 Part 2

IMY 220 • Lecture 9

ES6 – Promises

Promises give us a way to deal with asynchronous behaviour, such as timeouts, AJAX calls, etc.

“A Promise is an object representing the eventual completion or failure of an asynchronous operation”

“Essentially, a Promise is a returned object to which you attach callbacks, instead of passing callbacks into a function”

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

ES6 – Promises

The Promise object constructor is as follows

```
const examplePromise = new Promise(function(resolve, reject){  
    // asynchronous operation goes here  
  
    // if everything goes right:  
    resolve(optionalReturnData);  
  
    //if something goes wrong:  
    reject(otherReturnData);  
});
```

ES6 – Promises

How it works (broadly): instead of passing callback functions for a function to call when it finishes an asynchronous function (or fails)...

```
// example: loadUserDetails fetches user data with an AJAX call  
loadUserDetails(userid, successCallback, failCallback);
```

...the asynchronous function returns a *Promise*, which is dealt with when it finishes (or fails)

```
// in this example, we're assuming loadUserDetails returns a  
// promise (we'll look at returning promises next)  
loadUserDetails(userid) . then (successCallback, failCallback);
```

ES6 – Promises

The **resolve** parameter in the Promise corresponds to the first parameter (i.e. the “success-callback”) in the **then** function

```
const promise1 = new Promise(function(resolve, reject) {  
    setTimeout(resolve, 1000);  
});  
  
promise1.then(() => {alert("Success")}, () => {alert("Fail")});
```

This code will alert “Success” after waiting one second (since we don’t call **reject** in the Promise, it will never alert “Fail”)

ES6 – Promises

We can also send back data when calling **resolve**

```
const promise1 = new Promise(function(resolve, reject) {  
  setTimeout(resolve("Success"), 1000);  
});  
  
promise1.then(data => {alert(data)} , () => {alert("Fail")});
```

ES6 – Promises

And if we call **reject**, it corresponds to the second parameter (i.e. the “error-callback”) in the **then** function

```
const promise1 = new Promise(function(resolve, reject) {  
  setTimeout(reject, 1000);  
});  
  
promise1.then(() => {alert("Success")}, () => {alert("Fail")});
```

This code will alert “Fail” after waiting one second

ES6 – Promises

One of the useful features of Promises is how it deals with chaining of asynchronous events.

Previously, if we wanted to call asynchronous events after one another, we would have to pass them as callbacks to each other

```
firstAsyncFunction(function(result) {  
    secondAsyncFunction(result, function(secondResult) {  
        thirdAsyncFunction(secondResult, function(thirdResult) {  
            console.log(`This is not ideal: ${thirdResult}`);  
        });  
    });  
});
```


ES6 – Promises

So, generally speaking, when function1 finishes, it calls function2, which calls function3, etc.



Due to the way JS used to deal with asynchronous requests, calling each function as the previous function's callback was the only way to ensure that asynchronous functions finish in sequence

This is known as the “callback pyramid of doom” or “callback hell”

<http://callbackhell.com/>

Classic pyramid of doom

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



ES6 – Promises

Promises allow us to do this in a neater, more intuitive way due to the way it deals with chaining

```
firstAsyncFunction().then(function(result) {  
    return secondAsyncFunction(result);  
})  
.then(function(secondResult) {  
    return thirdAsyncFunction(secondResult);  
})  
.then(function(thirdResult) {  
    console.log(`This is a lot better: ${thirdResult}`);  
});
```

ES6 – Promises

Or even shorter with arrow functions

```
firstAsyncFunction()  
  .then(result => secondAsyncFunction(result))  
  .then(secondResult => thirdAsyncFunction(secondResult))  
  .then(thirdResult => {  
    console.log(`This is a lot better: ${thirdResult}`);  
  });
```

Each **.then** function returns another promise, which allows us to chain them together

ES6 – Promises

If we're referring to an existing function, we only need to pass the pointer to the function when it resolves or rejects

```
const p = new Promise((res, rej) => {  
    setTimeout(res, 1000);  
});  
const sayYes = function() {  
    alert('Done');  
}  
p.then(sayYes);
```

ES6 – Async/await

Technically part of ES7

Different way of dealing with asynchronous events

In other words, similar situations for what you'd want to use Promises for...

...BUT async/await is not the same as Promises

ES6 – Async

The `async` keyword is always used with a function like this

```
async function load() {  
    return 1;  
}
```

// Or with an arrow function expression

```
const load = async () => {  
    return 1;  
}
```

ES6 – Async

The use of the `async` keyword does two things:

- The function always returns a `Promise`, which can be called with `then`
- The function allows the use of the `await` functionality

ES6 – Async

```
const f = async () => {  
    return 1;  
}  
  
// can be used the same as  
const f = () => {  
    return new Promise((res, rej) => {  
        res(1);  
    })  
}  
  
// Both can be used as follows  
// (note that you have to execute the function in both cases)  
f().then(x => alert(x));
```

ES6 – Await

The `await` keyword is used to get the value from a `Promise`

Using `await` actually pauses execution of the script until the `Promise` returns a value

You can only use `await` inside an `async function`

ES6 – Await

This example alerts “Done” after 1 second

```
const myPromise = new Promise((resolve, reject) => {
    setTimeout(() => { resolve("Done") }, 1000);
});

const callAsync = async () => {
    let value = await myPromise;
    // execution pauses here until myPromise resolves
    alert(value);
}

callAsync();
```

ES6 – Await

Alternatively, if we want to use await without having to create and call a named function, we can wrap it in an anonymous self-invoking function

```
// assuming myPromise is declared here

( async () => {
    let value = await myPromise;
    // execution pauses here until myPromise resolves
    alert(value);
}) ();
```

ES6 – Await

Let's look at the difference between `await` and `then`

Given the following `Promise` declaration...

```
const setValue = () => {  
    return new Promise((res, rej) => {  
        setTimeout(() => {res("bar")}, 500);  
    });  
}
```

...we're going to call it using `await` and `then`

ES6 – Await

First, using `await`

```
(async () => {  
    let val = "foo";  
    val = await setValue();  
  
    alert(val);  
})();
```

Result: ???

ES6 – Await

First, using `await`

```
(async () => {  
    let val = "foo";  
    val = await setValue();  
  
    alert(val);  
})();
```

Result: bar

Execution is paused until `setValue` resolves

ES6 – Await

Using then

```
(async () => {  
    let val = "foo";  
    setValue().then(x => {val = x});  
  
    alert(val);  
})();
```

Result: ???

ES6 – Await

Using then

```
(async () => {  
    let val = "foo";  
    setValue().then(x => {val = x});  
  
    alert(val);  
})();
```

Result: foo

While `setValue` was completing in the background, execution continues and `val` is alerted before it is changed

ES6 – Promises + async/await

We'll revisit promises and async/await when we deal with AJAX

References

Banks, A. & Porcello, E. 2017. *Learning React: Functional Web Development with React and Redux*. O'Reilly Media, Inc.

<https://developer.mozilla.org/>

<https://github.com/lukehoban/es6features/blob/master/README.md>

<https://javascript.info/async-await>

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await>