Promises

So far we have mainly relied on callbacks for our asynchronous code, which has long been the standard. However you will have noticed that error handling can quickly become very messy as we chain several callbacks together. Luckily with the new ES6 standard a new feature known as Promise got introduced.

You can rationalize a Promise as the word it suggests. You give it some JavaScript code and it promises to return you something. It returns one of two things, either everything went well or an error occurred, either way you will get an answer from the Promise once it has finished executing and knows the output.

Learning Objectives

- Begin to develop an understanding concerning the power of Promises and how to implement them
- Be able to utilize both fetch and Axios to preform asynchronous calls
- Learn about ES7 Async and Await syntax
- Compare Async and Await to Promises

Promises

A Promise consist of three states:

- unresolved -> waiting for something to finish
- resolved -> something finished and all went well
- rejected -> something finished and an error occurred

By default a Promise is the unresolved state and waiting for something to occur. That something can be any kind of JavaScript function, for example a Get request.

Let’s declare a promise to see what is happening

```
var promise = new Promise((resolve, reject) => {
    resolve();
});
console.log(promise);
```

Try running the above code in your browser console. Afterwards replace the `resolve()` with `reject()`. What do you see?

You will notice that first of all you get back an object, which indicates whether the Promise was resolved or rejected. If you look at `promise.__proto__` you will see the object has two methods, `catch` and `then`.

You will further notice that you get an error message if you run the code with `reject()`. That error message basically tells you, you haven't handled the error. So let's solve that by making use of the two methods.

```
var promise = new Promise((resolve, reject) => {
    resolve();
});

promise.then(() => {
    console.log('Promise resolved.');
});

promise.catch(() => {
    console.log('An error occurred');
});
```

Run the above code, then replace the `resolve()` with `reject()`. Notice how `then()` only runs if the the promise was resolved and `catch()` only runs when it was rejected.

The real power of promises come to light when we start chaining different `.then()` together. Let's refactor to the code to ES6 standards and chain everything together.

```
const promise = new Promise((resolve, reject) => {
    resolve();
});

promise
    .then(() => console.log('I ran'))
    .then(() => console.log('I ran afterwards'))
    .then(() => console.log('Then I ran'))
    .catch((err) => console.log('uh oh error', err));
```

If you run the code above you will notice that you can do some pretty powerful chaining with promises, without having to worry about callbacks and error handling. You only need to handle the error once. To demonstrate that let's change our code

```
const promise = new Promise((resolve, reject) => {
    resolve();
});

promise
    .then(() => console.log('I ran'))
    .then(() => { throw new Error('Help I am an error'); })
    .then(() => console.log('I did not run'))
    .catch((err) => console.log('uh oh error', err));
```

Run the code above to test it.

Why did we alter one of the arrow functions by inserting curly brackets? It is because without the curly brackets the arrow function will automatically return the inner content. However we don't want to return anything, but instead throw an error. Therefore we need to wrap it in curly brackets to disable the return.

(Notice if you run the above code and get an error variable already declared, refresh the page. That error occurs because of the const declaration, the variable promise can only be declared once, replace it with var if you want, change the name or omit it);

fetch

Promises are often used for AJAX requests, as these are processes that are asynchronous and a Promise makes life a lot easier when handling their responses. Let's demonstrate it using a new ES6 feature `fetch()` which makes simple AJAX requests and returns a Promise.

```
const url = 'https://jsonplaceholder.typicode.com/posts';

fetch(url)
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(err => console.log(err))
```

With fetch we have to make two requests, in the first one we receive an object, however without a body, so we have to fetch the body using `response.json()`. Chaining the requests together makes this all a breeze to write.
Try manipulating the domain to something like typicode123.com to see how the errors are handled.

The implementation of fetch isn't ideal, especially it's error handling. While it is nice to have a native solution, other solutions like axios (https://github.com/mzabriskie/axios) make better use of Promises for API requests.

Axios

Axios is another package that makes a lot of uses of Promises. Here is an example of using `axios`.

```
// GET method
axios.get("/users")
    .then(res=> console.log(res))
    .catch(err=> console.log(err));

// POST method
axios.post("/users",{"id":2,"username":"newUserName"})
    .then(res=> console.log(res) )
    .catch(err=> console.log(err) );

// DELETE method
axios.delete("/users",{id:3})
    .then(res=> console.log(res))
    .catch(err=> console.log(err));

// PUT method
axios.put("/users",{"email":"user@user.com","password":"1234"})
    .then(res=> console.log(res))
    .catch(err=> console.log(res));
```

Applying Promise to asynchronous code

A lot of code we use in node is asynchronous, Promises can help us clean that up.

```
const fs = require('fs');

function readdir(path) {
    return new Promise((resolve, reject) => {
        fs.readdir(path, (err, files) => {
            if (err) {
                reject(err);
            } else {
                resolve(files);
            }
        });
    });
}

module.exports = readdir();
```

Now if you require the function `readdir()` in another module you can chain `.then()` and `.catch` to it.

Promise.all

If you have several Promises you can rely on `Promise.all` to determine once they finished running. `Promise.all` itself is a promise that resolves when all promises are resolved, if one fails it rejects them all. All you need to pass it is an array of promises.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'one');
});
const p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'two');
});

Promise.all([p1, p2])
    .then(values => console.log(values))
    .catch(err => console.log(err));
```

If you still have trouble understanding Promises or want to go all over it again, here are two videos each explaining them in a different manner. Do not worry if you don't understand Promises or the benefits that they bring to writing code, the concept is difficult and it will take some time to wrap your head around this abstract idea.

Here is one video (https://youtu.be/s6SH72uAn3Q) that uses real life examples to explain promises.

Here is another video (https://youtu.be/2d7s3spWAzo) which uses promises to render data to a HTML document.

Async & Await

ES7 supports new keywords called `async` and `await` to further simplify your promise method call.

```
// Async Version
async function callAPIs(){
    try{
        let res1 = await axios.get("http://example.com")
        let res2 = await axios.get(`http://example2.com?key1=${res1.key2}`)
        return res2.key2;
    }catch(err){
        console.log(err);
        throw err;
    }
}

// Promise version
function callAPIs(){
    return new Promise((resolve,reject)=>{
        axios.get("http://example.com")
            .then((res1)=>axios.get(`http://example2.com?key1=${res1.key2}`))
            .then((res2)=> resolve(res2.key2))
            .catch((err)=>{
                console.log(err);
                reject(err);
            });
    });
}
```

As shown above, the `async` function is quite similar in pattern to a new `Promise` object.

```
// Async Version
async function callAPIs(){
    return doSomething();
}

//Promise Version
function callAPIs(){
    return new Promise((resolve,reject)=>{
        resolve(doSomething());
    })
}
```

Each `await` corresponds to the `then` method called in the promised version.

```
//Async version
async function callAPIs(){
    let res1 = await axios.get("http://example.com")
    return res1;
}

// Promise version
function callAPIs(){
    return new Promise((resolve,reject)=>{
        axios.get('http://example.com')
            .then((res1)=>resolve(res1));
    });
}
```

Each `catch block` is correspondent to the `catch` method call in the promised version.

```
// Async version
async function callAPIs(){
    try{
        let res1 = await axios.get('http://example.com');
        return res1;
    }catch(err){
        console.log(err);
        throw err;
    }
}

// Promised version
function callAPIs(){
    return new Promise((resolve,reject)=>{
        axios.get("http://example.com")
        .then((res1)=>resolve(res1))
        .catch((err)=> reject(err))
    });
}
```

Each `return` in the async version corresponds to the `resolve` function call in the promised version.

Each `throw err` in the async version corresponds to the `reject` function call in the promised version.

Async version <-> Promise version Comparison

Here is a quick comparison between the two,

- `async` **function <-> new** `Promise` **Object**
- `await` **keyword <->** `then()` **method call**
- `catch` **block <->** `catch()` **method call**
- `return` **keyword <->** `resolve()` **function call**
- `throw` **error <->** `reject()` **function call**

Promise.all for Async

You can also use `Promise.all` in `async` function when you want to wait for multiple promises before proceeding.

```
async function callAPIs(){
    let combinedPromise = Promise.all([axios.get('http://example1.com'),
                                       axios.get('http://example2.com')]);
    let responses = await combinedPromise;
    return responses;
}
```

Exercise

A

Create a module that contains a promised version of fs.readdir (https://nodejs.org/api/fs.html#fs_fs_readdir_path_options_callback) and fs.stat (https://nodejs.org/api/fs.html#fs_class_fs_stats). Similar to the example above.

- Then download this file structure (https://drive.google.com/drive/folders/1GiUiO1dNj68x4qK7Dgyot4cbBu-xBMtQ?usp=sharing)
- Use the modules to determine whether each path leads to a directory or file. Read the documentation for `fs` properly you'll find all the necessary knowledge in there.
- Output should look like this

```
/files/project_A/ is a directory
/files/project_A/README.md is a file
/files/project_A/images/ is a directory
...
```

Completed. Let's continue.