104)HTTP Request from Javascript

First we initialize the request using the constructor function provide by browser. This constructor function takes no arguments. Xml was very popular in past. XML was just different way to structure data that you want to transfer. Now a days we use json for that. Json look more like js objects. Xml looked more lie html than js object.

const request = new XMLHttpRequest();

so xml was popular, now its not.name(xml) here is not relevant. E can use this constructor to transfer any information just not xml. The next thing we do down below is we actually use a few methods on request to set things up for example. Like we need to specify the url where json lives.we need to specify this somewhere in our code. This happens via request.open method. Open starts to initialize our request. And this is where we configure to important pieces of information. First is url and second is http method.

request.open('GET','http://puzzle.mead.io/puzzle');

the last method that we use is sent. We send off the request and this is going to actually initiate the process. Now the process does’nt take place right away. It takes a bit of time for you to connect with that server.then it takes some time for server to do some task. Then it takes some time for server to send its response back. We can see that in network tab of chrome dev tools. There is a timeline in ms. So we will get response back. But we are not doing anything with it. We ned to add event listerner that’s going to fire when we actually have that information. So with this code we are sending the request-

const request = new XMLHttpRequest();

request.open('GET','http://puzzle.mead.io/puzzle');

request.send();

now we need to do something when we get back response. Lets add and evnt listernet-

const request = new XMLHttpRequest();

request.open('GET','http://puzzle.mead.io/puzzle');

request.send();

request.addEventListener('readystatechange', (e) => {

});

So we are listening to readystatechnage event. Lets pull up mdn documentation for XMLHttpRequest object. now we re going to talk about ready state. Scroll down in doc to property section. there you can see ready state. So ready state is a property and when it changes our event handler fires. If we click on ready sate property we can see that there 5 state with value from 0 to 4. At 0 we have unsent request, open has’nt been called. At 1 open has been called. At 2 , we have called sent. At 3 we have strated the process of getting the response back. At 4 ware actually done. When it comes to figuring out what state we are in, we use these values. And only thnig that we are going to check for is 4. We want to do something when we have complete information back. Rest all values are not used. So our event handled will be fired five times, one for each state change. But we want to do something only if when 4 happens, so we ad if condition. Lets just print event.target. code –

const request = new XMLHttpRequest();

request.open('GET','http://puzzle.mead.io/puzzle');

request.send();

request.addEventListener('readystatechange', (e) => {

if(e.target.readyState === 4) {

console.log(e.target);

}

});

So e have XMLHttpRequest object printed on console. It has many properties. One property is reponseText, it contains our json response. We need to parse it as json. So w ehave to use Json.Parse method. Final code –

const request = new XMLHttpRequest();

request.open('GET','http://puzzle.mead.io/puzzle');

request.send();

request.addEventListener('readystatechange', (e) => {

if(e.target.readyState === 4) {

const data = JSON.parse(e.target.responseText);

console.log(data);

}

});

105)Headers and Erros

Every http response comes with status code. Its is numeric value that tells how things went. So we will check status code, before we try to access data in response.

request.addEventListener('readystatechange', (e) => {

if(e.target.readyState === 4 && e.target.status ===200) {

console.log(e.target.status);

const data = JSON.parse(e.target.responseText);

console.log(data);

}

});

Go to this website to get all status codes-

<https://httpstatuses.com/>

107)Callback Abstrcation

Here we craeted a new file called requests.js. now what we wwant to do is, we want to call our api in this newly created a file and use it in app.js. in case of synchronous code this was simple, like this-

Requestst.js-

const getPuzzle = () => {

return 'new puzzle'

};

App.js-

let puzzle = getPuzzle();

console.log(puzzle);

but this thing is not possible because our code is asynchronous. So we solve problem with this-

app.js-

getPuzzle((puzzle) => {

console.log(puzzle);

});

Requests.js-

const getPuzzle = (callback) => {

const request = new XMLHttpRequest();

request.open('GET', 'http://puzzle.mead.io/puzzle');

request.send();

request.addEventListener('readystatechange', (e) => {

if (e.target.readyState === 4 && e.target.status === 200) {

const data = JSON.parse(e.target.responseText);

callback(data.puzzle);

} else if (e.target.readyState === 4) {

console.log('An error occured');

}

});

}

So here we used callback approach. In requests.js we are calling function ( request event listerner) inside function. so even if we return value from inside function, value won’nt reach app.js. so we cannot use return statement in so we solved the problem with callback approach. We further improved our approach by handling error-

App.js-

getPuzzle((error,puzzle) => {

if (error) {

console.log(`Error: ${error}`);

} else {

console.log(puzzle);

}

});

Requests.js-

const getPuzzle = (callback) => {

const request = new XMLHttpRequest();

request.open('GET', 'http://puzzle.mead.io/puzzle');

request.send();

request.addEventListener('readystatechange', (e) => {

if (e.target.readyState === 4 && e.target.status === 200) {

const data = JSON.parse(e.target.responseText);

callback(undefined,data.puzzle);

} else if (e.target.readyState === 4) {

callback('An Error has taken place', undefined);

}

});

}

108)Asynchronous vs Synchronous Execution

When we execute something synchronously we start some sort of task like fetching a puzzle and then we have to wait for it to finish before we move on to next thing. When we execute something asynchronously , we can start some task then we can actually get other work done before the task completes.

There is synchronous way of sending the request . we norammly send request like this-

request.open('GET', 'http://puzzle.mead.io/puzzle');

by default it is asynchronous, so it means we can do other things while response comes back. We can pass third argument to this function which defines whether this request should be asynchronous or not. So we make synchronous request like this-

request.open('GET', 'http://puzzle.mead.io/puzzle', false);

so,with this we can also change our function like this-

const getPuzzleSync = (callback) => {

const request = new XMLHttpRequest();

request.open('GET', 'http://puzzle.mead.io/puzzle',false);

request.send();

if (request.readyState === 4 && request.status === 200) {

const data = JSON.parse(e.target.responseText);

return data.puzzle;

} else if (e.target.readyState === 4) {

throw new Error('The Error Occured');

}

}

Now our code will wait to execute next line till response comes back. So we can use return statement. Now we dnt even have to rely on event listerner. Note that this approach should never be used. It is just for understanding purpose. In time when request has not come back we will not be able to click on buttons or interact with UI. In time request is coming back entire browser is locked up.

This is something we want to avoid. So we will be stick with synchronous code which allows us to do other things while we are waiting for our long running tasks such as http request to complete. So term synchronous and asynchronous terms are also used as blocking and non-blocking.

110)Closures

Closures closely relate to function and fuction scope. In this course we have worked with closures but we did’nt know about the term. Lets see relatively straight example of closures.

const myFunction = () => {

const message = `This is my message`;

const printMessage = () => {

console.log(message);

}

printMessage();

};

myFunction();

this functions works as expected. This is very basic example so we cannot see closures. Lets see tweet it.

const myFunction = () => {

const message = `This is my message`;

const printMessage = () => {

console.log(message);

}

return printMessage;

};

const myPrintMessage = myFunction();

myPrintMessage();

so now we return function and then we call it. Here we call myPrintMessage only when we have returned from myFunction. Even we have returned from myFunction, we still have access to variable that is local to myFunction. so our function (myPrintMessage) still have access to message variable from parent function. so this s closure. **A closure is combination of function with lexical scope in which it was defined**. In our case when printMessage was defined it had access to message, so it’s always going to have access to message, even if myFunction completes. We have already used closures. In requests.js –

const getPuzzle = (wordCount,callback) => {

const request = new XMLHttpRequest();

request.open('GET',

`http://puzzle.mead.io/puzzle?wordCount=${wordCount}`);

request.send();

request.addEventListener('readystatechange', (e) => {

if (e.target.readyState === 4 && e.target.status === 200) {

const data = JSON.parse(e.target.responseText);

callback(undefined,data.puzzle);

} else if (e.target.readyState === 4) {

callback('An Error has taken place', undefined);

}

});

}

Both of them has callback function which is fired later when http response comes.

We call this function in app.js and pass callback. Now response comes back after some time. by that time function getPuzzle() has been executed. So getPuzzle is finished, then at some point down the line these callback function fires. It is because of closure, that our inner function(response handler) has access to callback that we have passed to getPuzzle function. so because we have closure our function has access to the lexical scope in which it was defined. In this case it has access to callback and wordCount.

So using closures and actually having support for them is essential for patters like this to actually work. This is one example where closure comes up. We will look at couple of other examples. lets use closure to create private variable in javascript. A variable that is accessible or modifiable via a very specific set of rules. Code-

const createCounter = () => {

let count = 0;

return {

increment() {

count++;

},

decrement() {

count--;

},

get() {

return count;

}

};

};

const counter = createCounter();

from this function we return an object. all of these function are using closure to access count , a variable defined in createCounter function. now count variable is only accessible via these metods. Lets see another example-

const createAdder = (a) => {

return (b) => {

return a+b;

}

};

const add10 =createAdder(10);

console.log(agg10(4));

so that are closures. W talked abiut them, because you are going to use them aalot. At starting of this lecture, we saw that how closures makes the eventHandler to excess callback passed to them.

111)Exploring Promises

Promises provide us better way to structure our asynchronous code. Right now in if you open app.js in hangman folder. you can see that we are using callback approach.in app.js, we call this function-

getPuzzle("1",(error,puzzle) => {

if (error) {

console.log(`Error: ${error}`);

} else {

console.log(puzzle);

}

});

In requests.js we define this function-

const getPuzzle = (wordCount,callback) => {

const request = new XMLHttpRequest();

request.open('GET',

`http://puzzle.mead.io/puzzle?wordCount=${wordCount}`);

request.send();

request.addEventListener('readystatechange', (e) => {

if (e.target.readyState === 4 && e.target.status === 200) {

const data = JSON.parse(e.target.responseText);

callback(undefined,data.puzzle);

} else if (e.target.readyState === 4) {

callback('An Error has taken place', undefined);

}

});

}

Here we used callback approach. We separated the logic of how data is obtained from how data is used. Now we will see that how we can structure it using promises. We will see a example, we will first use callback approach, then we will use prmise approach.

This is callback approach-

const getDataCallback = (callback) => {

setTimeout(() => {

callback(undefined,'this is data');

}, 2000);

};

getDataCallback((err, data)=> {

if(err) {

} else {

console.log(data);

}

});

We will do similar things by promises. First we create a promise. Promise constrcuctor function takes a single argument, this a argument is a function. this function gets called right away. This is where we can perform our long running process. So that can be making http request to save some data in db or any other operation. Here we will be using setTimeout. When promise constructor calls this function, it calls them with 2 arguments. These are resolve and reject. We can call resolve to say that things went well, we call reject to say that things have failed. So inside this callback function that we pass to constructor we pass either resolve or reject. So now our promise is set now we will use it. After that you know. Code-

const myPromise = new Promise((resolve,reject) => {

setTimeout(() => {

// resolve('this is promis data');

reject('This is error');

}, 2000);

});

myPromise.then((data) => {

console.log(data);

}, (error) => {

console.log(error);

});

Now with promises, you can either resolve or reject. There can be single outcome of promise and this can be either resolve or reject. So once promise is resolved it cannot be rejected and vice versa. If we do this-

resolve('this is promis data');

reject('This is error');

promise will be resolved, anything after that will be ignored. However if we do this-

myPromise.then((data) => {

console.log(data);

}, (error) => {

console.log(error);

});

myPromise.then((data) => {

console.log(data);

}, (error) => {

console.log(error);

});

Now our code will be executed 2 times.

Advantages of promises over callback-

1. Syntax is better.
2. We can callback 2 times accidentally, but with promise this cannot happen.
3. Here it is define that for success call resolve for error call reject. But in callback we define, which argument to call for error and success
4. Later we will see things we can do with promises nd not with callbacks.

112)Converting to promises

Here we will convert getPuzzle to promise pattern from callback pattern. Here we replace callbacks in our code with promise-

Requests.js-

const getPuzzle = (wordCount) => new Promise( (resolve,reject)=> {

const request = new XMLHttpRequest();

request.open('GET',

`http://puzzle.mead.io/puzzle?wordCount=${wordCount}`);

request.send();

request.addEventListener('readystatechange', (e) => {

if (e.target.readyState === 4 && e.target.status === 200) {

const data = JSON.parse(e.target.responseText);

resolve(data.puzzle);

} else if (e.target.readyState === 4) {

reject('An error has taken place ');

// callback('An Error has taken place', undefined);

}

});

})

App.js-

getPuzzle("1").then((puzzle) => {

console.log(puzzle );

}, (err) => {

console.log(err);

});

113)Chaining Promises

Promise chaining is useful when we are trying to do 2 things in a row. both end up being promise call. Lets ay first we want to get country, then we want to get all regios in that country. This would require us to create 2 functions with 2 separate promises where data for first promise needs to be received before we can start the second one. Lets ee how we will do this thing by callback-

const getDataCallback = (num,callback) => {

setTimeout(() => {

if(typeof num === 'number') {

callback(undefined, num\*2);

}

else {

callback('Number must be provided');

}

}, 2000);

};

getDataCallback(2,(err, data)=> {

if(err) {

} else {

getDataCallback(data, (err, data) => {

if (err) {

console.log('err');

} else {

console.log(data);

}

});

}

});

Thing you are seeing here is called callback hell. It is hard to follow, hard to maintain code. It would be pretty challenging task. If we want to expand it further, it would be a challenging task. The problem with this code is that it is deeply nested. We are 4 layers deep onto final code. Lets handle same thing by promise. One level is handled by promise like this-

const getDataPromise = (num) => new Promise((resolve, reject) => {

setTimeout(() => {

resolve('this is my success data');

// reject('This is error');

}, 2000);

})

const myPromise = getDataPromise(123);

myPromise.then((data) => {

console.log(data);

}, (error) => {

console.log(error);

});

One way to get this thing done by promises is-

const getDataPromise = (num) => new Promise((resolve, reject) => {

setTimeout(() => {

typeof num === 'number' ? resolve(num\*2) : reject('Number Must be provided')

// reject('This is error');

}, 2000);

})

const myPromise = getDataPromise(2).then((data)=> {

getDataPromise(data).then((data) => {

console.log(data);

}, (error) => {

console.log(error);

})

}, (err) => {

console.log(err);

});

Here code structure is better than callback. But there is room of improvement, we have duplicate functions for error handler, it would be nice to have one function since we are doing the exact same thing and we are still nesting. We can avoid all of that using promise chaining.

Here we are not putting **.then** inside our first level resolve handler. **When we return a promise from another promise hander we create promise chaining.** **So we can chain another then call to do something when our inner promise resolves. This is called promise chaining, because we are chaining multiple promise calls. Code-**

const getDataPromise = (num) => new Promise((resolve, reject) => {

setTimeout(() => {

typeof num === 'number' ? resolve(num\*2) : reject('Number Must be provided')

// reject('This is error');

}, 2000);

})

getDataPromise(10)

.then(data => {

return getDataPromise(data);

})

.then(data => {

console.log(data);

});

Here we have avoided nesting. Here all **then** are at same level. The nice thing is that we could expand it add third or fourth promise without adding any complexity. Like this-

getDataPromise(10)

.then(data => {

return getDataPromise(data);

})

.then(data => {

return getDataPromise(data);

})

.then(data => {

console.log(data);

});

Now I want to add about error handling. To do this we want to chain other methodcall at end. This method is catch. It takes only argument that is single error handler. This is exactly what we pass as second argument to then, although when we work with promise chaining we can actually use catch to set up a single error handler for all of our promises. So wuther of our promises fail, this error handler will be executed. Lets say if we get error in first promise, then success handler of other promises won’t be executed. Only catch block will be executed.

114)Fetch Api

Code-

fetch("http://puzzle.mead.io/puzzle",{})

.then((response) => {

if(response.status === 200) {

return response.json();

} else {

throw new Error();

}

})

.then( data => console.log(data))

.catch((err) => console.log(err));

Here we will see second way in which we can make http request. Currently only way we know to make request is via xmlHttpRequest object. this is not only way. In newer version of javascript we have access to fetch API. This is different way to make our request and this new api has promises built in.

As a first argument we pass url, then we can also pass second optional argument, which help us to customize our request. Refer to pdf guide about second argument. Fetch method returns promise. Good thing about fetch api is that we do not have to worry about readyState. Promise is going to resolve or reject when it is ready for us. So we know that request completed, we don’t have to figure out if it completed , we just have to figure out how it completed.so we need to check if things got well i.e we got status 200 or things go poorly. May be we got 404 for using one of those invalid query strings. When things go wrong we want our catch block to run. Our catch block will run either if our promise is rejected or we manually throw error. So if status is other than 200 , we manually throw error.

**On response we have acess to method called json(), it returns a promise**. Promise will resolve at some point in future. We have 2 options here. Either we can use promise chaining or we can use **.then**  inside then block of first promise. Here we used promise chaining.

Now we will convert call xmlHttp calls to fetch calls. In request.js-

const getPuzzle = (wordCount) => {

return fetch(`http://puzzle.mead.io/puzzle?wordCount=${wordCount}`)

.then(response => {

if (response.status === 200) {

return response.json();

}

else {

throw new Error(`Something went wrong`);

}

});

};

Here we do not want our second then chin inside this request.js file, so we return promise from this file and chain second then call in app.js. code-

getPuzzle("1")

.then((data) => {

console.log(data.puzzle );

})

.catch((err) => {

console.log(`Error: ${err}`);

})

Note that catch call appears at end, in app.js file.nowthere is one morething-

fetch("http://puzzle.mead.io/puzzle",{})

.then((response) => {

if(response.status === 200) {

return 'This is string';

} else {

throw new Error();

}

})

.then( data => console.log(data))

.catch((err) => console.log(err));

Here we do not return promise, we return a string. While working with promise chaining, we do not have to return a promise from then, I can return anything I would like and that’s going to be passed along to next step in promise chain. Lets use this thing, now in app.js we only want to get puzzle not whole request body object. so we use concept that we return just above-

const getPuzzle = (wordCount) => {

return fetch(`http://puzzle.mead.io/puzzle?wordCount=${wordCount}`)

.then(response => {

if (response.status === 200) {

return response.json();

}

else {

throw new Error(`Something went wrong`);

}

}).then((data) => data.puzzle);

};

Here we added one extra ten block, here we get data and pass it ito then block set up on app.js.

getPuzzle("2")

.then(puzzle => {

console.log(puzzle );

})

.catch((err) => {

console.log(`Error: ${err}`);

})

116)Promise Challenge

We are making request to 2 api-

In requests.js-

const getCountry = (countryCode) => {

return fetch(`https://restcountries.eu/rest/v2/all`)

.then(response => {

if (response.status === 200) {

return response.json();

} else {

throw new Error('Unable to fetch');

}

})

.then(data => {

return data.find(country => country.alpha2Code === countryCode);

});

};

const getLocation = () => {

return fetch(`https://ipinfo.io/json?token=1a11bd55cc8f9c`)

.then(response => {

if (response.status === 200) {

return response.json();

} else {

throw new Error("Unable to fetch");

}

});

};

App.js-

getCountry('US')

.then(country => {

console.log(country.name);

})

.catch(err => {

console.log(`Error: ${err}`);

})

getLocation()

.then(location => {

console.log(location);

console.log(`You are currently in

${location.city} ${location.region} ${location.country}`);

})

What we want to do is we want to chain these 2 api’s. what we want to do is do pass country code that we got from getLocation to getCountry and get details of this country.

This s how we do it-

getLocation()

.then(location => {

return getCountry(location.country)

})

.then(country => {

console.log(country.name);

})

.catch(err => {

console.log(err);

})

117)Async/Await

Async/await is combination of 2 new javascript tools. We have async function and await operator, when we use them togather we get a new way to structure and work with promise that makes code a whole lot easier to work with. This feature really builds up on what we already know about promises.

Lets see this code-

const processData = () => {};

console.log(processData());

output –undefined.

This because undefined is default return value if we do not return anything form function.

Now lets start introducing async/await. We are going to do this by adding single word to code and viewing how that how that changes the output. This is the async keyword. When we create a function we can choose to create a function as an sync function, to do that we just need to add **async** keyword before function definition. Like this-

const processData = async () => {};

now we have successfully created an async function that is the async portion of async await. We will see what it allows us to do in a second. but lets run this program-

const processData = async () => {};

console.log(processData());

output-

**Promise { undefined }**

So we are getting promise. That brings us to first important feature of async functions. Your async functions always returns promise. That promise is resolved with a value that you, the developer choose to return from the function. ex-

const processData = async () => {

return 10;

};

processData().then(data => {

console.log(data);

});

Output -**10**

We can also reject promise that is returned by throwing error like this-

Code-

const processData = async () => {

throw new Error('Something went wrong');

// return 10;

};

processData()

.then(data => {

console.log('Data',data);

})

.catch(error => {

console.log('Error', error);

});

In promises we had this-

const getDataPromise = num =>

new Promise((resolve, reject) => {

setTimeout(() => {

typeof num === "number" ? resolve(num \* 2): reject("Number Must be provided");

// reject('This is error');

}, 2000);

});

We are going to implement this using async/await.

We also have await operator. Await operator can only be used with async functions. That s why when people refer to this feature they usually refer to it as one thing async/await as opposed to 2 separate features- The async function and await operator. If you use one , you are most like to to use another.

This syntax should look familiar. We did very similar when we created a new instance of one of our classes. We used new operator with constructor function that allowed us to create a new instance, in this case we are using await operator with function that retuns a prmise. Now this is where things get cool. Now we do not have to use .**then** to setup callbcak function(when we call getDatapromise), this looks like synchronous code, which means if we want value that comes back from getDataPromise, we just store it in variable upfront.

Code-

const getDataPromise = num =>

new Promise((resolve, reject) => {

setTimeout(() => {

typeof num === "number" ? resolve(num \* 2) : reject("Number Must be provided");

// reject('This is error');

}, 2000);

});

const processData = async () => {

const data =await getDataPromise(2);

console.log(data);

};

processData()

.then(data => {

console.log('Data',data);

})

.catch(error => {

console.log('Error', error);

});

Output-

**4**

**Data undefined**

Now we can expand on it further. Previously if we wanted to use a promise multiple times, we have to use promise chaining. With async/ await we can just add few line to the program, nothing complex. Code –

const getDataPromise = num =>

new Promise((resolve, reject) => {

setTimeout(() => {

typeof num === "number" ? resolve(num \* 2) : reject("Number Must be provided");

// reject('This is error');

}, 2000);

});

const processData = async () => {

let data = await getDataPromise(2);

data = await getDataPromise(data);

return data;

};

processData()

.then(data => {

console.log('Data',data);

})

.catch(error => {

console.log('Error', error);

});

Output - **Data 8**

We start with 2, double it to 4. Then we call **getDataPromise** with 4. So we get 8. Then we return 8 from **processData** . what ever we returned from getProcess is going to be resolved value of promise.

So with async await operator we can structure our code that uses promises to look more like regular old synchronous code. We can perform one code after another.

let data = await getDataPromise(2);

data = await getDataPromise(data);

second line is not going to be executed till first promise resolves or rejects.

let data = await getDataPromise(2);

data = await getDataPromise(data);

return data;

return statement won’t run until, second promise resolves or rejects.

As of now we have seen happy path where all of promises are resolved, lets asee hat happens when one of them fails. Code-

const getDataPromise = num =>

new Promise((resolve, reject) => {

setTimeout(() => {

typeof num === "number" ? resolve(num \* 2) : reject("Number Must be provided");

// reject('This is error');

}, 2000);

});

const processData = async () => {

let data = await getDataPromise('two');

data = await getDataPromise(data);

return data;

};

processData()

.then(data => {

console.log('Data',data);

})

.catch(error => {

console.log('Error', error);

});

Error-

**Error Number Must be provided**

So when our promise fails, it is equivalent to we are a throwing an error from function and catch block(where we call process data) will be executed. This is same thing when we return an error manually from processData.

So we saw that promise chaining is daunting thing. We can add one more level just be-

let data = await getDataPromise('two');

data = await getDataPromise(data);

data = await getDataPromise(data);

now we will see how we can add this async/await to requests.js file in hangman folder. we will rename old getPuzzle function to getPuzzleOld. No change is required in app.js , where we call these functions.Code-

const getPuzzleOld = (wordCount) => {

return fetch(`http://puzzle.mead.io/puzzle?wordCount=${wordCount}`)

.then(response => {

if (response.status === 200) {

return response.json();

}

else {

throw new Error(`Something went wrong`);

}

}).then((data) => data.puzzle);

};

const getPuzzle = async (wordCount) => {

const response = await fetch(`http://puzzle.mead.io/puzzle?wordCount=${wordCount}`);

if (response.status === 200) {

const data = await response.json();

return data.puzzle;

// return response.json().puzzle;

}

else {

throw new Error(`Something went wrong`);

}

};

Async/await code is easy to understand. Promises are fine you can use them, but in case of promise chaining async/await code is better to understand.