26)Async Basics

Code-

console.log('Starting app');

setTimeout(() => {

console.log('Inside of call back');

}, 0);

setTimeout(() => {

console.log('Inside of call back');

},2000);

console.log('Finishing up');

**output-**

**Starting app**

**Finishing up**

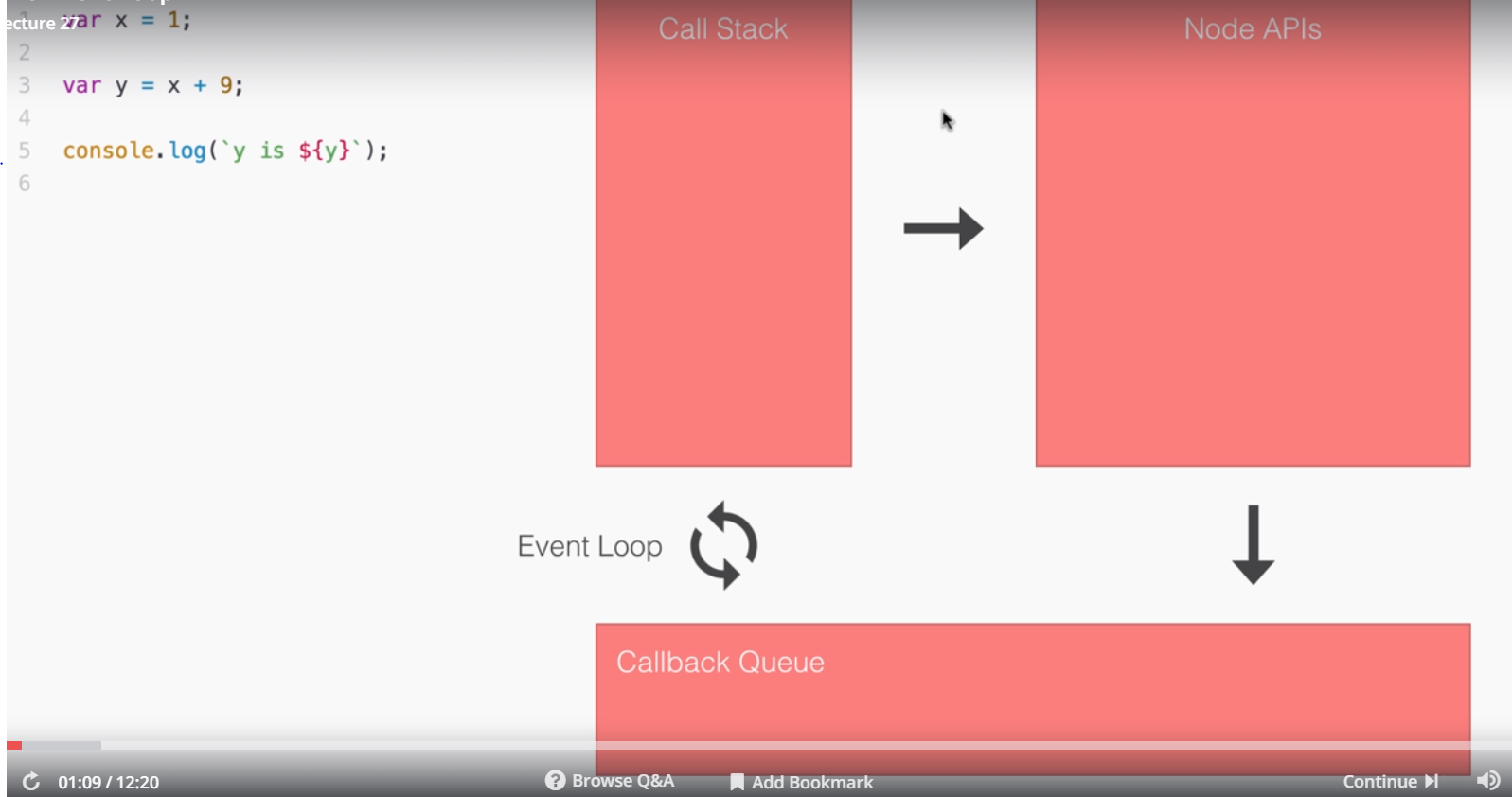
**Inside of call back**

**Inside of call back**

We will see why this output in a next video.

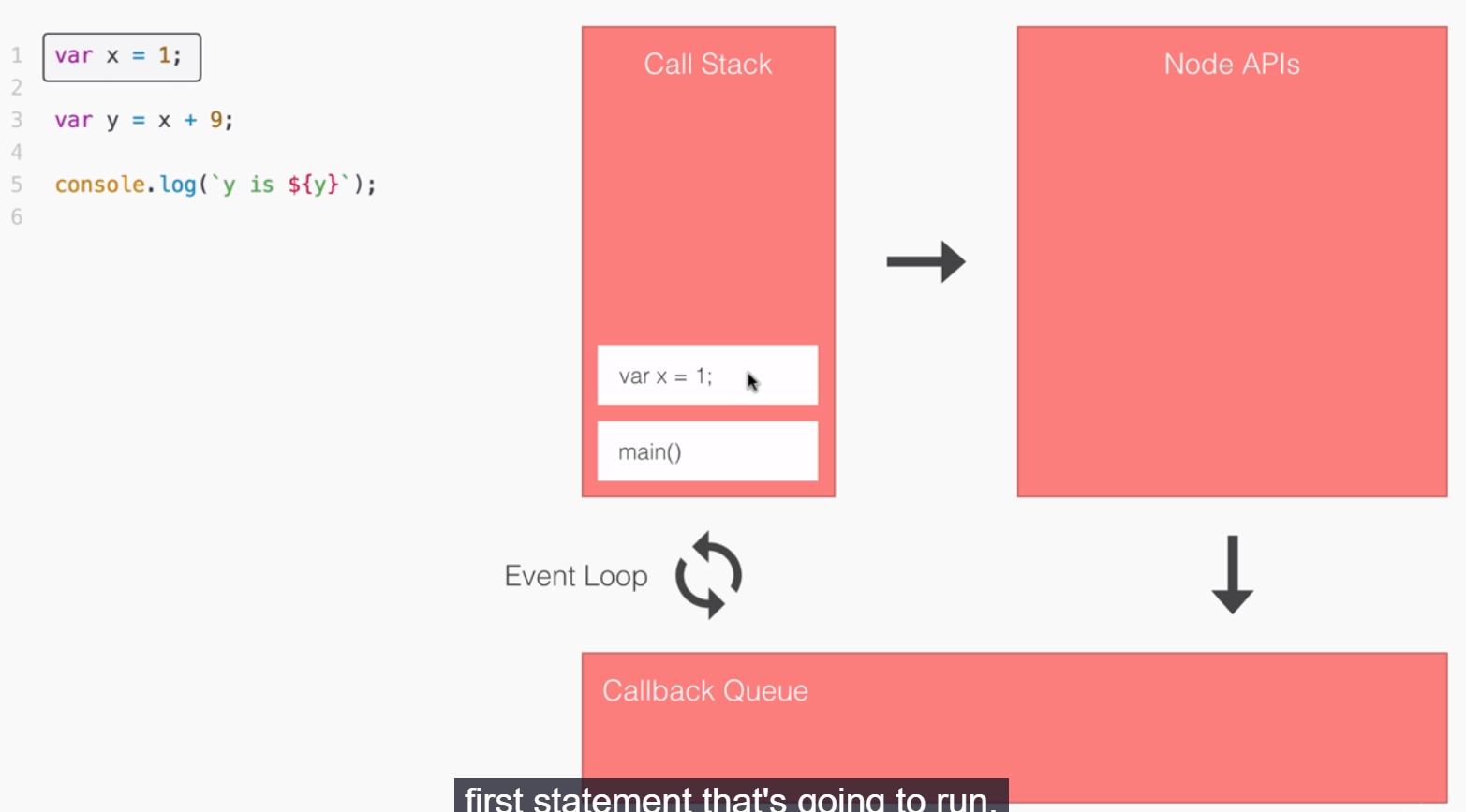
27)Call Stack & Event Loop

Here is basic example-

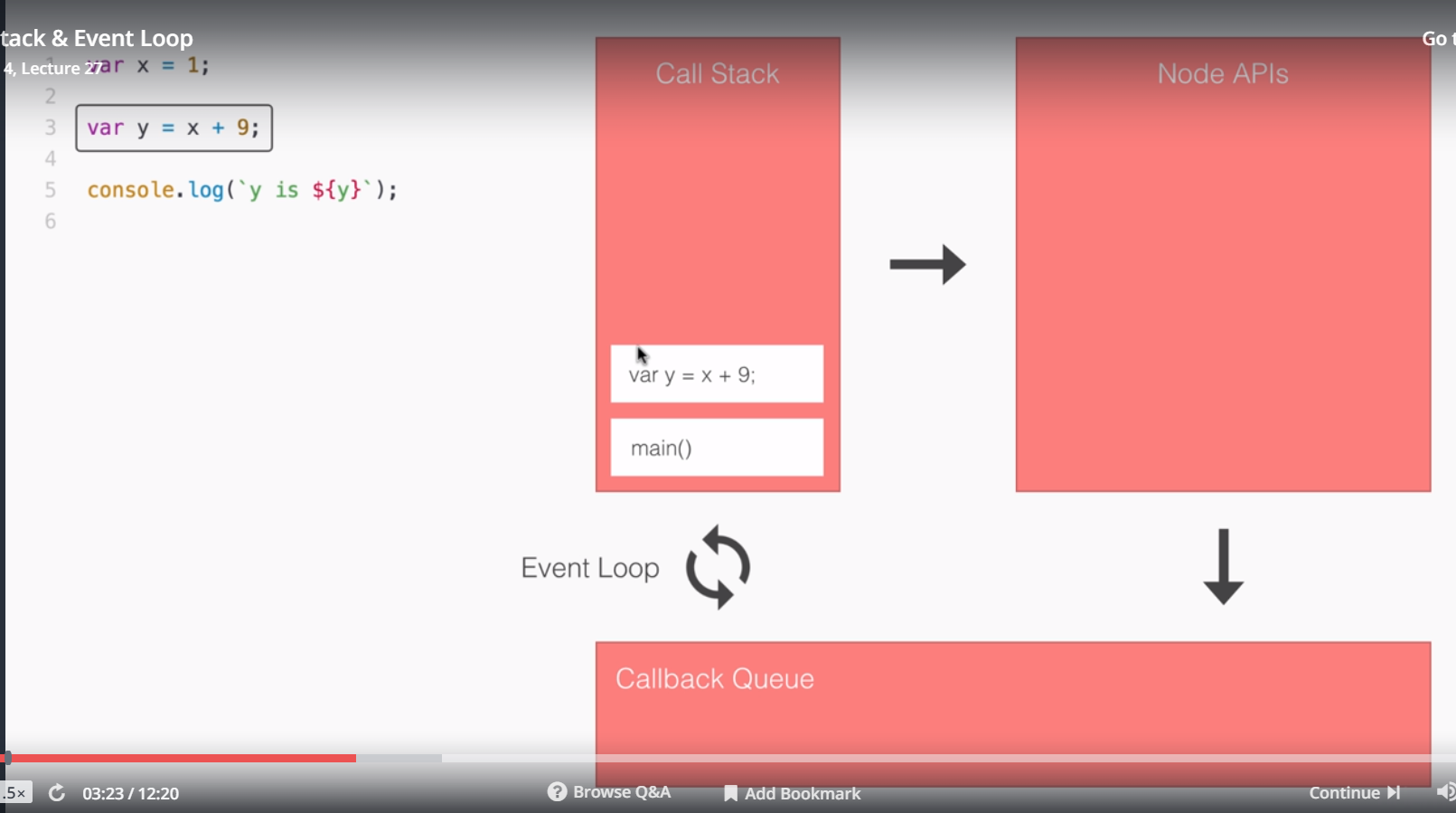


We will see how all these work togather to make node app. On left we have synchronous code. As our code is synchronous we just need to worry about call stack. Call stacj is part of v8 and for our synchronous example it’s only thing that’s going to run. We are not using any node api and we are not doing any asynchronous program. The call stacjk is simple data structure that keeps track of program execution inside of v8. It keeps track of functions currently executing and the statements that are fired. The callstack is simple data struvture that can do 2 things. One you can add something on top of it and 2 you can remove item on top of it, that is all it can do.

When we execute above program, first thing that happens is node runs the main function, this is function which is wrapped around all our files when we run them through node. In this case by telling v8 to run main program we are starting the program. Then we execute our first line. Notice that it comes on top of stack.



Once first statement is executed we can remove it and move onto next statement. So to execute second statement we put in in call stack. Like this-

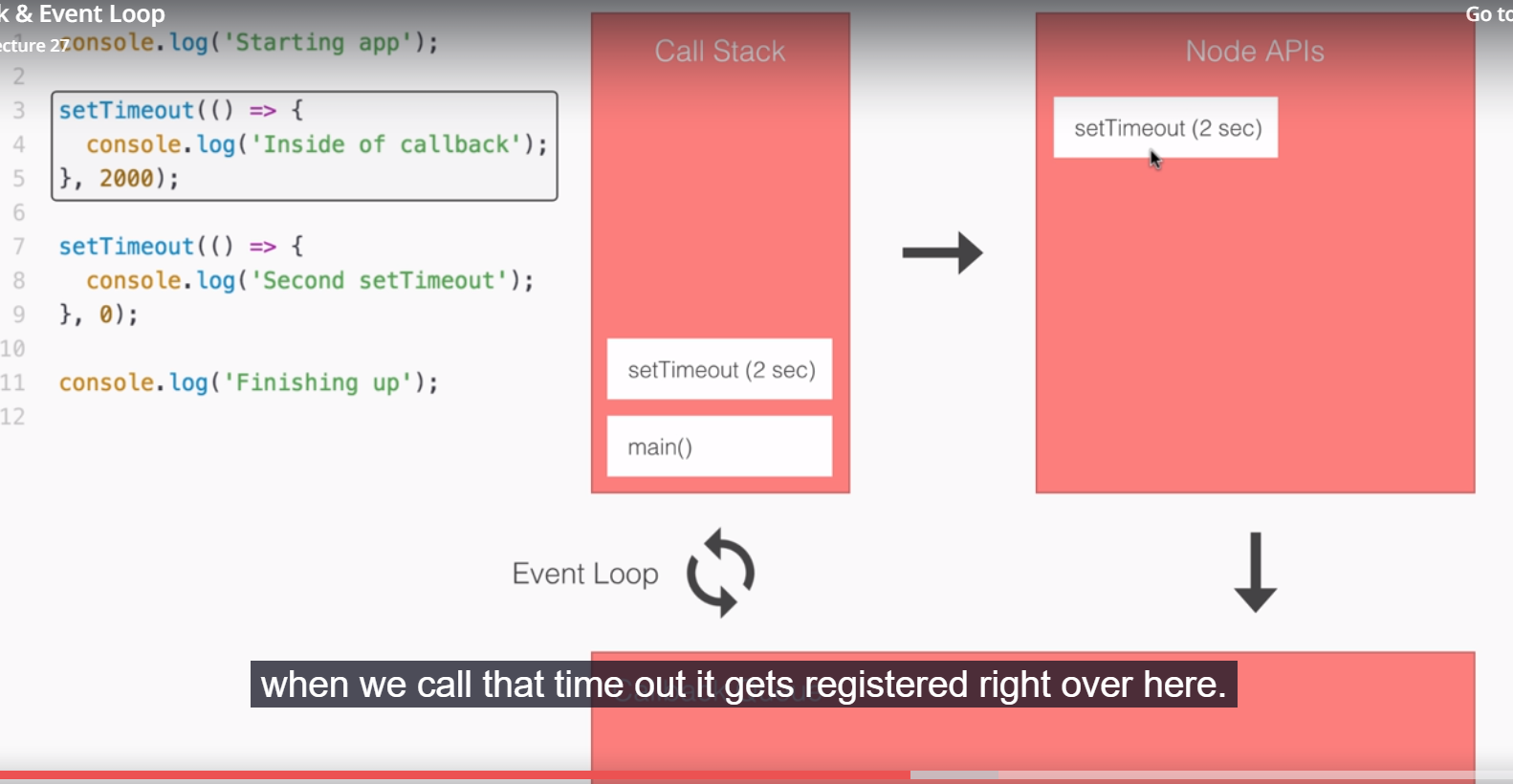


So like this we execute all statemtns. When all lines in main function is executed, this function automatically returns. Then main function is also removed from call stack. At this point our node process is closed.

In video now a second example of synchronous code using functions is hown . watch that time from 04:14 to 06:10.

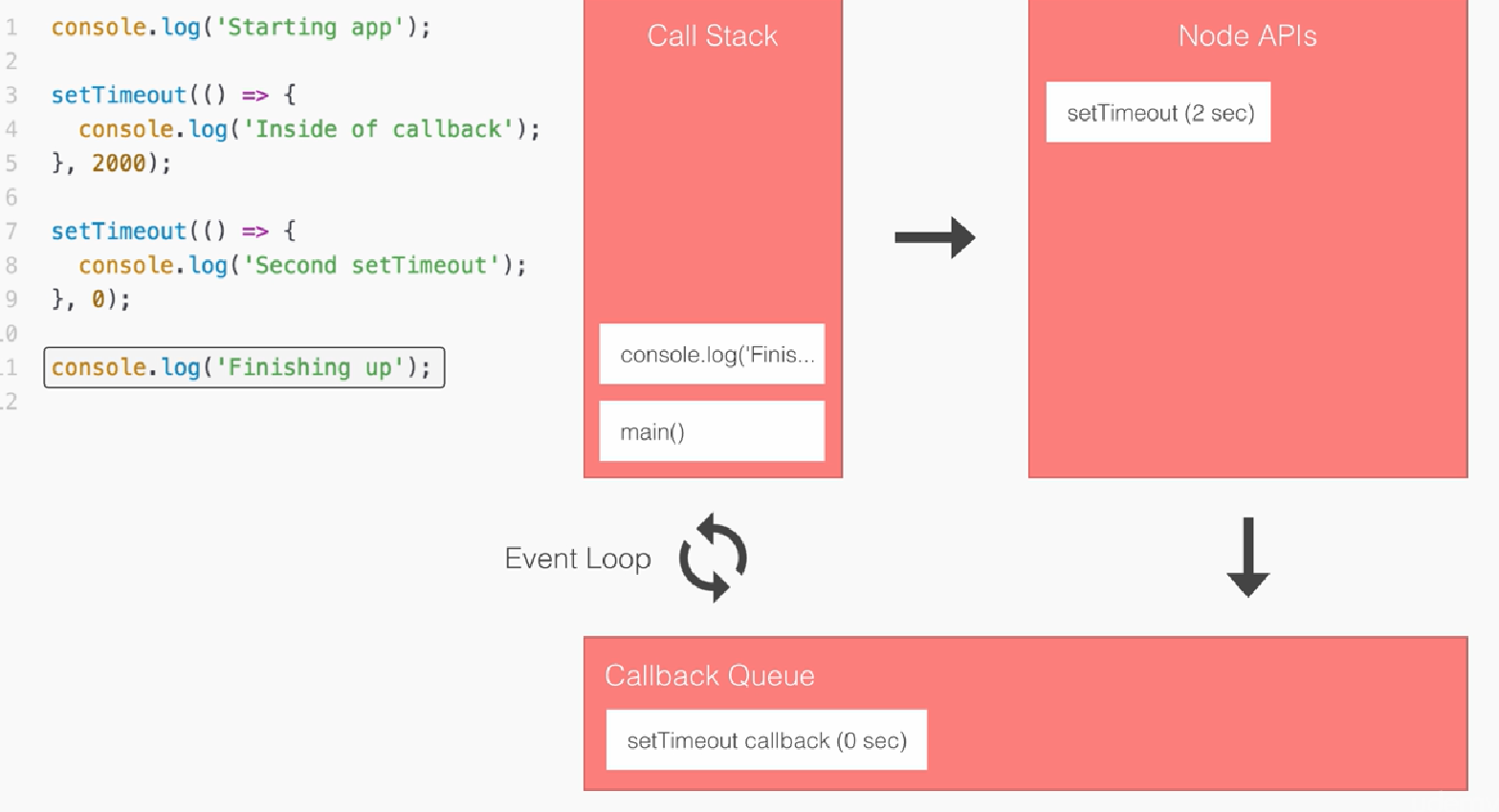
As of now we have not used node api’s, callback queue and event loop. They will be used in synce code. Lets see it.

First as we expect main is added to callstack. Then first line isexecuted. Then it is removed from callstack. Then 2nd line is executed. This is here things get interesting. This is call to setTimeout, which is a node api. It is not avalaible inside v8, it is something that node gives us access to. When we call it we are registering the event –callback pair in node api. The event is to wait 2 seconds and callback isfunction ithat we have provided to setTimeout. So when we call setTimeout it is registered in node API’s-



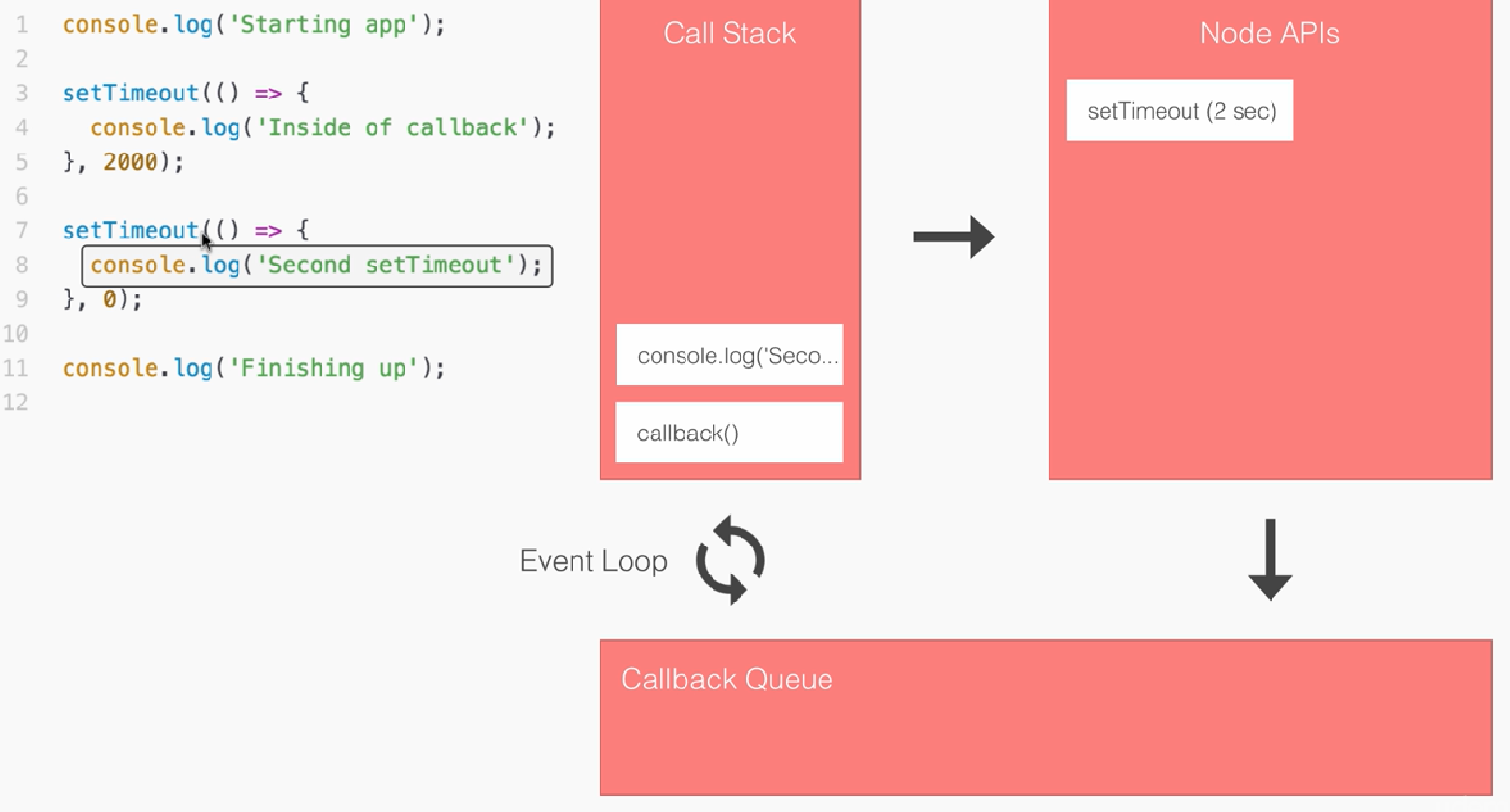
Now this statement is going to finish up and callstack is going to move on. So we remove statement from callstack but node api remains as it is. setTimeOut is going to strat counting count, just because timeOut is counting down, does not mean that callstack cannot continue to do its job. The callstack can only run one thing at a time but we could have events waiting to get processed even when callstack is executing.

Other statement that runs is another call to setTimeout. So same thing happens. It is registered in node api and call stack moves on. Callstack removes this setTimeOut statement. Now at this point lets assume that setTimeout that has delay of 0 seconds, finishes, when it finishes, it is not going to get executed right away. Now it is going to take that callback and is going to move it down to callback queue. The callback queue is all the callback functions that are ready to get fired.

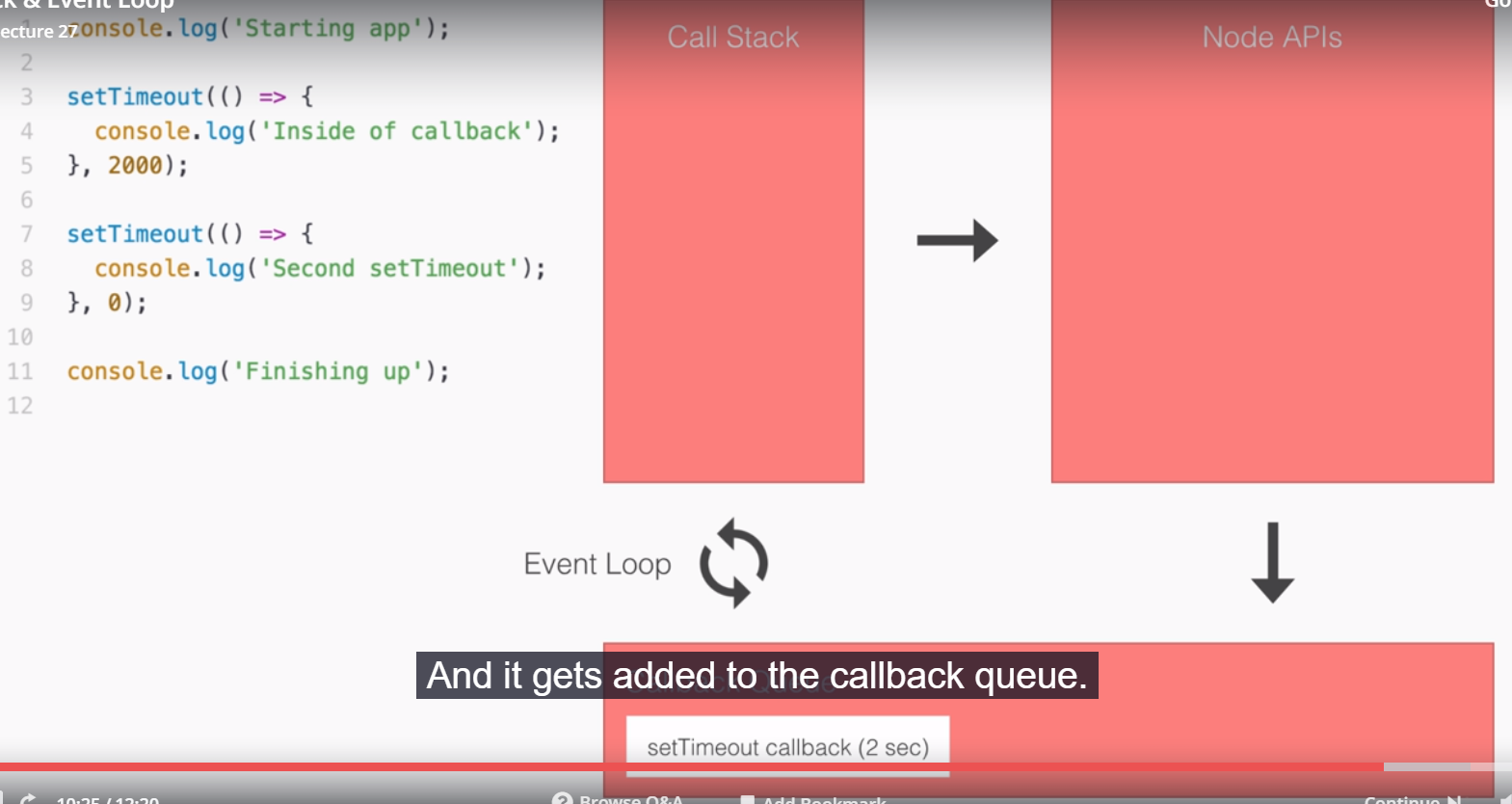


now callback queue is where callback functions are going to wait. They need to wait till callback stack is empty. When callback is empty we can run the first function. if there is another function after it, it has to wait till first function is executed. This is where event loop comes into play. The vent loop takes a look at call stack. If acll stack is not empty, it does not do anything. Because it cannot do something, you can only run one thing at a time. howevrr if call stack is empty, event loop says great, lets go ahead and see if there is anything to run in callback queue.

In ur example call stack is not empty so lets ,move ahead. So our last line that is consol.log runs and it is removed from call stack. Now all lines have been executed now our main function is also removed from call stack. At this point event loop says we have nothing in callstack and we do have something in callback queue. So let’s run that callback function. so it take callback and moves it into call stack.



This means function is executing. It prints “second timeout” on screen. This explains why second timeout appears last. It is because we cannot run callback till callstack is empty. Since last statement of our program is part of main function it is always going to appear before “second set Timeout”. Now when callback is executed then it is removed from call stack. At this time there is nothing in callstack and nothing in callback queue but there is still something in node pai. We still have a event listerner registered in node api. So node process is not yet complete. After 2 seconds event in node api is going to be firesd and callback will be moved to callback queue.



At this time event loop takes a look at call stack and sees it is empty, then it check callback queue. There we have a callback. It moves that callback to call stack. Then our function is executed. After callback is executed , our callback program implicitly returns and it is removed from call stack.

28)Callback Functions & API’s

Here we saw that we can pass callback to a function then we can execute it inside main function. this is what we saw in javascript course of Andrew mead.

This is api where we send our request and it returns the longitude and latitude of that address.

<https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20street%20Philadelphia>

as you can see it expects a query parameter. Where we can pass address. We can also use space in query parameters while entering URL in browser. Browser automatically replaces them with ‘%’. For this we are going to use third party module whichs lets us to make http request inside of your node application. That package is request. Google it-

npm/package/request

open first link. Here you can see all the documentation. Lets first create packa.json. run –

**npm init**

then install this package-

**npm install request –save**

now lets use it. Code-

const request = require('request');

request(

{

url: "https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20street%20Philadelphia",

json: true

},

(error, response, body) => {

console.log(body);

}

);

Output-

**{ results:**

**[ { address\_components: [Array],**

**formatted\_address: 'Lombard St, Philadelphia, PA, USA',**

**geometry: [Object],**

**partial\_match: true,**

**place\_id: 'ChIJf2BBgyTGxokR3oBiEh81IuU',**

**types: [Array] } ],**

**status: 'OK' }**

here we are making http request, by calling request function, it takes 2 argument. First argument is going to be options object where we can configure all sorts of information. The second one is callback function. this will be called when data comes back from Http endpoint. It takes some argumrnts, these are listed on official docs. Docs are in npm website. Now lets set up configuration object. first property is url, then we set json equal to true. This tells request that data coming back is json object and it should take that json string and convert it into js object. this will save us one step.

Now in output, that we aw on console we can see that w are seeing the value of objects. We just see [object], while if we type same url in browser we get whole object. we will correct it shorty.

29)Pretty Printing Objects

One solution to problem that we faced in last lecture is to use this-

console.log(JSON.stringify(body));

we can provide other arguments to stringyfy. Second one is to filter out properties. We don’t want to use it. Here we have to provide it because we want to provide third argument. Third argument is going to format the json and you are going to specify exactly how spaces you want to use per indentation.

console.log(JSON.stringify(body,undefined,2));

now we can see our object printed clearly on console.



30)What makes a Http Request

Here we saw that whatdoes body,response and error contin.

Body contains data, json or html.

Response has status code, body,headers. It also has request property which tells us detail about request, whose response we are sending.

Then we got error object. status code can reveal that error occurred but this is going to be an error on google serves, may be google server has syntax error and their program is crashing, may be the data that you sent is invalid for ex you sent an address that does not exist. Those errors are going to be evident via status code.

Now what error argument (to callback that we passed to request) contains is errors related to the process of even making the http request. Fir ex, may be domain is wrong, like we our url is wrong. In this case node cannot even connect to server.other erro can be our machine is not having internet access. Lets type the url wrong and print error object, we get-

**{**

**"errno": "ENOTFOUND",**

**"code": "ENOTFOUND",**

**"syscall": "getaddrinfo",**

**"hostname": "maps.gleapis.com",**

**"host": "maps.gleapis.com",**

**"port": 443**

**}**

The thing we really care about is error code. We have ENOTFOUND error, which means our machine cannot connect to host provided. So have to take care of cases when user does not have access to internet.

However if our request is successful, error object will be null.

Now in code we print latitude and longitude of our location. code-

const request = require('request');

request(

{

url:

"https://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20street%20Philadelphia$key=AIzaSyC8KvGqTf5XO0ZlfGAgTDxk\_hptqRohZfI",

json: true

},

(error, response, body) => {

console.log(`Latitude ${body.results[0].geometry.location.lat}`);

console.log(`Longitude ${body.results[0].geometry.location.lng}`);

}

);

31)Encoding User Input

User is going to type address 101 labboerd street. But while sending request we need to replace these spaces with % signs.

Lets install yargs first.

**npm install yargs –save**

then we set up otions for yargs-

const argv = yargs

.options({

a: {

demand: true,

alias: 'address',

describe: 'Address to fetch weather for',

string:true

}

});

We all know what first 3 means. Last option that is string, tells node to parse address argument as string. This is useful because we need input likethis-

node app.js –address “100 lombad street”

if we do not set string to true, user can pass this also-

node app.js –address

yargs will accept this, it will think I am trying to add Boolean flag which could be useful is some situations. For example do I want to fetch celcius or farneit. So in our case w edo not need any true false flag, we need data.to make sure we get data we set, string to true in options.

Then we set up help, we give give it alias. Then we call argv method to chain.

const argv = yargs

.options({

a: {

demand: true,

alias: "address",

describe: "Address to fetch weather for",

string: true

}

})

.help()

.alias("help", "h")

.argv;

We can use thee 2 methods for encoding and decoding-

**> encodeURIComponent('1301 lombard street philadelphia')**

**'1301%20lombard%20street%20philadelphia'**

**> decodeURIComponent('Sumeet%20Sood')**

**'Sumeet Sood'**

Then we use this encoded string in url. Code-

const request = require('request');

const yargs = require('yargs');

const argv = yargs

.options({

a: {

demand: true,

alias: "address",

describe: "Address to fetch weather for",

string: true

}

})

.help()

.alias("help", "h")

.argv;

console.log(argv);

let encodedAddress = encodeURIComponent(argv.a);

console.log(encodedAddress);

console.log(`https://maps.googleapis.com/maps/api/geocode/json

?address = ${ encodedAddress }`);

request(

{

url: `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`,

json: true

},

(error, response, body) => {

console.log(`Latitude ${body.results[0].geometry.location.lat}`);

console.log(`Longitude ${body.results[0].geometry.location.lng}`);

}

);

32)Callback Errors

Here we will see how we can handle errors in our callback functions.

Our app has many flaws, like if we try to search a address that does not exist it crashes. It is trying to read properties that dos not exist. To handle this we will see if our response is successful before we try to read some properties.

There are 2 types of errors, we are going to worry about right now. That’s going to be machine errors, things like being not able to connect to network and these are ususally going to show up in the error object, and then error coming from other server, the google server. This could be invalid address or something else. If we type invalid address, in browser url,this is what we get back-

{

* **results**: [ ],
* **status**: "ZERO\_RESULTS"

}

In case we pass real address, value of status is ok. So we can use status to see wheather request was successful or not. Code –

request(

{

url: `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`,

json: true

},

(error, response, body) => {

if (error) {

console.log("Unable to connect to google server");

} else if (body.status === "ZERO\_RESULTS") {

console.log("Unable to find address");

} else if (body.status === "OK") {

console.log(`Latitude ${body.results[0].geometry.location.lat}`);

console.log(`Longitude ${body.results[0].geometry.location.lng}`);

}

}

);

If we re not abe to reach api.i.e if our net connection is down or url is connected, then error object will be executed. If we pass a address and no match is found, then will get response , so we check response then.

33)Abstracting Callbacks

Here we want to move logic of fetching coordinates from api, into other file. app.js-

const request = require('request');

const yargs = require('yargs');

const argv = yargs

.options({

a: {

demand: true,

alias: "address",

describe: "Address to fetch weather for",

string: true

}

})

.help()

.alias("help", "h")

.argv;

let encodedAddress = encodeURIComponent(argv.a);

request(

{

url: `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`,

json: true

},

(error, response, body) => {

if (error) {

console.log("Unable to connect to google server");

} else if (body.status === "ZERO\_RESULTS") {

console.log("Unable to find address");

} else if (body.status === "OK") {

console.log(`Latitude ${body.results[0].geometry.location.lat}`);

console.log(`Longitude ${body.results[0].geometry.location.lng}`);

}

}

);

After refactoring we moved all logic into other geode.js. code-

const yargs = require('yargs');

const geocode = require('./geocode/geocode.js');

const argv = yargs

.options({

a: {

demand: true,

alias: "address",

describe: "Address to fetch weather for",

string: true

}

})

.help()

.alias("help", "h")

.argv;

geocode.geocodeAddress(argv.address, (errorMessage, results) => {

if (errorMessage) {

console.log(errorMessage);

} else {

console.log(JSON.stringify(results, undefined,2));

}

});

geocode.js-

const request = require("request");

let geocodeAddress =(address, callback) => {

let encodedAddress = encodeURIComponent(address);

request(

{

url: `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`,

json: true

},

(error, response, body) => {

if (error) {

callback("Unable to connect to google server");

} else if (body.status === "ZERO\_RESULTS") {

callback("Unable to find address");

} else if (body.status === "OK") {

callback(undefined, {

address: body.results[0].formatted\_address,

latitude:body.results[0].geometry.location.lat,

longitude: body.results[0].geometry.location.lng

});

}

}

);

};

module.exports = {

geocodeAddress

};

34)Wiring up weather search

Here Andrew opened forcast website created account, from there we got token that we are using on this app. Api structure is-

<https://api.forecast.io/forecast/token/lat,log>

if we url is wrong or we do not pass log and lat, we get 400 sttaus. We used in handling error. Here we made request on our own and got data back. Code-

code-

const request = require('request');

request(

{

url: `https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/90,91`,

json: true

},

(error, response, body) => {

if(error) {

console.log('Unable to connect to internet');

} else if (response.statusCode === 400) {

console.log('Unable to fetch weather');

} else if (response.statusCode === 200) {

console.log(body.currently.temperature);

}

}

);

35)Chaining callback togather

Here we use data of one api response to make other reqeuest.

Code-

App.js-

const yargs = require('yargs');

const geocode = require('./geocode/geocode.js');

const weather =require('./weather/weather.js');

const argv = yargs

.options({

a: {

demand: true,

alias: "address",

describe: "Address to fetch weather for",

string: true

}

})

.help()

.alias("help", "h")

.argv;

geocode.geocodeAddress(argv.address, (errorMessage, results) => {

if (errorMessage) {

console.log(errorMessage);

} else {

// console.log(JSON.stringify(results, undefined,2));

console.log(results.address);

weather.getWeather(results.latitude,results.longitude, (errorMessage, weatherResults) => {

if (errorMessage) {

console.log(errorMessage);

} else {

// console.log(JSON.stringify(weatherResults, undefined, 2));

console.log(`It's currently ${weatherResults.temperature}. it Feels like ${weatherResults.apparentTemperature}`);

}

});

}

});

Geocode.js-

const request = require("request");

let geocodeAddress =(address, callback) => {

let encodedAddress = encodeURIComponent(address);

request(

{

url: `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`,

json: true

},

(error, response, body) => {

if (error) {

callback("Unable to connect to google server");

} else if (body.status === "ZERO\_RESULTS") {

callback("Unable to find address");

} else if (body.status === "OK") {

callback(undefined, {

address: body.results[0].formatted\_address,

latitude:body.results[0].geometry.location.lat,

longitude: body.results[0].geometry.location.lng

});

}

}

);

};

module.exports = {

geocodeAddress

};

Weather.js-

const request = require('request');

let getWeather = (lat,long,callback) => {

request(

{

url: `https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/${lat},${long}`,

json: true

},

(error, response, body) => {

if (error) {

callback('Unable to connect to internet');

} else if (response.statusCode === 400) {

callback('Unable to fetch weather');

} else if (response.statusCode === 200) {

callback(undefined, {

temperature: body.currently.temperature,

apparentTemperature: body.currently.apparentTemperature

});

}

}

);

};

module.exports = {

getWeather

};

36)Intro to Promises

Promises are avalaible in javascript since ES6, althrough they have been around in third party libraries for quite some times. They finally made their way in core javascript in ES6.

For theory refer to js course. There we have promise notes.

37)Advanced Promises

There is one thing that is missed in javscript lectures of promises. See this code-

let asyncAdd = (a,b)=> {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (typeof a === 'number' && typeof b === 'number') {

resolve(a+b);

} else {

reject('Arguments should be numbers');

}

}, 1500);

});

};

asyncAdd(8,9).then((res) => {

console.log(`Result 1: ${res}`);

return asyncAdd(res,33);

}, (error) => {

console.log(error);

}).then(res => {

console.log(`Result 2: ${res}`);

}, error => {

console.log(error);

});

Outout-

**Result 1: 17**

**Result 2: 50**

Here we used promise chaining and it worked fine. Lets simulate error. First lets pass wrong arguments to second function call. Code-

let asyncAdd = (a,b)=> {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (typeof a === 'number' && typeof b === 'number') {

resolve(a+b);

} else {

reject('Arguments should be numbers');

}

}, 1500);

});

};

asyncAdd(8,9).then((res) => {

console.log(`Result 1: ${res}`);

return asyncAdd(res,'33');

}, (error) => {

console.log(error);

}).then(res => {

console.log(`Result 2: ${res}`);

}, error => {

console.log(error);

});

Output-

**Result 1: 17**

**Arguments should be numbers**

This works as expected. But things get little trickier when something earlier in promise chain gets rejected. Code-

let asyncAdd = (a,b)=> {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (typeof a === 'number' && typeof b === 'number') {

resolve(a+b);

} else {

reject('Arguments should be numbers');

}

}, 1500);

});

};

asyncAdd(8,'9').then((res) => {

console.log(`Result 1: ${res}`);

return asyncAdd(res,33);

}, (error) => {

console.log(error);

}).then(res => {

console.log(`Result 2: ${res}`);

}, error => {

console.log(error);

});

Output-

**Arguments should be numbers**

**Result 2: undefined**

First statement is expected, because this is error. Then in our promise chain, our success case runs. This is because the promise chain thinks that we have cleaned up the error because we provided the error handler(of first promise). So after running error handler , it says, ok things must be good now. Lets go ahead and move to next then call, calling the success case. Lets fix it. Code-

let asyncAdd = (a,b)=> {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (typeof a === 'number' && typeof b === 'number') {

resolve(a+b);

} else {

reject('Arguments should be numbers');

}

}, 1500);

});

};

asyncAdd(8,'9').then((res) => {

console.log(`Result 1: ${res}`);

return asyncAdd(res,33);

}).then(res => {

console.log(`Result 2: ${res}`);

}).catch(errorMessage => {

console.log(errorMessage);

});

Output-

**Arguments should be numbers**

To fix it remove both our error handlers from both then calls and replace both of them with with a call at very bottom to a different method. What we are doing to do down here is , call catch. The catch promise method is similar to then but it takes one argument and it is error handler. Here we specify error handler, if any of our promise calls fails.

Now we sued the request library to make http request. Request library does not support promises, we can wrap it in promise(see lecture end if you want to). Like we did in javascript course.

In next lecture we will see a library similar to request that do support promises.

38)Weather app with promises

As discussed, here we will use library similar to request but it supports promises. So here we will rebuilt our app. Create a new file

Lets see documentation of our new library. Google this- npm axios. Then open official npm website.

Here you can see the documentation. Official docs are good. When we make request using axios, there is no need to provide any other options like a option letting it know it’s json. Axios knows how to automatically parse our json data. This is how we can make request-

let encodedAddress = encodeURIComponent(argv.address);

let geocodeUrl = `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`;

axios.get(geocodeUrl).then((response) => {

console.log(response.data);

});

Final code is –

const yargs = require('yargs');

const axios = require('axios');

const argv = yargs

.options({

a: {

demand: true,

alias: "address",

describe: "Address to fetch weather for",

string: true

}

})

.help()

.alias("help", "h")

.argv;

let encodedAddress = encodeURIComponent(argv.address);

let geocodeUrl = `https://maps.googleapis.com/maps/api/geocode/json?address=${encodedAddress}`;

axios.get(geocodeUrl)

.then((response) => {

if(response.data.status === 'ZERO\_RESULTS') {

throw new Error('Unable to find that address');

}

let lat = response.data.results[0].geometry.location.lat;

let long = response.data.results[0].geometry.location.lng;

let weatherUrl = `https://api.forecast.io/forecast/4a04d1c42fd9d32c97a2c291a32d5e2d/${lat},${long}`;

console.log(response.data.results[0].formatted\_address);

return axios.get(weatherUrl);

})

.then(response => {

let temperature = response.data.currently.temperature;

let apparentTemperature =response.data.currently.apparentTemperature;

console.log(`It's currently ${temperature}. it feels like ${apparentTemperature}`);

})

.catch(e => {

if (e.code === 'ENOTFOUND') {

console.log('Unabale to connect with server');

} else {

console.log(e.message);

}

});

Here if we give invalid pincode or address, we will not get error,ZERO\_STATUS, so we manually throw an error. If we throw error the rest of code will not be excited and control will move to catch block. In case of second api call, we do not need to care about these things as we will laways get valid information if our address is valed.