We are going to make asynchronous http requests
Create a new folder for our new project. In the node-course folder, create weather-app folder
Create a new file under the new folder named app.js
Add the following code

```
console.log(’Starting’)


console.log(’Stopping’)
```

Add a basic asynchronous function in between. Using arrow function syntax

```
setTimeout( () => {
}, 2000)
```

It means, run this function after 2000 ms. Add something in the function

```
setTimeout( () => {
	console.log(‘2-second timer’)
}, 2000)
```

In synchronous Programming everything runs in order. But it is not the case for the asynchronous programs.
In the console, go to the correct folder. And run the app

```
node app.js
```

Now add a second timer.

```
console.log(’Starting’)

setTimeout( () => {
	console.log(‘2-second timer’)
}, 2000)

setTimeout( () => {
	console.log(‘0-second timer’)
}, 0)

console.log(’Stopping’)
```

Rerun the code.

0 second timer runs after the stopping message. Why?
2 results:
2 second timer does not delay the rest of the code
0 second timer runs after the stopping message.

Whenever we call a function, that function is added to the call stack. main function is at the bottom of the stack
When the function finishes, it is removed from the stack. When the main function is popped, the program finishes.

Main is added first at the bottom
Then console.log starting is added on top of it. Then removed once it is finished.

Settimeout is not part of the V8 script engine implementation. It is defined by node.js and part of its API implemented in c++. It is an event-callback pair. When the event is completed, callback is executed. Call stack is single threaded.

Settimeout 2 second is added to the call stack and forwarded to node api
Settimeout 0 second is added to the call stack and forwarded to node api

We are waiting for events to be completed. Settimeout 0 even is completed first and the callback is added to the callback queue.

Callback queue checks if the call stack is empty. If it is empty, it runs the callback directly. Otherwise, it waits for the functions in the call stack. Currently, main function is in the call stack.

console.log finishing is added on top of the main function in the call stack. The message is printed and the function is popped from the stack. Then main function is popped from the call stack.

Callback queue checks and see that the call stack is empty. It removes the callback and adds the timeout 0 to the call stack. Console log function for zero seconds is added on top of it in the stack. It is executed and popped from the stack. Then the callback is popped from the stack.

When the 2 seconds event is completed, its callback is added to the event loop. It takes the callback from the callback queue to the call stack. Console log two seconds is added on top of the callback. Then they are popped.

Node API extensions are stored in a separate queue and main function needs to finish first!

Let the app talk to the outside world.
Go to weatherstack.com and signup for the free account. It will generate a randomly created API access key.
We will find the endpoint for the current weather data. The base url is: `http://api.weatherstack.com/`

Copy the key: 81c6957beda8a6484399ecabcfdd7eae

We will make the request from the browser, add query strings with key-value pairs:
http://api.weatherstack.com/current?access_key=81c6957beda8a6484399ecabcfdd7eae&query=37.8267,-122.4233
For the Alcatraz island, CA.

We see the response in json format. We will make the request from out node.js app.

Delete everything in the app.js
We are going to use the request npm module. The package is deprecated but still functional. Postman forked the project and continue the project. Use postman-request instead.

First initialize app as an npm project in the folder. For default values, use -y flag to answer every question for default values.

```
npm init -y
```

Install the request module.

```
npm i request
```

In app.js, require the request module.

```
const request = require('request')
```

Copy the url from the browser and use it here:

```
const url =
'http://api.weatherstack.com/current?access_key=81c6957beda8a6484399ecabcfdd7eae&query=37.8267,-
122.4233'
```

Add url and the function to be called. There can be an error (network, etc.)

```
request({ url: url }, (error, response) => {
        console.log(response)
})
```

Run the program

```
node app.js
```

There are several properties of the response. We will use the body property which stores the data.

```
request({ url: url }, (error, response) => {
        const data = JSON.parse(response.body)
        console.log(data.current)
})
```

Rerun the app.

Now lets customize the request properties.
Require the request to be parsed as json.

```
request({ url: url, json:true }, (error, response) => {
        const data = JSON.parse(response.body)
        console.log(data.current)
})
```

Since the object will be json now, lets change the settings

```
request({ url: url, json:true }, (error, response) => {
        console.log(response.body.current)
})
```

Rerun the program. We get the same result.

Add a chrome extension to format json data properly.
Search for chrome extension json formatter: JSON formatter offered by callumlocke.co.uk
Add to chrome
Move back to the page and refresh the page. Check raw and parsed data.

Check the root object -> it is the response.body

What type of forecast we want to print?

```
request({ url: url, json:true }, (error, response) => {
        console.log('It is currently ' + response.body.current.temperature + ' degrees and it feels like ' +
response.body.current.feelslike + ' degrees.')
})
```

Run the app. node app.js

Change the units in the response.
Go to the guide in the weatherstack documentation page.
Under option look for the units parameter.
units = m for celcius, default
units = s for scientific, in kelvin, f for Fahrenheit.

```
const url =
'http://api.weatherstack.com/current?access_key=81c6957beda8a6484399ecabcfdd7eae&query=37.8267,-
122.4233&units=f'
```

Rerun the program.

Now pull weather description. Get from the array.

```
request({ url: url, json:true }, (error, response) => {
        console.log(response.body.current.weather_descriptipns[0] + '. It is currently ' +
response.body.current.temperature + ' degrees and it feels like ' + response.body.current.feelslike + '
degrees.')
})
```

Rerun the program.

Geocoding service
Convert address to lat and lon. User provides the address.
Go to https://account.mapbox.com/auth/signup/ and create a free account
Check access tokens and copy the default public token
Go to documentation. Under search service, look for the forward geocoding
Example request:

```
https://api.mapbox.com/geocoding/v5/mapbox.places/Los%20Angeles.json?access_token=pk.
eyJ1IjoiaWZzNSIsImEiOiJja215YnU1NHIwMmU4MnVvMnl0dnl2YndjIn0.iIitdw0flCDPjlnjP0Kl7g
```

The most relevant search results are in features property of the result.
Lat and lon are in the center.
Look for the optional parameters for the request. There is a language parameter, limit, etc. We will send the limit to 1. Change the query string above and refresh the page. In features there will be only one result.

Go to app.js and copy the url.

```
Const geocodeURL =
'https://api.mapbox.com/geocoding/v5/mapbox.places/Los%20Angeles.json?access_token=pk.eyJ1IjoiaWZ
zNSIsImEiOiJja215YnU1NHIwMmU4MnVvMnl0dnl2YndjIn0.iIitdw0flCDPjlnjP0Kl7g&limit=1'

request( { url: geocodeURL, json: true}, (error, response) => {
        const longitude = response.body.features[0].center[0]
        const latitude = response.body.features[0].center[1]
        console.log(latitude, longitude)
})
```

Rerun the program.