Introduction to Asynchronous JavaScript

## Day 01

Whenever you hear this term, asynchronous, immediately think about time or specifically time delays. This term involves how we manage multiple events where one event has to wait for another event to complete. It is especially important in networking and communications but essentially anywhere an event or task involves some kind of a delay. Think about opening a file, accessing a database or even about a button click.

Asynchronous programming involves applying strategies that enables your program to start a task that may take some time to complete, but at the same time be able to respond to other events happeining simultaneously. If you did not manage this properly, the risk is that the application or OS itself may become non-responsive and lead to poor user experience.

Some of the more advanced programming languages like Java, C++ and C# have the ability to spawn multiple threads. JavaScript needs an environment to operate such as the browser or the NodeJS environment. JS therefore is single-threaded and has to manage multiple tasks on one single tread or processing stream. The bootcamp is about how JS uses its environment to manage multiple tasks going on at the same time.

Note, all the code in this boot camp was produced using **VS Code**, it is a free code editor.

**Content for Day01**

**Part 1 – Callback Functions**

**Part 2 –Example with Promise Object**

**Part 3 – Async and Await**

**Part 4 – Error Handling**

**Part 5 – API Calls with fetch()**

# Part 1 – Callback Functions

A callback function can be an anonymous function that is passed to some other function, as a parameter. When the original function finishes what it was built for, it fires off the callback function (the one passed in).

For this first part, use the template provided, it is called HTML\_Template.zip. Once you unzip that file, there will be four individual files, open the index and scripts files in your favorite editor or just use Notepad. Make sure that all files are in one named folder.

Most of the HTML based examples in this bootcamp uses the *Live Server* plugin for VS Code. This is a plugin from *Ritwick Dey*. You may install it anytime, but the earlier the better. This is for users of VS Code. If you are using a different code editor, then a browser refresh may be necessary.

There is no code for this part but there is a starter zip file called **HTML\_Day01\_Part1\_Starter.zip** if you wish to use that.

1. In this first demo, on the left you see some JS code and on the right you see the result of running that code in a browser. The image on the right is the browser’s developer console which can usually be opened by hitting the **F12** key on the keypad:

|  |  |
| --- | --- |
| **function A() {  console.log("A") } function B() {  console.log("B") } function C() {  console.log("C") } A(); B(); C();** |  |

Note: if you are using VS Code, you can open the HTML file with Live Server, it’s a plugin for VSCode

1. Now what will happen if say, function B is delayed. In this case we have delayed its return for 1 second. The result is that C runs before B:

|  |  |
| --- | --- |
| **}**  **function B() {**  **setTimeout(() => {**  **console.log("B")**  **},1000);**  **}**  **function C() {**  **console.log("C")**  **}** |  |

1. What we need is to preserve the order. In other words we want to make sure that **C** does NOT run Before **B**.

This is where asynchronous programming comes in.

Before modern methods of dealing with this situation existed, we used callback functions to solve the problem.

A callback function simply means, go and run this piece of code, then when you are done running that code execute this other function (the callback function).

Of course we could simply move the C line (log()) to just below the B line and that would solve the problem. In a complex application however this may not be possible. Also you will be passing in the processing of the C function, it would be smoother to just pass in a reference to the function!

1. We could re-architect the **B()** function to accept another function to be executed, once B is done with it’s task. We could pass this other function into B *as a parameter*, then have B execute that function that was passed in. It does not matter what the function is, once it is passed in, that passed in function gets executed, AFTER B is done.

|  |  |
| --- | --- |
| **…**  **function B(cb) {**  **setTimeout(() => {**  **console.log("B");**  **cb();**  **},1000);**  **}**  **…** | *Notice that cb represents a function. That cb function is now called after B is done its work. This is the reason that the passed in function is called a callback function, it is called back once the current function finishes.* |

1. So, now that **cb** is inside of **B**, when we call B we can pass in **C**!

|  |  |
| --- | --- |
| …  **function C() {**  **console.log("C");**  **}**  **//**  **A(); B(C);** | *Notice when we pass C into B, we do NOT add the (). Adding the () will execute C, we don’t want that here, we want C to be executed inside of B where it belongs.* |

1. Here is the entire code:

|  |  |
| --- | --- |
| **function A() {**  **console.log("A")**  **};**  **function B(cb) {**  **setTimeout(() => {**  **console.log("B");**  **cb();**  **},1000);**  **};**  **function C() {**  **console.log("C")**  **};**  **//**  **A(); B(C); //two functions one line** |  |

|  |
| --- |
| End of section |

# Part 2 –Example with Promise Object

In this section we take a look at a different example and a different solution. This example creates a new **Promise** object. The function that does the creation will therefore **return** a Promise object that must be handled. In most cases however, you will be working with ordinary JS functions that return Promisses. All you have to do is handle the promise.

The code for this part is called **HTML\_Day01\_Part2.zip**. It contains the code in a **completed** state, so if you wanted to follow along, you can cope the code in the box on the left below.

1. In this next demo, on the left you see some JS code and on the right you see the result of running that code in a browser. The image on the right is the browser’s response. Alternatively you could use the developer console to view results.

|  |  |
| --- | --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **el.innerHTML = el.innerHTML + "I was first..."**  **};**  **doFirst();**  **el.innerHTML = el.innerHTML + "<br />And I am second!"** | *Note, in the previous example we used the developer’s console window to view the result, here we use the DOM.* |

1. Now, if we add a **setTimeout()** function to the **doFirst()** function, we could delay the output from that function. This forces the next output to print first, but it should be second!

|  |  |
| --- | --- |
| …  **let el = document.getElementById("response");**  **const doFirst = function() {**  **setTimeout(function(){**  **el.innerHTML = el.innerHTML + "<br />I was first..."**  **}, 2000);**  **};**  **doFirst();**  **el.innerHTML = el.innerHTML + "<br />And I am second!"** … |  |

Now that the order is affected, we could promissify the **doFirst()** function by returning a Promise object. This example is NOT related to the previous example of call back functions.

1. One way to solve the order issue, is to return a Promise object from the **doFirst()** function:

|  |  |
| --- | --- |
| …  **let el = document.getElementById("response");**  **const doFirst = function() {**  **const pf = new Promise();**  **setTimeout(function(){**  **el.innerHTML = el.innerHTML + "<br />I was first..."**  **}, 2000);**  **};**  **doFirst();**  **el.innerHTML = el.innerHTML + "<br />And I am second!"** … |  |

1. The reason we got the error in #3 is that we did not supply the *handler* function. The architecture of *promises* is that once we use them, we have to immediately provide a function to run after the promise completes. A promise can be in one and only one of three states, *pending, fulfilled* or *rejected.* Lets pass in an empty function that will get rid of the error:

|  |  |
| --- | --- |
| **…**  **let el = document.getElementById("response");**  **const doFirst = function() {  const pf = new Promise( () => {**    **} );**  **setTimeout(function(){**  **el.innerHTML = el.innerHTML + "<br />I was first..."**  **}, 2000);**  **};**  **doFirst();**  **el.innerHTML = el.innerHTML + "<br />And I am second!"**  **…** |  |

1. However, the handler function inside the Promise constructor, has to supply two variables that would represent two functions, **resolve()** and **reject().** These are parameters but they each represent a function.

|  |  |
| --- | --- |
| …  **let el = document.getElementById("response");**  **const doFirst = function() {  const pf = new Promise( (resolve, reject) => {**    **} );**  **setTimeout(function(){**  **el.innerHTML = el.innerHTML + "<br />I was first..."**  **}, 2000);**  **};**  **doFirst();**  **el.innerHTML = el.innerHTML + "<br />And I am second!**" … | *You will see how these two parameters come into the story a bit later.* |

1. Now we need to insert the **setTimeout()** into the **Promise** object, via the handler function. Basically once the **setTimeout()** function expires, the handler function will take over and handle the result of the **setTimeout()** function.

|  |  |
| --- | --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **const pf = new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **el.innerHTML = el.innerHTML + "<br />I was first..."**  **}, 2000);**  **});**  **};**  **doFirst();**  **el.innerHTML = el.innerHTML + "<br />And I am second!"** | *Although this setup will work in this case, it is not the proper way to handle output. We need to engage the resolve parameter, which remember, is a method.* |

1. Now we need to wrap our output, normally this would be data, into the resolve() function:

|  |  |
| --- | --- |
| **let el = document.getElementById("response");**  **const pf = new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **resolve(el.innerHTML = el.innerHTML + "<br />I was first...");**  **}, 2000);**  **}); doFirst();**  **el.innerHTML = el.innerHTML + "<br />And I am second!"** | *The output is still the same and we still have not yet solved the problem.* |

1. Lets continue by returning the promise, pf:

|  |  |
| --- | --- |
| **const doFirst = function() {**  **const pf = new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **resolve(el.innerHTML = el.innerHTML + "<br />I was first...")**  **}, 2000);**  **});**  **return pf;**  **};** | *Still no changes in output.* |

1. Now that **doFirst()** returns a promise, it means that we have to handle the result of that promise when we make the call to the **doFirst()** method. We now have to supply a **then()** method, chain it on to the **doFirst()** method and then provide a function to handle any output from the **doFirst()** method. In this case we don’t actually have data to handle, the **doFirst()** function simply prints some data to the DOM:

|  |  |
| --- | --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **setTimeout(function(){**  **el.innerHTML = el.innerHTML + "<br />I was first..."**  **}, 2000);**  **};**  **doFirst().then();**  … |  |

1. Notice that at this point, the output still has not changed. The change happens in the architectural design of our program. Before we change the structure of the program let us supply an anonymous function to the **then()** method.

|  |  |
| --- | --- |
| **doFirst().then( () => {**    **} );**  **…** | *This wont do anything to the output but it will give us the architectural structure in which to change the output.* |

1. Now we provide a variable for the data being returned from the Promise:

|  |  |
| --- | --- |
| **doFirst().then(data => {**  **data;**  **});** |  |

The variable we provide is called *data*.

1. With this strucutural change, we can move the last output line into the function that runs after the **doFirst()** function. This is the line that prints “And I am second!”

|  |  |
| --- | --- |
| **doFirst().then(data => {**  **data;**  **el.innerHTML = el.innerHTML + "<br />And I am second!";**  **});** | *Finally we see some change* |

1. Here is the entire block of code:

|  |
| --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **const pf = new Promise((resolve, reject)=>{**  **setTimeout(function() {**  **resolve(el.innerHTML = el.innerHTML + "<br />I was first...");**  **}, 2000);**  **});**  **return pf;**  **};**  **doFirst().then(data => {**  **data;**  **el.innerHTML = el.innerHTML + "<br />And I am second!";**  **});** |

1. (Optional) In this particular example, it is NOT necessary to actually create a new Promise object, we can have the object created automatically by simply returning a Promise:

|  |
| --- |
| **const doFirst = function() {**  **return new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **resolve("I was first...")**  **}, 2000);**  **}); …** |
| End of section | | |

# Part 3 – Async and Await

In this section we take a look at a more recent addition to the asynchronous JS world.

The code for this part is called **HTML\_Day01\_Part3.zip**. It contains the code in a **completed** state, so if you wanted to follow along, you can cope the code in the box on the left below.

1. We will continue with the previous example, but in the form shown in #15 above. Either comment out the call to **doFirst()** or delete it completely, so all you should have in the program is this code:

|  |
| --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **return new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **resolve("I was first...")**  **}, 2000);**  **});**  **};** |

1. Below the **doFirst()** function create a new function and decorate it with **async**:

|  |
| --- |
| **async function showMessages(){**  **}** |

We will use this function to do exactly what we did in the previous section, display two messages in order onto the browser window.

1. Call the **doFirst()** method from within **showMessages(),** but wait for it to finish so decorate the call with **await** keyword:

|  |
| --- |
| **async function showMessages(){**  **await doFirst();**  **}** |

1. After waiting for the **doFirst()** function to complete, we need to do something with the data being returned, so we could simply add it to a variable:

|  |
| --- |
| **async function showMessages(){**  **let firstMessage = await doFirst();**  **}** |

1. Now just show the message from the Promise onto the DOM:

|  |
| --- |
| **async function showMessages(){**  **let firstMessage = await doFirst();  el.innerHTML = firstMessage;**  **}** |

1. Nothing happens as yet because we have not called **showMessages(),** so lets do that now with a new line:

|  |
| --- |
| **async function showMessages(){**  **let firstMessage = await doFirst();  el.innerHTML = firstMessage;**  **} showMessages();** |

Well we got the first message but what about the second?

1. With this new architecture, all we have to do now is *plug-in* the second message:

|  |
| --- |
| **async function showMessages(){**  **let firstMessage = await doFirst();  el.innerHTML = firstMessage + "<br />And I am second!";**  **} showMessages();** |

1. Here is the entire code for this program:

|  |
| --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **return new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **resolve("I was first...")**  **}, 2000);**  **});**  **};**  **//**  **async function showMessages(){**  **let firstMessage = await doFirst();**  **el.innerHTML = firstMessage + "<br />And I am second!";**  **}**  **//**  **showMessages();** |

|  |
| --- |
| End of section |

# Part 4 – Error Handling

In this section we handle the negative side of Promisses, the errors that might be thrown during the execution of a Promise.

The code for this part is called **HTML\_Day01\_Part4.zip**. It contains the code in a **completed** state, so if you wanted to follow along, you can cope the code in the box on the left below. It would be better to follow along using the code at the end of Part3.

1. Actually in order to handle errors with the async/await construct, well it will be just like normal JS error handling, use a **try…catch** structure. Here is the code from the async and await section:

|  |
| --- |
| **try{**  **let firstMessage = await doFirst();**  **el.innerHTML = firstMessage;**  **el.innerHTML = firstMessage + "<br />And I am second!";**  **} catch(err){**  **el.innerHTML = "An Error Occured!";  }** |

If an error occurred, the **err** variable will hold that error object generated.

1. Here is the entire code for async/await

|  |
| --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **return new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **//resolve("I was first...");**  **reject("An error occured");**  **}, 2000);**  **});**  **};**  **//**  **async function showMessages(){**  **try{**  **let firstMessage = await doFirst();**  **el.innerHTML = firstMessage;**  **el.innerHTML = firstMessage + "<br />And I am second!";**  **} catch(err){**  **el.innerHTML = "An Error Occured!";**  **}**  **}**  **//**  **showMessages();** |

In this case, I am purposely generating an error in the **try** block, using the reject parameter.

1. Lets now return to the code from the *part2 #15*, so start with this code:

|  |
| --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **return new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **resolve("I was first...")**  **}, 2000);**  **});**  **};**  **//**  **doFirst().then((data)=>{**  **el.innerHTML = data + "<br />And I am second!"**  **});** |

For this example, just replace the current code with the code from #3

1. From the code above notice that we only used the *resolve* parameter, what about reject? Well, lets replace the **resolve**() function with a **reject**() function

|  |
| --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **return new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **reject("Oops we have a problem!")**  **}, 2000);**  **});**  **};**  **//**  **doFirst().then((data)=>{**  **el.innerHTML = data + "<br />And I am second!"**  **});** |

Notice that immediately we get an error in the Console window.

1. The error occurred in the Browser because we as programmers did not handle it. Luckily the Promise architecture has a built in error handling mechanism. This error handler must be called once the function that return the promise is called, so basically we add another method to the first call (chaining), the **catch()** method:

|  |
| --- |
| **doFirst()**  **.then((data)=>{**  **el.innerHTML = data + "<br />And I am second!"**  **})**  **.catch(err => {**  **console.log("An error occured!");**  **el.innerHTML = "An error occured!"**  **});** |

This is called chaining. The **catch()** method is chained to the first **then()** method which is chained to the Promise call.

1. Here is the entire block of code so far:

|  |
| --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **return new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **//resolve("I was first...");**  **reject("Oops we have a problem!");**  **}, 2000);**  **});**  **};**  **//**  **doFirst()**  **.then((data)=>{**  **el.innerHTML = data + "<br />And I am second!";**  **})**  **.catch(err => {**  **el.innerHTML = err;**  **});** |

1. There is a different (alternative) way to handle the **then()** method, basically pass in an object with two parameters, one for the positive and one for negative. **This is the recommended way according to the documentation**.

|  |
| --- |
| **doFirst()**  **.then(**  **resolve => { },**  **reject => { }**  **);** |

Each parameter represents functions so here we have empty functions for now.

1. Now we can complete each function

|  |
| --- |
| **doFirst()**  **.then(**  **resolve=>{**  **el.innerHTML = resolve + "<br />And I am second!"**  **},**  **reject => {**  **el.innerHTML = "<p style='color:red;'>" + reject + "...check with your Administrator";**  **}**  **);** |

So, now if we have data returned, it goes into the **resolve** block and if an error occurred, it goes into the **reject** block. Also I added a red font to the reject output.

1. Here is the entire program with the reject commented out:

|  |
| --- |
| **let el = document.getElementById("response");**  **const doFirst = function() {**  **return new Promise((resolve, reject)=>{**  **setTimeout(function(){**  **resolve("I was first...");**  **//reject("Oops we have a problem!");**  **}, 2000);**  **});**  **};**  **//**  **doFirst()**  **.then(**  **resolve=>{**  **el.innerHTML = resolve + "<br />And I am second!"**  **},**  **reject => {**  **el.innerHTML = "<p style='color:red;'>" + reject + "...check with your Administrator";**  **}**  **);** |

|  |
| --- |
| End of section |

# Part 5 – API Calls with fetch()

In this section we will make an API call using **fetch()** method which is part of JS. This function **returns** a Promise and must be handled using *one* of the promise architecture.

There is a fresh starter code for this section, its called “Starter for Fetch.zip”..

1. Once you get the starter code loaded, press the button to see that the code behind works. Now replace the code in the **showData**() function with the code below:

|  |
| --- |
| **function showData(){**  **fetch('https://jsonplaceholder.typicode.com/posts/1');**  **};** |

1. The **fetch()** method did its job but we have not provided the structure to accept the data and show it on the browser window, so one step at a time. First apply a **then()** method to handle the returned promise

|  |
| --- |
| **function showData(){**  **fetch('https://jsonplaceholder.typicode.com/posts/1')**  **.then();**  **};** |

1. Now inside the **then()** method, provide a function to handle any data being returned

|  |
| --- |
| **fetch('https://jsonplaceholder.typicode.com/posts/1')**  **.then( (data) => {**  **console.log(data)**  **} );** |

The return is a cryptic blob of a stream of data that is not recognizable

1. However, if we apply the **json()** method to the data object, we will parse the original object to extract just the *json* data from it

|  |
| --- |
| **fetch('jsonplaceholder.typicode.com/posts/1')**  **.then( (data) => {**  **console.log(data.json() )**  **} );** |

Notice that the json() method also produces a Promise object but the data we seek is inside

1. What we have to do is pass the original data to the **json()** method, then we will get the actual data:

|  |
| --- |
| **fetch('https://jsonplaceholder.typicode.com/posts/1')**  **.then( (data) => {**  **return data.json()**  **} );** |

1. Now that we returned the data, we have to chain on another **then()** method and supply another function to handle the data returned by the previous **then()** method:

|  |
| --- |
| **fetch('https://jsonplaceholder.typicode.com/posts/1')**  **.then((data)=>{**  **return(data.json())**  **})**  **.then(()=>{**    **})** |

1. At this point we supply a variable to hold the actual data, then we can show that data in the console window:

|  |
| --- |
| **.then((data)=>{**  **return(data.json())**  **})**  **.then((data)=>{**  **console.log(data);**  **})** |

1. Actually since we have a handle onto the DOM, we could start throwing our data onto the browser window via the **<div>** tag that we have access to:

|  |
| --- |
| **.then((data)=>{**  **return(data);**  **})**  **.then((post)=>{**  **el.innerHTML = post.title;**  **})** |

Remember *title* is part of the *post* available via the JSONPlaceholder data. What is returned to us is an Object. In order to interact with this object, we interact with it’s name/value pairs like title, body etc.

1. The only thing left is to cater for errors, so just like before simply chain on the **catch()** method for that:

|  |
| --- |
| **.then((post)=>{**  **el.innerHTML = post.title;**  **})**  **.catch(err => {**  **el.innerHTML = err;**  **})** |

To reproduce this error (in the image), just change the URL to something that does not exist, for example I simply removed the *m* from *com* and it produced this error

1. Here is the entire program with error handling, on the right is the same code but with regular functions:

|  |  |
| --- | --- |
| **let el = document.getElementById("response");**  **//**  **function showData(){**  **fetch('https://jsonplaceholder.typicode.com/posts/1')**  **.then((data)=>{**  **return(data.json())**  **})**  **.then((data)=>{**  **return(data);**  **})**  **.then((post)=>{**  **el.innerHTML = post.title;**  **})**  **.catch(err => {**  **el.innerHTML = err;**  **})**  **};** | **let el = document.getElementById("response");**  **//**  **function showData(){**  **fetch('https://jsonplaceholder.typicode.com/posts/1')**  **.then(function(data){**  **return(data.json())**  **})**  **.then(function(data){**  **return(data);**  **})**  **.then(function(post){**  **el.innerHTML = post.title;**  **})**  **.catch(function(){**  **el.innerHTML = err;**  **});**  **};** |

1. To use **async/await**, first remove all the **then()** methods but leave the **fetch**, also wrap the fetch into a new function and add the **async** decoration to it:

|  |
| --- |
| **async function getData(){**  **fetch('https://jsonplaceholder.typicode.com/posts/1');**  **};** |

1. Since we will now wait for the response from the **fetch** method, lets add a variable to hold that data in and at the same time decorate the **fetch** method with the **await** keyword:

|  |
| --- |
| **async function getData(){**  **const postData = await fetch('https://jsonplaceholder.typicode.com/posts/1');**  **};** |

Note: this is one long line but it looks like 2 lines, I will show you the entire line in smaller font so that it fits on this page:  
**const postData = await fetch('https://jsonplaceholder.typicode.com/posts/2');**

1. We still have to run the returned object through the **json()** method:

|  |
| --- |
| **async function getData(){**  **const postData = await fetch('https://jsonplaceholder.typicode.com/posts/1');  const finalData = await postData.json();**  **};** |

1. Check your html file and make sure you are calling the **getData()** function once the button is clicked:

|  |
| --- |
| **<p></p>**  **<div style="font-size: larger;" id="response"></div>**  **<button id="button1" onclick="getData()">Get Data</button>**  **</main>** |

1. At this point, if you now include a line to log the data you will see the data appear in the console window

|  |
| --- |
| **async function getData(){**  **const postData = await fetch('jsonplaceholder.typicode.com/posts/1');**  **const finalData = await postData.json();**  **console.log(finalData);**  **};** |

1. In order to get data such as the *title* or *body* of the post at the DOM level, simply extract the properties you want:

|  |
| --- |
| **async function getData(){**  **const postData = await fetch('https://jsonplaceholder.typicode.com/posts/1');**  **const finalData = await postData.json();**  **el.innerHTML = finalData.title;**  **};** |

1. Lets go further and add in error handling:

|  |
| --- |
| **let el = document.getElementById("response");**  **//**  **async function getData(){**  **try{**  **const postData = await fetch('https://jsonplaceholder.typicode.com/posts/1');**  **const finalData = await postData.json();**  **el.innerHTML = finalData.title;**  **}catch(err){**  **el.innerHTML = "Oops " + err;**  **}**  **};** |

|  |
| --- |
| End of section |

# Appendix A – Promise Object in Theory

A Promise is an object which **represents** the result of an asynchronous operation, either the success of that operation or its failure.

Promisses require handler functions and it is suggested that you **attach** these funtions to the function that returns the promise object.

A Promise can be in one and only one of these states:

pending: initial state, it has not been resolved.

fulfilled: meaning that the operation was completed successfully, it is resolved

rejected: meaning that the operation failed, an error object may be returned

After the pending state, a promise can either be **fulfilled** with the expected value or **rejected** as an error.

As soon as one of these two states are realized, the associated handler functions passed into the promise mechanism and stored in the promise's then method are called.

# Appendix B – Node Projects

When working with a Node project such as the one in #6 above, there are a few steps that you need to take to get the project up to date.

1. If you download the zipped file HTML\_Day01\_Part6.zip you will notice that it is NOT like the one in the document. This is because Node projects are designed to be slim and contain a very small footprint. It is up to us to re-create the original project.
2. First unzip the file and put it into a folder. Navigate into that folder and run the command **npm install**. The folder will re-create itself and you will see the node\_modules folder re-appear.
3. Once the folders have been reconstructed, you may continue following the instructions from #6, section 5.

# Appendix C – Technologies Used

1. HTML – any modern browser will work
2. JavaScript - any modern browser will work
3. Code Editor – instructor will use Visual Studio Code (free version)
4. NodeJS – environment to run JavaScript code

**APPENDIX D – FILE WATCHERS LIMIT REACHED (ENOSPC)**

1. If you are on Linux you may get an error about file watchers limit, this is due to the OS being not configured to handle heavy applications. Hidden from view is a package called inotify that Linux systems use to track this sort of activity. You would need to adjust this value using the steps below:

|  |
| --- |
| Run the following command at any terminal prompt: **sudo gedit /etc/sysctl.conf**  This would bring up the sysctl.conf file for editing    Add a new line with this value: **fs.inotify.max\_user\_watches=524288**  Make sure that there is no # sign in front of the line.  Save the file and try restarting the app with **npm start**  Or, temporarily:  **sudo sysctl -w fs.inotify.max\_user\_watches=524288** |
|  |

# Appendix E – Handle Response After POST Request

For those participants who are interested in what happens after a POST request is made, well you do get a response from the server. However, I was not able to cause this response to stay on the browser window using the json-server so here is the code using the json placeholder web endpoint:

|  |
| --- |
| let el = document.getElementById("response");  //  async function showData(){  try{  const postData = await fetch(  '*https://jsonplaceholder.typicode.com/posts*',  {  method:"POST",  headers: {  'Content-Type': 'application/json',  },  body:JSON.stringify(  {  "title":"Jennifer",  "body":"1234"  }  )  }  );  const response = await postData.json();  el.innerHTML = response.title + "<br />" + response.body;  }catch(err){  el.innerHTML = "Oops " + err;  }  }; |

# Appendix F – Technologies Used

From part 6, and any of the zipped files involving NodeJS, the unzipped code and the code that I show in the bootcamp will be different. The zipped code has to be built. Here are the steps to follow:

1. Unzip the file in a folder
2. Using a terminal window, navigate to the root folder. For example if you unzipped part6, the final code will be in a folder called HTML. Naviagate to that folder using the CD command.
3. Once you are at the folder, you will notice a package.json file. This is the manifest that Node will use to build your app. Run the command following command:  
   npm install
4. After that command is executed, you may or may not have a node\_modules folder created along with a package-lock.json file. Your folder is now ready for further development.

# Appendix G – Definitions

## **Synchronous Code**

Synchronous code is executed sequentially from the top of a file to the end of a file — with each line unable to execute until the previous line has finished executing.

Consider the building of a house. We would first need to first lay down the bricks that make our foundation. Then, we layer more bricks on top of each other, building the house from the ground up. We can’t skip a level and expect our house to be stable. Therefore, the laying of bricks happens *synchronously*, or in sequential order.

Likewise, *synchronous code* executes in sequential order — it starts with the code at the top of the file and executes line by line until it gets to the end of the file. This type of behavior is known as *blocking* (or blocking code) since each line of code cannot execute until the previous line finishes.

**Asynchronous Code**

Asynchronous code can be executed in parallel to other code that is already running. Let’s begin again with examining a real-life scenario, like baking a cake. We could start to preheat the oven and prepare our cake’s ingredients while we wait for our oven to heat up. The wait for the oven and the preparation of ingredients can happen at the same time, and thus these actions happen *asynchronously*.

Similarly, *asynchronous code* can be executed in parallel to other code that is already running. Without the need to wait for other code to finish before executing, our apps can save time and be more efficient. This type of behavior is considered *non-blocking*.

**Asynchronous Code Under the Hood**

For most programming languages, the ability to execute asynchronous code depends on the number of *threads* that an app has access to. We can think of a thread as a resource that a computer provides an app to do a task. Typically one thread allows for an app to complete one task. If we return to our house example, our computers thread tasks might look like this:

Thread 1: build house foundation -> build walls -> construct floor

A single thread could work for a synchronous task like building a house. However, in our cake baking example, our threads would have to look like this:

Thread 1: preheat oven  
Thread 2: prepare ingredients -> bake cake

We won’t discuss in-depth how many threads an app can access but we should note that the more threads we have, the more tasks we can run concurrently. Also, in most modern-day computers, multithreading is achieved by having a CPU that has multiple cores or by some other technology.

## **Asynchronous Code in Web Development**

Similar to how asynchronous behavior is useful in baking a cake, it can also be helpful for web programming. If we use synchronous (blocking) code in the browser, we might be stopping a user from being able to interact with a web app until the code is done running. This isn’t a great user experience. If our app takes a long time to load, our users might think that something’s wrong and might even opt to browse a different site!

However, if we opt for an asynchronous approach, we can cut down on the wait time. We’d load only the code that’s necessary for user interactions and then load up other bits of code in the background. With asynchronous code, we can create better user experiences and make apps that work more efficiently!

**Blocking**

Blocking occurs when code prevents a user from interacting with an app due to background code not finishing execution.

**Threads**

Generally, the amount of tasks a language can execute is limited by the amount of threads the language has access to.

**async-javascript-single-threaded**

JavaScript is a single-threaded language. However, it can handle asynchronous code using the event loop.

**async-javascript-syntaxes**

In JavaScript, asynchronous code can be written in a variety of different ways, including:

Callbacks, Promises, async/await

**async-javascript-set-timeout**

setTimeout() accepts two arguments:

* A callback function that is executed asynchronously
* The minimal number of milliseconds a program waits before executing the callback function

**async-javascript-set-interval**

setInterval() accepts two arguments:

* A callback function that is executed asynchronously
* The number of milliseconds for how frequently the callback function executes