

---

# 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 happening simultaneously on the system. 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 thread 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 6 – JSON Server**

**Part 7 – Post Requests**

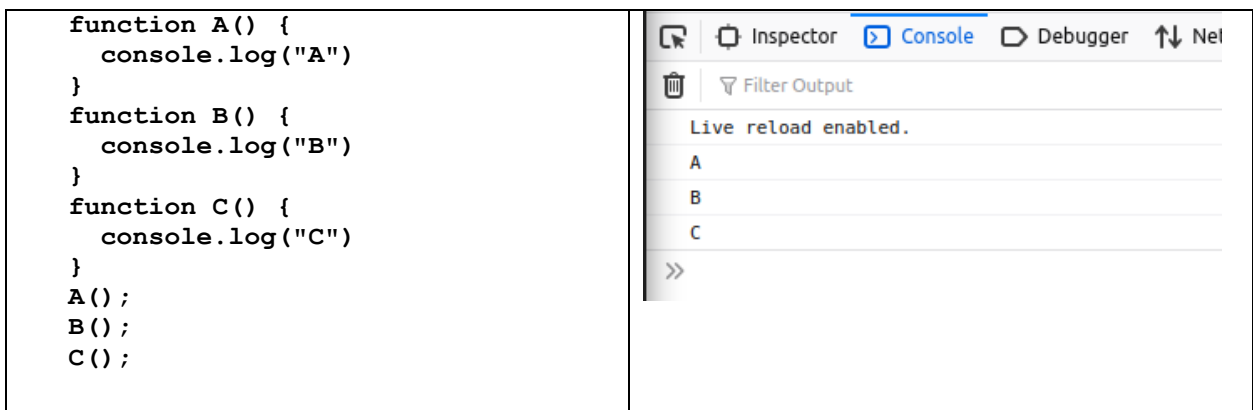
## Part 1 – Callback Functions

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).

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 **AJS\_Day01\_Part1\_Starter.zip** if you wish to use that. There is also the ending code.

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:



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

2. 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:

<pre> } function B() {   setTimeout(() =&gt; {     console.log("B")   }, 1000); } function C() {   console.log("C") } </pre>	
--	--

3. 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!

4. 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.

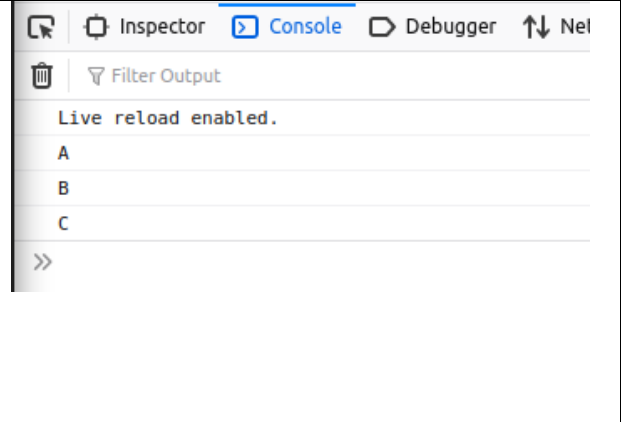
<pre> ... function B(cb) {   setTimeout(() =&gt; {     console.log("B");     cb();   }, 1000); } ... </pre>	<p><i>Notice that cb represents a function. That cb function is now called after B is done its work. This is the reason that the passed in function is called a callback function, it is called back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

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

<pre> ... function C() {   console.log("C"); } // A(); B(C); </pre>	<p><i>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.</i></p>
---	---

6. 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 Promises. All you have to do is handle the promise.

The code for this part is called `AJS_Day01_Part2_Starter.zip`. There is also the ending code.

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.

<pre>let el = document.getElementById("response"); const doFirst = function() {   el.innerHTML = el.innerHTML + "I was first..." };  doFirst();  el.innerHTML = el.innerHTML + "&lt;br /&gt;And I am second!"</pre>	<div> <div>The JavaScript Event Loop</div> <div>Event Loop, Callbacks, Promises, and Async/Await</div> <div>Output appears here</div> <div>I was first... And I am second!</div> <div>Note, in the previous example we used the developer's console window to view the result, here we use the DOM.</div> </div>
---	--

2. 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!

<pre>... let el = document.getElementById("response"); const doFirst = function() {   setTimeout(function(){     el.innerHTML = el.innerHTML + "&lt;br /&gt;I was first..."   }, 2000); }; doFirst();  el.innerHTML = el.innerHTML + "&lt;br /&gt;And I am second!" ...</pre>	<div> <div>The JavaScript Event Loop</div> <div>Event Loop, Callbacks, Promises, and Async/Await</div> <div>Output appears here</div> <div>And I am second! I was first...</div> </div>
---	---

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

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

<pre>... let el = document.getElementById("response"); const doFirst = function() {   const pf = new Promise();   setTimeout(function() {     el.innerHTML = el.innerHTML + "&lt;br /&gt;I was first..."   }, 2000); }; doFirst(); el.innerHTML = el.innerHTML + "&lt;br /&gt;And I am second!" ...</pre>	<div style="background-color: black; color: green; padding: 2px; text-align: center;"><b>The JavaScript Event Loop</b></div> <div style="background-color: #f0f0f0; padding: 2px; text-align: center; font-size: small;">Event Loop, Callbacks, Promises, and Async/Await</div> <div style="text-align: center; padding: 10px;"> <p><b>Output appears here</b></p> <p>And I am second! I was first...</p> </div>
---	--

4. 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:

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

5. 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.*

6. 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.*

7. 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 JavaScript Event Loop

Event Loop, Callbacks, Promises, and Async/Await

### Output appears here

And I am second!  
I was first...

*The output is still the same and we still have not yet solved the problem.*



8. Lets continue by returning the promise, pf:

<pre>const doFirst = function() {   const pf = new Promise((resolve, reject)=&gt;{     setTimeout(function(){       resolve(el.innerHTML = el.innerHTML + "&lt;br /&gt;I was first...")     }, 2000);   });   return pf; };</pre>	<div style="background-color: black; color: green; padding: 2px;"><b>The JavaScript Event Loop</b></div> <div style="background-color: #f0f0f0; padding: 2px;">Event Loop, Callbacks, Promises, and Async/Await</div> <p><b>Output appears here</b></p> <p>And I am second! I was first...</p> <p><i>Still no changes in output.</i></p>
---	--

9. 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:

<pre>let el = document.getElementById("response"); const doFirst = function() {   setTimeout(function(){     el.innerHTML = el.innerHTML + "&lt;br /&gt;I was first..."   }, 2000); }; doFirst().then(); ...</pre>	<div style="background-color: black; color: green; padding: 2px;"><b>The JavaScript Event Loop</b></div> <div style="background-color: #f0f0f0; padding: 2px;">Event Loop, Callbacks, Promises, and Async/Await</div> <p><b>Output appears here</b></p> <p>And I am second! I was first...</p>
--	--

10. 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.

<pre>doFirst().then( () =&gt; { } ); ...</pre>	<div style="background-color: black; color: green; padding: 2px;"><b>The JavaScript Event Loop</b></div> <div style="background-color: #f0f0f0; padding: 2px;">Event Loop, Callbacks, Promises, and Async/Await</div> <p><b>Output appears here</b></p> <p>And I am second! I was first...</p> <p><i>This wont do anything to the output but it will give us the architectural structure in which to change the output.</i></p>
--	---

11. With this structural 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!"

<pre>doFirst().then(()=&gt;{   el.innerHTML = el.innerHTML + "&lt;br /&gt;And I am second!"; });</pre>	<div data-bbox="906 279 1273 310" style="background-color: black; color: green; padding: 2px;"><b>The JavaScript Event Loop</b></div> <div data-bbox="906 310 1273 331" style="background-color: #f0f0f0; padding: 2px;">Event Loop, Callbacks, Promises, and Async/Await</div> <div data-bbox="938 346 1149 373" style="background-color: #f0f0f0; padding: 2px;"><b>Output appears here</b></div> <div data-bbox="938 384 1062 422" style="background-color: #f0f0f0; padding: 2px;">I was first... And I am second!</div> <div data-bbox="906 457 1273 485" style="background-color: #f0f0f0; padding: 2px;"><i>Finally we see some change</i></div>
--	---

12. Usually though, you would return data from the promise then do whatever you want with that data

```
const doFirst = function() {
  const pf = new Promise((resolve, reject)=>{
    setTimeout(function(){
      resolve("I was first...")
    }, 2000);
  });
  return pf;
};
doFirst().then((data)=>{
  el.innerHTML = el.innerHTML + data;
});
```

13. With this structural change we simply add on the rest of the data that we wish to be part of this promise architecture:

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

14. (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);
  }); ...
}
```

15. Here is the final code for this section:

```
let el = document.getElementById("response");
const doFirst = function() {
  return new Promise((resolve, reject)=>{
    setTimeout(()=>{
      resolve("<br />I was first...");
    }, 500);
  });
  //return pf;
};
doFirst().then((data)=>{
  el.innerHTML = data + "<br />And I am second!"
});
```

16. Here is the final code for this section, **using regular functions**:

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

*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 `AJS_Day01_Part3_Ending.zip`. It contains the code in a **completed** state, so if you wanted to follow along, just continue using the ending files from Part2.

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);
  });
};
```

2. 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.

3. 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();
}
```

4. 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();
}
```

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

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

6. Nothing happens as yet because we have not called `showMessages()`, so let's 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?

7. 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();
```

8. 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();
```

9. Using arrow functions:

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



## Part 4 – Error Handling

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

The code for this part is called **AJS\_Day01\_Part4\_A.zip**. It contains the code in a **completed** state, so if you wanted to, you can follow along using the code at the end of Part3. Note, there are two sets of code here, one where I used the **Async/Await** construct, that is the A part. The second zipped file, the B, part follows error handling using the **then().catch()** construct.

1. 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.

2. 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 occurred");
    }, 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();
```

### The JavaScript Event Loop

Event Loop, Callbacks, Promises, and Async/Await

#### Output appears here

An Error Occured!

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

3. 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

4. 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.

5. 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 occurred!");
    el.innerHTML = "An error occurred!"
  });
```

This is called chaining. The *catch()* method is chained to the first *then()* method which is chained to the Promise call. In this code I am NOT capturing the returned error message, but I do so in #6. You could also use *console.error(err)* which will provide more information.

6. 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;
  });
```

7. 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.

8. 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.



9. 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";
  }
);
```

### The JavaScript Event Loop

Event Loop, Callbacks, Promises, and Async/Await

#### Output appears here

I was first...  
And I am second!

*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 "AJS\_Day01\_Part5\_Starter.zip"..

1. Once you get the starter code loaded, press the "Get Data" 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');
};
```

2. 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();
};
```

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

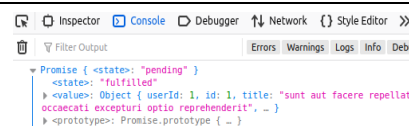
```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then( (data) => {
    console.log(data)
  } );
```



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

4. 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('https://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

5. 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()
  } );
```

6. 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(() => {
  })
})
```

7. 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);
})
```



The screenshot shows the browser's developer console with the 'Console' tab selected. It displays the JSON object returned by the previous `fetch` request: `{userId: 1, id: 1, title: 'sunt aut facere repellat provident...', body: 'quia et suscipit...'}`.

8. 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;
})
```



The screenshot shows a web application titled 'The JavaScript Event Loop'. It has a button labeled 'Get Data'. Below the button, the text 'Output appears here' is followed by the title of the first post: 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit'.

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.

9. 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;
})
```



The screenshot shows the same web application as before, but now it displays a network error: 'TypeError: NetworkError when attempting to fetch resource.' This error occurred because the URL was changed to something that does not exist.

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

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

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

11. 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');
};

```

12. 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');
```

13. 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();
};

```

14. 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>
```

15. 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);
};
```



16. 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;
};
```

17. Lets go further and add in error handling:

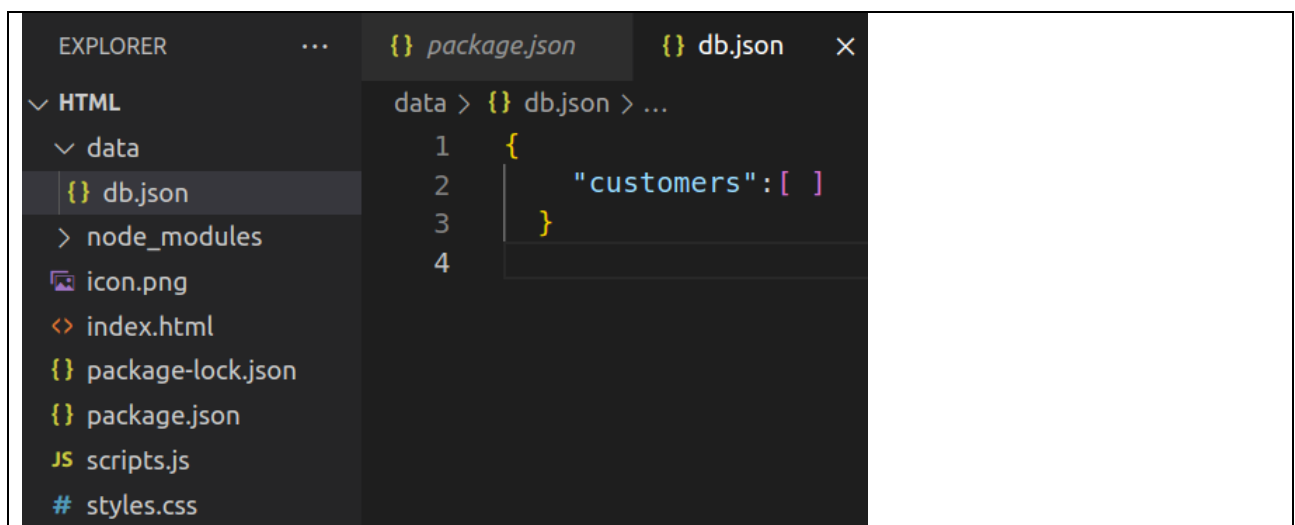
```
let el = document.getElementById("response");
//
async function getData(){
  try{
    const postData = await
    fetch('https://jsonplaceholder.typicode.com/posts/1');
    const finalData = await postData.json();
    el.innerHTML = finalData.title;
  }catch(err){
    el.innerHTML = "Oops " + err;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

*End of section*

## Part 6 – JSON Server

Lets configure a server so that we can make a *post* request to the server and test out the Promise in that situation. You may continue with the same project from Part5 or start a new one. For this part, you will need to be in a NodeJS environment. If you do not have access to this, just follow along. You can use `AJD_Day01_Part5_Ending` for this section. We wont be using the HTML file too much for the next two parts.

1. Install the JSON Server using the terminal window and the command: `npm install json-server`. If you started working with the HTML folder, then navigate to that folder before you install this code. Folders and paths are important for Node.
2. This server will need a database file to work with, lets simulate a database with just a plain text file holding JSON data in it. Back in the editor, create a new folder under the app folder called data. Inside of the data folder create a new text file called db.json. Alternatively, you can use the `db.json` file you downloaded, just make the data folder first.



Note: after you install the `json-server` using `npm`, you will see a few new folders and files, we can ignore most of this for now. There is also a zipped file called [data.zip](#) that you can just unzip and place in the appropriate directory.

3. From #2 above noticed that I started the `.json` file, it is configured with a pair of curly braces and an array inside of those braces. We can now define what our customer should look like:

```
{
  "employees": [
    {
      "username": "Axle",
      "password": "1234"
    }
  ]
}
```

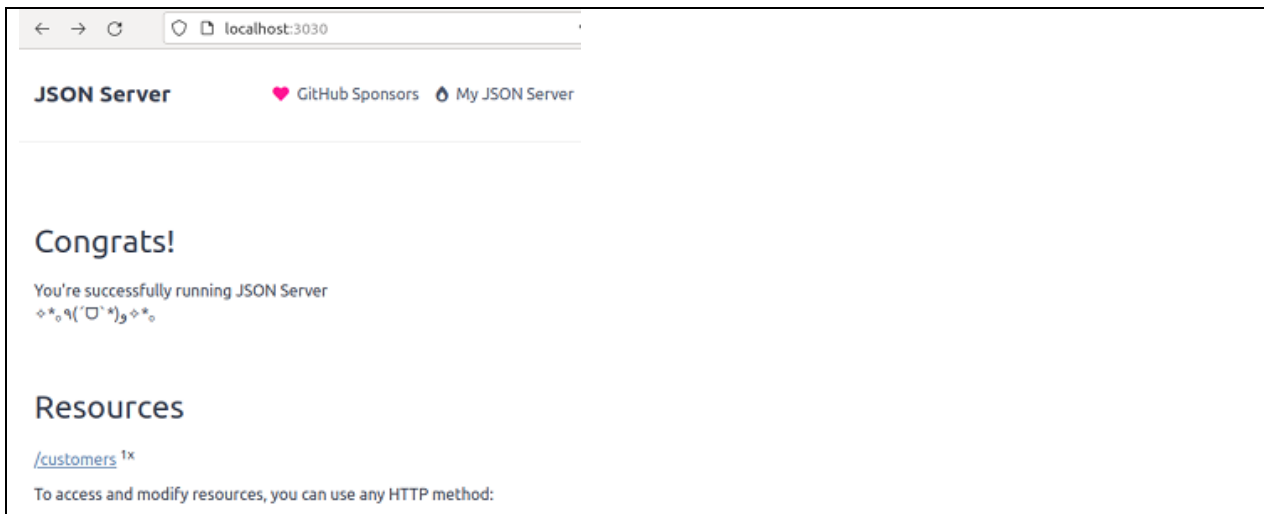
So now we have one customer, the Axle customer

4. Lets now start the json-server and point it to our employees database:

```
npx json-server --watch data/db.json --port 3030
```

Run this code from your terminal window, check that the path is correct. Although you can run this code from VS Code, it is not recommended.

5. If you do not get an error running this code, open a browser and navigate to the localhost address and add port 3030 at the end:



6. If you click on the employees link, it should show you the one customer we have in the database. The server and mock database is now setup for use.
7. Lets now reconfigure our code to hit the json-server instead of the public json placeholder website. If you already have the code from #17 part 5, then simply change the url:

```
async function getData() {  
  try {  
    const postData = fetch('http://localhost:3030/employees');  
    const finalData = await postData.json();  
    el.innerHTML = finalData[0].username;  
  } catch (err) {  
    el.innerHTML = "Oops " + err;  
  }  
};
```

Notice that *finalData* is an array, so we just interrogate the first element.

Special note about the zipped code for Part 6 ending. The code for this part has to be *installed*. That means you must unzip the files into a folder. Using a terminal window or if you open the folder as a project in VSCode, you must run the following command: `npm install`

That command will bring the folder and the project back to a runnable state. This means that you will see a `node_modules` folder and a `package_lock.json` file appear in the project folder. This is normal.



## Part 7 – Post Requests

The `fetch()` method also can be configured to make POST requests. The json-server that we just implemented into our project has the capability to accept data from the browser. We would need to configure the server itself as well as the code that hits that server with loaded data. Note, a POST request is also an asynchronous operation.

1. The json-server in its current configuration is not designed to accept data, we need to add a unique field. It's a simple task, just add an *id* field:

```
{
  "id": "1",
  "username": "Axle",
  "password": "1234"
}
```

Remember to insert a coma after the *id* field.

2. Now we need to configure the `fetch()` method. We do this by adding an object as the second parameter:

```
try{
  const postData = await fetch(
    'http://localhost:3030/employees',
    {
      method: "POST"
    }
  );
  const data = await postData.json();
  console.log(data);
}
```

So, the `fetch()` method, takes two arguments inside of the parentheses.

3. Now all we have to do is supply the data that will be posted. Of course this should match what we already have:

```
const postData = await fetch(
  'http://localhost:3030/customers',
  {
    method: "POST",
    body: {"username": "Jennifer", "password": "1234"}
  }
)
```

Notice that the data goes as part of the *body* property. However this still wont work completely.

4. It is better to wrap the data using a `JSON.stringify()` method:

```
const postData = await fetch(  
  'http://localhost:3030/customers',  
  {  
    method: "POST",  
    body: JSON.stringify({ "username": "Jennifer", "password": "1234" })  
  }  
)
```

This will ensure that the data reaches the server in the format it requires.

5. The above code did not throw any errors but it did not work properly. We now have to inform the server about the type of data we are supplying it with:

```
{  
  method: "POST",  
  headers: {  
    'Content-Type': 'application/json',  
  },  
  body: JSON.stringify({ "username": "Jennifer", "password": "1234" })  
}
```

This code should now post *Jennifer* and *1234* properly

6. The json-server does not respond with a *created* message. This has to be handled manually. Remember the server in this case is the Json Server. Also, I changed the button on the HTML file to read Post Data instead of Get Data.

Special note about the zipped code for Part 7 ending. The code for this part has to be *installed*. That means you must unzip the files into a folder. Using a terminal window or if you open the folder as a project in VSCode, you must run the following command: `npm install`

That command will bring the folder and the project back to a runnable state. This means that you will see a `node_modules` folder and a `package_lock.json` file appear in the project folder. This is normal.

## 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.

Promises require handler functions and it is suggested that you **attach** these functions 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

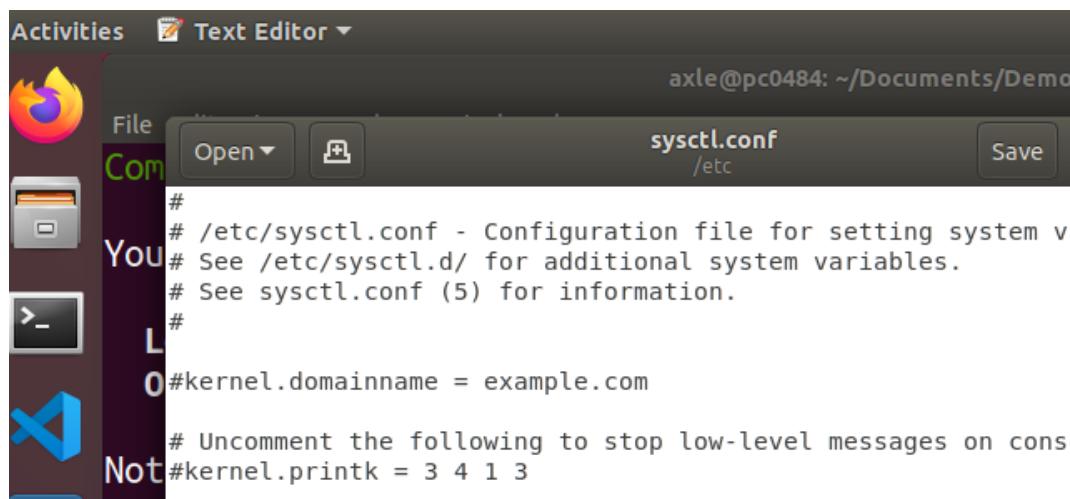
1. HTML – any modern browser will work
2. JavaScript – any modern browser will work
3. Code Editor – instructor will use Visual Studio Code (free version)
4. NodeJS – environment to run JavaScript code

## 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 – Working with the Zipped Files

To use any of the zipped files involving NodeJS, you will need to unzip those files. The code that I show in the bootcamp will be different from what the unzipped folder contains. 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. Navigate 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.