JavaScript Async/Await

JavaScript is always synchronous and single-threaded that provides the event loops. The event loops enable us to queue up an activity. This activity will not happen until the loops become available after the program that queued the action has completed the execution. However, our program contains a large number of functionalities, which causes our code to be asynchronous. The **Async/Await** functionality is one of them. **Async/Await** is an extension of **promises** that we get as language support.

In this article, we are going to discuss the JavaScript Async/Await with some examples.

JavaScript Async

An async function is a function that is declared with the async keyword and allows the await keyword inside it. The async and await keywords allow asynchronous, **promise-based** behavior to be written more easily and avoid configured promise chains. The async keyword may be used with any of the methods for creating a function.

Syntax:

The syntax of JavaScript may be defined as:

```
Async function myfirstfunction() {
return "Hello World"
}
```

It is the same as:

```
async function myfirstfunction() {
return Promise.resolve("Hello World");
}
```

Example:

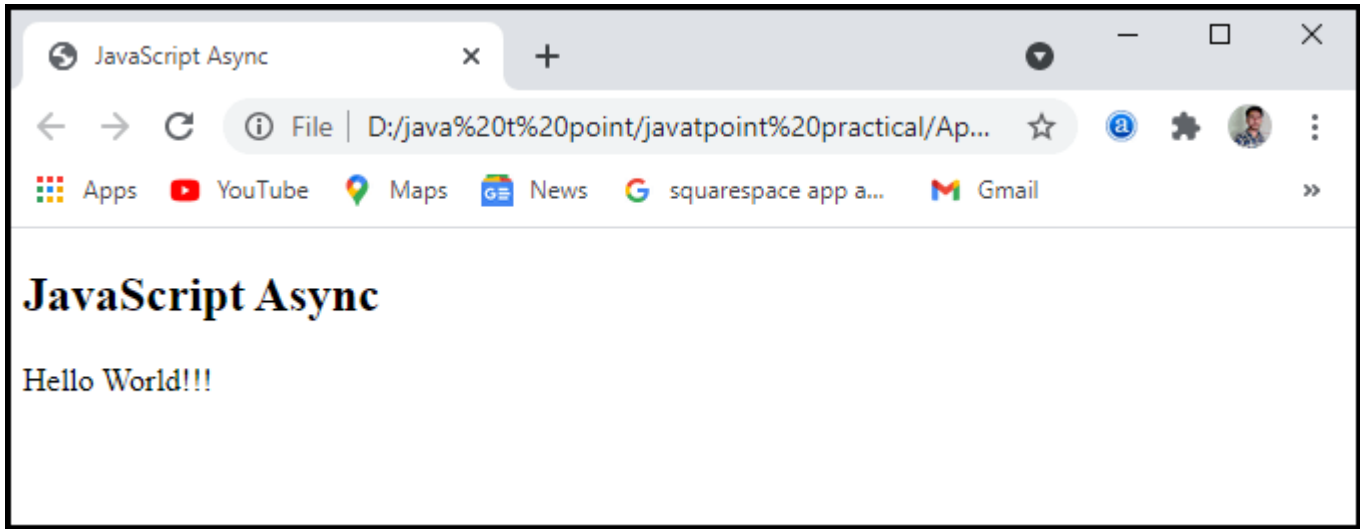
Let's take an example to understand how we can use a JavaScript Async in the program.

⇧ SCROLL TO TOP

```
<html>
<head>
   <meta charset="utf-8">
<title>JavaScript Async</title>
</head>
<body>
   <h2>JavaScript Async</h2>
   <p id="main"></p>
<script>
function myDisplayer(some) {
  document.getElementById("main").innerHTML = some;
}
async function myfirstFunction() {
    return "Hello World!!!";
    }
myfirstFunction().then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
</script>
</body>
</html>
```

Output: After executing the above code, we will get the output shown below in the screenshot.

⇧ SCROLL TO TOP



JavaScript Await

JavaScript Await function is used to wait for the promise. It could only be used inside the async block. It instructs the code to wait until the promise returns a response. It only delays the async block. Await is a simple command that instructs JavaScript to wait for an asynchronous action to complete before continuing with the feature. It's similar to a **"pause until done"** keyword. The await keyword is used to retrieve a value from a function where we will usually be used the **then()** function. Instead of calling after the asynchronous function, we'd use await to allocate a variable to the result and then use the result in the code as we will in the synchronous code.

Syntax:

The syntax of JavaScript Await function may be defined as:

```
// Await function works only inside the async function
let value = await promise;
```

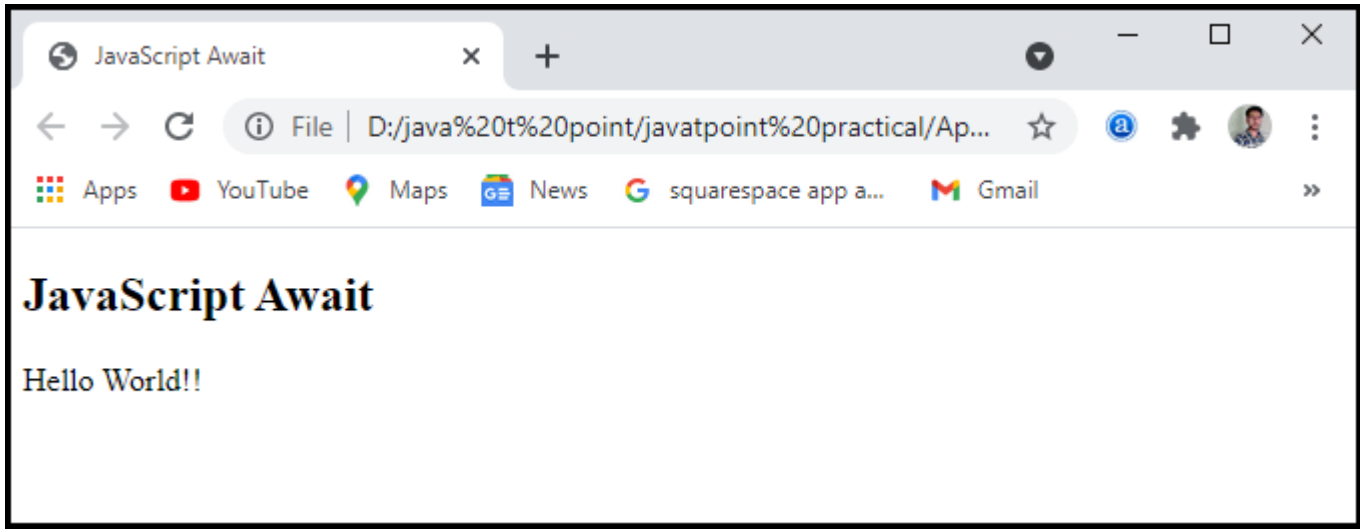
Example:

Let's take an example to understand how we can use the JavaScript Await function in the program.

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8">
<title>JavaScript Await</title>
</head>
<body>
   <h2>JavaScript Await</h2>
   <p id="main"></p>
<script>
async function myDisplay() {
  let myPromise = new Promise(function(myResolve, myReject) {
    myResolve("Hello World!!");
  });
  document.getElementById("main").innerHTML = await myPromise;
}
myDisplay();
</script>
</body>
</html>
```

⇧ SCROLL TO TOP

ing this code, we will get the output as shown below in the screenshot:



Example 2: Waiting for a timeout in the program

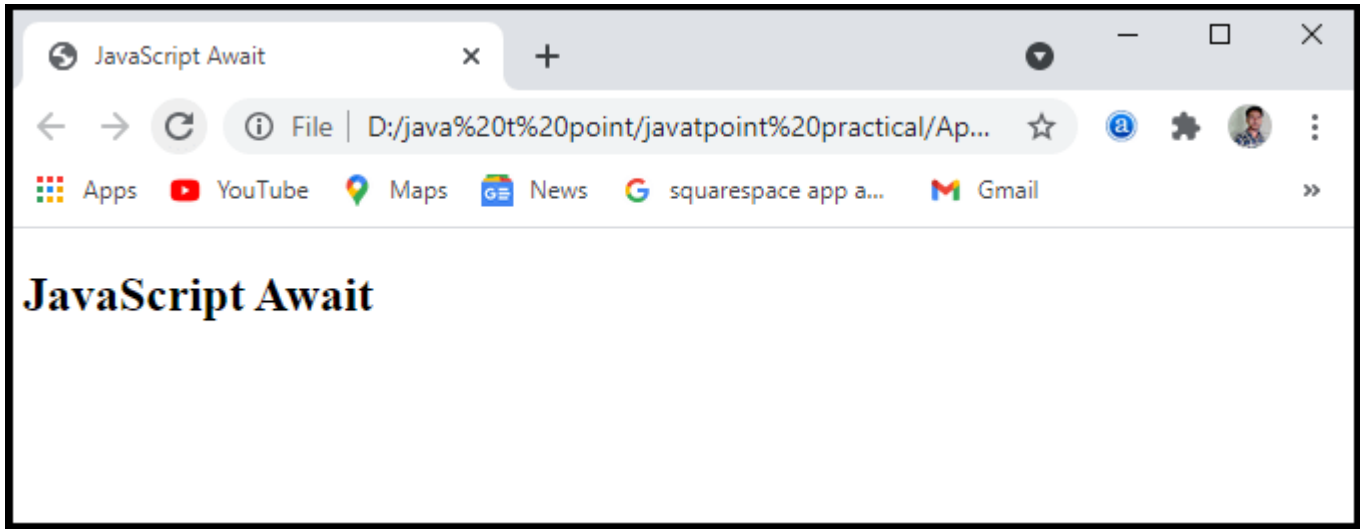
Let's take a program to understand the JavaScript Await using timeout waiting function.

```
<!DOCTYPE html>
```

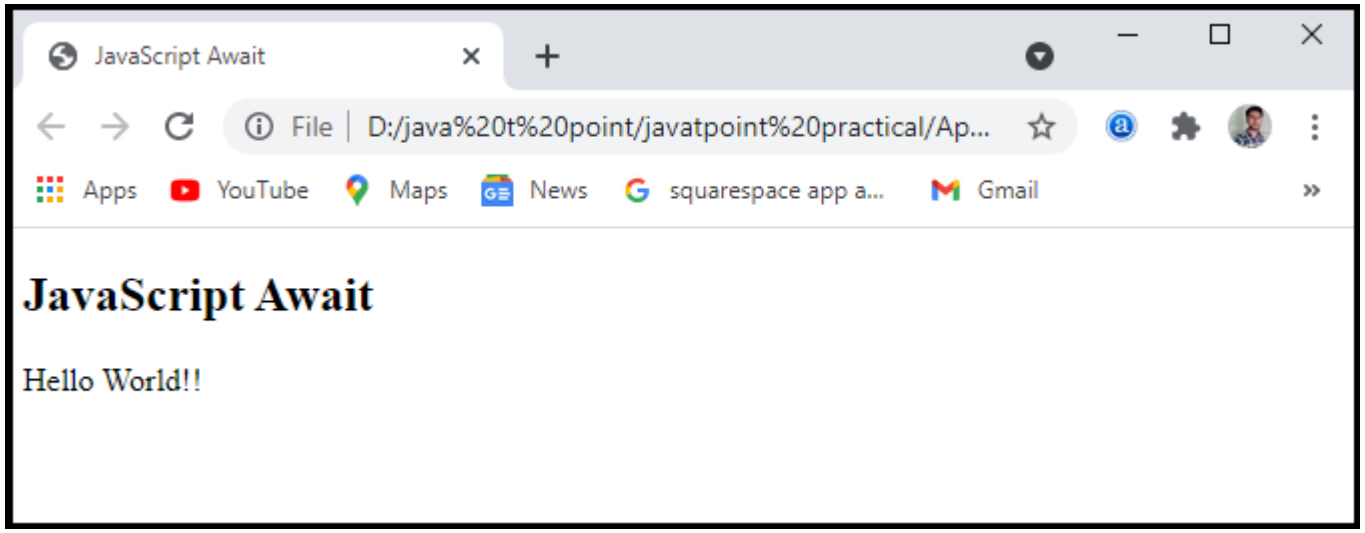
⇧ SCROLL TO TOP

```
<head>
   <meta charset="utf-8">
<title>JavaScript Await</title>
</head>
<body>
   <h2>JavaScript Await</h2>
   <p id="main"></p>
<script>
async function myDisplay() {
  let myPromise = new Promise(function(myResolve, myReject) {
    setTimeout(function() { myResolve("Hello World!!"); }, 2000);
  });
  document.getElementById("main").innerHTML = await myPromise;
}
myDisplay();
</script>
</body>
</html>
```

Output: After executing this code, we will get the output as shown below in the screenshot.



When we execute the code, it will show the result after **2** seconds. It uses the timeout function.



Error Handling

⇧ SCROLL TO TOP

It is very easy to handle errors in async functions. Promises have a **catch()** method for dealing with rejected promises, and because the async functions only return a promise, we may call the function and add a method to the end. We should use the promise's capture in the same way as we would any other catch. And all are easy to grasp. Remember that a then callback will fail. It can generate an error (with an explicit throw or by trying to access a property of a null variable). These crashes would also be caught by the grab process. Remind yourself that the promise's capture approach is similar to a standard catch.

Syntax:

The syntax of error handling may be defined as:

```
asyncFunc().catch(err =>
{
Console.error(err)
// catch error and do something
});
```

But there is another option: the all-powerful **try/catch block.** If we want to handle errors directly inside the async function, we may use try/catch in the same way we would in synchronous code.

Example:

Let's take an example to understand the error handling in the JavaScript Async and Await function.

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8">
<title>JavaScript Await</title>
</head>
<body>
   <h2>JavaScript Await</h2>
<script>
async function f() {
```

```
  let response = await fetch('http://no-url');
} catch(err) {
  alert(err); // TypeError: failed to fetch
}
}
f();
</script>
</body>
</html>
```

Output: After executing this code, we will get the output as shown below in the screenshot.

It may appear sloppy, but it is a very simple way to handle errors without appending. After the function calls, use the **catch()** method. It is up to us how we manage mistakes, and which approach we use should be dictated by how our code was written. Over time, we'll get a sense of what needs to be achieved.

⇧ SCROLL TO TOP