

## Free, Secure and Trusted Way to Authenticate Your Visitors

**FREE SIGN UP**

Add login to your website in **5 minutes** completely **for free!**

No hidden costs. No credit card needed.



## Callback vs Promises vs Async Await

This blog explains the fundamental concepts that JavaScript relies on to handle asynchronous operations. These concepts include **Callback functions**, **Promises** and the use of **Async**, and **Await** to handle deferred operations in JavaScript.

Before we decode the comparison between the three, let's get a brief understanding of synchronous (blocking) and asynchronous (non-blocking).

## Difference Between Sync and Async

To make it easy to understand, let's take a live example which probably will explain the difference between **asynchronous** and synchronous.

Imagine we go to a restaurant, a waiter comes to a table, takes your order and gives it to the kitchen. Then they move on to the server on another table, while the chef is preparing the meal. So the same person can serve the many different tables. The table has to wait for the chef to cook one meal before they serve another table. This is what we called asynchronous or non-blocking architecture. Here the waiter is like a thread allocated to handle requests. So a single thread is used to handle multiple requests.

In contrast to non-blocking or asynchronous architecture, we have blocking or synchronous architecture. Let's see how that works. So back to the restaurant example, imagine you go to another restaurant and in this restaurant, a waiter is allocated to you. He takes your order and gives it to the kitchen. Now he is sitting in the kitchen waiting for the chef to prepare your meal and this time he is not doing anything else he is just waiting for he is not going to take any order from another table until your meal is ready. This is what we called synchronous or blocking architecture.

Now, the first restaurant example represents an asynchronous process because you did not have to wait, the waiter takes the order from one table and goes to the next table to take the order. While the second example restaurant represents a synchronous operation because you had to wait until the resource (waiter in this case) can proceed with you. This is the single, most fundamental difference between sync and async processes.

The important thing to keep in mind is that the single-threaded event handling systems are usually implemented using an event or message queue. So when a program is being executed synchronously, the thread will wait until the first statement is finished to jump to the second one, while in asynchronous execution, even if the first one was not completed, the execution will continue.


There are different ways to handle the async code. Those are callbacks, promises, and async/await.

## Callbacks:

In JavaScript, functions are objects. So we can pass objects to functions as parameters.

We can also pass functions as parameters to other functions and call them inside the outer functions. So callback is a function that is passed to another function. When the first function is done, it will run the second function.

Let's take an example of callback function:



```
function printString(){  
  console.log("Tom");  
  setTimeout(function() { console.log("Jacob"); }, 300);  
  console.log("Mark")  
}  
  
printString();
```

That were sync code, we would have encountered the following output.

```
Tom  
Jacob  
Mark
```

But the setTimeout is an async function then the output of the above code will be:

```
Tom  
Mark  
Jacob
```

There is a built-in method in JavaScript called "setTimeout", which calls a function or evaluates an expression after a given period of time (in milliseconds).

In other words, the message function is being called after something happened (after 3 seconds passed for this example), but not before. So the callback is the function that is passed as the argument to `setTimeout`.

## Callback as an Arrow Function:

If you prefer, you can also write the above same callback function as an ES6 arrow function, which is a newer type of function in JavaScript:



```
function printString(){
  console.log("Tom");
  setTimeout(() => { console.log("Jacob"); }, 300);
  console.log("Mark")
}

printString();
```

The output will be the same as above.


The problem with callbacks is it creates something called "Callback Hell." Basically, you start nesting functions within functions within functions, and it starts to get really hard to read the code. So in this situation Promises came to handle the nested callback in a better way.

## Promises:

A promise in JavaScript is similar to a promise in real life. When we make a promise in real life, it is a guarantee that we are going to do something in the future. Because promises can only be made for the future.

A promise has two possible outcomes: it will either be kept when the time comes, or it won't.

This is also the same for promises in JavaScript. When we define a promise in JavaScript, it will be resolved when the time comes, or it will get rejected. It sounds like the IF condition. But there are huge differences between them.



A promise is used to handle the asynchronous result of an operation. JavaScript is designed to not wait for an asynchronous block of code to completely execute before other synchronous parts of the code can run. With Promises, we can defer the execution of a code block until an async request is completed. This way, other operations can keep running without interruption.

### States of Promises:

First of all, a Promise is an object. There are 3 states of the Promise object:

- Pending: Initial State, before the Promise succeeds or fails.
- Resolved: Completed Promise
- Rejected: Failed Promise, throw an error

For example, when we request data from the server by using a Promise, it will be in pending mode until we receive our data.

If we achieve to get the information from the server, the Promise will be resolved successfully. But if we don't get the information, then the Promise will be in the rejected state.

### Creating a Promise:

Firstly, we use a constructor to create a Promise object. The promise has two parameters, one for success (resolve) and one for fail (reject):

```
const myPromise = new Promise((resolve, reject) => {  
  // condition  
});
```

Let's create a promise:

```
const myFirstPromise = new Promise((resolve, reject) => {  
  const condition = true;  
  if(condition) {  
    setTimeout(function(){  
      resolve("Promise is resolved!"); // fulfilled  
    }, 300);  
  } else {  
    reject('Promise is rejected!');  
  }  
});
```

In the above Promise If Condition is true, resolve the promise returning the "Promise is resolved", else return an error "Promise is rejected". Now we have created our first Promise, Now let's use it.

## Using Promise:

To use the above create Promise we use `then()` for resolve and `catch()` for reject.

```
myFirstPromise
  .then((successMsg) => {
    console.log(successMsg);
  })
  .catch((errorMsg) => {
    console.log(errorMsg);
  });
```



Let's take this a step further:

```
const demoPromise= function() {
  myFirstPromise
    .then((successMsg) => {
      console.log("Success:" + successMsg);
    })
    .catch((errorMsg) => {
      console.log("Error:" + errorMsg);
    })
}

demoPromise();
```

In our created promise condition is "true" and we call demoPromise() then our console logs read:

```
Success: Promise is resolved!
```

So if the promise gets rejected, it will jump to the catch() method and this time we will see a different message on the console.

Error: Promise is rejected!

## What is Chaining?

Sometimes we need to call multiple asynchronous requests, then after the first promise is resolved (or rejected), a new process will start to which we can attach it directly by a method called chaining.

we create another promise:

```
const helloPromise = function() {  
  return new Promise(function(resolve, reject) {  
    const message = `Hi, How are you!`;   
  
    resolve(message)  
  });  
}
```

We chain this promise to our earlier “myFirstPromise” operation like so:



```
const demoPromise= function() {  
  
  myFirstPromise  
    .then(helloPromise)  
    .then((successMsg) => {  
      console.log("Success:" + successMsg);  
    })  
    .catch((errorMsg) => {  
      console.log("Error:" + errorMsg);  
    })  
  }  
  demoPromise();  
}
```

Once our condition is true, the output to our console is:

```
Hi, How are you!
```

Once the "hello" promise is chained with **.then**, subsequent **.then** utilizes data from the previous one.

## Async/Await:

Await is basically syntactic sugar for Promises. It makes your asynchronous code look more like synchronous/procedural code, which is easier for humans to understand.

Syntax of Async and Await:

```
async function printMyAsync(){  
  await printString("one")  
  await printString("two")  
  await printString("three")  
}
```



You can see that we use the "async" keyword for the wrapper function printMyAsync. This lets JavaScript know that we are using async/await syntax, and is necessary if you want to use Await. This means you can't use Await at the global level. It always needs a wrapper function. Or we can say await is only used with an async function.

The await keyword is used in an async function to ensure that all promises returned in the async function are synchronized, ie. they wait for each other. Await eliminates the use of callbacks in .then() and .catch(). In using async and await, async is prepended when returning a promise, await is prepended when calling a promise. try and catch are also used to get the rejection value of an async function.

Let's take an example to understand the Async and Await with our demoPromise:

```
async function demoPromise() {  
  try {  
    let message = await myFirstPromise;  
    let message = await helloPromise();  
    console.log(message);  
  
  } catch((error) => {  
    console.log("Error:" + error.message);  
  })  
}  
  
// finally, call our async function  
  
(async () => {  
  await myDate();  
})();
```

## Conclusion:

In this tutorial, we understand the concept of the callback, Promise, and Async/Await. We know how they can work with javascript asynchronous requests. Mainly they used API Request and event handling.

## Follow LoginRadius

 via feedly

 on twitter

## LoginRadius Docs