AppDividend

Home › Javascript › ECMA Script 6 ›

ECMA SCRIPT 6

Promises In ES6

By **Krunal** Last updated **Aug 15, 2017**

By clicking the subscribe button you will never miss the new articles!

Subscribe

Hello, web programmers, In today's **AppDividend** Tutorial, I have briefly described **Promises in ES6.**

Promises have arrived natively in **ES6.** It is also available in 3rd party libraries like **fetch** which is used in **HTTP calls** or **AJAX calls,** and also **jQuery 3.0** now supports **Promises.**

Content Overview [hide]

What is Promise

As its name suggests, It will return a **promise** as an **object** of any particular asynchronous actions. Promises are results of eventual operations, and it is **Object.** This object has **three** states.

1. **Pending State**
2. **Fulfilled State**
3. **Rejected State**

I will explain you one by one state but first, let me tell you its syntax

Syntax

```
let promise = new Promise((resolve, reject) =>  {
        // here async actions are performed
        resolve('your result');
});

// in case of success
promise.then((resolve) => console.log(resolve));

//in case of fail or an error
promise.catch((reject) => console.log(reject));
```

In promise, we have to pass two arguments to its constructor

1. **resolve**
2. **reject**

At this point, we do not know whether the promise will resolve or reject because in asynchronous calls will take some milliseconds to execute the promises. Asynchronous calls are simply AJAX calls, which will take some time to execute and will get a response from the server. So by that time, the state will be **pending.**

Pending State

While the Asynchronous task is performed, the promise object is in the holding state which does not contain any value. Instead, it will return a promise that any point of time in future it will get either resolve or reject with any error. **This state is called Pending state**

Resolve or Fulfilled State

If the operation of the Asynchronous task is successfully completed, then It will be resolved, and you can get its value by calling **then** function on promise object.

```
let promise = new Promise((resolve, reject) =>  {
        resolve('hello world');
});
promise.then((resolve) => console.log(resolve));
```

In above example, I am not performing any asynchronous call just show you a syntaxial example.

Rejected State

If the operation of the Asynchronous task is not completed successfully and an error occurred or thrown, then It will be dismissed, and you can get error value by calling **catch** function on promise object.

```
let promise = new Promise((resolve, reject) =>  {
        reject('an error occured');
});
promise.then((reject) => console.log(reject));
```

Example #1

```
//main.js

let promise = new Promise((resolve, reject) =>  {
        setTimeout(() => {
            resolve('Promises are working')
        }, 2000);
});
promise.then((data) => {
    console.log(data);
});
```

If you directly run above code in the browser, then you might face the following issue.

Possible Errors

1. **You can get any syntax error.**

2. **If you perform code directly in your browser, then chances are very high to fail the webpack compilation process.**

Possible Solutions

1. **Beginner's Guide To Setup ES6 Development Environment** Follow this article strictly and put above code in the **main.js** file.

Output

```
Promises are working //after 2 seconds
```

Here in above example, we have use **setTimeout** function to delay in output so that it will behave as an **asynchronous task**. After 2 seconds the promise will resolve and execute the **then** callback function which is accessible on the promise object. This callback has the data from returned from the **Asynchronous** Task. So, it is passed as an argument, and then we have the log that value and output will be above statement.

```
let promise = new Promise((resolve, reject) =>  {
        setTimeout(() => {
            reject('Aww there is something wrong')
        }, 2000);
});
promise.catch((error) => {
    console.log(error);
});
```

In this scenario, an **Asynchronous** task will give an error, and we can call **catch** method on promise object and take an argument as an error and log that error.

Output

```
Aww there is something wrong //after 2 seconds
```

Example #2

```
//main.js

let prom = () => {
    return new Promise((resolve, reject) =>  {
        setTimeout(() => {
            resolve(3);
            if(1){
                reject('error');
            }
        }, 1000);
    });
}
prom().catch(error => console.log(error));
prom().then(data =>console.log(data));
```

Output

```
3 //after 1 second
```

Here, first, we create a function that returns a promise and then depending on success or failure, we have called **then()** and **catch()** (methods respectively. If you close look at the example, whichever event occurs first like if the **resolve()** is called first then respectively **then()** method is called and if **reject()** method is called then respectively **catch()** method will be executed.

Callback Hell

Promises and **Callbacks** are pretty much the same but differ in one term called "**Callback Hell**." If you compose more than two **callbacks**, then things get complicated, and you will end up with mess code. **Promises** are composable, so we can create multiple **promises** and pass it to only one function, and it will get either resolve or reject one by one and also it removes the **callback hell** problem.

Let's take an example of what is called a callback hell.

```
//main.js

let mj = 'KingOfPop';
let prom = (data, callbackFunction) => {
    callbackFunction(null,data);
}
prom(mj, (error, data) => {
    console.log(data);
    prom(mj, (error, data) => {
        console.log(data);
        prom(mj, (error, data) => {
            console.log(data);
        });
    });
});
```

Output

```
KingOfPop
KingOfPop
KingOfPop
```

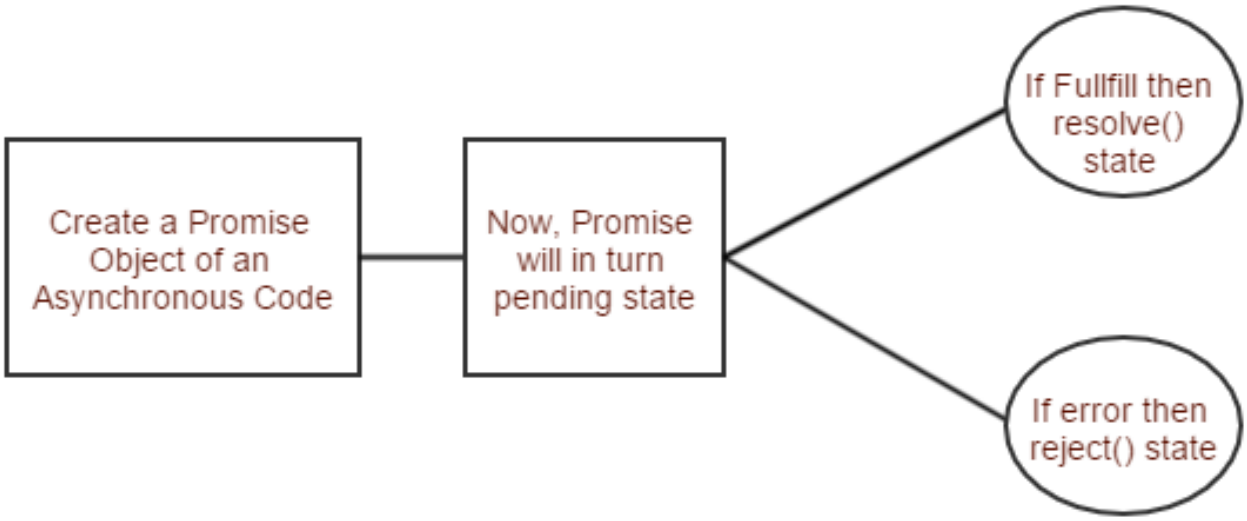
In above example, I have used **callbacks** instead of **Promises. So** the callback hell is created because first, we call prom function and then again when it completes, we called another and then another, so **pyramid of doom** is created which is also called **callback hell.** We can also get the same output using **Promises.**

```
//main.js

let prom = (data) => {
    return new Promise((resolve, reject) => {
        if(data){
            resolve(data)
        }
        reject('An Error occured');
    });
}
Promise.all([
    prom('KingOfPop'),
    prom('KingOfPop'),
    prom('KingOfPop')
]). then(data => data.forEach(d => console.log(d)));
```

```
KingofPop
KingOfPop
KingofPop
```

So, basically, we can compose multiple promises and then it will **resolve** or **reject** depending on the result. It is very cool. In above examples, I have not used **Asynchronous Code** because I am leaving it to you to perform faking ajax request or use timeout function on your own and see the magic.



Promises In ES6

Memorable Points

1. **Promises** are a way of saying that till now I have not any data while in a pending state, but in future, It will surely either resolve or reject the data depending on the result.
2. We can use multiple promises and compose it in a way that we can remove a **callback hell** problem.

If you still have doubt then ask in the comment below, I am happy to help you out.

ES6 pending state promises reject resolve

Krunal

I am Web Developer and Blogger. I have created this website for the web developers to understand complex concepts in an easy manner.