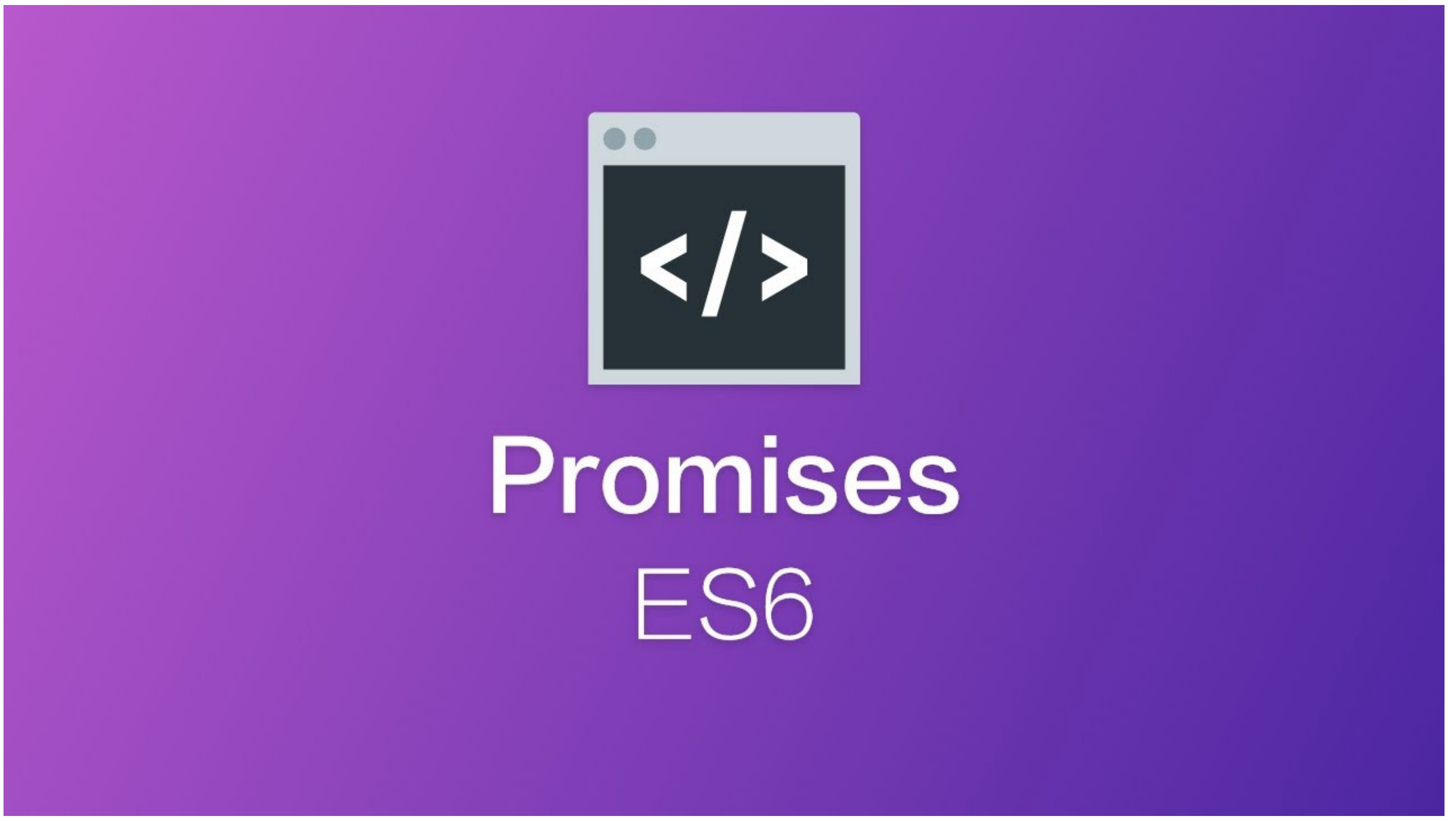
ES6 for beginners Part-2

Srebalaji Thirumalai Follow
Oct 14, 2017 · 7 min read



Part one of this article appeared here. I covered some interesting features there. :)

Topics I'm gonna cover in this post

1. Promises
2. Async / Await

Promises

Promises are one of the useful features in ES6. They are used to make **async** operations such as API request, file handling, downloading images, etc.

So, what is async ? (hold on if you already know)

Async operations are operations which take some time to complete.

For example, lets say you are defining a function that makes an API request to the server. That function doesn't return the output immediately. It takes few secs to get the response from the server.

So if you are calling that function and assigning it's value (i.e) output to some variable, it will be **undefined.** Because Javascript doesn't know that the function is handling some async operations.

So how do we handle it ?

Before that let's talk some history.

Before promises, programmers used to define **callbacks.** Callbacks are normal functions in Javascript which executes when the async operation is complete.

For example, you define a function which makes an API request to the server. And then you mention a callback function which will be executed when we get the response from the server.

So in the above case, Javascript doesn't halt the execution till we get the response from the API. And we have defined a function (callback) which will be executed after we get the response. I think you got it.

So, what are promises ?

Promises are objects that helps to do async operations.

Technically, they are objects that represents the completion of the async operation. (If you don't get it stay with me for a while.)

Before explaining how to define a promise I will explain the life cycle of the promise.

We have three states in promises

1. **Pending**: In this state the promise is just executing the async operation. For example, It's making some API request to the server or downloading some images from cdn. From this state promise can move to either to **Fulfilled** or to **Rejected**

2. **Fulfilled**: If the promise has reached this state, then it means that the async operation is complete and we have the output. For example, we have the response from the API.

3. **Rejected:** If the promise has reached this state, it means that the async operation is not successful and we have the error which caused the operation to fail.

Okay..Let's see some code.

```
const apiCall = new Promise(function(resolve, reject) {
 // async operation is defined here...
});
```

Promise is defined by creating a constructor using **new** keyword. Then the constructor will have a function (we call it **executor function**.)

The async operation is defined inside the executor function.

And note that the executor function have two parameters **resolve** and **reject**.

The first parameter **resolve** is actually a function. It it is called inside the executor function and it represents that the async operation is successful and we have got the output. **Resolve** function helps the promise to move from **pending** to **fulfilled** state. Hope you got it. :)

Like resolve, **reject** is also a function. It is also called inside the executor function and it represents that the async operation is not successful and we have got an error. **Reject** helps the promise to move from **pending** to **rejected** state. :)

```
const apiCall = new Promise(function(resolve, reject) {
 if ( API request to get some data ) {
  resolve("The request is successful and the response is "+
response);
 }
 else {
  reject("The request is not successful. The error is
"+error);
 }
});
```

In the above code, you can see we have done some async operation inside the executor function. Then the **resolve** function is called if we get the response from the server. And if there is some error **reject** function is called with the error message.

We are done defining the promise. Let's see how to execute promise and process the output.

```
// calling the promise.
apiCall
```

That's it. we are done. :) :)

Just kidding. It's not over yet.

In the above code, the function is called and the promise is executed (i.e) the **executor** function is executed. Then **resolve** or **reject** function is called based on the output.

But you can see that we didn't handle the output returned from the promise.

For example, if we get the response from the API then we have to process the response. Or if we get the error we need to handle it properly.

So how do we handle it ?

We use **handlers** to get the output from the promise.

Handlers are just functions which executes when some event occurs such as clicking a button, moving the cursor, etc.

So we can use handlers to handle when the **resolve** function is called or **reject** function is called.

Simple. :)

Let's see some code

```
// calling the promise with some handlers.
apiCall.then(function(x) {console.log(x); })
```

In the above code, we have attached a handler **then** to the promise. The handler **then** gets a function parameter. Then the function parameter itself has a parameter **x**.

So what's happening.

The handler **then** executes its **function parameter** when the **resolve** function is called inside the promise.

I will try to explain it again.

The **then** handler looks out for the event that is when the **resolve** function is called. So when the resolve function is called the **then** handler executes its function parameter.

```
apiCall.then(function(x) {console.log(x); })

// Output
The request is successful and the response is {name: "Jon
Snow"}
```

Likewise, there is another handler **catch**.

Catch handler looks out for **reject** function.

Catch function executes its function parameter when the **reject** function is called.

```
apiCall.then(function(x) {console.log(x);
}).catch(function(x) {console.log(x); })

// Assuming the request is not successful ( reject function
is called in the promise. )

Output:
The request is not successful
```

I think you got it.

The above code is not quite readable. So let's try to refactor it.

```
apiCall
.then(function(x) {
 console.log(x);
})
.catch(function(x) {
 console.log(x);
})
```

Ahh… It's readable now. Most programmers write like this.

Ok.. so I think you have come a long way.

Let's have a recap.

1. Promise is defined with **new** keyword with a function parameter. Then the function itself has two function parameters **resolve** and **reject.**
2. Function **resolve** should be called when the operation is successful.
3. Function **reject** should be called when the operation is a failure.
4. **Then** handler looks out for **resolve** function.
5. **Catch** handler looks out for **reject** function.
6. Make sure of the readability of the code :) :)

Here's the working example. Please practice if you are new to this.

Edit in JSFiddle

- Babel + JSX
- HTML
- CSS
- Result

```
var ApiCall = new Promise(function(resolve, reject) {

  var request = new XMLHttpRequest();
  request.open('GET', 'https://api.github.com/users/srebalaji');

  request.onload = function() {
    if (request.status == 200) {

      resolve(request.response);
    } else {
      reject(Error(request.statusText));
    }
  }

  request.send();
});

ApiCall
.then(function(x) {
  document.getElementById('response').innerHTML = x;
})
```

Promises in Javascript.

Hope you understand the example. Its straight forward.

Async / Await

If you understand Promises, then Async/Await is quite easy. And If you don’t get promises, Async/Await can help you understand it. Maybe you can get a clear escape from promises too. :)

Async

Async keyword makes any function to return only promises.

For example, consider the code below

```
async function hello() {
 return "Hello Promise..!"
}
```

The function **hello** will return a promise.

The above code is equivalent to the below code.

```
function hello() {
 return new Promise(function(resolve, reject) {
 // executor function body.
 });
}
```

Simple right?

Another Example:

```
async function hello(a, b) {
 if (a < b) {
  return "Greater";
 }
 else {
  return new Error("Not Greater");
 }
}
```

```
hello(14, 10)
.then(function(x) {
 console.log("Good..! " + x);
})
.catch(function(x) {
 console.log("Oops..! " + x);
})

Output:
Oops..! Not Greater.

// if you call hello(4, 10) you get "Good..! Greater"
```

In the above code, we have defined a **async** function and returned some value or returned some error.

If you are returning some value in the async function , it is equivalent to calling resolve function.

If you are returning some error by calling **error constructor (i.e) using 'new'** then it is equivalent to reject function.

Don't forget that async function will return a promise. So of course, you can call **resolve** and **reject** function inside **async** function too.

Lets see how that works.

```
async function Max(a, b) {
 if (a > b) {
  return Promise.resolve("Success");
 }
 else {
  return Promise.reject("Error");
 }
}

Max(4, 10)
.then(function(x) {
 console.log("Good " + x);
})
.catch(function(x) {
 console.log("Oops " + x);
});

Output:
Oops Error

// If we pass Max(14, 10) then we should get "Good Success"
:)
```

Await

As the name implies, it makes the Javascript to wait till the operation is done. Let's say you are making a API request with **await** keyword. It makes the Javascript to wait until you get the response from the endpoint. And then it resumes the execution.

Ok..Let's go deeper

Await can be used only inside async function. It doesn't work outside async function

Let's see a example

```
async function hello() {
 let response = await fetch('https://api.github.com/');
 // above line fetches the response from the given API
endpoint.
 return response;
}

hello()
.then(function(x) {
 console.log(x);
});
...
...

Output:
Response from the API.
```

In the above code, you can see we have used **await** while fetching response from API.

The fetch operation may take few secs to get the response so till that the execution will be halted and resumes later.

Note that await operation only halts the execution inside **hello function**. The remaining code outside the **hello** function will not be affected. The execution continues outside the function. And when we get the response the **function parameter inside then handler** is executed.

Hope you got it.

Let's see an example

Edit in JSFiddle

- Babel + JSX
- HTML
- CSS
- Result

```
function getResponse() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      resolve("Response from API. Executed after 5 secs
    }, 5000);
  });
}
```

Async/Await in Javascript

In the above example, you can see that we have used **await** for **getResponse** function. And the **getResponse** will return an output or an error after 5 seconds. So till that time the execution is halted and then the response is returned.

Lets see some real time example.

Edit in JSFiddle

- JavaScript
- HTML
- CSS
- Result

```
function getResponse(url) {
        return new Promise(function(success, failure) {
  var request = new XMLHttpRequest();
        request.open('GET', url);

  request.onload = function() {
    if (request.status == 200) {
      return success(request.response);
    } else {
      return failure("Error in processing..!" + request.status);
    }
  }
  request.onerror = function() {
    return failure("Error in processing ");
  }
  request.send();
  });
}

function getUsername(response) {
  response = JSON.parse(response);
```

Async/Await in Javascript

In the above example, you can see we have used multiple **awaits.** So for each await the execution stops till the response is received and then resumes.

Try the same example with some invalid url. You can see that the error is raised.

Error handling is very simple in **async** function. If the error is raised inside the async function or when the error is raised from other functions which is called inside async using **await** then the **reject function** is called. Simple.

Hope you enjoyed. I have covered much more interesting topics such as array map, array filter, reduce, etc in the next part. Check it out

ES6 for beginners part-3

ES6 array filter, array map, array reduce, template