JavaScript Promises

**Summary**: in this tutorial, you will learn about JavaScript promises and how to use them effectively.

Why JavaScript promises

The following example [defines a function](https://www.javascripttutorial.net/javascript-function/) getUsers() that returns a list of user [objects](https://www.javascripttutorial.net/javascript-objects/):

function getUsers() {

return [

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

];

}

Code language: JavaScript (javascript)

Each user object has two properties username and email.

To find a user by username from the user list returned by the getUsers() function, you can use the findUser() function as follows:

function findUser(username) {

const users = getUsers();

const user = users.find((user) => user.username === username);

return user;

}

Code language: JavaScript (javascript)

In the findUser() function:

* First, get a user array by calling the getUsers() function
* Second, find the user with a specific username by using the [find()](https://www.javascripttutorial.net/es6/javascript-array-find/) method of the Array object.
* Third, return the matched user.

The following shows the complete code for finding a user with the username 'john':

function getUsers() {

return [

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

];

}

function findUser(username) {

const users = getUsers();

const user = users.find((user) => user.username === username);

return user;

}

console.log(findUser('john'));

Code language: JavaScript (javascript)

Output:

{ username: 'john', email: 'john@test.com' }

Code language: CSS (css)

The code in the findUser() function is synchronous and blocking. The findUser() function executes the getUsers() function to get a user array, calls the find() method on the users array to search for a user with a specific username, and returns the matched user.

In practice, the getUsers() function may access a database or call an API to get the user list. Therefore, the getUsers() function will have a delay.

To simulate the delay, you can use the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function. For example:

function getUsers() {

let users = [];

*// delay 1 second (1000ms)*

setTimeout(() => {

users = [

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

];

}, 1000);

return users;

}

Code language: JavaScript (javascript)

How it works.

* First, define an array users and initialize its value with an empty array.
* Second, assign an array of the users to the users variable inside the callback of the setTimeout() function.
* Third, return the users array

The getUsers() won’t work properly and always returns an empty array. Therefore, the findUser() function won’t work as expected:

function getUsers() {

let users = [];

setTimeout(() => {

users = [

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

];

}, 1000);

return users;

}

function findUser(username) {

const users = getUsers(); *// A*

const user = users.find((user) => user.username === username); *// B*

return user;

}

console.log(findUser('john'));

Code language: JavaScript (javascript)

Output:

undefined

Code language: JavaScript (javascript)

Because the getUsers() returns an empty array, the users array is empty (line A). When calling the find() method on the users array, the method returns undefined (line B)

The challenge is how to access the users returned from the getUsers() function after one second. One classical approach is to use the [callback](https://www.javascripttutorial.net/javascript-callback/).

Using callbacks to deal with an asynchronous operation

The following example adds a callback argument to the getUsers() and findUser() functions:

function getUsers(callback) {

setTimeout(() => {

callback([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

}, 1000);

}

function findUser(username, callback) {

getUsers((users) => {

const user = users.find((user) => user.username === username);

callback(user);

});

}

findUser('john', console.log);

Code language: JavaScript (javascript)

Output:

{ username: 'john', email: 'john@test.com' }

Code language: CSS (css)

In this example, the getUsers() function accepts a callback function as an argument and invokes it with the users array inside the setTimeout() function. Also, the findUser() function accepts a callback function that processes the matched user.

The callback approach works very well. However, it makes the code more difficult to follow. Also, it adds complexity to the functions with callback arguments.

If the number of functions grows, you may end up with the callback hell problem. To resolve this, JavaScript comes up with the concept of promises.

Understanding JavaScript Promises

By definition, a promise is an **object** that encapsulates the result of an **asynchronous operation**.

A promise object has a state that can be one of the following:

* Pending
* Fulfilled with a **value**
* Rejected for a **reason**

In the beginning, the state of a promise is pending, indicating that the asynchronous operation is in progress. Depending on the result of the asynchronous operation, the state changes to either fulfilled or rejected.

The fulfilled state indicates that the asynchronous operation was completed successfully:

The rejected state indicates that the asynchronous operation failed.

Creating a promise

To create a promise object, you use the Promise() constructor:

const promise = new Promise((resolve, reject) => {

*// contain an operation*

*// ...*

*// return the state*

if (success) {

resolve(value);

} else {

reject(error);

}

});

Code language: JavaScript (javascript)

The promise constructor accepts a callback function that typically performs an asynchronous operation. This function is often referred to as an executor.

In turn, the executor accepts two callback functions with the name resolve and reject.

Note that the callback functions passed into the executor are resolve and reject by convention only.

If the asynchronous operation completes successfully, the executor will call the resolve() function to change the state of the promise from pending to fulfilled with a value.

In case of an error, the executor will call the reject() function to change the state of the promise from pending to rejected with the error reason.

Once a promise reaches either fulfilled or rejected state, it stays in that state and can’t go to another state.

In other words, a promise cannot go from the fulfilled state to the rejected state and vice versa. Also, it cannot go back from the fulfilled or rejected state to the pending state.

Once a new Promise object is created, its state is pending. If a promise reaches fulfilled or rejected state, it is *resolved*.

Note that you will rarely create promise objects in practice. Instead, you will consume promises provided by libraries.

Consuming a Promise: then, catch, finally

1) The then() method

To get the value of a promise when it’s fulfilled, you call the then() method of the promise object. The following shows the syntax of the then() method:

promise.then(onFulfilled,onRejected);

Code language: CSS (css)

The then() method accepts two callback functions: onFulfilled and onRejected.

The then() method calls the onFulfilled() with a value, if the promise is fulfilled or the onRejected() with an error if the promise is rejected.

Note that both onFulfilled and onRejected arguments are optional.

The following example shows how to use then() method of the Promise object returned by the getUsers() function:

function getUsers() {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

}, 1000);

});

}

function onFulfilled(users) {

console.log(users);

}

const promise = getUsers();

promise.then(onFulfilled);

Code language: JavaScript (javascript)

Output:

[

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' }

]

Code language: JavaScript (javascript)

In this example:

* First, define the onFulfilled() function to be called when the promise is fulfilled.
* Second, call the getUsers() function to get a promise object.
* Third, call the then() method of the promise object and output the user list to the console.

To make the code more concise, you can use an [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) as the argument of the then() method like this:

function getUsers() {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

}, 1000);

});

}

const promise = getUsers();

promise.then((users) => {

console.log(users);

});

Code language: JavaScript (javascript)

Because the getUsers() function returns a promise object, you can chain the function call with the then() method like this:

*// getUsers() function*

*//...*

getUsers().then((users) => {

console.log(users);

});

Code language: JavaScript (javascript)

In this example, the getUsers() function always succeeds. To simulate the error, we can use a success flag like the following:

let success = true;

function getUsers() {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (success) {

resolve([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

} else {

reject('Failed to the user list');

}

}, 1000);

});

}

function onFulfilled(users) {

console.log(users);

}

function onRejected(error) {

console.log(error);

}

const promise = getUsers();

promise.then(onFulfilled, onRejected);

Code language: JavaScript (javascript)

How it works.

First, define the success variable and initialize its value to true.

If the success is true, the promise in the getUsers() function is fulfilled with a user list. Otherwise, it is rejected with an error message.

Second, define the onFulfilled and onRejected functions.

Third, get the promise from the getUsers() function and call the then() method with the onFulfilled and onRejected functions.

The following shows how to use the arrow functions as the arguments of the then() method:

*// getUsers() function*

*// ...*

const promise = getUsers();

promise.then(

(users) => console.log,

(error) => console.log

);

Code language: JavaScript (javascript)

2) The catch() method

If you want to get the error only when the state of the promise is rejected, you can use the catch() method of the Promise object:

promise.catch(onRejected);

Code language: CSS (css)

Internally, the catch() method invokes the then(undefined, onRejected) method.

The following example changes the success flag to false to simulate the error scenario:

let success = false;

function getUsers() {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (success) {

resolve([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

} else {

reject('Failed to the user list');

}

}, 1000);

});

}

const promise = getUsers();

promise.catch((error) => {

console.log(error);

});

Code language: JavaScript (javascript)

3) The finally() method

Sometimes, you want to execute the same piece of code whether the promise is fulfilled or rejected. For example:

const render = () => {

*//...*

};

getUsers()

.then((users) => {

console.log(users);

render();

})

.catch((error) => {

console.log(error);

render();

});

Code language: JavaScript (javascript)

As you can see, the render() function call is duplicated in both then() and catch() methods.

To remove this duplicate and execute the render() whether the promise is fulfilled or rejected, you use the finally() method, like this:

const render = () => {

*//...*

};

getUsers()

.then((users) => {

console.log(users);

})

.catch((error) => {

console.log(error);

})

.finally(() => {

render();

});

Code language: JavaScript (javascript)

A practical JavaScript Promise example

The following example shows how to load a JSON file from the server and display its contents on a webpage.

Suppose you have the following JSON file:

https:*//www.javascripttutorial.net/sample/promise/api.json*

Code language: JavaScript (javascript)

with the following contents:

{

"message": "JavaScript Promise Demo"

}

Code language: JSON / JSON with Comments (json)

The following shows the HTML page that contains a button. When you click the button, the page loads data from the JSON file and shows the message:

<!DOCTYPE html>

**<html>**

**<head>**

**<meta charset="utf-8">**

**<title>**JavaScript Promise Demo**</title>**

**<link href="css/style.css" rel="stylesheet">**

**</head>**

**<body>**

**<div id="container">**

**<div id="message"></div>**

**<button id="btnGet">**Get Message**</button>**

**</div>**

**<script src="js/promise-demo.js">**

**</script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

The following shows the promise-demo.js file:

function load(url) {

return new Promise(function (resolve, reject) {

const request = new XMLHttpRequest();

request.onreadystatechange = function () {

if (this.readyState === 4 && this.status == 200) {

resolve(this.response);

} else {

reject(this.status);

}

};

request.open('GET', url, true);

request.send();

});

}

const url = 'https://www.javascripttutorial.net/sample/promise/api.json';

const btn = document.querySelector('#btnGet');

const msg = document.querySelector('#message');

btn.addEventListener('click', () => {

load(URL)

.then((response) => {

const result = JSON.parse(response);

msg.innerHTML = result.message;

})

.catch((error) => {

msg.innerHTML = `Error getting the message, HTTP status: ${error}`;

});

});

Code language: JavaScript (javascript)

How it works.

First, define the load() function that uses the XMLHttpRequest object to load the JSON file from the server:

function load(url) {

return new Promise(function (resolve, reject) {

const request = new XMLHttpRequest();

request.onreadystatechange = function () {

if (this.readyState === 4 && this.status == 200) {

resolve(this.response);

} else {

reject(this.status);

}

};

request.open('GET', url, true);

request.send();

});

}

Code language: JavaScript (javascript)

In the executor, we call resolve() function with the Response if the HTTP status code is 200. Otherwise, we invoke the reject() function with the HTTP status code.

Second, register the button click event listener and call the then() method of the promise object. If the load is successful, then we show the message returned from the server. Otherwise, we show the error message with the HTTP status code.

const url = 'https://www.javascripttutorial.net/sample/promise/api.json';

const btn = document.querySelector('#btnGet');

const msg = document.querySelector('#message');

btn.addEventListener('click', () => {

load(URL)

.then((response) => {

const result = JSON.parse(response);

msg.innerHTML = result.message;

})

.catch((error) => {

msg.innerHTML = `Error getting the message, HTTP status: ${error}`;

});

});

Code language: JavaScript (javascript)

Summary

* A promise is an object that encapsulates the result of an asynchronous operation.
* A promise starts in the pending state and ends in either fulfilled state or rejected state.
* Use then() method to schedule a callback to be executed when the promise is fulfilled, and catch() method to schedule a callback to be invoked when the promise is rejected.
* Place the code that you want to execute in the finally() method whether the promise is fulfilled or rejected.

Promise Chaining

**Summary**: in this tutorial, you will learn about the JavaScript promise chaining pattern that chains the promises to execute asynchronous operations in sequence.

Introduction to the JavaScript promise chaining

Sometimes, you want to execute two or more related asynchronous operations, where the next operation starts with the result from the previous step. For example:

First, create a new promise that resolves to the number 10 after 3 seconds:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

Code language: JavaScript (javascript)

Note that the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function simulates an asynchronous operation.

Then, invoke the then() method of the promise:

p.then((result) => {

console.log(result);

return result \* 2;

});

Code language: JavaScript (javascript)

The callback passed to the then() method executes once the promise is resolved. In the callback, we show the result of the promise and return a new value multiplied by two (result\*2).

Because the then() method returns a new Promise with a value resolved to a value, you can call the then() method on the return Promise like this:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

p.then((result) => {

console.log(result);

return result \* 2;

}).then((result) => {

console.log(result);

return result \* 3;

});

Code language: JavaScript (javascript)

Output:

10

20

In this example, the return value in the first then() method is passed to the second then() method. You can keep calling the then() method successively as follows:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

p.then((result) => {

console.log(result); *// 10*

return result \* 2;

}).then((result) => {

console.log(result); *// 20*

return result \* 3;

}).then((result) => {

console.log(result); *// 60*

return result \* 4;

});

Code language: JavaScript (javascript)

Output:

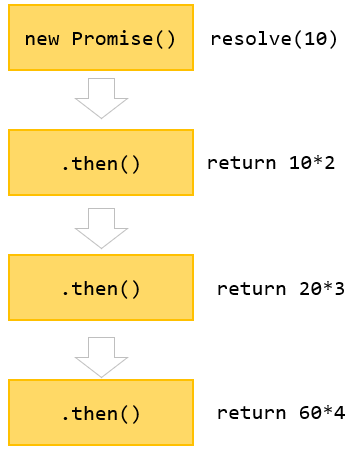
10

20

60

The way we call the then() methods like this is often referred to as a promise chain.

The following picture illustrates the promise chain:



Multiple handlers for a promise

When you call the then() method multiple times on a promise, it is not the promise chaining. For example:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

p.then((result) => {

console.log(result); *// 10*

return result \* 2;

})

p.then((result) => {

console.log(result); *// 10*

return result \* 3;

})

p.then((result) => {

console.log(result); *// 10*

return result \* 4;

});

Code language: JavaScript (javascript)

Output:

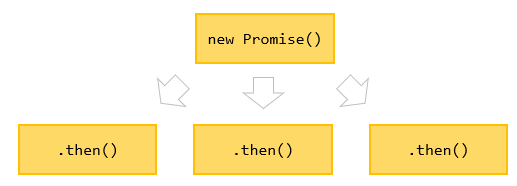
10

10

10

In this example, we have multiple handlers for one promise. These handlers have no relationships. Also, they execute independently and don’t pass the result from one to another like the promise chain above.

The following picture illustrates a promise that has multiple handlers:



In practice, you will rarely use multiple handlers for one promise.

Returning a Promise

When you return a value in the then() method, the then() method returns a new Promise that immediately resolves to the return value.

Also, you can return a new promise in the then() method, like this:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

p.then((result) => {

console.log(result);

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve(result \* 2);

}, 3 \* 1000);

});

}).then((result) => {

console.log(result);

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve(result \* 3);

}, 3 \* 1000);

});

}).then(result => console.log(result));

Code language: JavaScript (javascript)

Output:

10

20

60

This example shows 10, 20, and 60 after every 3 seconds. This code pattern allows you to execute some tasks in sequence.

The following modified the above example:

function generateNumber(num) {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve(num);

}, 3 \* 1000);

});

}

generateNumber(10)

.then((result) => {

console.log(result);

return generateNumber(result \* 2);

})

.then((result) => {

console.log(result);

return generateNumber(result \* 3);

})

.then((result) => console.log(result));

Code language: JavaScript (javascript)

Promise chaining syntax

Sometimes, you have multiple asynchronous tasks that you want to execute in sequence. In addition, you need to pass the result of the previous step to the next one. In this case, you can use the following syntax:

step1()

.then(result => step2(result))

.then(result => step3(result))

...

Code language: JavaScript (javascript)

If you need to pass the result from the previous task to the next one without passing the result, you use this syntax:

step1()

.then(step2)

.then(step3)

...

Code language: CSS (css)

Suppose that you want to perform the following asynchronous operations in sequence:

* First, get the user from the database.
* Second, get the services of the selected user.
* Third, calculate the service cost from the user’s services.

The following functions illustrate the three asynchronous operations:

function getUser(userId) {

return new Promise((resolve, reject) => {

console.log('Get the user from the database.');

setTimeout(() => {

resolve({

userId: userId,

username: 'admin'

});

}, 1000);

})

}

function getServices(user) {

return new Promise((resolve, reject) => {

console.log(`Get the services of ${user.username} from the API.`);

setTimeout(() => {

resolve(['Email', 'VPN', 'CDN']);

}, 3 \* 1000);

});

}

function getServiceCost(services) {

return new Promise((resolve, reject) => {

console.log(`Calculate the service cost of ${services}.`);

setTimeout(() => {

resolve(services.length \* 100);

}, 2 \* 1000);

});

}

Code language: JavaScript (javascript)

The following uses the promises to serialize the sequences:

getUser(100)

.then(getServices)

.then(getServiceCost)

.then(console.log);

Code language: CSS (css)

Output

Get the user from the database.

Get the services of admin from the API.

Calculate the service cost of Email,VPN,CDN.

300

Code language: JavaScript (javascript)

Note that ES2017 introduced the [async/await](https://www.javascripttutorial.net/es-next/javascript-async-await/) that helps you write the code that is cleaner than using the promise chaining technique.

In this tutorial, you have learned about the promise chain that executes multiple asynchronous tasks in sequence.

JavaScript Promise.all()

**Summary**: in this tutorial, you will learn how to use the Promise.all() static method to aggregate results from multiple asynchronous operations.

Introduction to the JavaScript Promise.all() method

The Promise.all() static method takes an [iterable](https://www.javascripttutorial.net/es6/javascript-iterator/) of [promises](https://www.javascripttutorial.net/es6/javascript-promises/):

Promise.all(iterable);

Code language: JavaScript (javascript)

The Promise.all() method returns a single promise that resolves when all the input promises have been resolved. The returned promise resolves to an array of the results of the input promises:

In this diagram, the promise1 resolves to a value v1 at t1 and the promise2 resolves to a value v2 at t2. Hence, the Promise.all(promise1, promise2) returns a promise that resolves to an array containing the results of the promise1 and promise2 [v1, v2] at t2.

In other words, the Promise.all() waits for all the input promises to resolve and returns a new promise that resolves to an array containing the results of the input promises.

If one of the input promise rejects, the Promise.all() method immediately returns a promise that rejects with an error of the first rejected promise:

In this diagram, the promise2 rejects at t1 with an error. Therefore, the Promise.all() returns a new promise that immediately rejects with the same error. Also, the Promise.all() doesn’t care other input promises, whether they will resolve or reject.

In practice, the Promise.all() is useful to aggregate the results from multiple asynchronous operations.

JavaScript Promise.all() method examples

Let’s take some examples to understand how the Promise.all() method works.

1) Resolved promises example

The following promises resolve to 10, 20, and 30 after 1, 2, and 3 seconds. We use the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) to simulate the asynchronous operations:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The first promise has resolved');

resolve(10);

}, 1 \* 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The second promise has resolved');

resolve(20);

}, 2 \* 1000);

});

const p3 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The third promise has resolved');

resolve(30);

}, 3 \* 1000);

});

Promise.all([p1, p2, p3]).then((results) => {

const total = results.reduce((p, c) => p + c);

console.log(`Results: ${results}`);

console.log(`Total: ${total}`);

});

Code language: JavaScript (javascript)

Output

The first promise has resolved

The second promise has resolved

The third promise has resolved

Results: 10,20,30

Total: 60

When all promises have resolved, the values from these promises are passed into the callback of the then() method as an array.

Inside the callback, we use the Array’s [reduce()](https://www.javascripttutorial.net/javascript-array-reduce/) method to calculate the total value and use the console.log to display the array of values as well as the total.

2) Rejected promises example

The Promise.all() returns a Promise that is rejected if any of the input promises are rejected.

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The first promise has resolved');

resolve(10);

}, 1 \* 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The second promise has rejected');

reject('Failed');

}, 2 \* 1000);

});

const p3 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The third promise has resolved');

resolve(30);

}, 3 \* 1000);

});

Promise.all([p1, p2, p3])

.then(console.log) *// never execute*

.catch(console.log);

Code language: JavaScript (javascript)

Output:

The first promise has resolved

The second promise has rejected

Failed

The third promise has resolved

In this example, we have three promises: the first one is resolved after 1 second, the second is rejected after 2 seconds, and the third one is resolved after 3 seconds.

As a result, the returned promise is rejected because the second promise is rejected. The catch() method is executed to display the reason for the rejected promise.

Summary

* The Promise.all() method accepts a list of promises and returns a new promsie that resolve to an array of results of the input promises if all the input promises resolved; or reject with an error of the first rejected promise.
* Use the Promise.all() method to aggregate results from multiple asynchronous operations.

JavaScript Promise.race()

**Summary**: in this tutorial, you will learn how to use the JavaScript Promise.race() static method.

Introduction to JavaScript Promise.race() static method

The Promise.race() static method accepts a list of [promises](https://www.javascripttutorial.net/es6/javascript-promises/) as an iterable object and returns a new promise that fulfills or rejects as soon as there is one promise that fulfills or rejects, with the value or reason from that promise.

Here’s the syntax of the Promise.race() method:

Promise.race(iterable)

Code language: JavaScript (javascript)

In this syntax, the iterable is an [iterable object](https://www.javascripttutorial.net/es6/javascript-iterator/) that contains a list of promises.

The name of Promise.race() implies that all the promises race against each other with a single winner, either resolved or rejected.

See the following diagram:

In this diagram:

* The promise1 is fulfilled with the value v1 at t1.
* The promise2 is rejected with the error at t2.
* Because the promise1 is resolved earlier than the promise2, the promise1 wins the race. Therefore, the Promise.race([promise1, promise2]) returns a new promise that is fulfilled with the value v1 at t1.

See another diagram:

In this diagram:

* The promise1 is fulfilled with v1 at t2.
* The promise2 is rejected with error at t1.
* Because the promise2 is resolved earlier than the promise1, the promise2 wins the race. Therefore, the Promise.race([promise1, promise2]) returns a new promise that is rejected with the error at t1.

JavaScript Promise.race() examples

Let’s take some examples of using the Promise.race() static method.

1) Simple JavaScript Promise.race() examples

The following creates two promises: one resolves in 1 second and the other resolves in 2 seconds. Because the first promise resolves faster than the second one, the Promise.race() resolves with the value from the first promise:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The first promise has resolved');

resolve(10);

}, 1 \* 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The second promise has resolved');

resolve(20);

}, 2 \* 1000);

});

Promise.race([p1, p2])

.then(value => console.log(`Resolved: ${value}`))

.catch(reason => console.log(`Rejected: ${reason}`));

Code language: JavaScript (javascript)

Output:

The first promise has resolved

Resolved: 10

The second promise has resolved

The following example creates two promises. The first promise resolves in 1 second while the second one rejects in 2 seconds. Because the first promise is faster than the second one, the returned promise resolves to the value of the first promise:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The first promise has resolved');

resolve(10);

}, 1 \* 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The second promise has rejected');

reject(20);

}, 2 \* 1000);

});

Promise.race([p1, p2])

.then(value => console.log(`Resolved: ${value}`))

.catch(reason => console.log(`Rejected: ${reason}`));

Code language: JavaScript (javascript)

Output

The first promise has resolved

Resolved: 10

The second promise has rejected

Note that if the second promise was faster than the first one, the return promise would reject for the reason of the second promise.

2) Practical JavaScript Promise.race() example

Suppose you have to show a spinner if the data loading process from the server is taking longer than a number of seconds.

To do this, you can use the Promise.race() static method. If a timeout occurs, you show the loading indicator, otherwise, you show the message.

The following illustrates the HTML code:

<!DOCTYPE html>

**<html>**

**<head>**

**<meta charset="utf-8">**

**<title>**JavaScript Promise.race() Demo**</title>**

**<link href="css/promise-race.css" rel="stylesheet">**

**</head>**

**<body>**

**<div id="container">**

**<button id="btnGet">**Get Message**</button>**

**<div id="message"></div>**

**<div id="loader"></div>**

**</div>**

**<script src="js/promise-race.js"></script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

To create the loading indicator, we use the CSS animation feature. See the [promise-race.css](https://www.javascripttutorial.net/sample/promise/css/promise-race.css) for more information. Technically speaking, if an element has the .loader class, it shows the loading indicator.

First, define a new function that loads data. It uses the setTimeout() to emulate an asynchronous operation:

const DATA\_LOAD\_TIME = 5000;

function getData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const message = 'Promise.race() Demo';

resolve(message);

}, DATA\_LOAD\_TIME);

});

}

Code language: JavaScript (javascript)

Second, develop a function that shows some contents:

function showContent(message) {

document.querySelector('#message').textContent = message;

}

Code language: JavaScript (javascript)

This function can also be used to set the message to blank.

Third, define the timeout() function that returns a promise. The promise will reject when a specified TIMEOUT is passed.

const TIMEOUT = 500;

function timeout() {

return new Promise((resolve, reject) => {

setTimeout(() => reject(), TIMEOUT);

});

}

Code language: JavaScript (javascript)

Fourth, develop a couple of functions that show and hide the loading indicator:

function showLoadingIndicator() {

document.querySelector('#loader').className = 'loader';

}

function hideLoadingIndicator() {

document.querySelector('#loader').className = '';

}

Code language: JavaScript (javascript)

Fifth, attach a click event listener to the**Get Message** button. Inside the click handler, use the Promise.race() static method:

*// handle button click event*

const btn = document.querySelector('#btnGet');

btn.addEventListener('click', () => {

*// reset UI if users click the 2nd, 3rd, ... time*

reset();

*// show content or loading indicator*

Promise.race([getData()

.then(showContent)

.then(hideLoadingIndicator), timeout()

])

.catch(showLoadingIndicator);

});

Code language: JavaScript (javascript)

We pass two promises to the Promise.race() method:

Promise.race([getData()

.then(showContent)

.then(hideLoadingIndicator), timeout()

])

.catch(showLoadingIndicator);

Code language: JavaScript (javascript)

The first promise gets data from the server, shows the content, and hides the loading indicator. The second promise sets a timeout.

If the first promise takes more than 500 ms to settle, the catch() is called to show the loading indicator. Once the first promise resolves, it hides the loading indicator.

Finally, develop a reset() function that hides the message and loading indicator if the button is clicked for the second time.

*// reset UI*

function reset() {

hideLoadingIndicator();

showContent('');

}

Code language: JavaScript (javascript)

Put it all together.

*// after 0.5 seconds, if the getData() has not resolved, then show*

*// the Loading indicator*

const TIMEOUT = 500;

const DATA\_LOAD\_TIME = 5000;

function getData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const message = 'Promise.race() Demo';

resolve(message);

}, DATA\_LOAD\_TIME);

});

}

function showContent(message) {

document.querySelector('#message').textContent = message;

}

function timeout() {

return new Promise((resolve, reject) => {

setTimeout(() => reject(), TIMEOUT);

});

}

function showLoadingIndicator() {

document.querySelector('#loader').className = 'loader';

}

function hideLoadingIndicator() {

document.querySelector('#loader').className = '';

}

*// handle button click event*

const btn = document.querySelector('#btnGet');

btn.addEventListener('click', () => {

*// reset UI if users click the second time*

reset();

*// show content or loading indicator*

Promise.race([getData()

.then(showContent)

.then(hideLoadingIndicator), timeout()

])

.catch(showLoadingIndicator);

});

*// reset UI*

function reset() {

hideLoadingIndicator();

showContent('');

}

Code language: JavaScript (javascript)

Summary

* The Promise.race(iterable) method returns a new promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or error from that promise.

JavaScript Promise.any()

**Summary**: in this tutorial, you’ll learn how to use the JavaScript Promise.any() method to compose promises.

Introduction to JavaScript Promise.any() method

The Promise.any() method accepts a list of [Promise](https://www.javascripttutorial.net/es6/javascript-promises/) objects as an [iterable object](https://www.javascripttutorial.net/es-next/javascript-asynchronous-iterators/):

Promise.any(iterable);

Code language: JavaScript (javascript)

If one of the promises in the iterable object is fulfilled, the Promise.any() returns a single promise that resolves to a value which is the result of the fulfilled promise:

In this diagram:

* The promise1 resolves to a value v1 at t1.
* The promise2 resolves to a value v2 at t2.
* The Promise.any() returns a promise that resolves to a value v1, which is the result of the promise1, at t1

The Promise.any() returns a promise that is fulfilled with any first fulfilled promise even if some promises in the iterable object are rejected:

In this diagram:

* The promise1 is rejected with an error at t1.
* The promise2 is fulfilled to value v2 at t2.
* The Promise.any() returns the a promise that resolves to a value v2 which is the result of the promise2. Note that the Promise.any() method ignores the rejected promise (promise1).

If all promises in the iterable object are rejected or if the iterable object is empty, the Promise.any() return a promise that rejects with an AggregateError containing all the rejection reasons. The AggregateError is a subclass of Error.

In this diagram:

* The promise1 is rejected for an error1 at t1.
* The promise2 is rejected for an error2 at t2.
* The Promise.any() returns a promise that is rejected at t2 with an AggregateError containing the error1 and error2 of all the rejected promises.

JavaScript Promise.any() examples

Let’s take some examples of using the Promise.any() method.

1) All promises fulfilled example

The following example demonstrates the Promise.any() method with all promises fulfilled:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 1 fulfilled');

resolve(1);

}, 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 2 fulfilled');

resolve(2);

}, 2000);

});

const p = Promise.any([p1, p2]);

p.then((value) => {

console.log('Returned Promise');

console.log(value);

});

Code language: JavaScript (javascript)

Output:

Promise 1 fulfilled

Returned Promise

1

Promise 2 fulfilled

Code language: JavaScript (javascript)

How it works.

* First, create a new promise p1 that will resolve to a value 1 after one second.
* Second, create a new promise p2 that will resolve to a value 2 after two seconds.
* Third, use the Promise.any() method that uses two promises p1 and p2. The Promise.any() returns a promise p that will resolve to the value 1 of the first fulfilled promise (p1) after one second.

2) One promise rejected example

The following example uses the Promise.any() method with list of promises that has a rejected promise:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 1 rejected');

reject('error');

}, 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 2 fulfilled');

resolve(2);

}, 2000);

});

const p = Promise.any([p1, p2]);

p.then((value) => {

console.log('Returned Promise');

console.log(value);

});

Code language: JavaScript (javascript)

Output:

Promise 1 rejected

Promise 2 fulfilled

Returned Promise

2

Code language: JavaScript (javascript)

In this example, the Promise.any() ignores the rejected promise. When the p2 resolves with the value 2, the Promise.any() returns a promise that resolves to the same value of the result of the p2.

3) All promises rejected example

The following example demonstrates how to use the Promise.any() method with all promises rejected:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 1 rejected');

reject('error1');

}, 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 2 rejected');

reject('error2');

}, 2000);

});

const p = Promise.any([p1, p2]);

p.catch((e) => {

console.log('Returned Promise');

console.log(e, e.errors);

});

Code language: JavaScript (javascript)

Output:

Promise 1 rejected

Promise 2 rejected

Returned Promise

[AggregateError: All promises were rejected] [ 'error1', 'error2' ]

Code language: JavaScript (javascript)

In this example, both p1 and p2 were rejected with the string error1 and error2. Therefore, the Promise.any() method was rejected with an AggregateError object that has the errors property containing all the errors of the rejected promises.

When to use the JavaScript Promise.any() method

In practice, you use the Promise.any() to return the first fulfilled promise. Once a promise is fulfilled, the Promise.any() method does not wait for other promises to be complete. In other words, the Promise.any() short-circuits after a promise is fulfilled.

For example, you have a resource served by two or more content delivery networks (CDN). To dynamically load the first available resource, you can use the Promise.any() method.

The following example uses the Promise.any() method to fetch two images and displays the first available image.

The index.html file

<!DOCTYPE html>

**<html lang="en">**

**<head>**

**<meta charset="UTF-8" />**

**<meta name="viewport" content="width=device-width, initial-scale=1.0" />**

**<title>**JavaScript Promise.any() Demo**</title>**

**</head>**

**<body>**

**<script src="js/app.js"></script>**

**</body>**

**</html>**

Code language: JavaScript (javascript)

The app.js file

function getImageBlob(url) {

return fetch(url).then((response) => {

if (!response.ok) {

throw new Error(`HTTP status: ${response.status}`);

}

return response.blob();

});

}

let cat = getImageBlob(

'https://upload.wikimedia.org/wikipedia/commons/4/43/Siberian\_black\_tabby\_blotched\_cat\_03.jpg'

);

let dog = getImageBlob(

'https://upload.wikimedia.org/wikipedia/commons/a/af/Golden\_retriever\_eating\_pigs\_foot.jpg'

);

Promise.any([cat, dog])

.then((data) => {

let objectURL = URL.createObjectURL(data);

let image = document.createElement('img');

image.src = objectURL;

document.body.appendChild(image);

})

.catch((e) => {

console.log(e.message);

});

Code language: JavaScript (javascript)

How it works.

* First, define the getImageBlob() function that uses the [fetch API](https://www.javascripttutorial.net/javascript-fetch-api/) to get the image’s blob from an URL. The getImageBlob() returns a Promise object that resolves to the image blob.
* Second, define two promises that load the images.
* Third, show the first available image by using the Promise.any() method.

Summary

* Use the JavaScript Promise.any() method to take a list of promises and return a promise that fulfills first.

JavaScript Promise.allSettled()

**Summary**: in this tutorial, you’ll learn about the Promise.allSettled() method to compose promises.

Introduction to the Promise.allSettled() method

ES2020 introduced the Promise.allSettled() method that accepts a list of [Promises](https://www.javascripttutorial.net/es6/javascript-promises/) and returns a new promise that resolves after all the input promises have settled, either resolved or rejected.

The following shows the syntax of the Promise.allSettled() method:

Promise.allSettled(iterable);

Code language: JavaScript (javascript)

The iterable contains the promises. The Promise.allSettled() returns a pending promise that will be asynchronously fulfilled once every input promise has settled.

The Promise.allSettled() method returns a promise that resolves to an [array](https://www.javascripttutorial.net/javascript-array/) of [objects](https://www.javascripttutorial.net/javascript-objects/) that each describes the result of the input promise.

Each object has two properties: status and value (or reason).

* The status can be either fulfilled or rejected.
* The value if case the promise is fulfilled or reason) if the promise is rejected.

The following diagram illustrates how the Promise.allSettled() method works:

In this diagram:

* The promise1 rejects to the error at t1.
* The promise2 resolves to a value at t2.
* The Promise.allSettled() method resolves to a array containing objects that describe the statuses and outcomes of the promise1 and promise2.

JavaScript Promise.allSettled() example

The following example uses the Promise.allSettled() to wait for all the input Promises to settle:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The first promise has resolved');

resolve(10);

}, 1 \* 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The second promise has rejected');

reject(20);

}, 2 \* 1000);

});

Promise.allSettled([p1, p2])

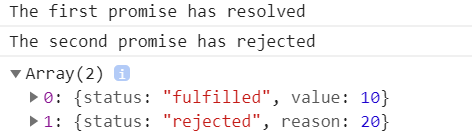
.then((result) => {

console.log(result);

});

Code language: JavaScript (javascript)

Output:



How it works:

* The first promise p1 resolves to the value 10 after one second
* The second promise p2 rejects for a reason with a value 20 after two seconds.
* The Promise.allSettled() returns a promise that resolves to the result array that has two elements. The first element is an object resolved by the p1 promise and the second one is another object which is rejected by the p2 promise.

Summary

* The Promise.allSettled() method accepts an iterable of promises and returns a new promise that resolves when every input promise has settled with an array of objects that describes the result of each promise in the iterable object.

JavaScript Promise finally()

**Summary**: in this tutorial, you will learn how to use the JavaScript Promise finally() method to execute the code once the promise is settled, regardless of its outcome.

Introduction to the JavaScript Promise finally() method

Suppose that you have a [promise](https://www.javascripttutorial.net/es6/javascript-promises/):

promise

.then(result => { ...})

.catch(error => { ... })

.finally(() => { ... })

Code language: JavaScript (javascript)

The finally() method is always executed whether the promise is fulfilled or rejected. In other words, the finally() method is executed when the promise is settled.

The finally() method was introduced in ES2018. In the finally() method, you can place the code that cleans up the resource when the promise is settled, regardless of its outcome.

By using the finally() method, you can avoid duplicate code in the then() and catch() methods like this:

promise

.then(result => {

*// process the result*

*// clean up the resources*

})

.catch(error => {

*// handle the error*

*// clean up the resources*

});

Code language: JavaScript (javascript)

Now, you can move the clean up the resources part to the finally() method as follows:

promise

.then(result => {

*// process the result*

})

.catch(error => {

*// handle the error*

})

.finally(() => {

*// clean up the resources*

});

Code language: JavaScript (javascript)

The finally() method is similar to the finally block in the [try...catch...finally](https://www.javascripttutorial.net/javascript-try-catch-finally/) statement. In synchronous code, you use the finally block to clean up the resources. In asynchronous code, you use the finally() method instead.

The JavaScript Promise finally() method example

The following defines a Connection class:

class Connection {

execute(query) {

if (query != 'Insert' && query != 'Update' && query != 'Delete') {

throw new Error(`The ${query} is not supported`);

}

console.log(`Execute the ${query}`);

return this;

}

close() {

console.log('Close the connection')

}

}

Code language: JavaScript (javascript)

The Connection class has two methods: execute() and close():

* The execute() method will only execute the insert, update, or delete query. It will issue an error if you pass into another query that is not in the list.
* The close() method closes the connection, kind of cleaning up the resource.

The following connect() function returns a promise that resolves to a new Connection if the success flag is set to true:

const success = true;

function connect() {

return new Promise((resolve, reject) => {

if (success)

resolve(new Connection());

else

reject('Could not open the database connection');

});

}

Code language: JavaScript (javascript)

The following example uses the finally() method to close the connection:

let globalConnection;

connect()

.then((connection) => {

globalConnection = connection;

return globalConnection.execute('Insert');

})

.then((connection) => {

globalConnection = connection;

return connection.execute('Select');

})

.catch(console.log)

.finally(() => {

if (globalConnection) {

globalConnection.`close()`;

}

});

Code language: JavaScript (javascript)

In this example:

* The connect() function resolves to a new Connection  object because the success flag is set to true.
* The first then() method executes the Insert query and returns a Connection object. The globalConnection is used to save the connection.
* The second then() method executes the Select query and issues an error. The catch() method shows the error message and the finally() method closes the connection.

Summary

* The finally() method schedule a function to execute when the promise is settled, either fulfilled or rejected.
* It’s good practice to place the code that cleans up the resources in the finally() method once the promise is settled, regardless of its outcome.

Promise Error Handling

**Summary**: in this tutorial, you will learn how to deal with error handling in promises.

Suppose that you have a function called getUserById() that returns a [Promise](https://www.javascripttutorial.net/es6/javascript-promises/):

function getUserById(id) {

return new Promise((resolve, reject) => {

resolve({

id: id,

username: 'admin'

});

});

}

Code language: JavaScript (javascript)

Normal error

First, change the getUserById() function to [throw an error](https://www.javascripttutorial.net/javascript-try-catch/) outside the promise:

function getUserById(id) {

if (typeof id !== 'number' || id <= 0) {

throw new Error('Invalid id argument');

}

return new Promise((resolve, reject) => {

resolve({

id: id,

username: 'admin'

});

});

}

Code language: JavaScript (javascript)

Second, handle the promise by using both then() and catch() methods:

getUserById('a')

.then(user => console.log(user.username))

.catch(err => console.log(err));

Code language: JavaScript (javascript)

The code throws an error:

Uncaught Error: Invalid id argument

Code language: JavaScript (javascript)

When you raise an exception outside the promise, you must catch it with try/catch:

try {

getUserById('a')

.then(user => console.log(user.username))

.catch(err => console.log(`Caught by .catch ${error}`));

} catch (error) {

console.log(`Caught by try/catch ${error}`);

}

Code language: JavaScript (javascript)

Output:

Caught by try/catch Error: Invalid id argument

Code language: JavaScript (javascript)

Errors inside the Promises

We change the getUserById() function to throw an error inside the promise:

let authorized = false;

function getUserById(id) {

return new Promise((resolve, reject) => {

if (!authorized) {

throw new Error('Unauthorized access to the user data');

}

resolve({

id: id,

username: 'admin'

});

});

}

Code language: JavaScript (javascript)

And consume the promise:

try {

getUserById(10)

.then(user => console.log(user.username))

.catch(err => console.log(`Caught by .catch ${error}`));

} catch (error) {

console.log(`Caught by try/catch ${error}`);

}

Code language: JavaScript (javascript)

Output:

Caught by .catch Error: Unauthorized access to the user data

Code language: JavaScript (javascript)

If you throw an error inside the promise, the catch() method will catch it, not the try/catch.

If you chain promises, the catch() method will catch errors that occurred in any promise. For example:

promise1

.then(promise2)

.then(promise3)

.then(promise4)

.catch(err => console.log(err));

Code language: JavaScript (javascript)

In this example, if any error in the promise1, promise2, or promise4, the catch() method will handle it.

Calling reject() function

Throwing an error has the same effect as calling the reject() as illustrated in the following example:

let authorized = false;

function getUserById(id) {

return new Promise((resolve, reject) => {

if (!authorized) {

reject('Unauthorized access to the user data');

}

resolve({

id: id,

username: 'admin'

});

});

}

try {

getUserById(10)

.then(user => console.log(user.username))

.catch(err => console.log(`Caught by .catch ${err}`));

} catch (error) {

console.log(`Caught by try/catch ${error}`);

}

Code language: JavaScript (javascript)

In this example, instead of throwing an error inside the promise, we called the reject() explicitly. The catch() method also handles the error in this case.

Missing the catch() method

The following example does not provide the catch() method to handle the error inside the promise. It will cause a runtime error and terminate the program:

function getUserById(id) {

return new Promise((resolve, reject) => {

if (!authorized) {

reject('Unauthorized access to the user data');

}

resolve({

id: id,

username: 'admin'

});

});

}

try {

getUserById(10)

.then(user => console.log(user.username));

*// the following code will not execute*

console.log('next');

} catch (error) {

console.log(`Caught by try/catch ${error}`);

}

Code language: JavaScript (javascript)

Output:

Uncaught (in promise) Unauthorized access to the user data

If the promise is resolved, you can omit the catch() method. In the future, a potential error may cause the program to stop unexpectedly.

Summary

* Inside the promise, the catch() method will catch the error caused by the throw statement and reject().
* If an error occurs and you don’t have the catch() method, the JavaScript engine issues a runtime error and stops the program.

JavaScript async/await

**Summary**: in this tutorial, you will learn how to write asynchronous code  using JavaScript  async/  await keywords.

Note that to understand how the async / await works, you need to know how [promises](https://www.javascripttutorial.net/es6/javascript-promises/) work.

Introduction to JavaScript async / await keywords

In the past, to deal with asynchronous operations, you often used the [callback functions](https://www.javascripttutorial.net/javascript-callback/). However, when you nest many callback functions, the code will be more difficult to maintain. And you end up with a notorious issue which is known as the callback hell.

Suppose that you need to perform three asynchronous operations in the following sequence:

1. Select a user from the database.
2. Get services of the user from an API.
3. Calculate the service cost based on the services from the server.

The following functions illustrate the three tasks. Note that we use the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function to simulate the asynchronous operation.

function getUser(userId, callback) {

console.log('Get user from the database.');

setTimeout(() => {

callback({

userId: userId,

username: 'john'

});

}, 1000);

}

function getServices(user, callback) {

console.log(`Get services of ${user.username} from the API.`);

setTimeout(() => {

callback(['Email', 'VPN', 'CDN']);

}, 2 \* 1000);

}

function getServiceCost(services, callback) {

console.log(`Calculate service costs of ${services}.`);

setTimeout(() => {

callback(services.length \* 100);

}, 3 \* 1000);

}

Code language: JavaScript (javascript)

The following shows the nested callback functions:

getUser(100, (user) => {

getServices(user, (services) => {

getServiceCost(services, (cost) => {

console.log(`The service cost is ${cost}`);

});

});

});

Code language: JavaScript (javascript)

Output:

Get user from the database.

Get services of john from the API.

Calculate service costs of Email,VPN,CDN.

The service cost is 300

Code language: JavaScript (javascript)

To avoid this callback hell issue, ES6 introduced the [promises](https://www.javascripttutorial.net/es6/javascript-promises/) that allow you to write asynchronous code in more manageable ways.

First, you need to return a Promise in each function:

function getUser(userId) {

return new Promise((resolve, reject) => {

console.log('Get user from the database.');

setTimeout(() => {

resolve({

userId: userId,

username: 'john'

});

}, 1000);

})

}

function getServices(user) {

return new Promise((resolve, reject) => {

console.log(`Get services of ${user.username} from the API.`);

setTimeout(() => {

resolve(['Email', 'VPN', 'CDN']);

}, 2 \* 1000);

});

}

function getServiceCost(services) {

return new Promise((resolve, reject) => {

console.log(`Calculate service costs of ${services}.`);

setTimeout(() => {

resolve(services.length \* 100);

}, 3 \* 1000);

});

}

Code language: JavaScript (javascript)

Then, you [chain the promises](https://www.javascripttutorial.net/es6/promise-chaining/):

getUser(100)

.then(getServices)

.then(getServiceCost)

.then(console.log);

Code language: CSS (css)

ES2017 introduced the async/await keywords that build on top of promises, allowing you to write asynchronous code that looks more like synchronous code and is more readable. Technically speaking, the async / await is syntactic sugar for promises.

If a function returns a Promise, you can place the await keyword in front of the function call, like this:

let result = await f();

Code language: JavaScript (javascript)

The await will wait for the Promise returned from the f() to settle. The await keyword can be used only inside the async functions.

The following defines an async function that calls the three asynchronous operations in sequence:

async function showServiceCost() {

let user = await getUser(100);

let services = await getServices(user);

let cost = await getServiceCost(services);

console.log(`The service cost is ${cost}`);

}

showServiceCost();

Code language: JavaScript (javascript)

As you can see, the asynchronous code now looks like the synchronous code.

Let’s dive in the async / await keywords.

The async keyword

The async keyword allows you to define a function that handles asynchronous operations.

To define an async function, you place the async keyword in front of the function keyword as follows:

async function sayHi() {

return 'Hi';

}

Code language: JavaScript (javascript)

Asynchronous functions execute asynchronously via the [event loop](https://www.javascripttutorial.net/javascript-event-loop/). It always returns a Promise.

In this example, because the sayHi() function returns a Promise, you can consume it, like this:

sayHi().then(console.log);

Code language: CSS (css)

You can also explicitly return a Promise from the sayHi() function as shown in the following code:

async function sayHi() {

return Promise.resolve('Hi');

}

Code language: JavaScript (javascript)

The effect is the same.

Besides the regular functions, you can use the async keyword in the function expressions:

let sayHi = async function () {

return 'Hi';

}

Code language: JavaScript (javascript)

[arrow functions](https://www.javascripttutorial.net/es6/javascript-arrow-function/):

let sayHi = async () => 'Hi';

Code language: JavaScript (javascript)

and methods of classes:

class Greeter {

async sayHi() {

return 'Hi';

}

}

Code language: JavaScript (javascript)

The await keyword

You use the await keyword to wait for a Promise to settle either in resolved or rejected state. And you can use the await keyword only inside an async function:

async function display() {

let result = await sayHi();

console.log(result);

}

Code language: JavaScript (javascript)

In this example, the await keyword instructs the JavaScript engine to wait for the sayHi() function to complete before displaying the message.

Note that if you use the await operator outside of an async function, you will get an error.

Error handling

If a promise resolves, the await promise returns the result. However, when the promise rejects, the await promise will throw an error as if there were a throw statement.

The following code:

async function getUser(userId) {

await Promise.reject(new Error('Invalid User Id'));

}

Code language: JavaScript (javascript)

… is the same as this:

async function getUser(userId) {

throw new Error('Invalid User Id');

}

Code language: JavaScript (javascript)

In the real scenario, it will take a while for the promise to throw an error.

You can catch the error by using the [try...catch](https://www.javascripttutorial.net/javascript-try-catch/) statement, the same way as a regular throw statement:

async function getUser(userId) {

try {

const user = await Promise.reject(new Error('Invalid User Id'));

} catch(error) {

console.log(error);

}

}

Code language: JavaScript (javascript)

It’s possible to catch errors caused by one or more await promise‘s:

async function showServiceCost() {

try {

let user = await getUser(100);

let services = await getServices(user);

let cost = await getServiceCost(services);

console.log(`The service cost is ${cost}`);

} catch(error) {

console.log(error);

}

}

Code language: JavaScript (javascript)

In this tutorial, you have learned how to use the JavaScript async / await keyword to write asynchronous code look like synchronous code.