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 `getUsers()` that returns a list of user objects:

```
function getUsers() {
  return [
    { username: 'john', email: 'john@test.com' },
    { username: 'jane', email: 'jane@test.com' },
  ];
}
```

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;
}
```

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()` 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'));
```

Output:

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

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()` 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;
}
```

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'));
```

Output:

```
undefined
```

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.

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);
```

Output:

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

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

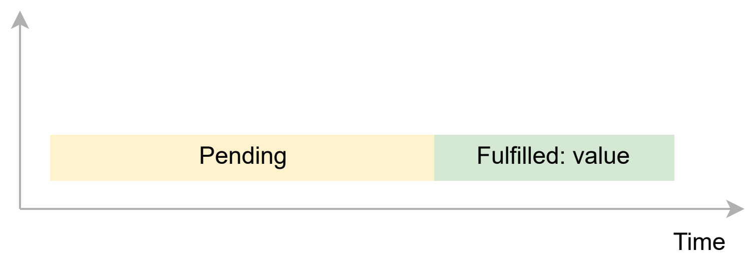
By definition, a promise is an **object** that encapsulates the result of an **asynchronous operation**.

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

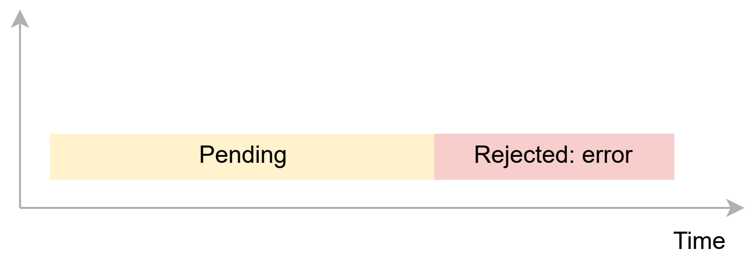
- Pending
- Fulfilled with a **value**
- Rejected for a **reason**

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);
  }
});
```

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.

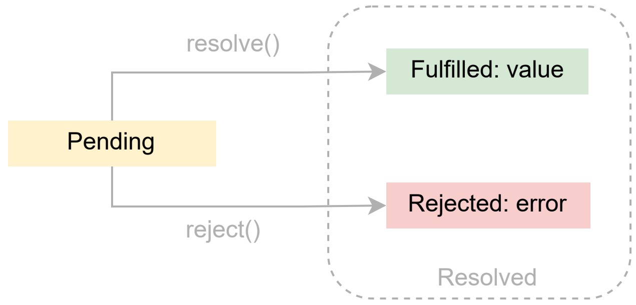
If the asynchronous operation completes successfully, the executor will call the `resolve()` function to change the state of the promise from pending to fulfilled with a value.

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 a 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);
```

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);
```

Output:

```
[
  { username: 'john', email: 'john@test.com' },
  { username: 'jane', email: 'jane@test.com' }
]
```

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 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);
});
```

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);
});
```

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);
```

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
);
```

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);
```

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);
});
```

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();
  });
```

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();
  });
```

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
```

with the following contents:

```
{
    "message": "JavaScript Promise Demo"
}
```

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>
```

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}`;
    });
});
```

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();
  });
}
```

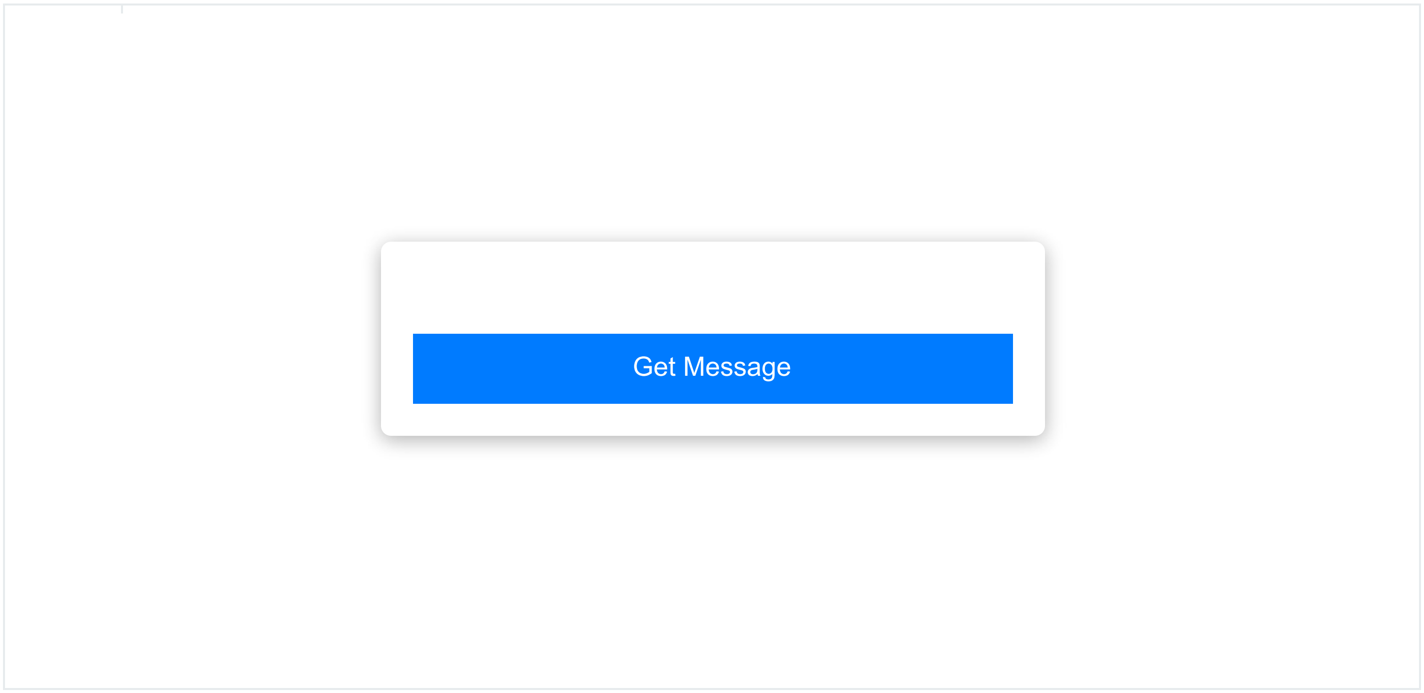
In the executor, we call `resolve()` function with the Response if the HTTP status code is 200. Otherwise, we invoke the `reject()` function with the HTTP status code.

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}`;
    });
});
```

Output



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 a fulfilled state or a 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.

Quiz