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 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.

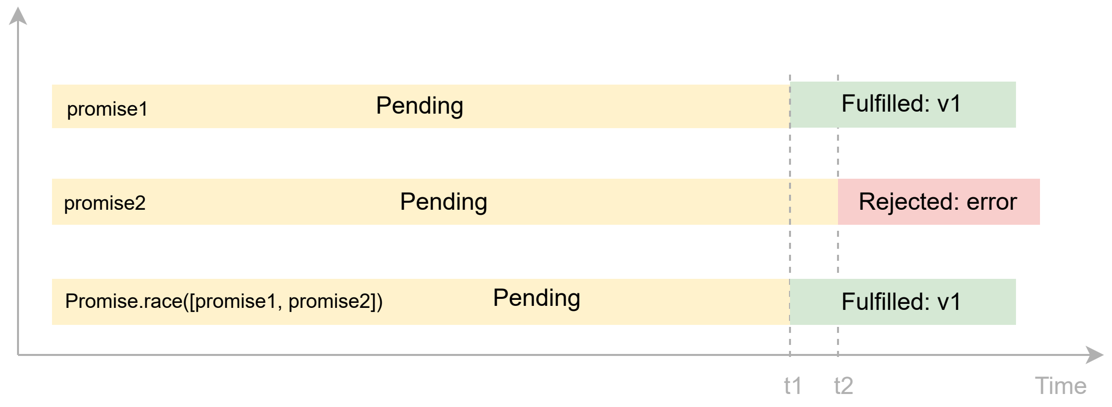
Here's the syntax of the `Promise.race()` method:

```
Promise.race(iterable)
```

In this syntax, the `iterable` is an iterable object 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:

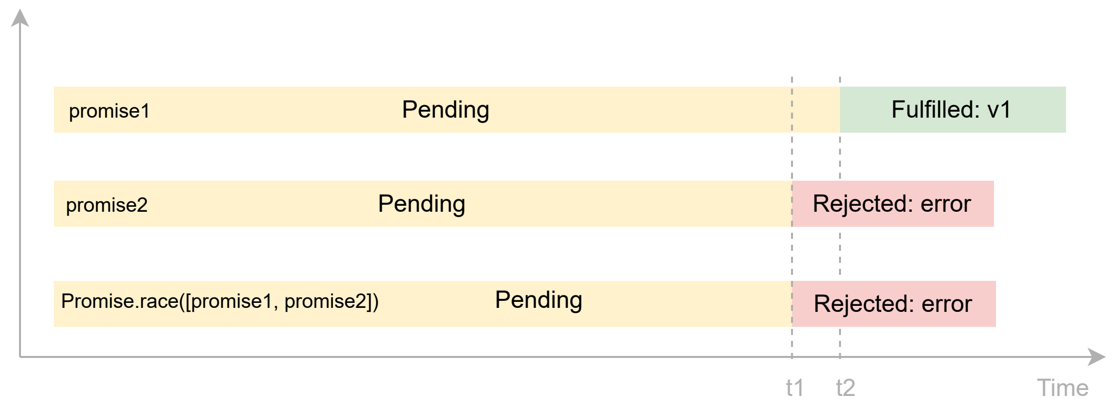


In this diagram:

- 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}`));
```

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}`));
```

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.

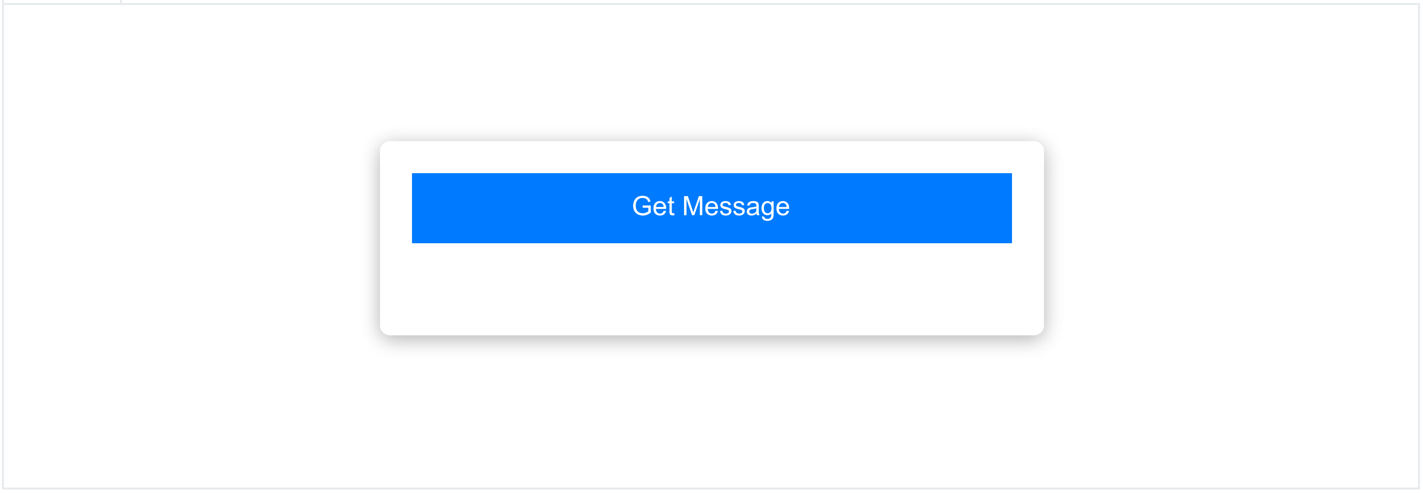
To do this, you can use the `Promise.race()` static method. If a timeout occurs, you show the loading indicator, otherwise, you show the message.

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>
```

Output



To create the loading indicator, we use the CSS animation feature. See the `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);
    });
}
```

Second, develop a function that shows some contents:

```
function showContent(message) {
    document.querySelector('#message').textContent = message;
}
```

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

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

```
const TIMEOUT = 500;

function timeout() {
    return new Promise((resolve, reject) => {
        setTimeout(() => reject(), TIMEOUT);
    });
}
```

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 = '';
}
```

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);
});
```

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

```
Promise.race([getData()
            .then(showContent)
            .then(hideLoadingIndicator), timeout()
        ])
        .catch(showLoadingIndicator);
```

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('');
}
```

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('');
}
```

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.

Quiz