Ivan Jovanovic

Do You Promise?

Posted on February 01, 2017 in JavaScript, Node.JS

Are you still writing your async JavaScript code using callbacks or async library? It's the time to start to **Promises**!

What Is A Promise?

As the word says, Promise is something that can be available now, or in future, or never. When someone promises you something, that can be fulfilled of rejected.

In JavaScript, Promise represents the eventual result of an asynchronous function.

It has 3 different states:

- `fulfilled` - Operation is successful.
- `rejected` - Operation failed.

Why Is Promise Better Than Classic Callbacks?

Promises are the new way of handling asynchronous functions. There are couple reasons why are Promises better than callbacks:

- With the extra callback parameter, we can be confused with what's input and what's the return value
- Callbacks don't handle errors thrown by functions that are used inside them (JSON.parse for example)
- Callback hell (executing functions in sequences)
- When we use callback function, they can depend on function that calls her

Promises give us the ability to write independent functions, that are understandable and that can handle all errors with ease.

Support

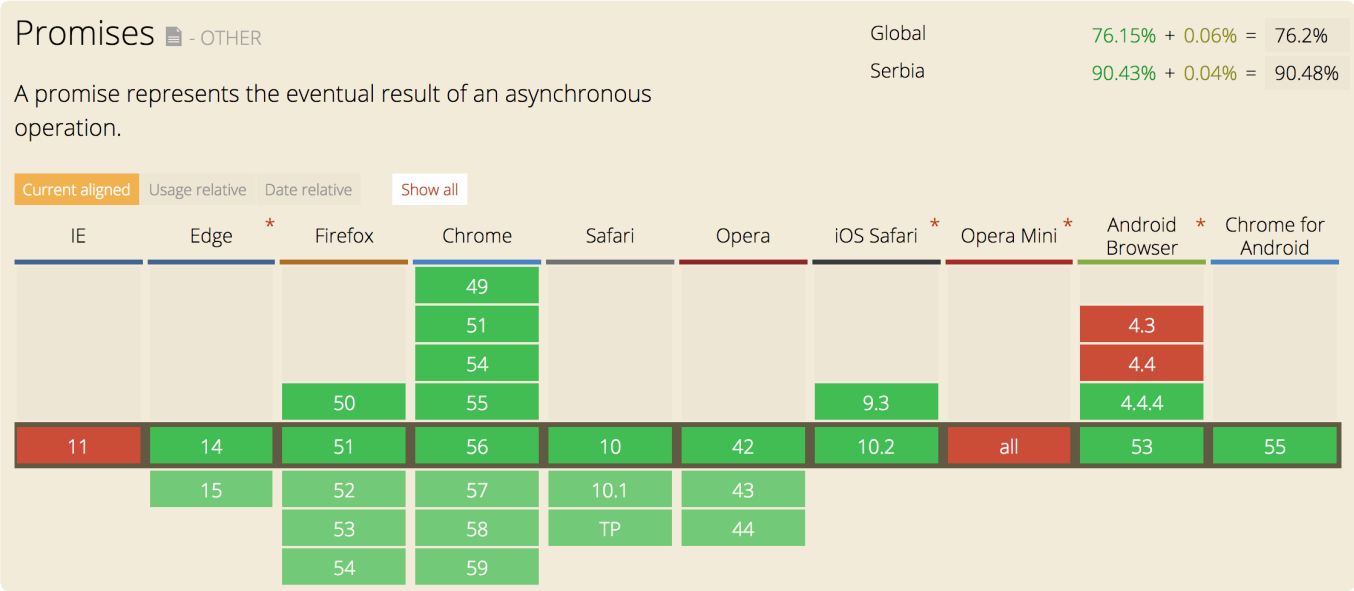
The newer version of browsers and NodeJS natively support Promises. If you want to make sure that Promises will work on older browsers or NodeJS versions, you can use Babel or some modules that imitate Promises:

- **bluebird**
- **Q**
- **when**

- **RSVP.js**

and much more. But since NodeJS and browsers got native support for Promises, you can use them without these modules.

Below, you can see browser support of promises:



As you can see, support is very good. The only problem is Internet Explorer, but that can be handled by using Babel or some other transpiler for ES6 to ES5 code.

Regarding the NodeJS support, Promises don't work only on `0.10.*` versions. But starting from `0.12.18` you can use native NodeJS Promises. If you want to find more about Promises support, you can check **http://node.green/#Promise**

Usage

Using Promises is very simple! The Promise comes in one class that handles everything. Check the example below:

```
new Promise( function(resolve, reject) { ... } );
```

That's all! Just one class that accepts function with two parameters, `resolve` and `reject`. Both of them are the functions that are called when a function finishes its execution. They accept one parameter. `resolve` should accept the value and `reject` should accept the reason why function didn't execute correctly. Check out the example below:

```
function readFile(filename) {
  return new Promise(function (resolve, reject) {
    asyncReadFile(filename, function (error, result) {
        if (error) {
          reject(error);
        } else {
          resolve(result);
        }
    });
  });
}
```

In the above example, we have `readFile` function that should read data from a file, specified in the argument `filename`. This function returns a Promise. Then inside the Promise, we call the async function called `asyncReadFile`. As you know, reading from files is an async process and we need to call some async function for it. After `asyncReadFile` is executed, it calls a callback that `resolves` (on success) or `rejects` (on error), depending on the result of the function.

Below, you can see the code that uses our `readFile` function:

```
readFile('some_file.txt')
  .then(function(result) {
    console.log(result);
```

```
    console.error(error)
});
```

After calling `readFile`, it returns a Promise that has `then` method. `then` accepts one function and passes data inside it, as one argument. That argument is the data that's passed to `resolve` method. In the example above, the `result` should be the text that's read from the file.

Promise also has `catch` method that is called after `reject` and it also accepts one function. Data that's passed inside the function is passed from `reject` function and that's usually the reason for the error.

`catch` will also handle all errors that are thrown in the function, like `try...catch` does.

`then` method should always return promises. That gives us an ability to chain our functions and run code sequentially. See the example:

```
readFile('some_file.txt')
  .then(function(result) {
    return getFirstParagraph(result);
  })
  .then(function(result) {
    return getFirstSentence(result);
  })
  .then(function(result) {
    console.log(result);
  })
  .catch(function(error) {
    console.error(error)
  });
```

transformed to Promise and we will be able to chain it. In the above example, we are first reading some text from the file, then getting the first paragraph, then getting the first sentence and at the end, we are logging the result in the console. Most important is that all errors that occur in the execution of those functions, will be handled with `catch`.

Shorter Way To Use Promises

There is a shorter way to return promise. `Promise` has also `resolve` and `reject` methods inside, so you can just call those functions, like this:

```
Promise.resolve(3)
  .then(function(result) {
    console.log(result); // 3
  });
```

Or

```
Promise.reject(new Error('error'))
  .catch(function(result) {
    console.log(result); // error
  });
```

As you can see, this might be the shorter way to use Promise, especially when you just want to `reject` or `resolve` an async function.

Promise.All And Promise.Race

two methods give us an ability to send the list of Promises and then receive results.

The `all` method accepts an array of functions that return Promises and gives us an array of the results in `then` method. It's resolved when all Promises are resolved. Check the code below:

```
var p1 = Promise.resolve('text');
var p2 = 1234;
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([p1, p2, p3])
  .then(values => {
    console.log(values); // ["text", 1234, 100]
  });
```

Promises also give us an ability to execute functions and then resolve after the first one finishes. For that purpose, we are going to use `race` method from `Promise` class. It accepts an array of Promises. Check below how it works:

```
var p1 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 2000, 'one');
});
var p2 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 1000, 'two');
});

Promise.race([p1, p2])
  .then(function(value) {
```

```
});
```

Promises In The ES6 Way!

After ES6 came, we got arrow functions. Arrow functions are awesome for making code shorter and readable. I am going to show you how to use arrow functions to make our first example much shorter.

This is how it looks like now:

```
function readFile(filename) {
  return new Promise(function (resolve, reject) {
    asyncReadFile(filename, function (error, result) {
        if (error) {
          reject(error);
        } else {
          resolve(result);
        }
    });
  });
}

readFile('some_file.txt')
  .then(function(result) {
    console.log(result);
  })
  .catch(function(error) {
    console.error(error)
  });
```

This is how it looks when we add arrow functions:

```
  asyncReadFile(filename, (error, result) => {
    if (error) {
      reject(error);
    } else {
      resolve(result);
    }
  });
});

readFile('some_file.txt')
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

We have saved couple lines and, in my opinion, made code nicer. It's your choice whether you are going to use arrow functions or not □

Promisify

Many Promise libraries have methods that transform callback-based functions to function that return Promises. Those methods are usually called `promisify`. An example for `bluebird` can be found here: **http://bluebirdjs.com/docs/api/promise.promisify.html**

If you're not a big fan of Promise libraries, there is one awesome library that can convert your callback-based function to function that returns native Promises. It's called **es6-promisify**. Here is an example:

```
const promisify = require("es6-promisify");
const fs = require("fs");

// Convert the stat function
```

```
// Now usable as a promise!
stat("example.txt").then(function (stats) {
  console.log("Got stats", stats);
}).catch(function (err) {
  console.error("Yikes!", err);
});
```

So cool! You can convert any function from NodeJS API to promise-based function!

Conclusion

Promises are awesome and if you aren't, you should start using them right now!

If you have an opinion about Promises, please leave a comment, I would love to hear what you think!

Next story

AWS Serverless Stack - API Gateway, Lambda And DynamoDB

Previous story

Building Serverless API With Claudia API Builder

Ivan is senior software engineer, JS advocate, mentor and team lead. Trying to explain why is JavaScript so awesome and why should everyone learn it!

Going up?