// Tutorial //

Understanding JavaScript Promises

Published on September 21, 2020 · Updated on September 15, 2020

JavaScript

By Jecelyn Yeen
Developer and author at DigitalOcean.

Introduction

Javascript Promises can be challenging to understand. Therefore, I would like to write down the way I understand promises.

Understanding Promises

A Promise in short:

“Imagine you are a **kid**. Your mom **promises** you that she’ll get you a **new phone** next week.”

POPULAR TOPICS ▼

Ubuntu
Linux Basics
JavaScript
React
Python
Security
Apache
MySQL
Databases
Docker
Kubernetes
Ebooks

You *don't know* if you will get that phone until next week. Your mom can *really buy* you a brand new phone, or *she doesn't*.

That is a **promise**. A promise has three states. They are:

1. Pending: You *don't know* if you will get that phone
2. Fulfilled: Mom is *happy*, she buys you a brand new phone
3. Rejected: Mom is *unhappy*, she doesn't buy you a phone

Creating a Promise

Let's convert this to JavaScript.

```
// ES5: Part 1

var isMomHappy = false;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // fulfilled
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }

    }
);
```

The code is quite expressive in itself.

Below is how a promise syntax looks normally:

```
// promise syntax look like this
new Promise(function (resolve, reject) { ... } );
```

Browse all topic tags

ALL TUTORIALS →

QUESTIONS ▼

Q&A

Ask a question

DigitalOcean Product Docs

DigitalOcean Support

EVENTS ▼

Tech Talks

Hacktoberfest

Deploy

GET INVOLVED ▼

Community Newsletter

Hollie's Hub for Good

Write for DOnations

Community tools and integrations

Hatch Startup program

CREATE YOUR FREE COMMUNITY ACCOUNT! →

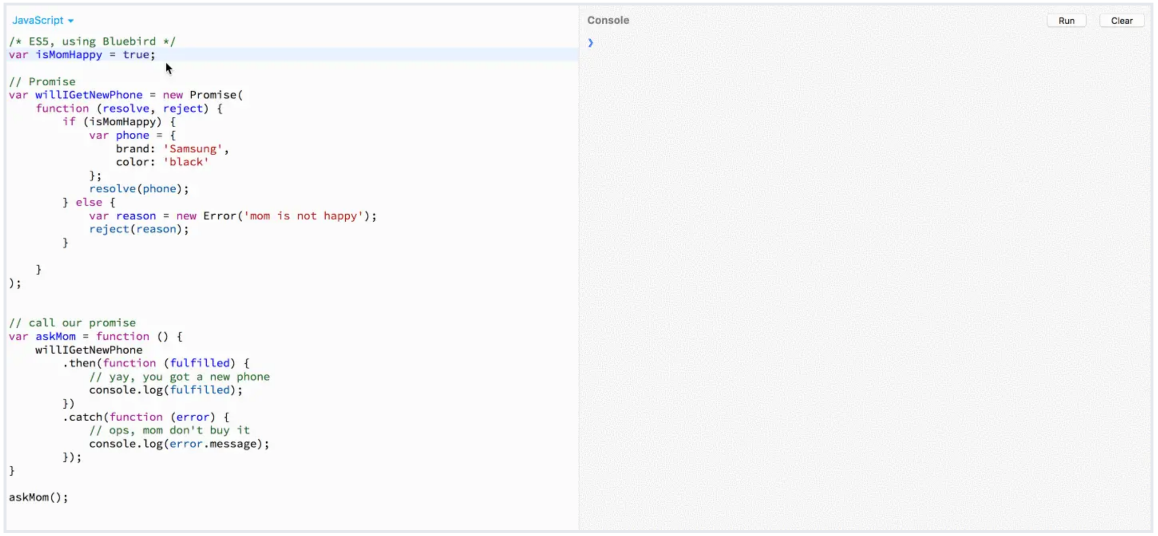
Consuming Promises

Now that we have the promise, let's consume it:

```
// ES5: Part 2

var willIGetNewPhone = ... // continue from part 1

// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
             // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {
            // oops, mom didn't buy it
            console.log(error.message);
             // output: 'mom is not happy'
        });
};

askMom();
```

Let's run the example and see the result!

Demo: https://jsbin.com/nifocu/1/edit?js,console



```
/* ES5, using Bluebird */
var isMomHappy = true;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            var reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);


// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
        })
        .catch(function (error) {
            // ops, mom don't buy it
            console.log(error.message);
        });
}

askMom();
```

Chaining Promises

Promises are chainable.

Let's say you, the kid, **promise** your friend that you will **show them** the new phone when your mom buys you one.

That is another promise. Let's write it!

```
// ES5

// 2nd promise
var showOff = function (phone) {
    return new Promise(
        function (resolve, reject) {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};
```

Notes: We can shorten the above code by writing as below:

```
// shorten it

// 2nd promise
var showOff = function (phone) {
    var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

    return Promise.resolve(message);
};
```

Let’s chain the promises. You, the kid, can only start the `showOff` promise after the `willIGetNewPhone` promise.

```
// call our promise
var askMom = function () {
    willIGetNewPhone
    .then(showOff) // chain it here
    .then(function (fulfilled) {
            console.log(fulfilled);
         // output: 'Hey friend, I have a new black Samsung phone.'
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
         // output: 'mom is not happy'
        });
};
```

That is how you can chain the promise.

Promises are Asynchronous

Promises are asynchronous. Let’s log a message before and after we call the promise.

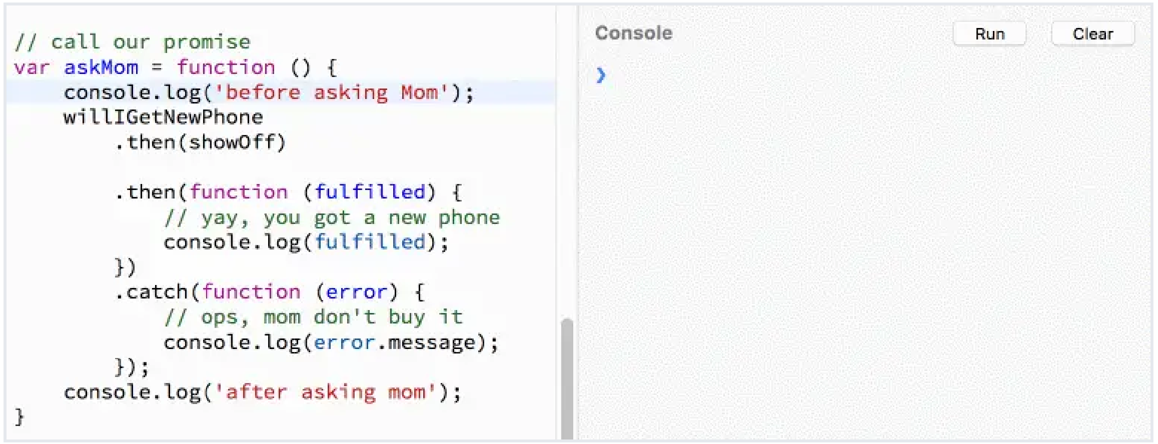
```
// call our promise
var askMom = function () {
    console.log('before asking Mom'); // log before
    willIGetNewPhone
        .then(showOff)
        .then(function (fulfilled) {
            console.log(fulfilled);
        })
        .catch(function (error) {
            console.log(error.message);
        });
    console.log('after asking mom'); // log after
}
```

What is the sequence of expected output? You might expect:

```
1. before asking Mom
2. Hey friend, I have a new black Samsung phone.
3. after asking mom
```

However, the actual output sequence is:

```
1. before asking Mom
2. after asking mom
3. Hey friend, I have a new black Samsung phone.
```



```
// call our promise
var askMom = function () {
    console.log('before asking Mom');
    willIGetNewPhone
        .then(showOff)

        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
        })
        .catch(function (error) {
            // ops, mom don't buy it
            console.log(error.message);
        });
    console.log('after asking mom');
}
```

You wouldn't stop playing while waiting for your mom's promise (the new phone). That's something we call **asynchronous**: the code will run without blocking or waiting for the result. Anything that needs to wait for a promise to proceed is put in `.then`.

Here is the full example in ES5:

```
// ES5: Full example

var isMomHappy = true;

// Promise
```

```
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // fulfilled
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }

    }
);

// 2nd promise
var showOff = function (phone) {
    var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

    return Promise.resolve(message);
};

// call our promise
var askMom = function () {
    willIGetNewPhone
    .then(showOff) // chain it here
    .then(function (fulfilled) {
            console.log(fulfilled);
            // output: 'Hey friend, I have a new black Samsung phone.'
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
            // output: 'mom is not happy'
        });
};

askMom();
```

Promises in ES5, ES6/2015, ES7/Next

ES5 - Majority browsers

The demo code is workable in ES5 environments (all major browsers + NodeJs) if you include Bluebird promise library. It's because ES5 doesn't support promises out of the box. Another famous promise library is Q by Kris Kowal.

ES6 / ES2015 - Modern browsers, NodeJs v6

The demo code works out of the box because ES6 supports promises natively. In addition, with ES6 functions, we can further simplify the code with an arrow function and use `const` and `let`.

Here is the full example in ES6 code:

```
//_ ES6: Full example_

const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => { // fat arrow
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);

// 2nd promise
const showOff = function (phone) {
    const message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';
    return Promise.resolve(message);
};

// call our promise
const askMom = function () {
    willIGetNewPhone
        .then(showOff)
        .then(fulfilled => console.log(fulfilled)) // fat arrow
        .catch(error => console.log(error.message)); // fat arrow
};
```

```
askMom();
```

Note that all the `var` are replaced with `const`. All of the `function(resolve, reject)` have been simplified to `(resolve, reject) =>`. There are a few benefits that come with these changes.

ES7 - Async/Await

ES7 introduced `async` and `await` syntax. It makes the asynchronous syntax easier to understand, without the `.then` and `.catch`.

Rewrite our example with ES7 syntax:

Copy

```
// ES7: Full example
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => {
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);

// 2nd promise
async function showOff(phone) {
    return new Promise(
        (resolve, reject) => {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};

// call our promise in ES7 async await style
```

```
async function askMom() {
    try {
        console.log('before asking Mom');

        let phone = await willIGetNewPhone;
        let message = await showOff(phone);

        console.log(message);
        console.log('after asking mom');
    }
    catch (error) {
        console.log(error.message);
    }
}

// async await it here too
(async () => {
    await askMom();
})();
```

Promises and When to Use Them

Why do we need promises? How did the world look before promises? Before answering these questions, let's go back to the fundamentals.

Normal Function VS Async Function

Let's take a look at these two examples. Both examples perform the addition of two numbers: one adds using normal functions, and the other adds remotely.

Normal Function to Add Two Numbers

Copy

```
// add two numbers normally

function add (num1, num2) {
    return num1 + num2;
}

const result = add(1, 2); // you get result = 3 immediately
```

Async Function to Add Two numbers

```
// add two numbers remotely

// get the result by calling an API
const result = getAddResultFromServer('http://www.example.com?num1=1&num2=2');
// you get result  = "undefined"
```

If you add the numbers with the normal function, you get the result immediately. However, when you issue a remote call to get the result, you need to wait, and you can't get the result immediately.

You don't know if you will get the result because the server might be down, slow in response, etc. You don't want your entire process to be blocked while waiting for the result.

Calling APIs, downloading files, and reading files are among some of the usual async operations that you'll perform.

You do not need to use promises for an asynchronous call. Prior to promises, we used callbacks. Callbacks are a function you call when you get the return result. Let's modify the previous example to accept a callback.

```
// add two numbers remotely
// get the result by calling an API

function addAsync (num1, num2, callback) {
    // use the famous jQuery getJSON callback API
    return $.getJSON('http://www.example.com', {
        num1: num1,
        num2: num2
    }, callback);
}

addAsync(1, 2, success => {
    // callback
    const result = success; // you get result = 3 here
});
```

Subsequent Async Action

Instead of adding the numbers one at a time, we want to add three times. In a normal function, we would do this:-

```
// add two numbers normally
```

```
let resultA, resultB, resultC;

 function add (num1, num2) {
    return num1 + num2;
}

resultA = add(1, 2); // you get resultA = 3 immediately
resultB = add(resultA, 3); // you get resultB = 6 immediately
resultC = add(resultB, 4); // you get resultC = 10 immediately

console.log('total' + resultC);
console.log(resultA, resultB, resultC);
```

This is how this looks with callbacks:

```
// add two numbers remotely
// get the result by calling an API

let resultA, resultB, resultC;

function addAsync (num1, num2, callback) {
    // use the famous jQuery getJSON callback API
        // https://api.jquery.com/jQuery.getJSON/
    return $.getJSON('http://www.example.com', {
        num1: num1,
        num2: num2
    }, callback);
}

addAsync(1, 2, success => {
    // callback 1
    resultA = success; // you get result = 3 here

    addAsync(resultA, 3, success => {
        // callback 2
        resultB = success; // you get result = 6 here

        addAsync(resultB, 4, success => {
            // callback 3
            resultC = success; // you get result = 10 here

            console.log('total' + resultC);
            console.log(resultA, resultB, resultC);
        });
    });
});
```

Demo: https://jsbin.com/barimo/edit?html,js,console

This syntax is less user-friendly due to the deeply nested callbacks.

Avoiding Deeply Nested Callbacks

Promises can help you avoid deeply nested callbacks. Let's look at the promise version of the same example:

```
// add two numbers remotely using observable

let resultA, resultB, resultC;

function addAsync(num1, num2) {
    // use ES6 fetch API, which return a promise
        // What is .json()? https://developer.mozilla.org/en-US/docs/Web/API/Body/json
    return fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
        .then(x => x.json());
}

addAsync(1, 2)
    .then(success => {
        resultA = success;
        return resultA;
    })
    .then(success => addAsync(success, 3))
    .then(success => {
        resultB = success;
        return resultB;
    })
    .then(success => addAsync(success, 4))
    .then(success => {
        resultC = success;
        return resultC;
    })
    .then(success => {
        console.log('total: ' + success)
        console.log(resultA, resultB, resultC)
    });
```

With promises, we flatten the callback with `.then`. In a way, it looks cleaner because there is no callback nesting. With ES7 `async` syntax, you could further enhance this example.

Observables

Before you settle down with promises, there is something that has come about to help you deal with async data called `Observables`.

Let’s look at the same demo written with Observables. In this example, we will use RxJS for the observables.

```
let Observable = Rx.Observable;
let resultA, resultB, resultC;

function addAsync(num1, num2) {
    // use ES6 fetch API, which return a promise
    const promise = fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
        .then(x => x.json());

    return Observable.fromPromise(promise);
}

addAsync(1,2)
  .do(x => resultA = x)
  .flatMap(x => addAsync(x, 3))
  .do(x => resultB = x)
  .flatMap(x => addAsync(x, 4))
  .do(x => resultC = x)
  .subscribe(x => {
    console.log('total: ' + x)
    console.log(resultA, resultB, resultC)
  });
```

Observables can do more interesting things. For example, `delay` add function by `3 seconds` with just one line of code or retry so you can retry a call a certain number of times.

```
...

addAsync(1,2)
  .delay(3000) // delay 3 seconds
  .do(x => resultA = x)
  ...
```

You may read one of my RxJs posts here.

Conclusion

Familiarizing yourself with callbacks and promises is important. Understand them and use them. Don’t worry about Observables just yet. All three can factor into your development depending on the situation.

About the authors

Jecelyn Yeen Author
Developer and author at DigitalOcean.

Comments

2 Comments

B I U H1 H2 H3 1.

Leave a comment...

Login to Comment

chandragie • March 10, 2021

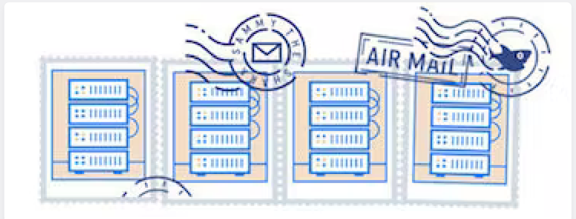
This comment has been deleted

Reply

Artur Zhilinskiy • May 26, 2021

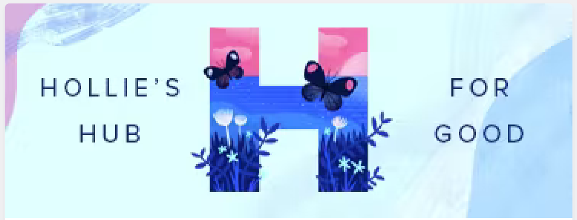
Thank you for the explanation! It really helped me to get basic knowledge how to use promises and modern async/await keywords. Thank you a lot!

Reply



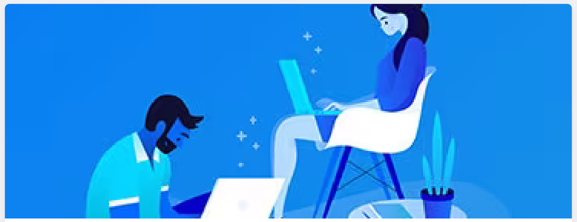
GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a Newsletter.



HOLLIE'S HUB FOR GOOD

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.



BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.

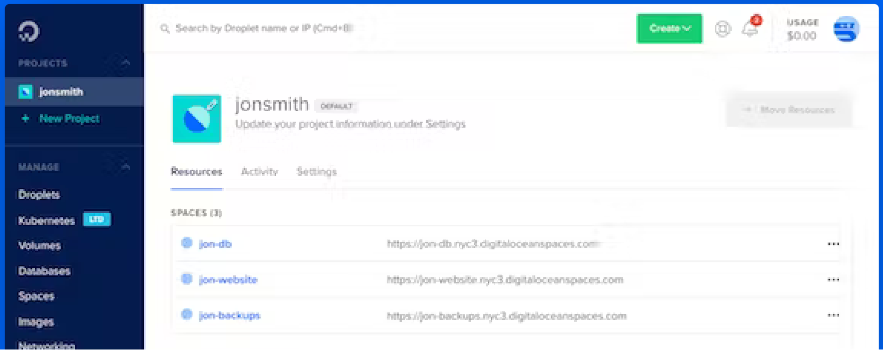
Featured on Community Kubernetes Course Learn Python 3 Machine Learning in Python Getting started with Go Intro to Kubernetes

DigitalOcean Products Virtual Machines Managed Databases Managed Kubernetes Block Storage Object Storage Marketplace VPC Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn More



jonsmith

Update your project information under Settings

SPACES (3)

jon-db	https://jon-db.nyc3.digitaloceanspaces.com
jon-website	https://jon-website.nyc3.digitaloceanspaces.com
jon-backups	https://jon-backups.nyc3.digitaloceanspaces.com

Company

About
Leadership
Blog
Careers
Partners
Referral Program
Press
Legal
Security & Trust Center

Products

Pricing
Products Overview
Droplets
Kubernetes
Managed Databases
Spaces
Marketplace
Load Balancers
Block Storage
API Documentation
Documentation
Release Notes

Community

Tutorials
Q&A
Tools and Integrations
Tags
Write for DigitalOcean
Presentation Grants
Hatch Startup Program
Shop Swag
Research Program
Open Source
Code of Conduct

Contact

Get Support
Trouble Signing In?
Sales
Report Abuse
System Status
Share your ideas