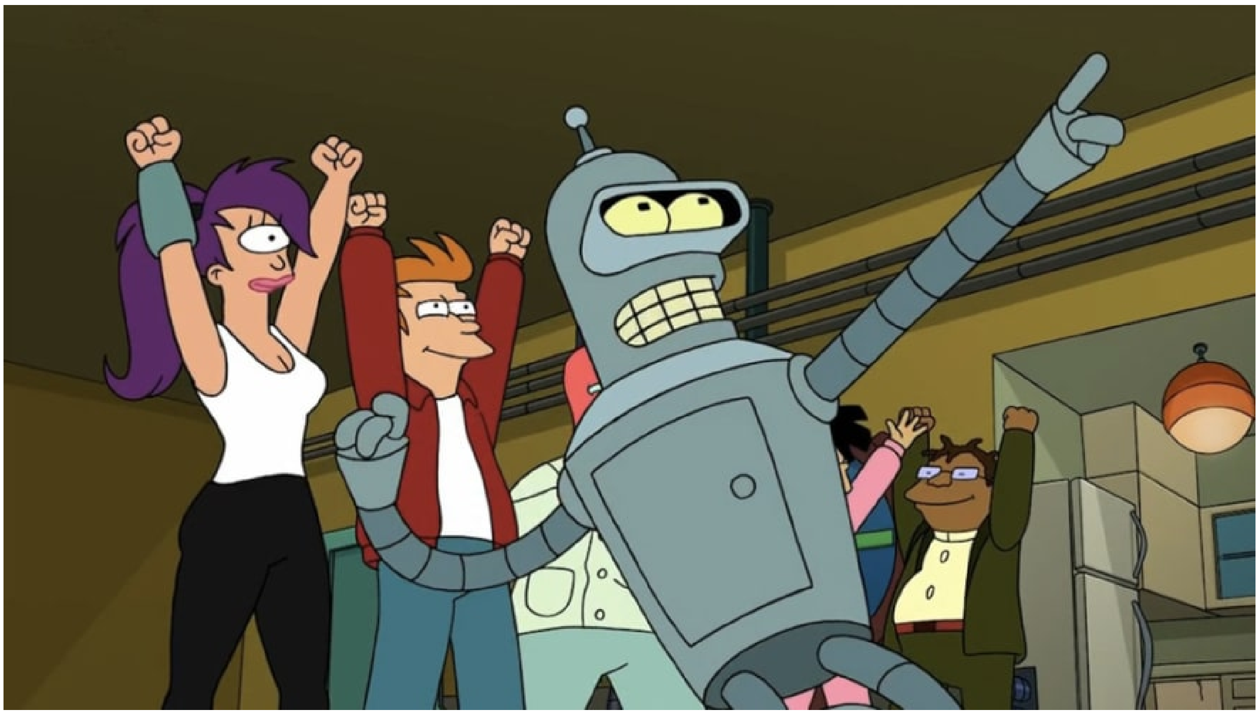
9 Promising Promise Tips

Kushan Joshi

Feb 21 *Updated on May 22, 2018*

#javascript **#promises** **#webdev**

Promises are great to work with! Or so does your fellow developer at work says.



This article would give you to the point no bullshit tips on how to improve your relationship with the Promises.

1. You can return a Promise inside a .then

Let me make the most important tip standout

> **Yes! you can return a Promise inside a .then**

Also, the returned promise is automatically unwrapped in the next `.then`

```
.then(r => {
    return serverStatusPromise(r); // this is a promise of { statusCode: 20
})
.then(resp => {
    console.log(resp.statusCode); // 200; notice the automatic unwrapping of
})
```

2. You create a new Promise

search

If you are familiar with the dot chaining style of javascript you would feel at home. But for a newcomer this might not be obvious.

In promises whenever you `.then` or `.catch` you are creating a new Promise. This promise is a composition of the promise you just chained and the `.then` / `.catch` you just attached.

Let us look at an example:

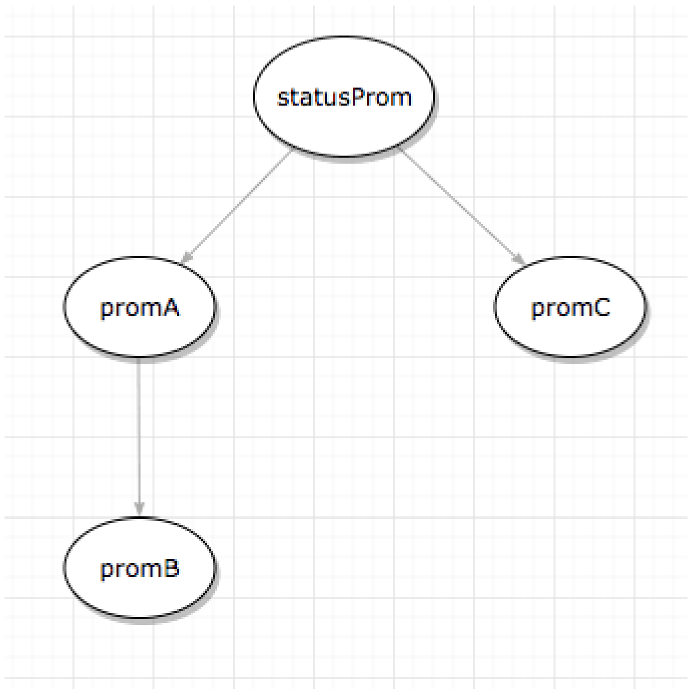
```
var statusProm = fetchServerStatus();

var promA = statusProm.then(r => (r.statusCode === 200 ? "good" : "bad"));

var promB = promA.then(r => (r === "good" ? "ALL OK" : "NOTOK"));

var promC = statusProm.then(r => fetchThisAnotherThing());
```

The relationship of above promises can be described neatly in a flow chart:



The important thing to note here is that `promA`, `promB` and `promC` are all different promises but related.

I like to think of `.then`ing as a big massive plumbing where water will stop flowing to the children when the parent node malfunctions. For eg. if `promB` fails, no other node will be affected but if `statusProm` fails all the nodes will be affected i.e. `rejected`.

3. A Promise is resolved/rejected for EVERYONE

I find this as one of the most important thing that makes promises great to work with. To put it simply, if a promise is get notified when it gets `resolved/rejected`.

> This also means nobody can ever mutate your promise, so please feel free to pass it around without worrying.

```
function yourFunc() {
  const yourAwesomeProm = makeMeProm();

  yourEvilUncle(yourAwesomeProm); // rest assured you promise will work, reg

  return yourAwesomeProm.then(r => importantProcessing(r));
}

function yourEvilUncle(prom) {
  return prom.then(r => Promise.reject("destroy!!")); // your evil uncle
}
```

In the example above you can see that promise by design makes it difficult for anyone to do nefarious things. As I said above, `Keep calm and pass the promise around`

4. Promise Constructor is not the solution

I have seen fellow developers exploiting the constructor style everywhere, thinking they are doing it the promise way. But this is a big lie, the actual reason is that the constructor API is very similar to the good old callback API and old habits die hard.

To actually take a step forward and move away from callbacks, you need to carefully minimize the amount of Promise constructor's you use.

Let us jump to the actual use case of a `Promise constructor`:

```
return new Promise((res, rej) => {
  fs.readFile("/etc/passwd", function(err, data) {
    if (err) return rej(err);
    return res(data);
  });
});
```

`Promise constructor` should **only be used when you want to convert a callback to promise**.
Once you have grasped this beautiful way of creating promises, it can become really tempting to use it at other places which are already promisified!

Let us look at a redundant `Promise constructor`

☠️Wrong

```
return new Promise((res, rej) => {
    var fetchPromise = fetchSomeData(.....);
    fetchPromise
        .then(data => {
            res(data); // wrong!!!
        })
```

Correct

```
return fetchSomeData(...); // when it looks right, it is right!
```

Wrapping a promise with `Promise constructor` is just **redundant and defeats the purpose of the promise itself**.

Protip

If you are a **nodejs** person, I recommend checking out util.promisify. This tiny thing helps you convert your node style callback into promises.

```
const {promisify} = require('util');
const fs = require('fs');

const readFileAsync = promisify(fs.readFile);

readFileAsync('myfile.txt', 'utf-8')
  .then(r => console.log(r))
  .catch(e => console.error(e));
```

5. Use Promise.resolve

Javascript provides `Promise.resolve`, which is a short hard for writting something like this:

```
var similarProm = new Promise(res => res(5));
// ^^ is equivalent to
var prom = Promise.resolve(5);
```

This has multiple use cases and my favourite one being able to convert a regular (sync) javascript object into a promise.

```
// converting a sync function to an async function
function foo() {
  return Promise.resolve(5);
}
```

You can also use it as a safety wrapper around a value which you are unsure whether it is a promise or regular value.

```
function goodProm(maybePromise) {
  return Promise.resolve(maybePromise);
}

goodProm(5).then(console.log); // 5

goodProm(Promise.resolve(Promise.resolve(5))).then(console.log); // 5, Notic
```

6.Use Promise.reject

Javascript also provides `Promise.reject`, which is a short hand for this

```
var rejProm = new Promise((res, reject) => reject(5));

rejProm.catch(e => console.log(e)) // 5
```

One of my favourite use case is rejecting early with `Promise.reject`.

```
function foo(myVal) {
    if (!mVal) {
        return Promise.reject(new Error('myVal is required'))
    }
    return new Promise((res, rej) => {
        // your big callback to promie conversion!
    })
}
```

In simple words, use `Promise.reject` wherever you want to reject promise.

In the example below I use inside a `.then`

```
.then(val => {
  if (val != 5) {
    return Promise.reject('Not Good');
  }
})
.catch(e => console.log(e)) // Not Good
```

Note: You can put any value inside `Promise.reject` just like `Promise.resolve`. The reason you often find `Error` in a rejected promise is that it is primarily used for throwing an async error.

Javascript provides Promise.all, which is a shorthand for …. well I can't come up with this 😁.

In a pseudo algorithm, `Promise.all` can be summarised as

```
Takes an array of promises

    then waits for all of them to finish

    then returns a new Promise which resolves into an Array

    catches if even a single fails/rejects.
```

The following example shows when all the promises resolve:

```
var prom1 = Promise.resolve(5);
var prom2 = fetchServerStatus(); // returns a promise of {statusCode: 200}

Proimise.all([prom1, prom2])
.then([val1, val2] => { // notice that it resolves into an Array
    console.log(val1); // 5
    console.log(val2.statusCode); // 200
})
```

This one shows when one of them fails:

```
var prom1 = Promise.reject(5);
var prom2 = fetchServerStatus(); // returns a promise of {statusCode: 200}

Proimise.all([prom1, prom2])
```

```
    console.log(val2.statusCode);
})
.catch(e =>  console.log(e)) // 5, jumps directly to .catch
```

Note: `Promise.all` is smart! In case of a rejection, it doesn't wait for all of the promises to complete!. Whenever any promise rejects, it immediately aborts without waiting for other promises to complete.

😎**Protip**

`Promise.all` does not provide a way to execute promises in batches(concurrency), since by design promises are executed the moment they are created. If you want to control the execution, I recommend trying out Bluebird.map. (*Thanks* Mauro *for this tip.*)

8. Do not fear the rejection *OR*

Do not append redundant `.catch` after every .then

How often do we fear errors being gobbled up somewhere in between?

To overcome this fear, here's a very simple tip:

> **Make the rejection handling the problem of the parent function.**

and all the promise rejections trickle down to it.

Do not fear writing something like this

```
return fetchSomeData(...);
```

Now if you do want to handle the rejection in your function, decide whether you want to resolve things or continue the rejection.

💘 Resolving a rejection

Resolving rejection is simple, in the `.catch` whatever you return would be assumed to be resolved. However there is a catch (pun intended), if you return a `Promise.reject` in a `.catch` the promise will be rejected.

```
.then(() => 5.length) // <-- something wrong happenned here
.catch(e => {
        return 5;  // <-- making javascript great again
})
.then(r => {
    console.log(r); // 5
})
.catch(e => {
    console.error(e); // this function will never be called :)
})
```

💔Rejecting a Rejection

To reject a rejection is simple, **don't do anything.** As I said not, parent functions have a better way to handle the rejection than your current function.

The important thing to remember is, once you write a catch it means you are handling the error. This is similar to how sync `try/catch` works.

If you do want to intercept a rejection: (I highly recommend not!)

```
.then(() => 5.length) // <-- something wrong happenned here
.catch(e => {
  errorLogger(e); // do something impure
  return Promise.reject(e); // reject it, Yes you can do that!
})
.then(r => {
    console.log(r); // this .then (or any subsequent .then) will never be ca
})
.catch(e => {
    console.error(e); //<-- it becomes this catch's problem
})
```

The fine line between .then(x,y) and then(x).catch(x)
The `.then` accepts a second callback parameter which can also be used to handle errors. This might look similar to doing something like `then(x).catch(x)`, but both these error handlers differ in which error they catch.

I will let the following example speak for itself.

```
  return Promise.reject(new Error('something wrong happened'));
}).catch(function(e) {
  console.error(e); // something wrong happened
});


.then(function() {
  return Promise.reject(new Error('something wrong happened'));
}, function(e) { // callback handles error coming from the chain above the c
   console.error(e); // no error logged
});
```

The `.then(x,y)` comes really handy when you want to handle an error coming from the promise you are `.then`ing and not want to handle from the `.then` you just appended to the promise chain.

Note: 99.9% of the times you are better off using the simpler `then(x).catch(x)`.

9. Avoid the .then hell

This tip is pretty simple, try to avoid the `.then` inside a `.then` or `.catch`. Trust me it can be avoided more often than you think.

☠️Wrong

```
request(opts)
.catch(err => {
  if (err.statusCode === 400) {
```

```
              .catch(err2 => console.error(err2))
  }
})
```

Correct

```
request(opts)
.catch(err => {
  if (err.statusCode === 400) {
    return request(opts);
  }
  return Promise.reject(err);
})
.then(r => r.text())
.catch(err => console.erro(err));
```

Sometimes it does happen that we need multiple variables in a `.then` scope and there is no option but to create another `.then` chain.

```
.then(myVal => {
    const promA = foo(myVal);
    const promB = anotherPromMake(myVal);
    return promA
          .then(valA => {
              return promB.then(valB => hungryFunc(valA, valB)); // very hur
          })
})
```

I recommend using the ES6 destructuring power mixed with `Promise.all` to the rescue!

```
    const promA = foo(myVal);
    const promB = anotherPromMake(myVal);
    return Promise.all([prom, anotherProm])
})
.then(([valA, valB]) => {   // putting ES6 destructing to good use
    console.log(valA, valB) // all the resolved values
    return hungryFunc(valA, valB)
})
```

Note: You can also use async/await to solve this problem if your node/browser/boss/conscious allows!

I really hope this article helped you in understanding Promises.

Please check out my previous blog posts.

- A toddlers guide to memory leaks in Javascript
- Understanding Default Parameters in Javascript

If you ♡ this article, please share this article to spread the words.

Reach out to me on Twitter @kushan2020.

❤️ 79 🦄 12 🔖 45 ···

Kushan Joshi + FOLLOW

The world would be a better place without backends.

kushan2020 kepta github.com/kepta

Add to the discussion

PREVIEW SUBMIT

Mauro Gabriel Titimoli Feb 23

Hi Kushan,

Thanks for having taken the time to write this, great post.

Just a couple of things that they are not correct:

1. When you wrote the pseudo code to describe Promise.all, you wrote: then runs all of them simultaneously; which it's not true, as promises given to Promise.all have already started
2. The "correct" example you wrote for (9), is not entirely OK, it should be like follows:

```
request(opts)
.catch(err =>
  err.statusCode === 400
    ? request(opts)
    : Promise.reject(err)) // you missed rejecting in your example
.then(r => r.text())
.catch(err => console.error(err));
```

2 REPLY

Kushan Joshi Feb 23

Hey Mauro,
Thanks for giving the feedback.

1. Yes I agree, I think I tried to convey that it doesn't do any sort of batch execution, but then promises by design run the moment they are created. I think I should have an additional tip which clarifies about how promises run.

REPLY

Chris West

Feb 22

Thanks so much for the post. I always love learning more about JavaScript. In fact, I recently created a topsites site that I think this link belongs on. It is at ciphly.com?languages=JavaScript. Please consider adding this post and other informative ones like it!

1 REPLY

Bruno Scopelliti

Feb 23

Hi, since we're talking of Promise, these days I'm publishing a series of videos about how to build a promise polyfill, with the stated goal of making clear how promise works under the hood.
If this sounds as something interesting, here's the first video: youtube.com/watch?v=E_p-PVNqhZE

1 REPLY

Ben Halpern

Hey there, we see you aren't signed in. (Yes you, the reader. This is a fake comment.)

Please consider creating an account on dev.to. It literally takes a few seconds and ***we'd appreciate the support so much***. ♡

Plus, no fake comments when you're signed in. 🙃

JOIN THE DEV COMMUNITY

Anna Rankin

Apr 2

Thanks so much for this! I never knew about `Promise.reject` / `Promise.resolve` . Awesome :D

1 REPLY

Yujin

Feb 27

thanks for this post! Gave me some idea how to understand Promises on open-source projects :D

That is a very informative post on Promises.
Thanks for writing :)

2 REPLY

happy julien Mar 19

Thank you very much... You great

1 REPLY

code of conduct - report abuse

Classic DEV Post from Apr 19

Give me your best programming haiku

Ben Halpern

...

49 33

READ POST SAVE FOR LATER

From one of our Community Sponsors

Under the Hood of the Most Powerful Video JavaScript API

The JW Player Team

In this article, our goal is to demonstrate how to leverage our JavaScript API effectively to deliver a better video experience on your website through code walkthroughs & demos. We'll then wrap up with some details under the hood of JW Player, explaining how we're the fastest player on the web.

36 3

Another Post You Might Like

Responsive Windows Login page UI Tutorial— Part 2

Muna Mohamed

Using HTML and SCSS

54 2

READ POST SAVE FOR LATER

What is the best, Flutter, Ionic or react native?
moumni - Jun 1

Building a Pomodoro Timer with Vue.js on CodePen
Tori Pugh - May 31

Play With the React 🐘Router
Sai gowtham - Jun 1

Under the Hood of the Most Powerful Video JavaScript API
The JW Player Team - Jun 1

Home About Sustaining Membership Privacy Policy Terms of Use

Contact Code of Conduct The DEV Community copyright 2018