JavaScript Promises: A Tutorial with Examples

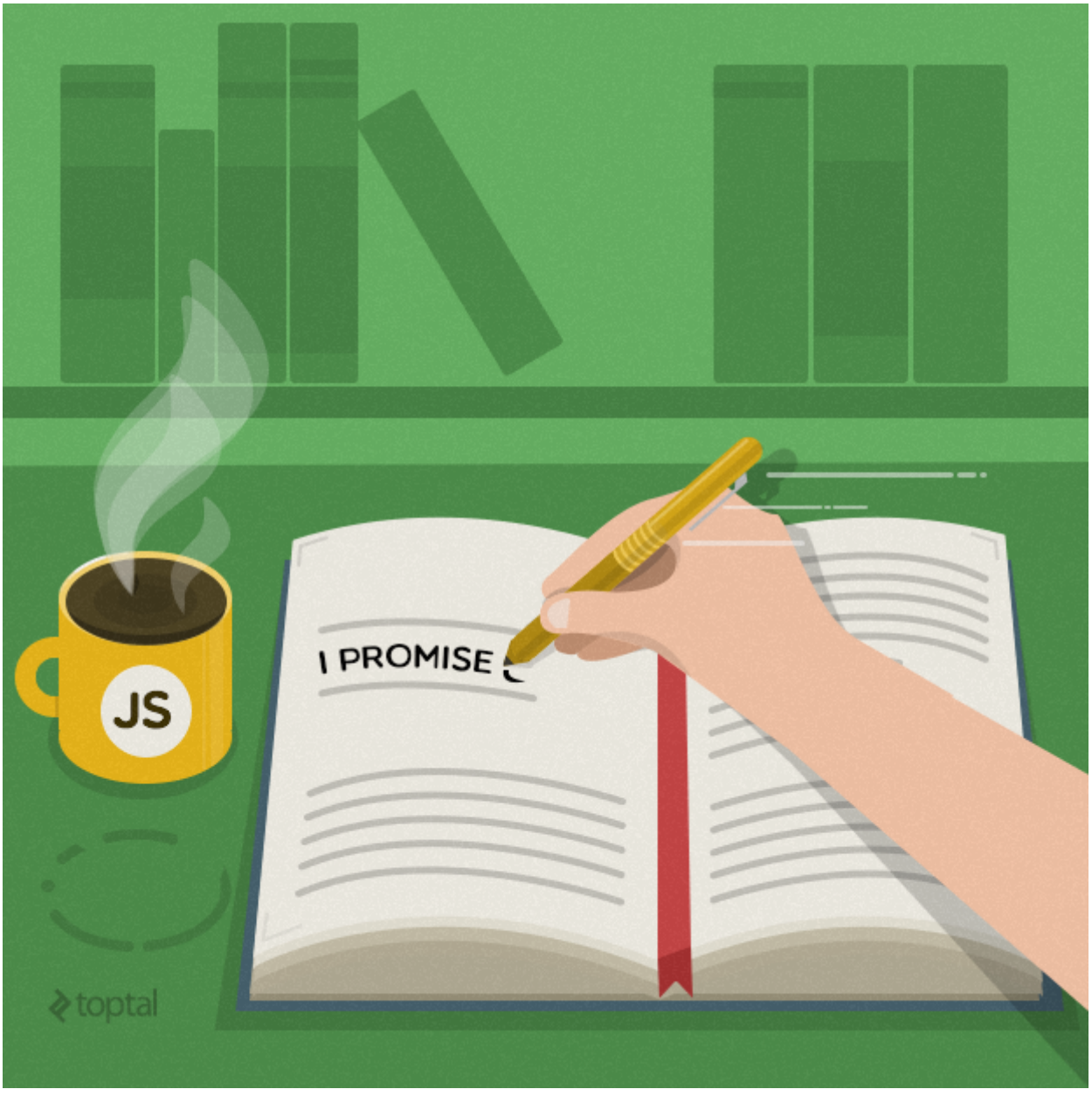
View all articles

by **Balint Erdi** - Software engineer @ Toptal

#BackEnd #FrontEnd #JavaScript #Promises #PromisesAPlus #RSVP

Promises are a hot topic in JavaScript development circles, and you should definitely get acquainted with them. They are not easy to wrap your head around; it can take a few tutorials, examples, and a decent amount of practice to comprehend them.

My aim with this tutorial is to help you understand JavaScript Promises, and nudge you to practice using them more. I will be explaining what promises are, what problems they solve, and how they work. Each step, described in this article, is accompanied by a `jsbin` code example to help you work along, and to be used as the base for further exploration.



What is a JavaScript promise?

A promise is a method that eventually produces a value. It can be considered as the asynchronous counterpart of a getter function. Its essence can be explained as:

```
promise.then(function(value) {
  // Do something with the 'value'
});
```

Promises can replace the asynchronous use of callbacks, and they provide several benefits over them. They start to gain ground as more and more libraries and frameworks embrace them as their primary way to handle asynchronicity. Ember.js is a great example of such a framework.

There are several libraries that implement the Promises/A+ specification. We'll learn the basic vocabulary, and work through a few JavaScript promises examples to introduce the concepts behind them in a practical way. I'll use one of the more popular implementation libraries, rsvp.js, in the code examples.

Get ready, we'll roll a lot of dice!

Getting the rsvp.js library

Promises, and thus rsvp.js, can be used both on the server and on the client side. To install it for *nodejs*, go to your project folder and type:

```
npm install --save rsvp
```

If you work on the front-end and use bower, it's just a

```
bower install -S rsvp
```

away.

If you just want to get right in the game, you can include it via simple script tag (and with `jsbin`, you can add it via the "Add library" dropdown):

```
<script src="//cdn.jsdelivr.net/rsvp/3.0.6/rsvp.js"></script>
```

What properties does a promise have?

A promise can be in one of three states: **pending**, **fulfilled**, or **rejected**. When created, the promise is in pending state. From here, it can either go to the fulfilled or rejected state. We call this transition the **resolution of the promise**. The resolved state of a promise is its final state, so once it is fulfilled or rejected, it stays there.

The way to create a promise in rsvp.js is via what is called a revealing constructor. This type of constructor takes a single function parameter and immediately calls it with two arguments, `fulfill` and `reject`, that can transition the promise to either the `fulfilled` or the `rejected` state:

```
var promise = new RSVP.Promise(function(fulfill, reject) {
  (...)
});
```

This JavaScript promises pattern is called a revealing constructor because the single function argument reveals its capabilities to the constructor function, but ensures that consumers of the promise cannot manipulate its state.

Consumers of the promise can react to its state changes by adding their handler through the `then` method. It takes a fulfillment and a rejection handler function, both of which can be missing.

```
promise.then(onFulfilled, onRejected);
```

Depending on the result of the promise's resolution process, either the `onFulfilled` or the `onRejected` handler is called **asynchronously**.

Let's see an example that shows in which order things get executed:

```
function dieToss() {
  return Math.floor(Math.random() * 6) + 1;
}

console.log('1');
var promise = new RSVP.Promise(function(fulfill, reject) {
  var n = dieToss();
  if (n === 6) {
    fulfill(n);
  } else {
    reject(n);
  }
  console.log('2');
});

promise.then(function(toss) {
  console.log('Yay, threw a ' + toss + '.');
}, function(toss) {
  console.log('Oh, noes, threw a ' + toss + '.');
});
console.log('3');
```

This snippet prints output similar to the following:

```
1
2
3
Oh, noes, threw a 4.
```

Or, if we get lucky, we see:

```
1
2
3
Yay, threw a 6.
```

This promises tutorial demonstrates two things.

First, that the handlers we attached to the promise were indeed called after all other code ran, asynchronously.

Second, that the fulfillment handler was called only when the promise was fulfilled, with the value it was resolved with (in our case, the result of the dice toss). The same holds true for the rejection handler.

Chaining promises and trickling down

The specification requires that the `then` function (the handlers) must return a promise, too, which enables chaining promises together, resulting in code that looks almost synchronous:

```
signupPayingUser
  .then(displayHoorayMessage)
  .then(queueWelcomeEmail)
  .then(queueHandwrittenPostcard)
  .then(redirectToThankYouPage)
```

Here, `signupPayingUser` returns a promise, and each function in the promise chain gets called with the return value of the previous handler once it has completed. For all practical purposes, this serializes the calls without blocking the main execution thread.

To see how each promise gets resolved with the return value of the previous item in the chain, we return to tossing dice. We want to toss the dice a maximum of three times, or until the first six comes up jsbin:

```
function dieToss() {
  return Math.floor(Math.random() * 6) + 1;
}

function tossASix() {
  return new RSVP.Promise(function(fulfill, reject) {
    var n = Math.floor(Math.random() * 6) + 1;
    if (n === 6) {
      fulfill(n);
    } else {
      reject(n);
    }
  });
}

function logAndTossAgain(toss) {
  console.log("Tossed a " + toss + ", need to try again.");
  return tossASix();
}

function logSuccess(toss) {
  console.log("Yay, managed to toss a " + toss + ".");
}

function logFailure(toss) {
  console.log("Tossed a " + toss + ". Too bad, couldn't roll a six");
}

tossASix()
  .then(null, logAndTossAgain)   //Roll first time
  .then(null, logAndTossAgain)   //Roll second time
  .then(logSuccess, logFailure); //Roll third and last time
```

When you run this promises example code, you'll see something like this on the console:

```
Tossed a 2, need to try again.
Tossed a 1, need to try again.
Tossed a 4. Too bad, couldn't roll a six.
```

The promise returned by `tossASix` is rejected when the toss is not a six, so the rejection handler is called with the actual toss. `logAndTossAgain` prints that result on the console and returns a promise that represents another dice toss. That toss, in turn, also gets rejected and logged out by the next `logAndTossAgain`.

Sometimes, however, you get lucky*, and you manage to roll a six:

```
Tossed a 4, need to try again.
Yay, managed to toss a 6.
```

* You don't have to get *that* lucky. There is a ~42% chance to roll at least one six if you roll three dice.

That example also teaches us something more. See how no more tosses were made after the first successful rolling of a six? Note that all fulfillment handlers (the first arguments in the calls to `then`) in the chain are `null`, except the last one, `logSuccess`. The specification requires that if a handler (fulfillment or rejection) is not a function then the returned promise must be resolved (fulfilled or rejected) with the same value. In the above promises example, the fulfillment handler, `null`, is not a function and the value of the promise was fulfilled with a 6. So the promise returned by the `then` call (the next one in the chain) is also going to be fulfilled with 6 as its value.

This repeats until an actual fulfillment handler (one that is a function) is present, so the fulfillment **trickles down** until it gets handled. In our case, this happens at the end of the chain where it is cheerfully logged out onto the console.

Handling errors

The Promises/A+ specification demands that if a promise is rejected or an error is thrown in a rejection handler, it should be handled by a rejection handler that is “downstream” from the source.

Leveraging the below trickle down technique gives a clean way to handle errors:

```
signupPayingUser
  .then(displayHoorayMessage)
  .then(queueWelcomeEmail)
  .then(queueHandwrittenPostcard)
  .then(redirectToThankYouPage)
  .then(null, displayAndSendErrorReport)
```

Because a rejection handler is only added at the very end of the chain, if any fulfillment handler in the chain gets rejected or throws an error, it trickles down until it bumps into `displayAndSendErrorReport`.

Let’s return to our beloved dice and see that in action. Suppose we just want to throw dice asynchronously and print out the results:

```
var tossTable = {
  1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five', 6: 'six'
};

function toss() {
  return new RSVP.Promise(function(fulfill, reject) {
    var n = Math.floor(Math.random() * 6) + 1;
    fulfill(n);
  });
}

function logAndTossAgain(toss) {
  var tossWord = tossTable[toss];
  console.log("Tossed a " + tossWord.toUppercase() + ".");
}
```

```
toss()
  .then(logAndTossAgain)
  .then(logAndTossAgain)
  .then(logAndTossAgain);
```

When you run this, nothing happens. Nothing is printed on the console and no errors are thrown, seemingly.

In reality, an error does get thrown, we just don't see it since there are no rejection handlers in the chain. Since code in the handlers gets executed asynchronously, with a fresh stack, it does not even get logged out to the console. Let's fix this:

```
function logAndTossAgain(toss) {
  var tossWord = tossTable[toss];
  console.log("Tossed a " + tossWord.toUpperCase() + ".");
}

function logErrorMessage(error) {
  console.log("Oops: " + error.message);
}

toss()
  .then(logAndTossAgain)
  .then(logAndTossAgain)
  .then(logAndTossAgain)
  .then(null, logErrorMessage);
```

Running the above code does show the error now:

```
"Tossed a TWO."
"Oops: Cannot read property 'toUpperCase' of undefined"
```

We forgot to return something from `logAndTossAgain` and the second promise is fulfilled with `undefined`. The next fulfillment handler then blows up trying to call `toUpperCase` on that. That's another important thing to remember: always return something from the handlers, or be prepared in subsequent handlers to have nothing passed.

Building higher

We have now seen the basics of JavaScript promises in this tutorial's example code. A great benefit of using them is that they can be composed in simple ways to produce "compound" promises with the behavior we would like. The `rsvp.js` library provides a handful of them, and you can always create your own using the primitives and these higher-level ones.

For the final, most complex example, we travel to the world of AD&D role playing and toss dice to get character scores. Such scores are obtained by rolling three dice for each skill of the character.

Let me paste the code here first and then explain what is new:

```
function toss() {
  var n = Math.floor(Math.random() * 6) + 1;
  return new RSVP.resolve(n); // [1]
}

function threeDice() {
  var tosses = [];

  function add(x, y) {
    return x + y;
  }

  for (var i=0; i<3; i++) { tosses.push(toss()); }
```

```
  return RSVP.all(tosses).then(function(results) { // [2]
    return results.reduce(add); // [3]
  });
}

function logResults(result) {
  console.log("Rolled " + result + " with three dice.");
}

function logErrorMessage(error) {
  console.log("Oops: " + error.message);
}

threeDice()
  .then(logResults)
  .then(null, logErrorMessage);
```

We are familiar with `toss` from the last code example. It simply creates a promise that is always fulfilled with the result of casting a dice. I used `RSVP.resolve`, a convenient method that creates such a promise with less ceremony (see [1] in the code above).

In `threeDice`, I created 3 promises that each represent a dice toss and finally combined them with `RSVP.all`. `RSVP.all` takes an array of promises and is resolved with an array of their resolved values, one for each constituent promise, while maintaining their order. That means we have the result of the tosses in `results` (see [2] in the code above), and we return a promise that is fulfilled with their sum (see [3] in the code above).

Resolving the resulting promise then logs the total number:

```
"Rolled 11 with three dice"
```

Using promises to solve real problems

JavaScript promises are used to solve problems in applications that are far more complex than *asynchronous-for-no-good-reason dice tosses*.

If you substitute rolling three dice with sending out three ajax requests to separate endpoints and proceeding when all of them have returned successfully (or if any of them failed), you already have a useful application of promises and `RSVP.all`.

Promises, when used correctly, produce easy-to-read code that is easier to reason about, and thus easier to debug than callbacks. There is no need to set up conventions regarding, for example, error handling since they are already part of the specification.

We barely scratched the surface of what promises can do in this JavaScript tutorial. Promise libraries provide a good dozen of methods and low level constructors that are at your disposal. Master these, and the sky is the limit in what you can do with them.

About the author

Balint Erdi was a great role-playing and AD&D fan a long time ago, and is a great promise and Ember.js fan now. What has been constant is his passion for rock & roll. That's why he decided to write a book on Ember.js that uses rock & roll as the theme of the application in the book. Sign up here to know when it launches.

About the author

View full profile »
Hire the Author
Balint Erdi, Hungary
member since August 26, 2013
RubyEmber.jsRuby on Rails
Balint lives on the cutting edge of technology, and has been practicing TDD since before it became popular. He was a classic PHP developer, and has since moved on to Java, Python, and Ruby. Currently, he is in love with Ember.js. He is fascinated by functional programming and also enjoys Clojure. [click to continue...]
Hiring? Meet the Top 10 Freelance JavaScript Developers for Hire in October 2016

Comments

liuxiaolei
awesome! thanks for sharing. could be even better if add some introduction to catch , finally and deferred.
Balint Erdi
Glad you liked it. I agree that there is a lot that is left out but it's really meant to be an introduction, something that could be built upon (possibly in a future post? :)).
尤川豪
This is a great article. You really know how to demonstrate difficult topics in an easy way. Thanks a lot. I like those dices. :)
Brennan
Solid introduction article on the topic. Have read a few so far and like the flow building on the first example, along with the tips and problems that can be run into.
Balint Erdi
Glad you liked it!
Fritz Schenk
I enjoyed your article and examples. Seems that examples should use true asynchronism as would webworkers or ajax provide. Also, how is it different from using callbacks in promise.then(callback, callback)?
Niels C. Nielsen Jr.
Very very good examples!
Byte Archer

Yeah, asynchronity is definitely the way to go. Something caught my eye in your text: promise.then(callback, callback) There is actually a case where doing it this way leads to a bit unexpected results. If there's an exception thrown in the success callback (the first one), then it will not be passed on to the rejection handler (the second callback). This is due to design how Promises work. A better improved version would be to separate the rejection handler to its own and then chain it immediately after original .then(). Like this promise.then(callback).catch(callback) You can find more information on this at http://bytearcher.com/articles/using-promise-then-callback-callback-misses-errors/

arun

hi balint, really this things are well and good, Actually i have small doubt i want to change the script tag source file dynamically, i create the element and also set the src file also, but this file is executed before another file is execute. What i do for this scenario ??

bdo customer

In the last example there is a code: return results.reduce(add); // [3] why is it not: resolve(results.reduce(add)); // [3] is resolve(n) the same as return(n) in a promise?


days ago How Sequel and Sinatra Solve Ruby's API Problem28 days ago 
The 10 Most Common Mistakes That WordPress Developers Makeabout 1 month ago 
The Six Commandments of Good Code: Write Code that Stands the Test of Timeabout 1 month ago 
Meet RxJava: The Missing Reactive Programming Library for Androidabout 1 month ago 
Celebrating 25 Years of Linux Kernel Developmentabout 1 month ago

Relevant technologies

- Back-end
- JavaScript
- Front-end

About the author

Balint Erdi
Ruby Developer

Balint lives on the cutting edge of technology, and has been practicing TDD since before it became popular. He was a classic PHP developer, and has since moved on to Java, Python, and Ruby. Currently, he is in love with Ember.js. He is fascinated by functional programming and also enjoys Clojure.