JavaScript Promise API

By David Walsh on November 2, 2015

19

While synchronous code is easier to follow and debug, async is generally better for performance and flexibility. Why "hold up the show" when you can trigger numerous requests at once and then handle them when each is ready? Promises are becoming a big part of the JavaScript world, with many new APIs being implemented with the promise philosophy. Let's take a look at promises, the API, how it's used!

Promises in the Wild

The XMLHttpRequest API is async but does *not* use the Promises API. There are a few native APIs that now use promises, however:

- Battery API
- fetch API (XHR's replacement)
- ServiceWorker API (post coming soon!)

Promises will only become more prevalent so it's important that all front-end developers get used to them. It's also worth noting that Node.js is another platform for Promises (obviously, as Promise is a core language feature).

Testing promises is probably easier than you think because `setTimeout` can be used as your async "task"!

Basic Promise Usage

The `new Promise()` constructor should only be used for legacy async tasks, like usage of `setTimeout` or `XMLHttpRequest`. A new Promise is created with the `new` keyword and the promise provides `resolve` and `reject` functions to the provided callback:

```
var p = new Promise(function(resolve, reject) {

 // Do an async task async task and then...

 if(/* good condition */) {
  resolve('Success!');
 }
 else {
  reject('Failure!');
 }
});

p.then(function() {
 /* do something with the result */
}).catch(function() {
 /* error :( */
})
```

It's up to the developer to manually call `resolve` or `reject` within the body of the callback based on the result of their given task. A realistic example would be converting XMLHttpRequest to a promise-based task:

```
// From Jake Archibald's Promises and Back:
// http://www.html5rocks.com/en/tutorials/es6/promises/#toc-promisifying-xmlhttprequest

function get(url) {
  // Return a new promise.
  return new Promise(function(resolve, reject) {
    // Do the usual XHR stuff
    var req = new XMLHttpRequest();
    req.open('GET', url);

    req.onload = function() {
      // This is called even on 404 etc
      // so check the status
      if (req.status == 200) {
        // Resolve the promise with the response text
        resolve(req.response);
      }
      else {
        // Otherwise reject with the status text
        // which will hopefully be a meaningful error
        reject(Error(req.statusText));
      }
    };

    // Handle network errors
    req.onerror = function() {
      reject(Error("Network Error"));
    };

    // Make the request
    req.send();
  });
}

// Use it!
get('story.json').then(function(response) {
  console.log("Success!", response);
}, function(error) {
  console.error("Failed!", error);
});
```

Sometimes you don't *need* to complete an async tasks within the promise -- if it's *possible* that an async action will be taken, however, returning a promise will be best so that you can always count on a promise coming out of a given function. In that case you can simply call `Promise.resolve()` or `Promise.reject()` without using the `new` keyword. For example:

```
var userCache = {};

function getUserDetail(username) {
  // In both cases, cached or not, a promise will be returned

  if (userCache[username]) {
   // Return a promise without the "new" keyword
    return Promise.resolve(userCache[username]);
  }
  // Use the fetch API to get the information
  // fetch returns a promise
  return fetch('users/' + username + '.json')
    .then(function(result) {
      userCache[username] = result;
      return result;
    })
    .catch(function() {
      throw new Error('Could not find user: ' + username);
    });
}
```

Since a promise is always returned, you can always use the `then` and `catch` methods on its return value!

then

All promise instances get a `then` method which allows you to react to the promise. The first `then` method callback receives the result given to it by the `resolve()` call:

```
new Promise(function(resolve, reject) {
 // A mock async action using setTimeout
 setTimeout(function() { resolve(10); }, 3000);
})
.then(function(result) {
 console.log(result);
});

// From the console:
// 10
```

The `then` callback is triggered when the promise is resolved. You can also chain `then` method callbacks:

```
new Promise(function(resolve, reject) {
 // A mock async action using setTimeout
 setTimeout(function() { resolve(10); }, 3000);
})
.then(function(num) { console.log('first then: ', num); return num * 2; })
.then(function(num) { console.log('second then: ', num); return num * 2; })
.then(function(num) { console.log('last then: ', num);});

// From the console:
// first then:  10
// second then:  20
// last then:  40
```

Each `then` receives the result of the previous `then`'s return value.

If a promise has already resolved but `then` is called again, the callback immediately fires. If the promise is rejected and you call `then` after rejection, the callback is never called.

catch

The `catch` callback is executed when the promise is rejected:

```
new Promise(function(resolve, reject) {
 // A mock async action using setTimeout
 setTimeout(function() { reject('Done!'); }, 3000);
})
.then(function(e) { console.log('done', e); })
.catch(function(e) { console.log('catch: ', e); });

// From the console:
// 'catch: Done!'
```

What you provide to the `reject` method is up to you. A frequent pattern is sending an `Error` to the `catch`:

```
reject(Error('Data could not be found'));
```

`Promise.all`

Think about JavaScript loaders: there are times when you trigger multiple async interactions but only want to respond when all of them are completed -- that's where `Promise.all` comes in. The `Promise.all` method takes an array of promises and fires one callback once they are all resolved:

```
Promise.all([promise1, promise2]).then(function(results) {
 // Both promises resolved
})
.catch(function(error) {
 // One or more promises was rejected
});
```

An perfect way of thinking about `Promise.all` is firing off multiple AJAX (via `fetch`) requests at one time:

```
var request1 = fetch('/users.json');
var request2 = fetch('/articles.json');

Promise.all([request1, request2]).then(function(results) {
 // Both promises done!
});
```

You could combine APIs like `fetch` and the Battery API since they both return promises:

```
Promise.all([fetch('/users.json'), navigator.getBattery()]).then(function(results) {
 // Both promises done!
});
```

Dealing with rejection is, of course, hard. If any promise is rejected the `catch` fires for the first rejection:

```
var req1 = new Promise(function(resolve, reject) {
 // A mock async action using setTimeout
 setTimeout(function() { resolve('First!'); }, 4000);
});
var req2 = new Promise(function(resolve, reject) {
 // A mock async action using setTimeout
 setTimeout(function() { reject('Second!'); }, 3000);
});
Promise.all([req1, req2]).then(function(results) {
 console.log('Then: ', results);
}).catch(function(err) {
 console.log('Catch: ', err);
});
```

```
// From the console:
// Catch: Second!
```

`Promise.all` will be super useful as more APIs move toward promises.

`Promise.race`

`Promise.race` is an interesting function -- instead of waiting for all promises to be resolved or rejected, `Promise.race` triggers as soon as any promise in the array is resolved or rejected:

```
var req1 = new Promise(function(resolve, reject) { 
 // A mock async action using setTimeout
 setTimeout(function() { resolve('First!'); }, 8000);
});
var req2 = new Promise(function(resolve, reject) { 
 // A mock async action using setTimeout
 setTimeout(function() { resolve('Second!'); }, 3000);
});
Promise.race([req1, req2]).then(function(one) {
 console.log('Then: ', one);
}).catch(function(one, two) {
 console.log('Catch: ', one);
});

// From the console:
// Then: Second!
```

A use case could be triggering a request to a primary source and a secondary source (in case the primary or secondary are unavailable).

Get Used to Promises

Promises have been a hot topic for the past few years (or the last 10 years if you were a Dojo Toolkit user) and they've gone from a JavaScript framework pattern to a language staple. It's probably wise to assume you'll be seeing most new JavaScript APIs being implemented with a promise-based pattern...

...and that's a great thing! Developers are able to avoid callback hell and async interactions can be passed around like any other variable. Promises take some time getting used to be the tools are (natively) there and now is the time to learn them!

Recent Features

An Interview with Eric Meyer

Your early CSS books were instrumental in pushing my love for front end technologies. What was it about CSS that you fell in love with and drove you to write about it? At first blush, it was the simplicity of it as compared to the table-and-spacer...

Write Simple, Elegant and Maintainable Media Queries with Sass

I spent a few months experimenting with different approaches for writing simple, elegant and maintainable media queries with Sass. Each solution had something that I really liked, but I couldn't find one that covered everything I needed to do, so I ventured into creating my...

Incredible Demos

Create an Animated Sliding Button Using MooTools

Buttons (or links) are usually the elements on our sites that we want to draw a lot of attention to. Unfortunately many times they end up looking the most boring. You don't have to let that happen though! I recently found a...

Dijit’s TabContainer Layout: Easy Tabbed Content

One of Dojo's major advantages over other JavaScript toolkits is its Dijit library. Dijit is a UI framework comprised of JavaScript widget classes, CSS files, and HTML templates. One very useful layout class is the TabContainer. TabContainer allows you to quickly create a tabbed content...

Discussion

Stephen

Thanks, nice article summarizing promises, very useful as a refresher!

Max

Nice article, as Stephen said, good refresher.

Lars

Hi David, good summary! Just one thing.. `Promise.all` resolves with an array of all results. So instead of something like this here:

```
Promise.all([promise1, promise2]).then(function(result1, result2) {...})
```

it should be more like that:

```
Promise.all([promise1, promise2]).then(function(results) {...})
```

..or with ES6 destructuring sugar:

```
Promise.all([promise1, promise2]).then(function([result1, result2]) {...})
```

Paweł Słomka

Great catch, I just wanted to point this out;)

Francis Kim

Probably hands down the best tutorial on promises I’ve seen. Also loved the fact that you mentioned Dojo Toolkit – I feel like it doesn’t get enough mentions.

Adam

Mocha/Dojo Deferreds used to let the caller register a `then` callback prior to execution of the code and the subsequent `accept/reject` so that you could pass around and register callbacks on without even knowing if it was in progress yet, avoiding a lot of awkward conditional logic. Is this possible with the new API? If not, was this by design?

Drew

At present, it looks like Promises unnest, which means that:

```
Promise.resolve(Promise.resolve(0)) = Promise.resolve(0)
Promise.resolve(Promise.reject(0)) = Promise.reject(0)
```

This means that if you have a function that takes a argument that could be either a Promise itself OR a bare value, you can immediately wrap that value in `Promise.resolve()` to ensure that it's a Promise when you go to use it later in the function (without using any conditionals). Mega-flexible!

Example of a function that could receive either a value or a Promise that returns a value, treating all of these the same way:

```
asyncIncrement(5).then(log); //-> [Promise:6]
asyncIncrement(Promise.resolve(5)).then(log); //-> [Promise:6]
asyncIncrement(fetch('getfiveapi')).then(log);  //-> [Promise:6]
```

Domenic Denicola

Hi David,

Unfortunately your very first example falls into the [“explicit promise construction antipattern”](), which is pretty sad for a tutorial. Your getUserDetail should be rewritten as follows to avoid it: https://gist.github.com/domenic/70df41c0b2e89712b3eb

In general the Promise constructor should only be used when adapting legacy APIs, like setTimeout. (Even then it is better to do it in a single place, e.g. `function delay(ms) { return new Promise(...) }`, instead of doing it every time.)

David Walsh

Thank you for the heads up!

Tomas van Rijsse

I don't quite get what the `getUserDetail` function returns.
Is it the Promise from the `fetch()` method or the result from the `.then()` method?
Which of the two is actually returned to the caller of `getUserDetail`?

I expect that the `Promise` is returned but who would one retrieve the results in that case?

David Walsh

The `getUserDetail` function in all cases returns the promise with the user information passed as the first argument to `then()`.

Anh Tran

Is there a way to make consequent promises? I want to send an ajax request (promise A) and if it's success then send another ajax request (promise B).

ikram

Thanks, nice article summarizing promises, very useful as a refresher!
I have written also about angular promises, plaese have a look,
http://growthefuturenow.com/promises-in-an-angularjs-earth/

Fagner Brack

Also, be careful not to fall in this kind of trap which I have seen pretty often lately:
https://medium.com/@fagnerbrack/promises-are-not-proxies-fd00751eb980

Jonathan

Is “One or more promises was rejected” right? shouldn’t we use were in this case

Michail Almyros

Great article well done, I really like how you are explaining things.

Paweł Słomka

Awesome article that sums up everything that one has to know about Promises! The only thing missing is that `catch` also catches errors that are thrown in `then`s that come before.

Great article, David!

Rom Grk

« Dealing with rejection is, of course, hard. »

Totally agree with that.

Dylan Scott

Thanks for the great article – it has validated what I wrote about in http://dylanscott.me/promise-parallelism-with-nodejs/. Even taught me something new about .race() which I didn’t quite get!

Name | Email | Website

Wrap your code in `<pre class="{language}"></pre>` tags, link to a GitHub gist, JSFiddle fiddle, or CodePen pen to embed!

□ **Continue this conversation via email**

Post Comment! | Use Code Editor