ARTICLE

Promises in Node.js: An alternative to callbacks

Learn why and how to use promises in Node.js

by Marc Harter | Published June 20, 2019

Node.js

Callbacks are the simplest possible mechanism for handling asynchronous code in JavaScript. Yet, raw callbacks sacrifice the control flow, exception handling, and function semantics that developers are familiar with in synchronous code:

```
// Asynchronous operations return no meaningful value
var noValue = fs.readFile('file1.txt', function(err, buf) {
  // Errors are explicitly handled every time
  if (err) return handleErr(err)
  fs.readFile('file2.txt', function(err2, buf2) {
    if (err2) return handleErr(err2)
    data.foo.baz = 'bar' // Exceptions like this ReferenceError are not c

    // Sequential operations encourage heavy nesting
    fs.readFile('file3.txt', function(err3, buf3) {
      if (err3) return handleErr(err3)
    })
  })
})
```

Show more

Promises offer a way to get that control back with:

- More powerful control flow
- Better exception handling
- Functional programming semantics

Still, promises can be confusing, so you may have written them off or skipped directly to `async/await`, which adds new syntax to JavaScript for promises.

However, understanding how promises work and behave at a fundamental level will help you make the most of them. In this article, we cover the basics of promises, including what they are, how to create them, and how to use them most effectively.

Promises in the abstract

First, let’s look at the **behavior** of promises: What are they and how can they be useful? Then we’ll discuss how to create and use promises.

What is a promise? Let’s look at a definition:

> A promise is an abstraction for asynchronous programming. It’s an object that proxies for the return value or the exception thrown by a function that has to do some asynchronous processing. — Kris Kowal on JSJ

The core component of a promise object is its `then` method. The `then` method is how you get the return value (known as the *fulfillment value*) or the exception thrown (known as the *rejection reason*) from an asynchronous operation. `then` takes two optional callbacks as arguments, which we’ll call `onFulfilled` and `onRejected`:

```
let promise = doSomethingAync()
promise.then(onFulfilled, onRejected)
```

`onFulfilled` and `onRejected` trigger when the promise resolves (the asynchronous processing has completed). One of these functions will trigger because *only* one resolution is possible.

Callbacks to promises

Given this basic knowledge of promises, let’s take a look at a familiar asynchronous Node.js callback:

```
readFile(function(err, data) => {
  if (err) return console.error(err)
  console.log(data)
})
```

If our `readFile` function *returned a promise*, we would write the same logic as:

```
let promise = readFile()
promise.then(console.log, consoler.error)
```

At first glance, it looks like the aesthetics changed. But, we now have access to a **value** representing the asynchronous operation (the promise). We can pass around the promise in code like any other value in JavaScript. Anyone with access to the promise can consume it using `then` *regardless of whether the asynchronous operation has completed or not*. We also have guarantees that the result of the asynchronous operation won't change somehow, as the promise will resolve once (either fulfilled or rejected).

> It's helpful to think of `then` not as a function that takes two callbacks (`onFulfilled` and `onRejected`), but as a function that *unwraps* the promise to reveal what happened from the asynchronous operation. Anyone with access to the promise can use `then` to unwrap it. For more about this idea, read Callbacks are imperative, promises are functional: Node's biggest missed opportunity.

Chaining and nesting promises

The `then` method itself *returns a promise*:

```
let promise = readFile()
let promise2 = promise.then(readAnotherFile, console.error)
```

This promise represents the return value for its `onFulfilled` or `onRejected` handlers, if specified. Since one resolution is possible, the promise proxies the triggered handler:

```
let promise = readFile()
let promise2 = promise.then(
  function(data) {
    return readAnotherFile() // If readFile was successful, let's readAno
  },
  function(err) {
    console.error(err) // If readFile was unsuccessful, let's log it but
    return readAnotherFile()
  }
)
promise2.then(console.log, console.error) // The result of readAnotherFil
```

Since `then` returns a promise, it means promises can chain together to avoid the deep nesting of callback hell:

```
readFile()
  .then(readAnotherFile)
  .then(doSomethingElse)
  .then(...)
```

Still, promises can nest if keeping a closure alive is important:

```
readFile().then(function(data) {
  return readAnotherFile().then(function() {
    // Do something with `data`
  })
})
```

Promises and synchronous functions

Promises model synchronous functions in important ways. One such way is using `return` for continuation instead of calling another function. The previous examples returned `readAnotherFile()` to signal what to do after `readFile()`.

If you return a promise, it will signal the next `then` when the asynchronous operation completes. You can also return any other value and the next `onFulfilled` will get the value as an argument:

```
readFile()
  .then(function (buf) {
    return JSON.parse(buf.toString())
  })
  .then(function (data) => {
    // Do something with `data`
  })
```

Error handling in promises

You also can use the `throw` keyword and get `try/catch` semantics. This may be one of the most powerful features of promises. For example, consider the following synchronous code:

```
try {
  doThis()
  doThat()
} catch (err) {
  console.error(err)
}
```

In this example, if `doThis()` or `doThat()` would `throw` an error, we would `catch` and log the error. Since `try/catch` blocks allow grouped operations, we can avoid having to explicitly handle errors for each operation. We can do this same thing asynchronously with promises:

```
doThisAsync()
  .then(doThatAsync)
  .then(undefined, console.error)
```

If `doThisAsync()` is unsuccessful, its promise rejects, and the next `then` in the chain with an `onRejected` handler triggers. In this case, it’s the `console.error` function. And like `try/catch` blocks, `doThatAsync()` would never get called. This is an improvement over raw callbacks where you have to handle errors explicitly at each step.

But, it gets better! Any thrown exception–implicit or explicit–from the `then` callbacks is also handled in promises:

```
doThisAsync()
  .then(function(data) {
    data.foo.baz = 'bar' // Throws a ReferenceError as foo is not defined
  })
  .then(undefined, console.error)
```

Here, the raised `ReferenceError` triggers the *next* `onRejected` handler in the chain. Pretty neat! Of course, this works for explicit `throw` as well:

```
doThisAsync()
  .then(function(data) {
    if (!data.baz) throw new Error('Expected baz to be there')
  })
  .catch(console.error) // The catch(fn) is shorthand for .then(undefined
```

An important note with error handling

As stated earlier, promises mimic `try/catch` semantics. In a `try/catch` block, it’s possible to mask an error by never explicitly handling it:

```
try {
  throw new Error('Never will know this happened')
} catch (e) {}
```

The same goes for promises:

```
readFile().then(function(data) {
  throw new Error('Never will know this happened')
})
```

To expose masked errors, a solution is to end the promise chain with a simple `.catch(onRejected)`clause:

```
readFile()
  .then(function(data) {
    throw new Error('Now I know this happened')
  })
  .catch(console.error)
```

Third-party libraries include options for exposing unhandled rejections.

Promises in the concrete

Our examples have used promise-returning dummy methods to illustrate the `then` method from ES6/2015 and Promises/A+. Let’s turn now and look at more concrete examples.

Converting callbacks to promises

You may be wondering how to create a promise in the first place. The API for creating a promise isn’t specified in Promise/A+ because it’s not necessary for interoperability. ES6/2015 did standardize a `Promise` constructor which we will come back to. One of the most common cases for using promises is converting existing callback-based libraries. Here, Node has a built-in utility function, `util.promisify`, to help us.

Let’s convert one of Node’s core asynchronous functions, which take callbacks, to return promises instead using `util.promisify`:

```
const util = require('util')
const fs = require('fs')
let readFile = util.promisify(fs.readFile)

let promise = readFile('myfile.txt')
promise.then(console.log, console.error)
```

Creating raw promises

You can create a promise using the `Promise` constructor as well. Let’s convert the same `fs.readFile` method to return promises without using `util.promisify`:

```
const fs = require('fs')
function readFile(file, encoding) {
  return new Promise(function(resolve, reject) {
    fs.readFile(file, encoding, function(err, data) {
      if (err) return reject(err) // Rejects the promise with `err` as th
      resolve(data) // Fulfills the promise with `data` as the value
    })
  })
}

let promise = readFile('myfile.txt')
promise.then(console.log, console.error)
```

Making APIs that support both callbacks and promises

We have seen two ways to turn callback code into promise code. You can also make APIs that provide both a promise and callback interface. For example, let’s turn `fs.readFile` into an API that supports both callbacks and promises:

```
const fs = require('fs')
function readFile(file, encoding, callback) {
  if (callback) return fs.readFile(file, encoding, callback) // Use callb
  return new Promise(function(resolve, reject) {
    fs.readFile(file, encoding, function(err, data) {
      if (err) return reject(err)
      resolve(data)
    })
  })
}
```

If a callback exists, trigger it with the standard Node style `(err, result)` arguments.

```
readFile('myfile.txt', 'utf8', function(er, data) {
  // ...
})
```

Doing parallel operations with promises

We've talked about sequential asynchronous operations. For parallel operations, ES6/2015 provides the `Promise.all` method which takes in an array of promises and returns a new promise. The new promise fulfills after *all* the operations have completed. If *any* of the operations fail, the new promise rejects.

```
let allPromise = Promise.all([readFile('file1.txt'), readFile('file2.txt'
allPromise.then(console.log, console.error)
```

> It's important to note again that promises mimic functions. A function has one return value. When passing `Promise.all` two promises that complete, `onFulfilled` triggers with one argument (an array with both results). This may surprise you; yet, consistency with synchronous counterparts is an important guarantee that promises provide.

Making promises even more concrete

The best way to understand promises is to use them. Here are some ideas to get you started:

- Wrap some standard Node.js library functions, converting callbacks into promises. No cheating using the `node.promisify` utility!
- Take a function using `async/await` and rewrite it without using that syntactic sugar. This means you will return a promise and use the `then` method.
- Write something recursively using promises (a directory tree would be a good start).
- Write a passing Promise A+ implementation. Here is my crude one.

SOCIAL

CONTENTS

Making promises even more concrete

RESOURCES

ES6 Promise Specification

Promises with Domenic Denicola and Kris Kowal

You're Missing the Point of Promises

Redemption from Callback Hell

Callbacks are imperative, promises are functional

List of Promise/A+ compatible implementations

Related content

MEETUP

JAMstack Lagos Meetup - September

September 14, 2019

API Management Cloud +

CODE PATTERN | SEP 09, 2019

Build a network to support blockchain-enabled crowdfunding

Get the Code »

Blockchain Hyperledger Fabric +

MEETUP

Kubernetes on IBM Cloud

September 7, 2019

Containers DevOps +