

Simplifying Asynchronous Coding with ES7 Async Functions

By [Joe Zimmerman](#) April 06, 2015

The debut of [Promises](#) in JavaScript has lit the internet on fire—they help developers break out of [callback hell](#) and solve a lot of problems that have plagued the asynchronous code of JavaScript programmers everywhere. Promises are far from flawless, though. They still require callbacks, can still be messy in complex situations, and are incredibly verbose.

With the advent of ES6, which not only makes promises native to the language without requiring one of the countless available libraries, we also get [generators](#). Generators have the ability to pause execution within a function, which means that by [wrapping them in a utility function](#), we have the ability to wait for an asynchronous operation to finish before moving on to the next line of code. Suddenly your asynchronous code can start to look synchronous!

But this is just the first step. In ES7, [async functions](#) will be released. Async functions take the idea of using generators for asynchronous programming and give them their own simple and semantic syntax. Consequently, you don't have to use a library to get that wrapping utility function, because that is handled in the background.

To run the ES7 code samples from this article, you'll need to use the Traceur compiler. [Show me how.](#)

Async Functions vs Generators

Here is an example of using generators for asynchronous programming. It uses the [Q](#) library:

```
var doAsyncOp = Q.async(function* () {
  var val = yield asynchronousOperation();
  console.log(val);
  return val;
});
```

`Q.async` is the wrapper function that handles everything behind the scenes. The `*` is what denotes the function as a generator function and `yield` is how you pause the function and let the wrapper function take over. `Q.async` will return a function that you can assign—as I have done—to `doAsyncOp` and subsequently invoke.

Here's what it looks like when you get rid of the cruft by using the new syntax included in ES7:

```
async function doAsyncOp () {
  var val = await asynchronousOperation();
  console.log(val);
  return val;
};
```

It's not ~~60~~ different, but we removed the wrapper function and the asterisk and replaced them with the `async` keyword. The `yield` keyword was also replaced by `await`. These two examples will do the exactly same thing: wait for `asynchronousOperation` to complete before assigning its value to `val`, logging it, and returning it.

Converting Promises to Async Functions

What would the previous example look like if we were using vanilla promises?

```
function doAsyncOp () {
    return asynchronousOperation().then(function(val) {
        console.log(val);
        return val;
    });
};
```

This has the same number of lines, but there is plenty of extra code due to `then` and the callback function passed to it. The other nuisance is the duplication of the `return` keyword. This has always been something that bugged me, because it makes it difficult to figure out exactly what is being returned from a function that uses promises.

As you can see, this function returns a promise that will fulfill to the value of `val`. And guess what ... so do the generator and `async` function examples! Whenever you return a value from one of those functions, you are actually implicitly returning a promise that resolves to that value. If you don't return anything at all, you are implicitly returning a promise that resolves to `undefined`.

Chaining Operations

One of the aspects of promises that hooks many people is the ability to chain multiple asynchronous operations without running into nested callbacks. This is one of the areas in which `async` functions excel even more than promises.

This is how you would chain asynchronous operations using promises (admittedly we're being silly and just running the same `asynchronousOperation` over and over again).

```
function doAsyncOp () {
    return asynchronousOperation().then(function(val) {
        return asynchronousOperation(val);
    }).then(function(val) {
        return asynchronousOperation(val);
    }).then(function(val) {
        return asynchronousOperation(val);
    });
};
```

With `async` functions, we can just act like `asynchronousOperation` is synchronous:

```
async function doAsyncOp () {
    var val = await asynchronousOperation();
    val = await asynchronousOperation(val);
    val = await asynchronousOperation(val);
    return await asynchronousOperation(val);
};
```

You don't even need the `await` keyword on that return statement because either way it will return a promise resolving to the final value.

Parallel Operations

One of the other great features of promises is the ability to run multiple asynchronous operations at once and continue on your way once all of them have completed. `Promise.all` is the way to do this according to the new ES6 spec. Here's an example:

```
function doAsyncOp () {
  return Promise.all([asynchronousOperation(), asynchronousOperation()])
    .then(function(vals) {
      vals.forEach(console.log);
      return vals;
    });
}
```

This is also possible with async functions, though you may still need to use `Promise` directly:

```
async function doAsyncOp () {
  var vals = await Promise.all([asynchronousOperation(), asynchronousOperation()]);
  vals.forEach(console.log.bind(console));
  return vals;
}
```

It's still much cleaner even with the `Promise.all` bit in there, but notice that I said "may" in the previous paragraph. I say this because there is a feature that has been discussed—not confirmed—that will allow [parallelism using `await*`](https://github.com/lukehoban/ecmascript-asyncawait#await-and-parallelism) (<https://github.com/lukehoban/ecmascript-asyncawait#await-and-parallelism>). The idea is that `await* EXPRESSION` would be converted to `await Promise.all(EXPRESSION)` behind the scenes, which allows us to be more terse and avoid using the `Promise` API directly. In this case the previous example would look like this:

```
async function doAsyncOp () {
  var vals = await* [asynchronousOperation(), asynchronousOperation()];
  vals.forEach(console.log.bind(console));
  return vals;
}
```

Handling Rejection

Promises have the ability to be resolved or rejected. Rejected promises can be handled with the second function passed to `then` or with the `catch` method. Since we're not using any `Promise` API methods, how would we handle a rejection? We do it with a `try` and `catch`. When using async functions, rejections are passed around as errors and this allows them to be handled with built-in JavaScript error handling code.

```
function doAsyncOp () {
  return asynchronousOperation().then(function(val) {
    return asynchronousOperation(val);
  }).then(function(val) {
    return asynchronousOperation(val);
  }).catch(function(err) {
    console.error(err);
  });
}
```

That's pretty similar to our chaining example except we replaced the final chained call with a `catch`. Here's what it would look like with async functions.

(1)

```
≡ async function doAsyncOp () {
  try {
    var val = await asynchronousOperation();
    val = await asynchronousOperation(val);
    return await asynchronousOperation(val);
  } catch (err) {
    console.error(err);
  }
};
```



It's not as terse as the other conversions to async functions, but it *is* exactly how you would do it with synchronous code. If you don't catch the error here, it'll bubble up until it is caught in the caller functions, or it will just not be caught and you'll kill execution with a run-time error. Promises work the same way, except that rejections don't *need* to be errors; they can just be a string explaining what went wrong. If you don't catch a rejection that was created with an error, then you will see a run-time error, but if you just use a string, then it will fail silently.

Broken Promises

To reject an ES6 promises you can use `reject` inside the `Promise` constructor, or you can throw an error—either inside the `Promise` constructor or within a `then` or `catch` callback. If an error is thrown outside of that scope, it won't be contained in the promise.

Here are some examples of ways to reject ES6 promises:

```
function doAsyncOp () {
  return new Promise( function(resolve, reject) {
    if ( somethingIsBad ) {
      reject('something is bad');
    }
    resolve('nothing is bad');
  });
}

/*-- or --*/

function doAsyncOp () {
  return new Promise( function(resolve, reject) {
    if ( somethingIsBad ) {
      reject(new Error('something is bad'));
    }
    resolve('nothing is bad');
  });
}

/*-- or --*/

function doAsyncOp () {
  return new Promise( function(resolve, reject) {
    if ( somethingIsBad ) {
      throw new Error('something is bad');
    }
    resolve('nothing is bad');
  });
}
```

Generally it is best to use the `new Error` whenever you can because it will contain additional information about the error, such as the line number where it was thrown, and a potentially useful stack trace.

Here are some examples where throwing an error will not be caught by the promise:

(1)

```
≡ function doAsyncOp () {
    // the next line will kill execution
    throw new Error('something is bad');
    return new Promise( function(resolve, reject) {
        if ( somethingIsBad ) {
            throw new Error('something is bad');
        }
        resolve('nothing is bad');
    });
}

// assume `doAsyncOp` does not have the killing error
function x () {
    var val = doAsyncOp.then(function() {
        // this one will work just fine
        throw new Error("I just think an error should be here");
    });
    // this one will kill execution
    throw new Error("The more errors, the merrier");
    return val;
}
```

With async functions promises are rejected by throwing errors. The scope issue doesn't arise—you can throw an error anywhere within an async function and it will be caught by the promise:

```
async function doAsyncOp () {
    // the next line is fine
    throw new Error('something is bad');

    if ( somethingIsBad ) {
        // this one is good too
        throw new Error('something is bad');
    }
    return 'nothing is bad';
}

// assume `doAsyncOp` does not have the killing error
async function x () {
    var val = await doAsyncOp;

    // this one will work just fine
    throw new Error("I just think an error should be here");

    return val;
}
```

Of course, we'll never get to that second error or to the `return` inside the `doAsyncOp` function because the error will be thrown and will stop execution within that function.

Gotchas

If you're new to async functions, one gotcha to be aware of is using nested functions. For example, if you have another function within your async function (generally as a callback to something), you may think that you can just use `await` from within that function. You can't. You can only use `await` directly within an `async` function. This does not work:

(1)

```
≡ async function getAllFiles (files) {
    return await* files.map(function(filename) {
        var file = await getFileAsync(filename);
        return parse(file);
    });
}
```



The `await` on line 3 is invalid because it is used inside a normal function. Instead, the callback function must have the `async` keyword attached to it.

```
async function getAllFiles (fileNames) {
    return await* fileNames.map(async function(fileName) {
        var file = await getFileAsync(fileName);
        return parse(file);
    });
}
```

It's obvious when you see it, but nonetheless it's something that you need to watch out for. In case you're wondering, here's the equivalent using promises:

```
function getAllFiles (fileNames) {
    return Promise.all(fileNames.map(function (fileName) {
        return getFileAsync(fileName).then(function (file) {
            return parse(file);
        });
    }));
}
```

The next gotcha relates to people thinking that `async` functions are synchronous functions. Remember, the code *inside* the `async` function will run as if it is synchronous, but it will still immediately return a promise and allow other code to execute outside of it while it works to fulfillment. For example:

```
var a = doAsyncOp(); // one of the working ones from earlier
console.log(a);
a.then(function() {
    console.log(`a` finished');
});
console.log('hello');

/* -- will output -- */
Promise Object
hello
`a` finished
```

You can see that `async` functions still utilize built-in promises, but they do so under the hood. This gives us the ability to think synchronously while within an `async` function, although others can invoke our `async` functions using the normal promises API or using `async` functions of their own.

But ES7 Doesn't Exist!

More from this author

[How to Solve the Global npm Module Dependency Problem \(https://www.sitepoint.com/solve-global-npm-module-dependency-problem/?utm_source=sitepoint&utm_medium=relatedinline&utm_term=&utm_campaign=relatedauthor\)](https://www.sitepoint.com/solve-global-npm-module-dependency-problem/?utm_source=sitepoint&utm_medium=relatedinline&utm_term=&utm_campaign=relatedauthor)

ES6 hasn't even been released yet (due June 2015), so why am I teaching you how to use a feature slated to release with ES7? Because, even if you can't use it natively, you can write it and use tools to compile it down to ES5. Async functions are all about making your code more readable and therefore more maintainable. As long as we have source maps, we can always work with the cleaner ES7 code.

There are several tools that can compile async functions (and other ES6/7 features) down to ES5 code, the most notable of which is [traceur](https://github.com/google/traceur-compiler/) (<https://github.com/google/traceur-compiler/>), which is a pure CLI tool. If you prefer to use a build tool, like Grunt, there are various plug-ins that utilize traceur. For example:

[grunt-traceur](https://www.npmjs.org/package/grunt-traceur) (<https://www.npmjs.org/package/grunt-traceur>) for Grunt
[gulp-traceur](https://www.npmjs.org/package/gulp-traceur) (<https://www.npmjs.org/package/gulp-traceur>) for Gulp
[es6ify](https://www.npmjs.org/package/es6ify) (<https://www.npmjs.org/package/es6ify>) for Browserify

Are you already taking advantage of the amazing power yielded to us through async functions? Is this something you'd consider using today? Let us know in the comments.

Happy Coding!

Was this helpful?  

More: [async functions](https://www.sitepoint.com/tag/async-functions/) (<https://www.sitepoint.com/tag/async-functions/>), [es7](https://www.sitepoint.com/tag/es7/) (<https://www.sitepoint.com/tag/es7/>), [generators](https://www.sitepoint.com/tag/generators/) (<https://www.sitepoint.com/tag/generators/>), [Promises](https://www.sitepoint.com/tag/promises/) (<https://www.sitepoint.com/tag/promises/>)



Meet the author

[Joe Zimmerman](https://www.sitepoint.com/author/joezimmerman/) (<https://www.sitepoint.com/author/joezimmerman/>)  (<https://twitter.com/JoeZimJS>)  (<https://plus.google.com/+Jozimjs/posts>)  (<https://www.linkedin.com/profile/view?id=34738916>)  (<https://www.reddit.com/user/jozimjs/>)  (<https://www.flickr.com/photos/122971964@N07/>)  (<https://github.com/jozimjs>)  (<https://www.youtube.com/user/jozim007/>)

Joe Zimmerman has been doing web development since he was 12. Since then, JavaScript has become his passion. He also loves to teach though his blog, spend time with his wife and children and lead them in God's Word.

No Reader comments

LATEST COURSES >

([/premium/courses/](https://www.sitepoint.com/premium/courses/))

PREMIUM COURSE

1h 1m

[Diving into ES2015](https://www.sitepoint.com/premium/courses/diving-into-es2015-2924)

(<https://www.sitepoint.com/premium/courses/diving-into-es2015-2924>)

PREMIUM COURSE

3h 7m

[JavaScript: Next Steps](https://www.sitepoint.com/premium/courses/javascript-next-steps-2921)

(<https://www.sitepoint.com/premium/courses/javascript-next-steps-2921>)

PREMIUM COURSE

1h 11m

[React The ES6 Way](https://www.sitepoint.com/premium/courses/react-the-es6-way-2914)

(<https://www.sitepoint.com/premium/courses/react-the-es6-way-2914>)

LATEST BOOKS >

[\(/premium/books/\)](/premium/books/)

PREMIUM BOOK

[ECMAScript 2015: A SitePoint Anthology](#)

(<https://www.sitepoint.com/premium/books/ecmascript-2015-a-sitepoint-anthology>)

PREMIUM BOOK

[Jump Start Git](#)

(<https://www.sitepoint.com/premium/books/jump-start-git>)

PREMIUM BOOK

[Full Stack JavaScript Development with MEAN](#)

(<https://www.sitepoint.com/premium/books/full-stack-javascript-development-with-mean>)

Get the latest in JavaScript, once a week, for free.

Enter your email

Subscribe

[Facebook](#) [LinkedIn](#) [Twitter](#)

About

[Our Story \(/about-us/\)](#)

[Advertise \(/advertise/\)](#)

[Press Room \(/press/\)](#)

[Reference \(<http://reference.sitepoint.com/css/>\)](#)

[Terms of Use \(/legals/\)](#)

[Privacy Policy \(/legals/#privacy\)](#)

[FAQ \(<https://sitepoint.zendesk.com/hc/en-us>\)](#)

[Contact Us \(<mailto:feedback@sitepoint.com>\)](#)

[Contribute \(/write-for-us/\)](#)

Visit

[SitePoint Home \(/\)](#)

[Themes \(/basetheme/\)](#)

[Podcast \(/versioning-show/\)](#)

[Forums \(<https://www.sitepoint.com/community/>\)](#)

[Newsletters \(/newsletter/\)](#)

[Premium \(/premium/\)](#)

[References \(/sass-reference/\)](#)

Connect

(<https://www.sitepoint.com/feed/>) (</newsletter/>) (<https://www.facebook.com/sitepoint>) (<http://twitter.com/sitepointdotcom>) (<https://plus.google.com/+sitepoint>)

© 2000 – 2016 SitePoint Pty. Ltd.