CODE > JAVASCRIPT

A Primer on ES7 Async Functions

by Joe Zimmerman 7 Dec 2014

Difficulty: Intermediate Length: Long Languages: English

JavaScript Programming Web Development ES6

If you've been following the world of JavaScript, you've likely heard of promises. There are some great tutorials online if you want to learn about promises, but I won't explain them here; this article assumes you already have a working knowledge of promises.

Promises are touted as the future of asynchronous programming in JavaScript. Promises really are great and help solve a lot of issues that arise with asynchronous programming, but that claim is only somewhat correct. In reality, promises are the *foundation* of the future of asynchronous programming in JavaScript. Ideally, promises will be tucked away behind the scenes and we'll be able to write our asynchronous code as if it were synchronous.

In ECMAScript 7, this will become more than some fanciful dream: It will become reality, and I will show you that reality—called async functions—right now. Why are we talking about this now? After all, ES6 hasn't even been completely finalized, so who knows how long it will be before we see ES7. The truth is you can use this technology right now, and at the end of this post, I will show you how.

The Current State of Affairs

Before I begin demonstrating how to use async functions, I want to go through some examples with promises (using ES6 promises). Later, I'll convert these examples to use async functions so you can see what a big difference it makes.

Examples

For our first example, we'll do something really simple: calling an asynchronous function and logging the value it returns.

```
function getValues() {
    return Promise.resolve([1,2,3,4]);
}

getValues().then(function(values) {
    console.log(values);
});
```

Now that we have that basic example defined, let's jump into something a bit more complicated. I'll be using and modifying examples from a post on my own blog that goes through some patterns for using promises in different scenarios. Each of the examples

asynchronously retrieves an array of values, performs an asynchronous operation that transforms each value in the array, logs each new value, and finally returns the array filled with the new values.

First, we'll look at an example that will run multiple asynchronous operations in parallel, and then respond to them immediately as each one finishes, regardless of the order in which they finish. The `getValues` function is the same one from the previous example. The `asyncOperation` function will also be reused in the upcoming examples.

```
function asyncOperation(value) {
    return Promise.resolve(value + 1);
}

function foo() {
    return getValues().then(function(values) {
        var operations = values.map(function(value) {
            return asyncOperation(value).then(function(newValue) {
                console.log(newValue);
                return newValue;
            });
        });

        return Promise.all(operations);
    }).catch(function(err) {
        console.log('We had an ', err);
    });
}
```

We can do the exact same thing, but make sure the logging happens in the order of the elements in the array. In other words, this next example will do the asynchronous work in parallel, but the synchronous work will be sequential:

```
function foo() {
    return getValues().then(function(values) {
        var operations = values.map(asyncOperation);

        return Promise.all(operations).then(function(newValues) {
            newValues.forEach(function(newValue) {
                console.log(newValue);
            });

            return newValues;
        });
    }).catch(function(err) {
        console.log('We had an ', err);
    });
}
```

Our final example will demonstrate a pattern where we wait for a previous asynchronous operation to finish before starting the next one. There is nothing running in parallel in this example; everything is sequential.

```
function foo() {
    var newValues = [];
    return getValues().then(function(values) {
        return values.reduce(function(previousOperation, value) {
            return previousOperation.then(function() {
                return asyncOperation(value);
            }).then(function(newValue) {
                console.log(newValue);
                newValues.push(newValue);
            });
        }, Promise.resolve()).then(function() {
        return newValues;
        });
    }).catch(function(err) {
        console.log('We had an ', err);
    });
}
```

Even with the ability of promises to reduce callback nesting, it doesn't really help much. Running an unknown number of sequential asynchronous calls will be messy no matter what you do. It's especially appalling to see all of those nested `return` keywords. If we

passed the `newValues` array through the promises in the `reduce`'s callback instead of making it global to the entire `foo` function, we'd need to adjust the code to have even more nested returns, like this:

```
function foo() {
    return getValues().then(function(values) {
        return values.reduce(function(previousOperation, value) {
            return previousOperation.then(function(newValues) {
                return asyncOperation(value).then(function(newValue) {
                    console.log(newValue);
                    newValues.push(newValue);
                    return newValues;
                });
            });
        }, Promise.resolve([]));
    }).catch(function(err) {
        console.log('We had an ', err);
    });
}
```

Don't you agree we need to fix this? Let's look at the solution.

Async Functions to the Rescue

Even with promises, asynchronous programming isn't exactly simple and doesn't always flow nicely from A to Z. Synchronous programming is so much simpler and is written and read so much more naturally. The Async Functions specification looks into a means (using ES6 generators behind the scenes) of writing your code as if it were synchronous.

How Do We Use Them?

The first thing that we need to do is prefix our functions with the `async` keyword. Without this keyword in place, we cannot use the all-important `await` keyword inside that function, which I'll explain in a bit.

The `async` keyword not only allows us to use `await`, it also ensures that the function will return a `Promise` object. Within an async function, any time you `return` a value, the function will *actually* return a `Promise` that is resolved with that value. The way to reject is to throw an error, in which case the rejection value will be the error object. Here's a simple example:

```
async function foo() {
    if( Math.round(Math.random()) )
        return 'Success!';
    else
        throw 'Failure!';
}

// Is equivalent to...

function foo() {
    if( Math.round(Math.random()) )
        return Promise.resolve('Success!');
    else
        return Promise.reject('Failure!');
}
```

We haven't even gotten to the best part and we've already made our code more like synchronous code because we were able to stop explicitly messing around with the `Promise` object. We can take any function and make it return a `Promise` object just by adding the `async` keyword to the front of it.

Let's go ahead and convert our `getValues` and `asyncOperation` functions:

```
async function getValues() {
    return [1,2,3,4];
}

async function asyncOperation(value) {
    return value + 1;
}
```

Easy! Now, let's take a look at the best part of all: the `await` keyword. Within your async function, every time you perform an operation that returns a promise, you can throw the `await` keyword in front of it, and it'll stop executing the rest of the function until the returned

promise has been resolved or rejected. At that point, the `await promisingOperation()` will evaluate to the resolved or rejected value. For example:

```
function promisingOperation() {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            if( Math.round(Math.random()) )
                resolve('Success!');
            else
                reject('Failure!');
        }, 1000);
    }
}

async function foo() {
    var message = await promisingOperation();
    console.log(message);
}
```

When you call `foo`, it'll either wait until `promisingOperation` resolves and then it'll log out the "Success!" message, or `promisingOperation` will reject, in which case the rejection will be passed through and `foo` will reject with "Failure!". Since `foo` doesn't return anything, it'll resolve with `undefined` assuming `promisingOperation` is successful.

There is only one question remaining: How do we resolve failures? The answer to that question is simple: All we need to do is wrap it in a `try...catch` block. If one of the asynchronous operations gets rejected, we can `catch` that and handle it:

```
async function foo() {
    try {
        var message = await promisingOperation();
        console.log(message);
    } catch (e) {
        console.log('We failed:', e);
    }
}
```

Now that we've hit on all the basics, let's go through our previous promise examples and convert them to use async functions.

Examples

The first example above created `getValues` and used it. We've already re-created `getValues` so we just need to re-create the code for using it. There is one potential caveat to async functions that shows up here: The code is *required* to be in a function. The previous example was in the global scope (as far as anyone could tell), but we need to wrap our async code in an async function to get it to work:

```
async function() {
    console.log(await getValues());
}(); // The extra "()" runs the function immediately
```

Even with wrapping the code in a function, I still claim it's easier to read and has fewer bytes (if you remove the comment). Our next example, if you remember correctly, does everything in parallel. This one is a little bit tricky, because we have an inner function that needs to return a promise. If we're using the `await` keyword inside of the inner function, that function also needs to be prefixed with `async`.

```
async function foo() {
    try {
    var values = await getValues();

        var newValues = values.map(async function(value) {
            var newValue = await asyncOperation(value);
            console.log(newValue);
            return newValue;
        });

        return await* newValues;
    } catch (err) {
        console.log('We had an ', err);
    }
}
```

You may have noticed the asterisk attached to the last `await` keyword. This seems to still be up for debate a bit, but it looks like `await*` will essentially auto-wrap the expression to its right in `Promise.all`. Right now, though, the tool we'll be looking at later doesn't support `await*`, so it should be converted to `await Promise.all(newValues);` as we're doing in the next example.

The next example will fire off the `asyncOperation` calls in parallel, but will then bring it all back together and do the output sequentially.

```
async function foo() {
    try {
    var values = await getValues();
        var newValues = await Promise.all(values.map(asyncOperation));

        newValues.forEach(function(value) {
            console.log(value);
        });

        return newValues;
    } catch (err) {
        console.log('We had an ', err);
    }
}
```

I love that. That is extremely clean. If we removed the `await` and `async` keywords, removed the `Promise.all` wrapper, and made `getValues` and `asyncOperation` synchronous, then this code would still work the exact same way except that it'd be synchronous. That's essentially what we're aiming to achieve.

Our final example will, of course, have everything running sequentially. No asynchronous operations are performed until the previous one is complete.

```
async function foo() {
    try {
    var values = await getValues();

        return await values.reduce(async function(values, value) {
            values = await values;
            value = await asyncOperation(value);
            console.log(value);
            values.push(value);
            return values;
        }, []);
    } catch (err) {
        console.log('We had an ', err);
    }
}
```

Once again, we're making an inner function `async`. There is an interesting quirk revealed in this code. I passed `[]` in as the "memo" value to `reduce`, but then I used `await` on it. The value to the right of `await` isn't required to be a promise. It can take any value, and if it isn't a promise, it won't wait for it; it'll just be run synchronously. Of course, though, after the first execution of the callback, we'll actually be working with a promise.

This example is pretty much just like the first example, except that we're using `reduce` instead of `map` so that we can `await` the previous operation, and then because we are using `reduce` to build an array (not something you'd normally do, especially if you're building an array of the same size as the original array), we need to build the array within the callback to `reduce`.

Using Async Functions Today

Now that you've gotten a glimpse of the simplicity and awesomeness of async functions, you might be crying like I did the first time I saw them. I wasn't crying out of joy (though I almost did); no, I was crying because ES7 won't be here until I die! At least that's how I *felt*. Then I found out about Traceur.

Traceur is written and maintained by Google. It is a transpiler that converts ES6 code to ES5. That doesn't help! Well, it wouldn't, except they've also implemented support for async functions. It's still an experimental feature, which means you'll need to explicitly tell the

compiler that you're using that feature, and that you'll definitely want to test your code thoroughly to make sure there aren't any issues with the compilation.

Using a compiler like Traceur means that you'll have some slightly bloated, ugly code being sent to the client, which isn't what you want, but if you use source maps, this essentially eliminates most of the downsides related to development. You'll be reading, writing, and debugging clean ES6/7 code, rather than having to read, write, and debug a convoluted mess of code that needs to work around the limitations of the language.

Of course, the code size will still be larger than if you had hand-written the ES5 code (most likely), so you may need to find some type of balance between maintainable code and performant code, but that is a balance you often need to find even without using a transpiler.

Using Traceur

Traceur is a command-line utility that can be installed via NPM:

```
npm install -g traceur
```

In general, Traceur is pretty simple to use, but some of the options can be confusing and may require some experimentation. You can see a list of the options for more details. The one we're really interested in is the `--experimental` option.

You need to use this option to enable the experimental features, which is how we get async functions to work. Once you have a JavaScript file (`main.js` in this case) with ES6 code and async functions included, you can just compile it with this:

```
traceur main.js --experimental --out compiled.js
```

You can also just run the code by omitting the `--out compiled.js`. You won't see much unless the code has `console.log` statements (or other console outputs), but at the very least, you can check for errors. You'll likely want to run it in a browser, though. If that's the case, there are a few more steps you need to take.

1. Download the `traceur-runtime.js` script. There are many ways to get it, but one of the easiest is from NPM: `npm install traceur-runtime`. The file will then be available as `index.js` within that module's folder.
2. In your HTML file, add a `script` tag to pull in the Traceur Runtime script.
3. Add another `script` tag below the Traceur Runtime script to pull in `compiled.js`.

After this, your code should be up and running!

Automating Traceur Compilation

Beyond just using the Traceur command-line tool, you can also automate the compilation so you don't need to keep returning to your console and re-running the compiler. Grunt and Gulp, which are automated task runners, each have their own plugins that you can use to automate Traceur compilation: grunt-traceur and gulp-traceur respectively.

Each of these task runners can be set up to watch your file system and re-compile the code the instant you save any changes to your JavaScript files. To learn how to use Grunt or Gulp, check out their "Getting Started" documentation.

Conclusion

ES7's async functions offer developers a way to *actually* get out of callback hell in a way that promises never could on their own. This new feature allows us to write asynchronous code in a way that is extremely similar to our synchronous code, and even though ES6 is still awaiting its full release, we can already use async functions today through transpilation. What are you waiting for? Go out and make your code awesome!

Joe Zimmerman

I have been doing web development ever since I found an HTML book on my dad's shelf when I was 12. Since then, JavaScript's popularity has grown and so has my passion for it. I also love to teach others though blogging, here and elsewhere. When I'm not writing code, I'm spending time with my wife and children and leading them in God's Word.

joezimjs

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

15 Comments **Tuts+ Hub** **Login**

Recommend 1 **Share** **Sort by Best**

Join the discussion…

James Kyle • 2 years ago

Hey Joe, great article. You should check out 6to5 as it doesn't have the problems you mentioned with Tracuer. It tries to keep the compiled output as close to the original source as possible. https://6to5.github.io/

4 • Reply • Share ›

stevebennett James Kyle • a year ago

Btw, anyone else a bit confused: 6to5 is what Babel used to be called. https://babeljs.io

• Reply • Share ›

Joe Zimmerman James Kyle • 2 years ago

I looked at their project a while back and at the time they didn't have as many features supported as Traceur and were specifically missing some that I wanted, but you're right. That is a great alternative to Traceur and may actually be nicer than Traceur, especially now that they handle everything I want. I'll need to take a look and do some comparisons with my own code.

• Reply • Share ›

James Kyle Joe Zimmerman • 2 years ago

Well I'm a contributor to 6to5, so let me know if you have any questions about it, or drop by our gitter room (https://gitter.im/6to5/6to5).

• Reply • Share ›

Iman • 2 years ago

There are some awesome features like single callback for multiple async functions when you use jQuery deffered promise functions. Is there anything similar in ES7?

Reply • Share ›

skrat • 2 years ago

Yet again, no matter how hard the ECMA folks try, CSP wins on readability, simplicity, composability. We had deferreds, futures, promises, now async functions :) but still, have a look at how asynchronicity is tamed using js-csp (backed by generators), or ClojureScript's elegant core.async (backed by state machines). Go and find out how awesome CSP is!

Reply • Share ›

Joe Zimmerman ➚ skrat • 2 years ago

ClojureScript is an entirely new language that many JS developers (including myself) can't and/or don't want to learn. If you try to find a job with CSP, you're probably not going to find much, whereas JavaScript is in extremely high demand. Instead of bashing JavaScript or the "ECMA folks", how about you use that voice to try to get browsers to implement other languages or work with the standards bodies to make JavaScript better? Or, if you really think CSP is so great, get people to love it by writing about it in articles instead of at the bottom of an article where only a minority will see it.

1 Reply • Share ›

skrat ➚ Joe Zimmerman • 2 years ago

Bashing - Violent physical assault. I really do that when I propose an alternative? Async functions is just a sugar around generators and promises. Where do async/await come from? C#, Microsoft is pushing it to ECMA. Do we need another language feature when the same can be accomplished with library code? Hardly. Is Promise the right primitive for async. programming? I leave that up to you. From my experience, Promise is nice when your program is low on asynchronicity, otherwise you end up in Promise hell. Replacing it with CSP and its channels as async. primitives in a complex WebGL rendering engine, greatly simplified the entire code base, including asset loading, pre/post-processing and even the rendering loop.

Reply • Share ›

panesofglass ➚ skrat • 2 years ago

I agree CSP probably offers a better, more reliable model for asynchronous programming than the syntactic sugar of async/await. I am curious to learn how the following hold for the JS implementation of async/await: http://tomasp.net/blog/csharp-...

Reply • Share ›

Alxandr • 2 years ago

Your rewrites to async are extremely crude. When using async and await you should really use standard control structures such as loops. Especially for the last one that uses `await values.reduce`. You could just do a simple for loop over the array and you'd have the exact same effect, and normal mortals would understand it.

Reply • Share ›

Joe Zimmerman ➚ Alxandr • 2 years ago

I actually greatly prefer functional ways of looping, and with the proliferation of libraries like jQuery and Underscore that push their own looping methods, I would have to assume that `map`, `reduce`, and `forEach`/`each` and how they are used would be common knowledge.

I'm curious to see how you would approach the problems if you think my conversions to async are so crude.

4 Reply • Share ›

Alxandr ➚ Joe Zimmerman • 2 years ago

Straight forward: https://gist.github.com/Alxand...
Note, since we're talking ES7, I'm assuming stuff like `for..of` and `let`. Should be really easy to rewrite to a ES5 loop though, and just replace `let` with `var`.

1 Reply • Share ›

Joe Zimmerman ➚ Alxandr • 2 years ago

That would certainly work for the last example, and it is quite readable and offers the synchronicity implicitly by not deferring the other asynchronous task to additional functions. I like that, but you did say that my "rewrites" (plural) were crude, so I'd love to learn what else you think could be improved. Using `await` inside standard loops won't work if you're trying to run them in parallel, so using `map` seems the cleaner option there, still. Of course, I could replace the `forEach` in the examples with a `for..of` loop, but that shouldn't make much difference.

1 Reply • Share ›

Alxandr ➚ Joe Zimmerman • 2 years ago

Sorry for delayed answer. Had exams this week :-/.

Anyways, I spoke a bit early, I agree on that. Using map on the second example is actually really elegant.

The first sample though, is something that I would generally never do. It's this weird mix of logging when you get a new answer (ie. not caring about order for the logging), while at the same time waiting for all of them to complete before returning the resolved array (suddenly, order does matter again).

If order is unimportant, I would argue that something akin to observables (I think in the JS world they are normally referred to as "streams") is a much more fitting model.

It was mainly the last example I had an issue with, and I spoke to harshly, and I'm sorry for that. I read this post as a developer that has used async/await for a long time (in .NET), and saw someone who had just discovered them for JS and spoke out "guys, here's how you use this shiny new feature", except it was all wrong (which ofcause it wasn't). So I spoke too early.

Though I still mean that the first example is not something that should be put up as "here's how you do it", because (in my opinion at least) it's not something that should be done.

[Edit]
Also, I'd still like to see the last example be changed. If people end up coming here (from google or whatever) in search for how things are done, I'd argue that having the best examples possible is important.

 • Reply • Share ›

Joe Zimmerman ➚ Alxandr • 2 years ago

Thank you for the apology. I knew you meant no ill will, but it's still appreciated. The 3 examples I used were just 3 patterns I have run across:
1) "Fully" parallel: It doesn't matter what order things happen in, though I did return an array that was in the original order. The main thing was that you were able to perform an action (just console.log in the examples) on a value the moment it comes in regardless of order
2) Parallel to Sequential: retrieve the values using async operations all in parallel, but make sure we don't work on them until we have them all so we can do that in the correct sequence.
3) Fully Sequential: Don't start the next asynchronous operation until the previous one has finished and we have processed it.

There's nothing inherently wrong with the first example. Promise.all/await* on the array of promises is the simplest way to return a promise that only fulfills once all of the operations are finished, which is what you want. We get the values in the original order at the end by default, not necessarily by design. Also, the way it is done helps to show that inner functions that use `await` also need to use the `async` prefix.

I will see what I can do to "fix" the last example, though I'll probably just add an additional example saying "here's how to do it even better/cleaner!". Thank you very much for your honest feedback. I've learned a few things through this discussion. :)

1 • Reply • Share ›

Help and Support +

Email Newsletters

Get Envato Tuts+ updates, news, surveys & offers.

Email Address

Subscribe

Privacy Policy