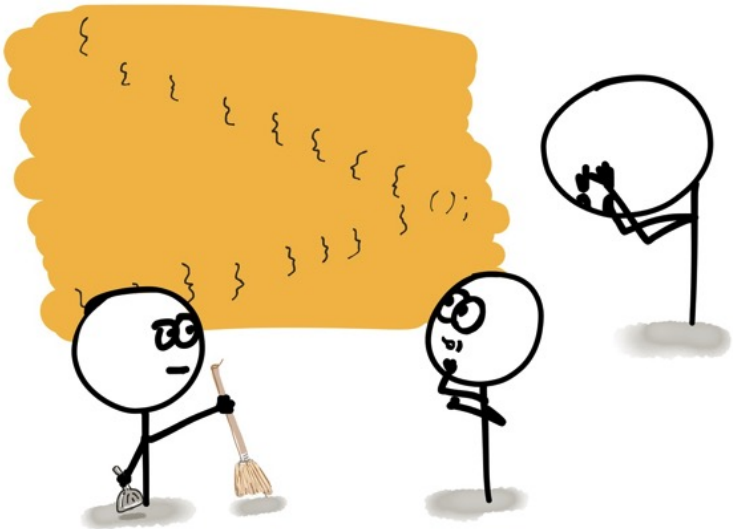
You Need ES2017's Async Functions. Here's Why ...

April 18, 2017 By Derick



If you've ever written code like this, you know the pain that is asynchronous workflow in JavaScript.

```
1  function createEmployeeWorkflow(cb){
2
```

```
  createEmployee(function(err, employee){
    If (err) { return cb(err); }

    if (employee.needsManager()) {

      selectManager(employee, function(err, manager){
        If (err) { return cb(err); }

        employee.manager = manager;
        saveEmployee(employee, function(err){
          If (err) { return cb(err); }

          cb(undefined, employee);
        });
      });

    } else {

      saveEmployee(employee, function(err){
        If (err) { return cb(err); }
        cb(undefined, employee);
      });

    }
  });
}
```

1.js hosted with ♥ by GitHub view raw

Nested function after nested function. Multiple redundant (but probably necessary) checks for errors.

It's enough to make you want to quit JavaScript… and this is a simple example!

Now imagine how great it would be if your code could look like this.

```
function createEmployeeWorkflow(cb){
  var err;

  try {
    var employee = createEmployee();
```

```
    if (employee.needsManager()){
      var manager = selectManager(employee);
      employee.manager = manager;
    }

    saveEmployee(employee);
  } catch (ex) {
    err = ex;
  }

  cb(err, employee);
}
```

2.js hosted with ♥ by GitHub view raw

Soooo much easier to read… as if the code were entirely synchronous! I'll take that any day, over the first example.

Using Async Functions

With async functions, that second code sample is incredibly close to what you can do. It only takes a few additional keywords to mark these function calls as async, and you're golden.

```
async function createEmployeeWorkflow(cb){
  var err;

  try {
    var employee = await createEmployee();

    if (employee.needsManager()){
      var manager = await selectManager(employee);
      employee.manager = manager;
    }

    await saveEmployee(employee);
  } catch (ex) {
    err = ex;
  }

  cb(err, employee);
}
```

3.js hosted with ❤ by GitHub view raw

Did you notice the difference, here?

With the addition of “async” to the outer function definition, you can now use the “await” keyword to call your other async functions.

By doing this, the JavaScript runtime will now invoke the async functions in a manner that allows you to wait for a response without using a callback. The code is still asynchronous where it needs to be, and synchronous where it can be.

This code does the same thing, has the same behavior from a functionality perspective. But visually, this code is significantly easier to read and understand.

The question now, is how do you create the async functions that save so much extra code and cruft, allowing you to write such simple workflow?

Writing Async Functions

If you’ve ever used a JavaScript Promise, then you already know how to create an async function.

Look at how the “createEmployee” function might be written, for example.

```
async function createEmployee(){
  return new Promise((resolve, reject) => {

    // do stuff here to create the employee
    var employee = // ...

    // now check if it worked or not
    if (/* some success case */) {
      resolve(employee);
    } else {
      reject(someError);
```

```
    }

  });
}
```

4.js hosted with ❤ by GitHub view raw

This code immediately creates and returns a promise. Once the work to create the employee is done, it then checks for some level of success and resolves the promise with the employee object. If there was a problem, it rejects the promise.

The only difference between this function and any other function where you might have returned a promise, is the use of the “async” keyword in the function definition.

But it’s this one keyword that solves the nested async problem that JavaScript has suffered with, forever.

Async With Flexibility

Beyond the simplicity of reading and understanding this code, there is one more giant benefit that needs to be stated.

With the use of promises in the the async functions, you have options for how you handle them. You are not required to “await” the result. You can still use the promise that is returned.

This code is just as valid as the previous code.

```
function createEmployeeWorkflow(cb){

  createEmployee()
    .then((employee) => {

      // ... all the other code

      cb(null, employee);

```

```
    }).catch((ex) {
      return cb(ex);
    });

}
```

5.js hosted with ❤ by GitHub view raw

Yes, this code still calls the same async createEmployee function. But we're able to take advantage of the promises that are returned when we want to.

And if you look back at the 3rd code sample above, you might remember that I was calling async functions but ultimately using a callback to return the result. Yet again, we see more flexibility.

Reevaluating My Stance On Promises

In the past, I've made some pretty strong statements about how I never look to promises as my first level of async code. I'm seriously reconsidering my position.

If the use of promises allows me to write such easily readable code, then I'm in.

Of course, now the challenge is getting support for this in the majority of browsers, as I'm not about to drop a ton of terrible pre-compiler hacks and 3rd party libraries into a browser to make this work for the web.

Node.js on the other hand? Well, it's only a matter of time before v8.0 is stable for release.

For now, though, I'll play with v7.6+ in a Docker container and get myself prepared for the new gold standard in asynchronous JavaScript.

RELATED POST

Docker for JavaScript Developers: On-Site Training

3 Features of ES7 (and Beyond) That You Should Be ...

How a 650MB Node.js Image for Docker Uses Less Spa...

3 Rules For When A New JavaScript Feature Is Ready...

Never Use The :latest Image From Docker Hub

Filed Under: Async Await, Callbacks, Error Handling, ES2017, Functions, JavaScript, Node.js, Promises

About Derick

Derick Bailey is a developer, entrepreneur, author, speaker and technology leader in central Texas (north of Austin). He's been a professional developer since the late 90's, and has been writing code since the late 80's. Derick has built software for organizations of all shapes and sizes, including contributions to Microsoft's MDSN library, running several very highly regarded open source projects, creating software solutions for large financial organizations, healthcare orgnaizations, world-class airlines, the U.S. government, and more. These days, Derick spends most of his time working on content for his own entrepreneurial efforts at WatchMeCode.net, playing video games when he gets a chance, and writing code for for his few remaining clients. You can reach Derick at DerickBailey.com or on twitter, @derickbailey.