Serially Iterating An Array, Asynchronously

March 12, 2015 By derickbailey

I recently found myself looking at an array of functions in JavaScript. I needed to iterate through this list, process each function in an asynchronous manner (providing a “done” callback method), and ensure that I did not move on to the next item in the list until the current one was completed.

The answer I found was in recursion and shifting through the array (grabbing the first item out of the array on each pass).

```
1  // "steps" is an array of functions... work to be done
2  function run(steps, data, done){
3
```

```
4    // hide the recursion in the parent function
5    function runStep(steps, data, done){
6
7      // our recursion exit strategy:
8      // no items left? we're done.
9      if (steps.length === 0) {
10       return done();
11     }
12
13     // get the first item in the array
14     var step = steps.shift();
15
16     // process the next item using recursion
17     function next(){
18       // setImmediate prevents call stack overflow
19       setImmediate(function(){
20         runStep(steps, data, done);
21       });
22     }
23
24     // run the step, providing a "done" callback
25     step(data, function(err){
26       if (err) { return done(err); }
27       // call the recursion for the next item
28       next();
29     });
30   }
31
32   // kick it off, here
33   runStep(steps, data, done);
34 };
```

1.js hosted with ♥ by GitHub view raw

It's a little ugly, granted, but it works.

Breaking It Down

The first thing I'm doing is hiding the recursion inside of the parent function. This keeps the public API for the method clean.

At the bottom of the parent function, I'm kicking off the nested recursive function by making a call to it and passing in the original list of steps that need to be run.

The recursive function does a number of things, including an initial check for the recursion's exit strategy. If there are no items left in the array, then exit the recursion by returning and calling the original "done" method that was passed in to the parent function.

If there are steps to process, grab the first one in the list. Then set up a "next" function, using a closure around the updated "steps" list and the other parameters that I need. This "next" function is responsible for the recursive method call, and it does this using a setImmediate call to ensure we avoid any call stack limitations. Note: If you're using this in a browser, you'll probably want setTimeout instead of setImmediate.

Once the "next" method is setup, the actual "step" is processed by calling it as a function. I pass in the "data" argument that I need, and provide a callback for it. The callback checks for an error, and forwards it if it finds one. If no errors happened, it calls the "next" function which invokes the recursive call to process the next item in the list.

The Important Part

The critical point in this code, which allowed me to ensure that I am only processing one step at a time, is the callback function that I passed in to the "step" function.

```
// run the step, providing a "done" callback
step(data, function(err){
  if (err) { return done(err); }
  // call the recursion for the next item
  next();
});
```

With this callback function, I'm allowing the step to run asynchronously. It doesn't matter how long the step takes to complete its work. When it is done, the step calls the callback that I provided. Within this callback, the "next()" function call kicks off the processing of the next item in the list.

Could Be Improved

This code works. In fact, it works well. I've got it running as a very important part of my Migroose framework for MongoDB / MongooseJS Data Migrations.

But I know this code could probably be improved… in several ways.

First off, the use of setImmediate means it is possible to enter an infinite loop and never exit. I avoided a call stack limitation problem with this, but introduced another potentially disastrous problem.

Second, this code isn't easy to read. I could improve this a little bit by using a more explicit queue construct on top of my array, but I know there are other ways in which I could improve readability and understandability.

Then there's the problem of array.shift() to get the next item. This is a destructive call, changing the contents of the array. If you're using an array that needs to be maintained the way it is, you don't want to use this method. In my case, the array is temporal – I built it just for this iteration /

step execution call. So in my case, it's ok for me to destroy this array as I'm processing it. I'm never going to use the array again. In any other circumstance, though, I would suggest keeping an index of where you're at, and using the index to process the next item.

Lastly, there are probably some performance considerations and memory considerations for the closures and use of setImmediate, among other things. As yet, I have not run into performance or memory problems with this. However, the introduction of a closure always includes the possibility of a memory leak – and that possibility is multiplied by the number of iterations (length of my array, in this case).

How Would You Do It?

I'm curious – what improvements would you make in this code?

What alternative methods would you have used, to process these tasks sequentially and asynchronously?

I'm sure there are a dozen libraries out there, that would take care of this for me. Sometimes it's nice to not bother with a third party library, but I'd be interested in seeing how you would handle this.

So drop some comments in the box below!

RELATED POST

10 Myths About Docker That Stop Developers Cold

Docker Recipes Update: Speed Up npm install In Mou...

Update and a Bonus Recipe for the Docker Recipes e...

A Sneak Peak at Docker Recipes for Node.js Develop...

Docker Recipes for Node.js: Pre-sale Dates & ...

Filed Under: JavaScript, NodeJS, Patterns, Recursion, Tips And Tricks

About derickbailey

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. In his spare time, he gets called a spamming marketer by people on Twitter, and blurts out all of the stupid / funny things he's ever done in his career on his email newsletter.

DERICK BAILEY AROUND THE WEB

Twitter: @derickbailey

Google+: DerickBailey

Screencasts: WatchMeCode.net

eBook: Building Backbone Plugins