Asynchronous Programming Techniques

In this lab, you will familiarize yourself with various asynchronous programming techniques, a mainstay of Node development.

# Objectives

In this lab, you will learn to

* manage scope when using the this keyword in the presence of asynchronous callbacks, and
* see how to break up long-running tasks in order to improve Node's performance.

# Managing Scope in Callbacks

First, let's have a look at how you might write a class to encapsulate a filename and a method to determine whether or not that file exists to see how the use of this might be problematic.

## Using this in callbacks

1. Open file scope-this.js in your IDE or text editor. Notice that we're using the function keyword to define a class called FileThing with a name property and a method called exists that asynchronously determines whether the file referred to by this.name exists.

In particular, have a look at how we're calling fs.open then inspecting the error given to us in the supplied onOpened callback. If we couldn't open the file, we ensure that it was because it didn't exist, log that fact, then inform our caller appropriately. Otherwise, we successfully opened the file, so we close it via fs.close (skipping a close callback) and invoke the supplied callback with no error and a true flag.

## Observing the loss of this

1. Now, use node to execute scope-this.js. You should see output similar to the following (highlighting added for emphasis).

$ node scope-this.js   
Opening: thisFileDoesNotExist  
Failed opening file: undefined  
file does NOT exist

You should see that the file indeed does not exist, but notice the second line of output, where we log that the file wasn't found. It says "Failed opening file: undefined". This meant that the expression this.name returned the JavaScript literal undefined! Why?

The scope of this, at this location, is the enclosing function, onOpened. Since it doesn't define a property named name, JavaScript correctly returns undefined! Let's fix this (pun intended).

1. The first change we will make is to add 'use strict'; to the top of scope-this.js. Run the program again and you will see a very different result:

$ node scope-this.js  
Opening: thisFileDoesNotExist  
/lesson-30-asynchronous-programming-techniques/lab/scope-this.js:13  
 console.log('Failed opening file: ' + this.name);  
 ^  
TypeError: Cannot read property 'name' of null  
 at onOpened (/lesson-30-asynchronous-programming-techniques/lab/scope-this.js:13:53)  
 at Object.oncomplete (fs.js:107:15)  
  
Process finished with exit code 8

The 'use strict' directive alters how the interpreter deals with this.

## Set a reference to the this we meant

1. Create a new file called scope-self.js and copy the contents of scope-this.js into it.

Remember that the scope of this in functions defined in a class is the class's members.

1. Since the this we meant in the onOpened function was FileThing's this, which is present in the function exists, create a new variable inside function exists called self and set it to the this that we meant (that was a mouthful).
2. Replace all uses of this in exists and in the onOpened callback with self.
3. Execute scope-self.js via node, and you should the following output, which properly echoes the filename in the "Failed opening file" message.

$ node scope-self.js   
Opening: thisFileDoesNotExist  
Failed opening file: thisFileDoesNotExist  
file does NOT exist

When you see the output, you may move on to the next step.

# Breaking up long-running tasks

Remember that Node is single-threaded, meaning that it is possible for our code to cause Node to halt all work. This is obviously undesirable, so let's have a look at how we can refactor code to yield its processing, letting Node catch up on I/O that may have been queued while Node was processing our logic.

Note that the code for this lab is already completely written and serves instead as an example of yielding.

This example presents a brute-force means of calculating prime numbers up to a given exclusive limit, in order to provide a good example of long-running, compute-intensive code.

1. Open file prime-obnoxious.js, which contains the function computePrimes. Notice that the function takes a limit greater than or equal to one, handles some boundary conditions, then jumps into the main loop of calculating prime numbers. The loop starts with prime candidate 3 & factor 2 and increments each candidate once it's found to have an integral factor other than itself or all factors have been exhausted.
2. Run file prime-obnoxious.js via node. You should see output similar to the following. Your timing may vary depending on your hardware and current load.

$ node prime-obnoxious.js   
9593 primes  
3.033 secs

During the period that function computePrimes is running, Node can do nothing else. Other I/O-bound tasks, if present, would be queuing up and waiting for Node's event loop to finish executing computePrimes before they could be handled.

1. Now open file prime-polite.js. The first thing that you should notice is that we've introduced into computePrimes an inner function computePrimeBatch, also known as a JavaScript *closure*, that breaks up the task of computing primes into smaller chunks.

A JavaScript closure is roughly a function defined within another function that captures (or "closes around") and remembers all independent variables that are within scope at the time the closure is executed.

After each chunk has evaluated a certain number of prime candidates, it yields control to Node via the setImmediate function so that Node can allow other tasks to execute.

There are two ways of yielding control in Node: setImmediate and process.nextTick. setImmediate queues the given function behind whatever I/O event callbacks that are already in the event queue, ensuring I/O is not being starved. process.nextTick, however, queues the given function at the head of the event queue so that it executes immediately after the current function completes, which may starve I/O. In general, prefer setImmediate.

1. Run prime-polite.js via node. You should see output similar to the following. Again, your timing may vary depending on hardware & load.

$ node prime-polite.js  
9593 primes  
3.681 secs

Notice the time it took the polite version of the script to complete was about 20% slower than the obnoxious version. Why? Because we're yielding our algorithm's execution to Node so that Node can process any I/O-bound tasks that may be queued, which takes time.

While the performance of our algorithm suffers, the overall throughput of Node remains high because we're giving Node a chance to service any I/O-bound tasks that may have been waiting during our algorithm's processing.

This lab is now complete!