Extracted from:

CoffeeScript

Accelerated JavaScript Development

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

CoffeeScript

Accelerated JavaScript Development

Trevor Burnham

Foreword by Jeremy Ashkenas

edited by Michael Swaine

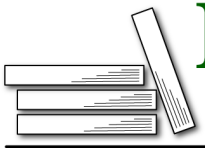
CoffeeScript

Accelerated JavaScript Development

Trevor Burnham

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-78-4
Printed on acid-free paper.
Book version: P1.0—July 2011

Running JavaScript on the server has long been a dream of web developers. Rather than switching back and forth between a client-side language and a server-side language, a developer using a JavaScript-powered server would only need to be fluent in that lingua franca of web apps—or in its twenty-first-century offshoot, CoffeeScript.

Now that dream is finally a reality. In this chapter, we'll take a brief tour of Node.js, starting with its module pattern (part of the CommonJS specification). Then we'll figure out just what an "evented architecture" is, with its implications for both server performance and our sanity. Finally, we'll add a Node back end to our 5x5 project from the last chapter, with real-time multiplayer support powered by WebSocket.

6.1 What Is Node.js?

Ignore the name: Node.js isn't a JavaScript library. Instead, Node.js is a JavaScript *interpreter* (powered by V8, the engine used by Google's Chrome browser) that interfaces with the underlying operating system. That way, JavaScript run by Node.js can read and write files, spawn processes, and—most enticingly—send and receive HTTP requests.

Like CoffeeScript, Node is a new project (dating to early 2009) that's taken off rapidly and attracted all kinds of excitement. Witness the Node.js Knockout, a Rails Rumble-inspired competition to develop the best Node app in forty-eight hours.[1]

A number of awesome projects have already been written with Node and CoffeeScript. The following is a small, select sampling. You might want to come back to this list after you've completed the book; reading real-world code is a great way to take your mastery to the next level:

- *Docco* [Ash11]: Uber-computer scientist Donald Knuth advocated "literate programming," in which code and comments are written so that someone encountering the program for the first time can understand it just by reading it once. Docco, written by Jeremy Ashkenas, supports this methodology by generating beautiful web pages in which comments and code are displayed side-by-side.
- *Eco* [Ste11]: Say you're writing a Node-based web application. You have all of these HTML skeletons and a mess of application code, but you're not sure how to combine the two. Eco lets you embed CoffeeScript *within* your markup, turning it into a server-side templating language.

1. http://nodeknockout.com/

- *Zappa* [NM11]: Creating web applications from scratch has never been simpler. Zappa is a layer on top of Node's popular Express framework that lets you succinctly define how your web server should respond to arbitrary HTTP requests.[2] Works great with Eco, too!
- *Zombie.js* [Ark11]: There's a new kid on the full-stack web app testing block: Zombie.js. Zombie lets you validate your application's behavior with the power of Sizzle, the same selection engine that powers jQuery. Not only is it easy to use, it's also insanely fast.

You can find a more comprehensive list of CoffeeScript-powered apps of all kinds at http://github.com/jashkenas/coffee-script/wiki/In-The-Wild.

6.2 Modularizing Code with 'exports' and 'require'

In past chapters, we've used global to put variables in an application-wide namespace. While global has its place, Noders generally prefer to keep their code nice and modular, with each file having its own namespace. How, then, do you share objects from one file with another?

The answer is a special object called exports, which is part of the CommonJS module standard. A file's exports object is returned when another file calls require on it. So, for instance, let's say that I have two files:

Download Nodejs/app.coffee

```
util = require './util'

console.log util.square(5)
```

Download Nodejs/util.coffee

```
console.log 'Now generating utility functions...'
exports.square = (x) -> x * x
```

When you run coffee app.coffee, require './util' executes util.coffee and then returns its exports object, giving you the following:

```
Now generating utility functions...
25
```

You might be wondering why we didn't need to specify a file extension. A .js file extension is always optional under Node.js. .coffee is also optional but only if the running application has loaded the coffee-script library, which is always implicitly done when we use coffee to run a file. coffee-script also tells Node how to handle CoffeeScript files. So if we compiled app to JavaScript but not util, then we'd have to write this:

2. http://expressjs.com

Download Nodejs/app.js

```
require('coffee-script');
var util = require('./util');
console.log(util.square(5));
```

When a library's name isn't prefixed with . or /, Node looks for a matching file in one of its library paths, which you can see by looking at require.paths.

By convention, a library's name for require is the same as its name for npm. Recall, for instance, that we used npm install -g coffee-script to install CoffeeScript. That gave us the coffee binary, but also the coffee-script library. We'll be using npm to install some more libraries for our project at the end of this chapter.

6.3 Thinking Asynchronously

One of the most common complaints about JavaScript has always been its lack of support for threading. While popular languages like Java, Ruby, and Python allow several tasks to be carried out simultaneously, JavaScript is strictly linear.

Yet what might seem on its surface to be JavaScript's greatest weakness is now widely seen as a blessing in disguise. Without threads, there are no mutexes, no race conditions, no endless sleep loops. Many of the most common sources of software bugs are banished. What's more, multithreading often adds significant overhead to an application, especially to web servers, which is one reason why Node.js has a reputation as an efficient alternative to frameworks in languages that typically rely on threads for concurrency.

(Of course, without threads there's no way to take advantage of multiple processors. The good news is that there are already projects out there, such as multi-node and cluster, that effectively bind multiple instances of your app to the same server port, giving you the performance advantages of parallel processing without the headaches of sharing data across threads.[3])

Because JavaScript is event-oriented rather than thread-oriented, events only run when all other execution has stopped. Imagine how frustrating it would be if every time your application made a request (say, to the file system or to an HTTP server), it froze up completely until the request was completed! For that reason, nearly every function in the Node.js API uses a callback: you make your request, Node.js quickly passes it along, and your application continues as if nothing happened. When your request is completed (or goes awry), the function you passed to Node.js gets called.

3. http://github.com/kriszyp/multi-node and http://github.com/learnboost/cluster, respectively.

For example, if you wanted to show the contents of the current directory, you would write the following:

```
fs = require 'fs'
fs.readdir '.', (err, files) ->
  console.log files
console.log 'This will happen first.'
```

Here's what happens:

1. We ask Node.js to read the current directory with fs.readdir, passing a callback.
2. Node.js passes the request along to the operating system, then immediately returns.
3. We print 'This will happen first.' to the console.
4. Once our code has run, Node.js checks to see if the operating system has answered our request yet. It has, so it runs our callback, and a list of files in the current directory is printed to the console.

You got that? This is very important to understand. *Your code is never interrupted.* No matter how many RPMs your hard drive has, that callback isn't getting run until after *all* of your code has run. JavaScript code never gets interrupted. Even the seemingly precise setTimeout and setInterval will wait forever if your code gets stuck in an infinite loop.

All of that is as true in the browser as it is in Node, but it's doubly important to understand in Node because your application logic *will* take the form of tangled chains of callbacks. Have no doubt about it. The challenge is to manage them in a way that humans can understand.

Consider how a simple form submission to a web application gets handled:

1. We get the user's information from the database to check that they have permission to make the request.
2. If so, we update the database accordingly.
3. We read a template from the file system, customize it appropriately, and send it to the user.

Then, at the very least, our application skeleton looks like this:

```
formRequestReceived = (req) ->
  checkDatabaseForPermissions req, ->
    updateDatabase req, ->
      renderTemplate req, (tmpl) ->
        sendResponse tmpl
```

And that's without error-handling at each step!

Unfortunately, that matryoshka doll feeling is never quite going to go away. The fact is, in most languages you'd rely on threads so that you could just write something like this:

```
formRequestReceived = (req) ->
  if checkDatabaseForPermissions req
    updateDatabase req
    tmpl = renderTemplate req
    sendResponse tmpl
```

But those languages are pretending to synchronize the asynchronous. Somewhere in each of those database-calling and file-reading functions, there's a `sleep` loop saying, "I hope someone else does something useful while I wait to hear from the database." It's simpler on the surface, but at a price in memory, CPU, and—more often than not—unpleasant surprises.

Note that many NodeJS API functions *do* offer a synchronous version for convenience. For instance, instead of using `fs.readdir` with a callback, you can call `fs.readdirSync` and simply get the list of filenames returned to you. If your application doesn't have any events waiting to fire, then there's no reason not to use these convenient alternatives.

Unfortunately, there's no way to implement a synchronous version of an arbitrary asynchronous function in JavaScript or CoffeeScript. It's only possible using native extensions (typically written in C++), which are beyond the scope of this book.

Scope in Loops

Remember what we learned in Section 2.2, *Scope: Where You See 'Em*, on page ?: *only functions create scope.* Expecting loops to create scope leads otherwise mild-mannered programmers to summon forth horrific bugs when dealing with asynchronous callbacks. In fact, this is probably the most common source of confusion in asynchronous code.

For instance, let's say that we have an application that loads numbers from some (synchronous) source and keeps a running tally of those numbers until the sum meets or exceeds `limit`. Each time a number is loaded, that number—and the sum thus far—needs to be saved. Also, due to overzealous security requirements, each save needs to be encrypted using a key unique to the given number. That key must be fetched *asynchronously* via the `getEncryptionKey` function.

A first attempt might look like this:

```
sum = 0
while sum < limit
  sum += x = nextNum()
  getEncryptionKey (key) ->
    saveEncrypted key, x, sum  # FAIL!
```

The problem here is that by the time the getEncryptionKey callback is called, x and sum have moved on—in fact, the entire loop has been run. So for each x the loop goes through, the values of x and sum after the loop has finished running will be saved (most likely with the wrong encryption key).

The solution is to *capture* the values of x and sum. The easiest way to do that is to use an anonymous function. The do keyword was added to CoffeeScript for precisely this purpose:

```
sum = 0
while sum < limit
  sum += x = nextNum()
  do (x, sum) ->
    getEncryptionKey (key) ->
      saveEncrypted key, x, sum  # Success!
```

If you're familiar with Lisp, this use of do should remind you of the let keyword. do (x, sum) -> ... is shorthand for ((x, sum) -> ...)(x, sum). So now the line saveEncrypted key, x, sum references the copies of x and sum created by the do instead of the x and sum used by the loop.

Note that this form shadows the outer x and sum, making them inaccessible. If you want to have access to the original variables while still capturing their values, then you might write something like this:

```
do ->
  capturedX = x; capturedSum = sum
  ...
```

Now it's time for a little project of our own, extending the jQuery version of 5x5 with a Node-powered back end.