Extracted from:

CoffeeScript

Accelerated JavaScript Development, Second Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

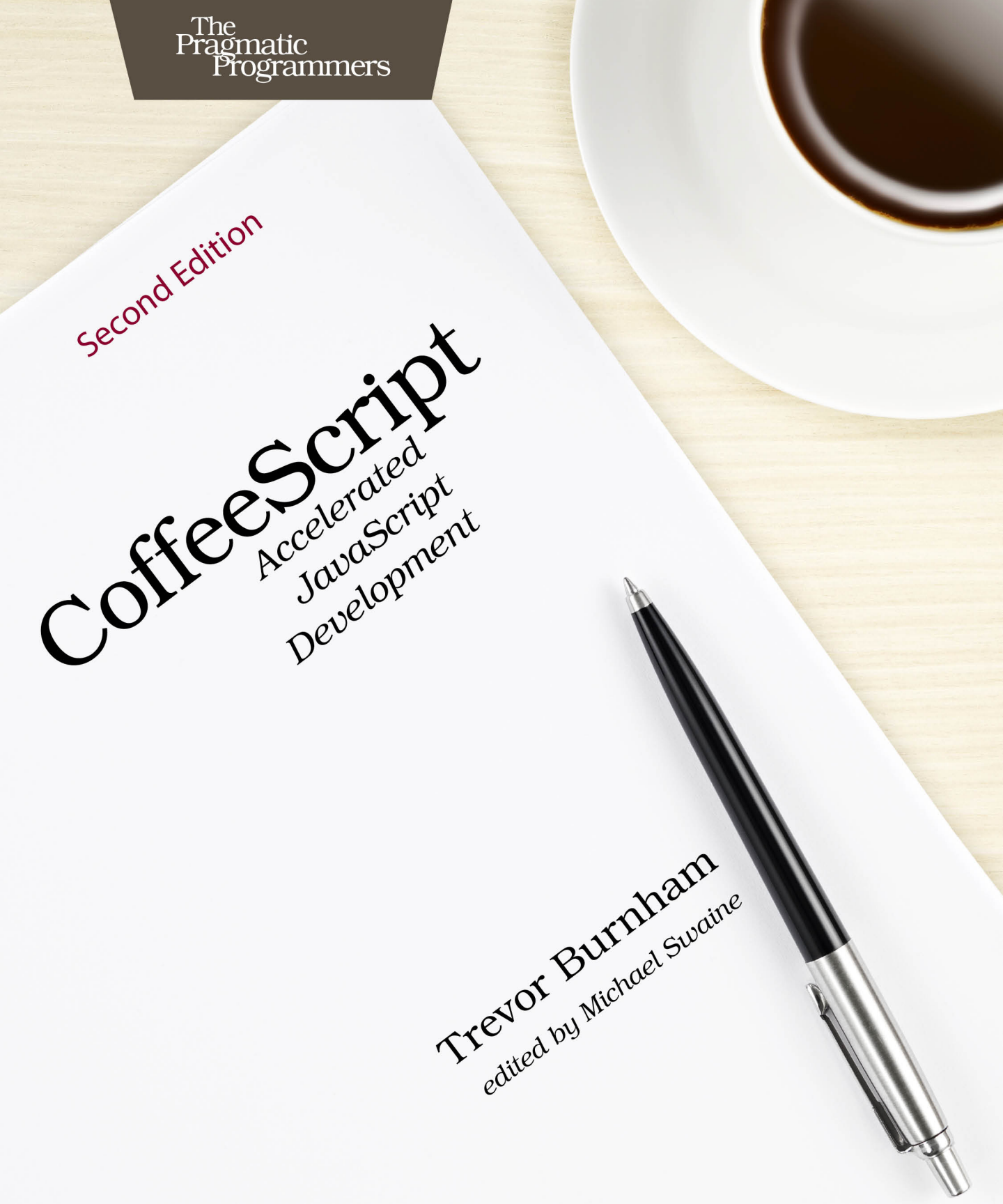
Second Edition

CoffeeScript

Accelerated JavaScript Development

Trevor Burnham

edited by Michael Swaine



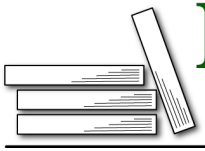
CoffeeScript

Accelerated JavaScript Development, Second Edition

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 *https://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing (indexer)
Cathleen Small (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-94122-226-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2015

CHAPTER 6

Web Servers with Node and Express

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 need to be fluent only in that lingua franca of web apps—or in a dialect such as CoffeeScript.

Now that dream is finally a reality. In this chapter, we'll take a brief tour of Node, the preeminent environment for running JavaScript outside of the browser. Then we'll figure out just what an "evented architecture" is, with its implications for both server performance and our sanity. Finally, we'll create a Node back end for our project from the previous chapter, allowing us to persist our task cards in a proper database.

What Is Node?

Node (also called Node.js in contexts where the term "Node" might be ambiguous) is a JavaScript runtime environment powered by V8, the engine used by Google's Chrome browser. While browsers provide a JavaScript environment with an API that allows code to interact with a web page, the Node API gives JavaScript access to the underlying operating system. That means that scripts running in Node can read and write files, spawn processes, and bind to TCP ports. In fact, any functionality that can be accessed by a C program can be added to Node via addons.[1]

But the most exciting thing about Node isn't the technology, it's the community. Node developers have accomplished amazing feats since Node's debut in early 2009, building a rich ecosystem of open-source packages and using Node in production at high-profile companies such as PayPal, LinkedIn, and

1. http://nodejs.org/api/addons.html

Uber. To see some of the cool mini-projects the community has built, look no further than the Node Knockout, an annual competition to develop the best Node app in forty-eight hours.[2]

Node also has the best package manager on the planet, npm.[3] Although originally started independently, npm was quickly embraced as an official accompaniment to Node, and the two projects are now developed in tandem to complement each other. I like npm so much that I wrote a (short) book about it: *The npm Book*.[4] You'll get a taste of npm as we build this chapter's project.

Writing Node Modules

In the browser, isolating scripts is a pain. Ultimately, every variable a script defines is either scoped in a function or attached to the global object, called window. Expressing dependencies from one script on another is a pain, too. How many times have you written if (window.x) ...? These problems are addressed by the proposed standard for ES6 modules, but it will take years before browser support is widespread.

Thankfully, Node has its own solution to this problem. Every file is its own module with isolated variable scope. There is a global object, simply called global (more semantic than window, wouldn't you agree?), but it's rarely used. Instead, each module attaches data it wants to share to a special object called exports. When a script wants to load a module as a dependency, it uses Node's require function, passing in a partial file path. (More details on that in a moment.) Here's a simple example:

```
# strings.js
exports.hello = 'Hello, Node modules!';

# main.js
var strings = require('./strings');
console.log(strings.hello);

$ node main.js
Hello, Node modules!
```

When Node came across the function call require('./strings'), it blocked execution (a rare thing in Node) while it looked for any of the following in the same directory as main.js, in this order:

2. http://nodeknockout.com/
3. https://www.npmjs.org/
4. https://leanpub.com/npm

1. strings.js (a JavaScript module)
2. strings.json (a JSON data file)
3. strings.node (a native addon)
4. A directory named strings with a file named index.js, index.json, or index.node (or a different file specified as "main" by a package.json in that directory)

In our example, of course, Node didn't have to look very far. It found strings.js, executed it, and returned the object corresponding to exports in strings.js from the require function in main.js.

When require is used with a nonrelative path, it looks for a file with that name in the directory where npm installs packages—or, more accurately, a hierarchy of directories, all named node_modules. It starts with ./node_modules, then ../node_modules, and so on until it gets down to /node_modules. (None of these directories has to exist, of course. Node simply skips the ones that don't.) Finally, it looks in a handful of global locations, although using global modules is discouraged. I mentioned some of the potential problems with global npm modules in *Building the Project with Grunt*, on page ?.

The rules for using require with a relative path combine beautifully with the node_modules hierarchy: when you npm install coffee-script, you get a node_modules/coffee-script directory with a package.json that points to the "main" JavaScript file that defines the API for the CoffeeScript compiler. That package.json also lists the project's own dependencies, which npm installs in node_modules/coffee-script/node_modules. Those, in turn, can have their own dependencies. But you don't have to worry about any of that from the root project: require('coffee-script') just works. And that's why npm is the best package manager on the planet.

You may be asking: "Okay, but how do I load a CoffeeScript file?" That's a reasonable question that's been an area of surprising controversy in Node-land. Early on, Node offered the ability to register extensions to require. After registering the .coffee file extension, you could write require('./script.coffee'), and Node would know to pass that file into the CoffeeScript compiler before executing it. However, the JavaScript purists disliked this feature, and it is now marked as deprecated. No big deal: compiling CoffeeScript before runtime is perfectly sensible. It allows us to deal with compile-time bugs and runtime bugs separately. With the project setup we'll create in the next section, we'll enjoy automatic recompilation, plus source maps for debugging. Who could ask for more?

Compiling a Node Project with Grunt

In the previous chapter, we used Grunt to compile our CoffeeScript files into JavaScript (and source maps) to be served to the browser. For this chapter's project, we'll have two kinds of CoffeeScript files: files that define JavaScript to be sent to the browser, and files that should be run locally in Node. So we'll keep them in two separate directories, /assets and /src. We'll use our Node server to serve our compiled assets to the browser. Any time any of our files change, we'll restart the Node server to ensure that it reflects our changes.

We'll keep our assets that don't have to be compiled (our CSS and HTML files, as well as the external JS we'll install through Bower) in the assets directory as well, and copy them into /lib with everything else we would need to deploy our project (except for the third-party packages in /node_modules). To do that, we'll use another Grunt plugin, grunt-contrib-copy.[5]

To manage our local Node server, we'll use the grunt-nodemon[6] plugin, which wraps around the excellent nodemon.[7] nodemon watches files and restarts our Node server.

One wrinkle: both grunt-contrib-watch (which we'll be using to automatically recompile our project when source files change, as in the previous chapter) and grunt-nodemon keep running indefinitely, and Grunt tasks normally run in a one-at-a-time fashion. To run them at the same time, we need to use yet another plugin, grunt-concurrent.[8]

If you think this sounds quite complicated, well, you're right. Setting up the perfect Grunt configuration for a project can take quite a bit of work, because the possibilities for custom tailoring are endless. But once it's set up, the dividends (compared to compiling manually) are enormous.

As in the last chapter, let's start by setting up our project directory:

```
$ mkdir coffee-tasks
$ cd coffee-tasks
$ npm init
```

And now let's install Grunt and the plugins we need, including the ones from the previous chapter:

5. https://github.com/gruntjs/grunt-contrib-copy
6. https://github.com/ChrisWren/grunt-nodemon
7. https://github.com/remy/nodemon
8. https://github.com/sindresorhus/grunt-concurrent

```
$ npm install -g grunt-cli
$ npm install --save-dev grunt
$ npm install --save-dev grunt-eco
$ npm install --save-dev grunt-concurrent
$ npm install --save-dev grunt-contrib-watch
$ npm install --save-dev grunt-contrib-copy
$ npm install --save-dev grunt-contrib-coffee
$ npm install --save-dev grunt-nodemon
```

Whew! Okay, that's all set. Now here's our Gruntfile:

Node/Gruntfile.coffee

```
module.exports = (grunt) ->
  grunt.loadNpmTasks('grunt-eco')
  grunt.loadNpmTasks('grunt-concurrent')
  grunt.loadNpmTasks('grunt-contrib-watch')
  grunt.loadNpmTasks('grunt-contrib-copy')
  grunt.loadNpmTasks('grunt-contrib-coffee')
  grunt.loadNpmTasks('grunt-nodemon')

  grunt.initConfig
    watch:
      coffeeAssets:
        files: 'assets/coffee/*.coffee'
        tasks: ['coffee:compileAssets']
      coffeeServer:
        files: 'src/*.coffee'
        tasks: ['coffee:compileServer']
      eco:
        files: 'assets/templates/*.eco'
        tasks: ['eco:compile']
      css:
        files: 'assets/css/*.css'
        tasks: ['copy:css']
      html:
        files: 'assets/html/*.html'
        tasks: ['copy:html']

    coffee:
      compileAssets:
        expand: true
        flatten: true
        options:
          sourceMap: true
        cwd: 'assets/coffee/'
        src: ['*.coffee']
        dest: 'lib/public/js/'
        ext: '.js'
      compileServer:
        expand: true
        flatten: true
```

```
    options:
      sourceMap: true
    cwd: 'src/'
    src: ['*.coffee']
    dest: 'lib/'
    ext: '.js'

eco:
  compile:
    options:
      basePath: 'assets'
    src: 'assets/templates/*.eco'
    dest: 'lib/public/js/templates.js'

copy:
  css:
    files: [{
      expand: true
      cwd: 'assets/css/'
      src: ['*.css']
      dest: 'lib/public/css/'
    }]
  html:
    files: [{
      expand: true
      cwd: 'assets/html/'
      src: ['*.html']
      dest: 'lib/public/'
    }]
  bower:
    files: [{
      expand: true
      flatten: true
      cwd: 'bower_components/'
      src: [
        'jquery/dist/jquery.js'
        'underscore/underscore.js'
        'backbone/backbone.js'
      ]
      dest: 'lib/public/js/'
    }]

nodemon:
  dev:
    script: 'lib/server.js'
    watch: 'lib'
    ext: '*'
    options:
      nodeArgs: ['--debug']
```

```
    concurrent:
      dev:
        tasks: ['nodemon', 'watch']
        options:
          logConcurrentOutput: true

  grunt.registerTask('build', ['coffee', 'eco', 'copy'])
  grunt.registerTask('default', ['build', 'concurrent'])
```

It's a lot to take in, but in practice it should feel pretty straightforward: you'll be editing files in /src (for the server) and /assets (for the front end), all of which will go into /lib. The root of /lib is reserved for the files that make up our Node server, while the contents of /lib/public will be directly available to the browser. This new directory structure allows us to simplify our index.html nicely:

Node/assets/html/index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>CoffeeTasks</title>

  <!-- Libraries -->
  <script src="js/jquery.js"></script>
  <script src="js/underscore.js"></script>
  <script src="js/backbone.js"></script>

  <!-- Templates -->
  <script src="js/templates.js"></script>

  <!-- Backbone models/views -->
  <script src="js/card.js"></script>
  <script src="js/column.js"></script>
  <script src="js/board.js"></script>

  <!-- Application core -->
  <script src="js/application.js"></script>

  <!-- Stylesheets -->
  <link rel="stylesheet" href="css/normalize.css">
  <link rel="stylesheet" href="css/style.css">
</head>
<body>
  <!-- All content is rendered client-side -->
</body>
</html>
```

In the next section, we'll get our server up and running.