Building CoffeeScript Frameworks

Building CoffeeScript Frameworks

Build your own Backbone.js or Spine.js

K-2052

This book is for sale at http://leanpub.com/building-coffeescript-frameworks

This version was published on 2014-04-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By K-2052

Markdown To Ebook

Contents

Introduction

The web today demands much of its keepers, we have abstraction after abstraction to keep up with; frameworks, template engines, packaging systems, compile to JS languages, and even entirely new platforms (think TypeScript) that just happen to use the web. It is daunting just too keep up with all the new doodads much less learn how they work.

The three types of CoffeeScript/JS Frameworks

Modern Web frameworks fall into three main categories;

1. *Component/Module based*

Frameworks designed for large applications. They have an opinionated way of structuring your application but when you follow it you get a lot of stuff for free. Rather than simply carry over patterns from other types of app development, they often have invented entirely new ones. Ember.js[1] is the most well known representative of this category.

2. *Platforms*

Platforms typically combine a compile-to-js language with a framework or library. Sometimes they even abstract away HTML or even CSS. Dart[2], Cappuccino[3], ClojureScript[4] and Elm[5] fall into this category.

3. *Design Pattern Based/Minimalist*

These frameworks are built with varying combinations of ModelView*. Some have controllers some use presenters. Some simply opt for placing most of their logic in routes and a controller. What they all have in common is their focus on design patterns and structure. They give you a good starting point and then get out of your way. Backbone.js was the first (and remains the most popular) of this kind but many many have sprung since then, including the one we will build in this book Ryggrad[6]

[1] http://emberjs.com/
[2] https://www.dartlang.org/
[3] http://www.cappuccino-project.org/
[4] https://github.com/clojure/clojurescript
[5] http://elm-lang.org
[6] https://github.com/ryggrad/Ryggrad

How to use this book

Code

You'll find all the code for this book on Github. Each completed chapter has a corresponding branch. You clone the the repo, switch to the chapters branch and then the code should run with no hiccups.

For example;

```
$ git clone https://github.com/k2052/BuildingCoffeeScriptFrameworks.src
$ cd BuildingCoffeeScriptFrameworks.src
```

Then checkout the branch:

```
$ git checkout chapter-2
```

Assembling CoffeeScript Frameworks

Before we go any further we are going to learn how to structure, organize, and test our framework. Only once we have some solid foundations will we get started building.

Let's create a folder for our framework and then cd into it:

```
$ mkdir Ryggrad && cd Ryggrad
```

Setting Up A Dev Environment

Ubuntu

Node.js is available in the standard repo but it's outdated so instead we will install the version from Chris Lea's repo.

Make sure you update apt and install the prerequisites:

```
$ sudo apt-get update
$ sudo apt-get install -y python-software-properties python g++ make
```

Then add the repo:

```
$ sudo add-apt-repository -y ppa:chris-lea/node.js
$ sudo apt-get update
```

And finally install node (npm will be installed with node):

```
$ sudo apt-get install nodejs
```

Now install Grunt (we will install CoffeeScript later with our grunt tasks):

```
$ npm install -g grunt-cli
```

OS X

Install Homebrew:

```
$ ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
```

Update Homebrew:

```
$ brew update
```

Add brew to your path:

```
export PATH="/usr/local/bin:$PATH"
```

And finally install node (npm will be installed with node):

```
$ brew install node
```

Now install Grunt (we will install CoffeeScript later with our grunt tasks):

```
$ npm install -g grunt-cli
```

Introducing Bower

The first tool in our arsenal is Bower[7]. Bower is a simple package manager for web assets. Stuff like; javaScript libraries, css frameworks, icon fonts etc. When we install a package with bower it simply places it in a folder called `bower_components`. It is left up to other systems to integrate these packages. There are integrations for everything from Grunt to Sprockets.

Bower is a npm package, so to install it we simply need to run:

```
$ npm install -g bower
```

Then to install a package using bower we simple need to run:

```
$ bower install [package name]
```

Let's install our first package (*Note*: run this in your project's folder):

```
$ bower install jquery
```

This will install jQuery and place it in bower_components in the root of our project. Since we intend for this to be a package installable via bower, let's go ahead and generate a bower.json file by running:

[7] http://bower.io/

```
$ bower init
```

You'll see something like the following:

```
----------------------------------------
Update available: 1.2.8 (current: 1.2.6)
Run npm update -g bower to update
----------------------------------------

[?] name: Ryggrad
[?] version: 0.0.1
[?] description: "A CoffeeScript FrameWork"
[?] main file:
[?] keywords:
[?] authors:
[?] license: MIT
[?] homepage:
[?] set currently installed components as dependencies? Yes
[?] add commonly ignored files to ignore list? No
[?] would you like to mark this package as private which prevents it from being [\
?] would you like to mark this package as private which prevents it from being ac\
cidentally published to the registry? Yes

{
  name: 'Ryggrad',
  version: '0.0.1',
  description: '"A CoffeeScript FrameWork"',
  license: 'MIT',
  private: true,
  dependencies:{
    jquery: '~2.0.3'
  }
}

[?] Looks good? Yes
```

That does it for bower, for the moment.

Assembling Our framework Using Grunt

There was once a dark time in the JS world. Assembling things was hard back then. We built our JS projects using giant monstrosities of such enormous complexity and size that only wisest wizards

among us could operate them. These monsters [8]: took XMl files to compile our xml files to compile our JSON to a task file that was finally ran by a horde of json-to-xml parsers that then ran a shell script. It was hard, complex and rarely worked smoothly. Building with these tools, was simply put, grunt work.

Times have changed. The days have gotten brighter and the nights shorter. The darkness is held back for a time at least, by a plethora of new and simple tools that now populate our land. One tool in particular, rose up to save us, a tool called grunt.

Grunt is a task runner designed to take the grunt work out of assembling a JS project. It has tasks for everything from minification to unit testing. It will make building even the most complicated of JS projects leisurely work.

Installation

Installing grunt is simple:

```
$ npm install grunt-cli -g
```

Installing *grunt-cli* will make a grunt wrapper command available globally. When you run run $ `grunt` *grunt-cli* will look for the local/project scoped version of Grunt and then run it. How do we install a local version of Grunt to our project? Simple, we use a package.json file

package.json

If you have used npm to manage your JS projects then you are familiar with the package.json [9] format. It holds meta data about your project and any dependencies it has. A basic grunt project looks like this:

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.2",
    "grunt-contrib-jshint": "~0.6.3",
    "grunt-contrib-nodeunit": "~0.2.0",
    "grunt-contrib-uglify": "~0.2.2"
  }
}
```

If you have an existing Grunt project you can use npm to append the grunt dependency to package.json:

[8] Monsters like Ant and Apache build.

[9] An excellent interactive guide on the package.json format is available at http://package.json.nodejitsu.com/

```
$ npm install grunt --save-dev
```

Once you have your package.json you can install grunt by running:

```
$ npm install .
```

We can then run grunt against our project:

```
$ grunt
```

For now, running this will do nothing. How do we make it do something? We need to add a Gruntfile

Gruntfile

Grunt is configured with a Gruntfile.coffee or Gruntfile.js file in the root of a project.

A basic Gruntfile looks like:

```
module.exports = (grunt) ->
  # Project configuration.
  grunt.initConfig
    pkg: grunt.file.readJSON("package.json")
    uglify:
      options:
        banner: "/*! <%= pkg.name %> <%= grunt.template.today(\"yyyy-mm-dd\") %> \
*/\n"

      build:
        src: "src/<%= pkg.name %>.js"
        dest: "build/<%= pkg.name %>.min.js"


  # Load the plugin that provides the "uglify" task.
  grunt.loadNpmTasks "grunt-contrib-uglify"

  # Default task(s).
  grunt.registerTask "default", ["uglify"]
```

Note: No metadata about our project is placed in the Gruntfile. Stuff like the project's name and version is pulled from the *package.json* file using `pkg: grunt.file.readJSON("package.json")`.

What if we want to do something more interesting with our Gruntfile? Like for example, run tests.

The Anatomy of a Grunt Project

Let's say you have a project and it looks like this;

- *src/* Which contains a bunch of CoffeeScript source files
- *spec* Which contains some specs that need to be ran in a browser

The first thing we need to do is define some vars that hold the paths to those files:

```
resources:
  src: [
    'src/*.coffee'
  ]
  spec: [
    'spec/*_spec.coffee'
  ]
```

Paths can both be patterns or specific files. If we wanted to control source order we could do something like: {lang=coffeescript} ~~~~~~~ resources: src: ['src/file1.coffee', 'src/file2.coffee'] ~~~~~~~

To compile our CoffeeScript src files we do:

```
coffee:
  options:
    join: true
  src:
    files:
      '<%= pkg.name %>.debug.js': '<%= resources.src %>'
  test:
    files:
      'test/js/<%= pkg.name %>.js': '<%= resources.src %>'
      'test/spec/spec.js': ['<%= resources.spec %>']
```

This uses the coffee task to output a build file to the root called `pkg.name.debug.js` (pkg.name will be replaced with pkg.name) and our specs to *root/test/spec*. Where does the *coffee* task come from? We load tasks by doing the following:

```
grunt.loadNpmTasks 'grunt-contrib-coffee'
```

This loads the tasks into our GruntFile and makes them available. Let's load some for minification, watching files and running our tests:

```
grunt.loadNpmTasks 'grunt-contrib-uglify'
grunt.loadNpmTasks 'grunt-contrib-watch'
grunt.loadNpmTasks 'grunt-mocha'
```

Then configure them:

```
uglify:
  options:
    compress: false
    banner: '<%= meta.banner %>'
  endpoint:
    files: '<%=meta.file%>.js': '<%= meta.file %>.debug.js'

watch:
  src:
    files: '<%= resources.src %>'
    tasks: ['coffee:src', 'uglify']

mocha:
  test:
    src: [ 'test/test.html' ],
    options:
      # Select a Mocha reporter
      # http://visionmedia.github.com/mocha/#reporters
      reporter: 'Spec',

      # Indicates whether 'mocha.run()' should be executed in
      # 'bridge.js'. If you include `mocha.run()` in your html spec,
      # check if environment is PhantomJS. See example/test/test2.html
      run: true,

      # Override the timeout of the test (default is 5000)
      timeout: 10000
```

Now how do we run them? We have to register them with grunt and then we call them using the correct scope. For example, our coffee:src task would be ran using grunt coffee:src. Here is how we register them:

```
grunt.registerTask 'default', ['coffee:src', 'uglify']
grunt.registerTask 'spec', ['coffee:test', 'mocha:test']
```

Our final Gruntfile looks like this:

```coffeescript
module.exports = (grunt) ->
  grunt.initConfig
    pkg: grunt.file.readJSON "package.json"

    meta:
      file: '<%= pkg.name %>'
      endpoint: 'package',
      banner: '/* <%= pkg.name %> v<%= pkg.version %> - <%= grunt.template.today(\
"yyyy/m/d") %>\n' +
              ' <%= pkg.homepage %>\n' +
              ' Copyright (c) <%= grunt.template.today("yyyy") %> <%= pkg.author.\
name %>' +
              ' - Licensed under <%= pkg.license %> */\n'

    resources:
      src: [
        'src/*.coffee',
      ]
      spec: [
        'spec/*_spec.coffee'
      ]


    coffee:
      options:
        join: true
      src:
        files:
          '<%= pkg.name %>.debug.js': '<%= resources.src %>'
      test:
        files:
          'test/js/<%= pkg.name %>.js': '<%= resources.src %>'
          'test/spec/spec.js': ['<%= resources.spec %>']

    uglify:
      options:
        compress: false
        banner: '<%= meta.banner %>'
      endpoint:
        files: '<%=meta.file%>.js': '<%= meta.file %>.debug.js'

    watch:
```

```
43        src:
44          files: '<%= resources.src %>'
45          tasks: ['coffee:src', 'uglify']
46
47      mocha:
48        test:
49          src: [ 'test/test.html' ],
50          options:
51            # Select a Mocha reporter
52            # http://visionmedia.github.com/mocha/#reporters
53            reporter: 'Spec',
54
55            # Indicates whether 'mocha.run()' should be executed in
56            # 'bridge.js'. If you include `mocha.run()` in your html spec,
57            # check if environment is PhantomJS. See example/test/test2.html
58            run: true,
59
60            # Override the timeout of the test (default is 5000)
61            timeout: 10000
62
63    grunt.loadNpmTasks 'grunt-contrib-coffee'
64    grunt.loadNpmTasks 'grunt-contrib-uglify'
65    grunt.loadNpmTasks 'grunt-contrib-watch'
66    grunt.loadNpmTasks 'grunt-mocha'
67
68    grunt.registerTask 'default', ['coffee:src', 'uglify']
69    grunt.registerTask 'spec', ['coffee:test', 'mocha:test']
```

Grunt Tips

With any technology you are going to experience some pains and fustrations, hopefully some of the tips below can help you troubleshoot common issues.

Tests Hang or Never Run

This is usually due to errors that only the browser will catch. Grunt is bad about telling you when they're happening so you'll need to open test.html in a real browser and view them there. The console should reveal any issues.

Testing Our Framework

There are so many options out there for testing and for good reason; testing is both incredibly important and something you have to do a lot. When you do something a lot you want it to work

just so and when it's important you want it to just work! Everyone has different styles and different things will make it click for them, so it's no wonder that there are so many options. An option for every workflow and style.

I don't want to push my way of testing on you. You'll find your own formula soon enough. I just want share my own little recipe and what works for me. When you see something work, there is a chance that it'll connect and you'll want to work that way.

Mocha, Chai and Sinon are born from what one might call the post-mega-testing framwork era. There was a time when packaging JS was really annoying and distributing it was even more annoying. The first testing frameworks where birthed in this era, they needed to have everything and kitchen sink because every separate bit meant one more hassle. A modular testing framework meant one more shell script, one more configuration file; which meant, two more places for things to go wrong.

In an arena where things just need to work (nothing is worse nor more ironic than your tests failing because of a failing test framework) monolithic testing frameworks were a necessary evil.

Mocha and their ilk are born of the modern era and are thus modular and broken up. Mocha is just a test framework, you can match it up with whatever you like. Chai is just the assertion library, it can be matched up with whatever else you need. Sinon is just mocking and stubs you can match… Well, you get the point.

No matter what testing frameworks you use there are two pieces of magic that make them all work; 1. PhantomJS and 2. Grunt.

PhantomJS is a headless browser used to run tests in a, you guessed it, browser. Grunt is a task runner which is used to automate your testing so you don't have to remember the long and verbose CLI arguments to pass to PhantomJS; also, to make sure you don't forget to run your tests you lazy developer.

Getting Started with Mocha, Chai and Sinon

When Mocha is ran against a test suite it generates a series of failing/passing messages. These are appended to the DOM and can easily be parsed. So, if you are clever you can write a script to run PhantomJS against a test page and then you can extract the results output them to the console. Unsurprisingly, someone has already written scripts to do this. The come in the form of Grunt tasks. We just need a test page to run the tasks against.

We will place our tests in *spec/* and our test pages will be placed in *test/*

The first thing we need to do is add mocha, chai and sinon to our project. Since we are testing in the browser we are going to use bower to manage the packages. This way we can link to the files directly. If we were testing backend code only we could forgo PhantomJS the browser and bower altogether and simply utilize npm and mocha via node.js.

One could utilize node-browserify[10] instead of bower but that is a subject for another chapter.

Install them by first adding them to our bower.json file:

```
"devDependencies": {
  "sinon": "~1.7.0",
  "chai": "~1.6.0",
  "chai-jquery": "~1.1.0",
  "mocha": "~1.9.0",
  "sinon-chai": "~2.4.0"
}
```

Then run:

```
$ bower install
```

Now we need to add some Grunt tasks to compile our tests and dependencies and add them to a page with Mocha.

Let's define an array (in our Gruntfile.coffee) with all our test dependencies:

```
test_dependecies: [
  'bower_components/mocha/mocha.js',
  'bower_components/chai/chai.js',
  'bower_components/sinon-chai/lib/sinon-chai.js',
  'bower_components/sinon/lib/sinon.js'
]
```

And one for all our specs:

```
spec: [
  'spec/*_spec.coffee'
]
```

It's probably best to scope these to something that semantically makes sense:

[10] https://github.com/substack/node-browserify

```yaml
resources:
  test_dependecies: [
    'bower_components/mocha/mocha.js',
    'bower_components/chai/chai.js',
    'bower_components/sinon-chai/lib/sinon-chai.js',
    'bower_components/sinon/lib/sinon.js'
  ]

  spec: [
    'spec/*_spec.coffee'
  ]
```

We need to get these test dependencies joined into one file. We can do this with `grunt-contrib-concat`. To install it and add it to our npm file we run:

```
$ npm install grunt-contrib-concat --save-dev
```

Then we add a task for it:

```yaml
concat:
  options:
    separator: ";"

  test:
    src: '<%= resources.test_dependecies%>'
    dest: 'test/js/lib.js'
```

And load/register it:

```
grunt.loadNpmTasks 'grunt-contrib-concat'
```

Then we can utilize the coffee task to compress and output our tests:

```yaml
coffee:
  test:
    files:
      'test/js/spec.js':   ['<%= resources.spec %>']
```

Let's register a task that compiles all our dependencies to *test/js/lib.js* and all our specs to *test/js/spec.js*:

```
grunt.registerTask 'spec', ['coffee:test', 'concat:test', 'mocha:test']
```

Now we only need to add a test.html page and a task to run it.

Our test page will look like this:

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Tests</title>
    <link rel="stylesheet" href="../bower_components/mocha/mocha.css" type="text/\
css" charset="utf-8" />
</head>
<body>
    <!-- Required for browser reporter -->
    <div id="mocha"></div>

    <!-- mocha and other dependencies-->
    <script src="js/lib.js" type="text/javascript" charset="utf-8"></script>
    <script type="text/javascript" charset="utf-8">
      // This will be overridden by mocha-helper if you run with grunt
      mocha.setup('bdd');
    </script>

    <script src="../../phantomjs/bridge.js" type="text/javascript" charset="utf-8\
"></script>
    <script type="text/javascript" charset="utf-8">
      // Setup chai
      var expect = chai.expect;
      chai.should();
    </script>

    <!-- Spec files -->
    <script src="js/spec.js" type="text/javascript" charset="utf-8"></script>

    <!-- run mocha -->
    <script type="text/javascript" charset="utf-8">
      mocha.run();
    </script>
</body>
</html>
```

Note: You'll also need to add a task to compile the thing you're testing and link to from the html file. I typically link directly to the debug file that is built in the build task. For example;

I'll have a Grunt task like:

```
coffee:
  src:
    files:
      '<%= meta.file %>.debug.js': '<%= resources.src %>'
```

And then link to it in my test.html:

```
<script src="../ryggrad.debug.js" type="text/javascript" charset="utf-8"></script>
```

Once we have this we only need to configure Mocha and load the Mocha task:

```
mocha:
  test:
    src: [ 'test/test.html' ],
    options:
      # Select a Mocha reporter
      # http://visionmedia.github.com/mocha/#reporters
      reporter: 'Spec',

      # Indicates whether 'mocha.run()' should be executed in
      # 'bridge.js'. If you include `mocha.run()` in your html spec,
      # check if environment is PhantomJS. See example/test/test2.html
      run: true,

      # Override the timeout of the test (default is 5000)
      timeout: 10000
```

Then load the task:

```
grunt.loadNpmTasks 'grunt-mocha'
```

Add register a spec task that compiles our specs and then runs the mocha task:

```
grunt.registerTask 'spec', ['coffee:test', 'mocha:test']
```

Then you can run it using:

```
$ grunt spec
```

We wont get anything yet, we need to write some tests first:

Writing Tests

Chai and Mocha are both rather flexible with the style in which you can use them but for our purposes we focus on the should style. The should style is not only the most popular way to use Chai but should be familiar to anyone that has used shoulda style frameworks in say ruby.

We start off by defining what we are we testing:

```
describe "Foo", ->
```

Then we write our tests by passing them into into an `it` function:

```
describe "Foo", ->
  it "should create a new foo instance", ->
    foo = new Foo()
    foo.should.be.an.instanceOf Foo
```

Chai prototypes the JS Object to include the should method, which provides us with a convenient chaining DSL. A few examples:

```
foo.should.be.a('string')
foo.should.equal('bar')
foo.should.have.length(3)
tea.should.have.property('flavors')
  .with.length(3)
```

Once you have a hang of these basic principles constructing a Chai test is simply a matter of consulting the docs. There are plugins that make testing everything from jQuery selectors to Ajax a breeze.

Overview

We should now have a project with a structure like the following:

- *bower_components/*: Contains assets installed via bower.
- *bower.json*: A bower manifest

- *node_modules*: Contains npm managed dependencies. *Note:* you should not git commit this repo
- *GruntFile.coffee*: Con
- *package.json*: A npm manifest
- *ryggrad.debug.js*: The compiled framework, not minified or compressed, suitable for debugging.
- *ryggrad.js*: The compiled framework. This is in the root of the project so that it can be easily used in a bower asset pipeline.
- *spec/*: Contains our specs.
- *src/*: Contains the source CoffeeScript files of the framework
- *test/* A folder with our test web pages meant to be run by a browse (can be headless).

The source for this completed chapter is available on Github[11]

[11] https://github.com/k2052/BuildingCoffeeScriptFrameworks.src/tree/chapter-2