AngularJS Testing Cookbook

AngularJS stepped up to offer a comprehensive solution to frontend development with minimal dependencies and a clear set of objectives.

This book follows the AngularJS philosophy and offers guidance on how to approach testing components that make up the AngularJS framework. At the start of the book, you will explore how to configure your system to run unit and end-to-end tests. Following this, you'll become familiar with fundamental principles on testing AngularJS with Jasmine. Then, you'll understand how spies can enable you to test your code with greater coverage and simplicity throughout your application. The final result is an AngularJS application that is tested with integrity, helping facilitate a cleaner and more reliable codebase.

What this book will do for you...

- Install and configure all required tools and dependencies to begin testing your AngularJS application
- Understand the anatomy of a unit test and integrate core fundamentals to begin testing AngularJS
- Discover navigation and routing using both the AngularJS router and the ui-router library
- Explore how to test controllers
- Analyze test directives and user-based interaction
- Explain the uses of spies and test events
- Test services using mocks and spies
- Animate test cases, both synchronously and asynchronously

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Eliminate volatile code by taking control and understanding how to test AngularJS applications.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Setup and Configuration'
- A synopsis of the book’s content
- More information on AngularJS Testing Cookbook

About the Author

Simon Bailey is a frontend developer based in the UK, specializing in JavaScript development and application architecture. He founded Newtriks Ltd. and has been remotely contracting for the last 10 years for global corporations and venture-backed start-ups. He regularly consults Angular, Backbone, and React, and trains programmers in test-driven development. He is an enthusiastic open source contributor and maintains a blog at http://newtriks.com and is the cofounder and lead developer of the live webcasting platform Sayansho Ltd. He is a husband, father, and lover of the golden age of hip hop.
AngularJS Testing Cookbook

It's 2015 and the world has accelerated at a whirlwind pace towards the open source paradigm. Software is now also analyzed and gauged based on GitHub stars, codebase contributors and consistency in Git commits. When Google open sourced AngularJS, it offered a declarative approach for web application development and was welcomed by an ever-growing community. It's been over 4 years since the first commit to the AngularJS codebase and it has evolved at a breakneck speed to become one of the most popular web application frameworks in the world of open-source.

As the AngularJS team cultivated the framework's documentation, it became abundantly clear that they wanted to facilitate testing of web applications. The documented guidance offers solutions on how developers should test but avoids an overly prescriptive testing process. This allows the developer to decide when and where to test, offering a more approachable development workflow, reducing potential concerns surrounding testing.

This book follows the AngularJS philosophy and offers guidance on how to approach testing components that make up the AngularJS framework. Testing need not be a chore. The recipes contained within this book will provide simple accessible solutions to testing, with a clear demarcation of AngularJS-specific functionality between chapters. The benefit of a cookbook is that it can be used as a point of reference for continuous development of your AngularJS applications. Regardless of whether you're starting a project from scratch or looking to introduce tests into an existing codebase, this book will provide you with a reference point to build the testing foundation.

What This Book Covers

Chapter 1, Setup and Configuration, is crucial in order to get started with the rest of the book and will detail various options for alternative tools that can be used to test AngularJS applications.

Chapter 2, Getting Started with Testing and AngularJS, will form the basis for all tests throughout the book, explaining basics principles required to use Jasmine and AngularJS plus the anatomy of a unit test. Specifics detailed here are crucial to get a test running, for example, dependency injection.

Chapter 3, How to Test Navigation and Routing, will explain how to test routing and page rendering, starting from the view perspective of the application. This chapter also shows how to use Protractor and Page Objects.

Chapter 4, Testing Controllers, explains that the business logic is contained within the controllers, hence it should be addressed early in this book and is the logical early step following routing. Data binding and updating the view based on model data will be tackled ensuring that the UI behaves as expected.
Chapter 5, *Testing User Interaction and Directives*, focuses on testing changes within a directive based on interaction from either UI events or application updates to the model. Directives are one of the cornerstones of AngularJS and can range in complexity. They can provide the foundation to many aspects of the application and therefore require comprehensive tests.

Chapter 6, *Using Spies to Test Events*, shows that an application's state can be updated via events, whether from user interaction in the browser or via API changes. We need to ensure that the correct events are sent and with the correct data. We will also test a third party component wrapped in a directive and mimic its events.

Chapter 7, *Testing Filters*, explains that manipulated output are easily integrated with bound values in the view. They are important to test as they essentially display the data to the user. After looking at a couple of simple examples, this chapter will provide examples on how to run end-to-end tests on filters simulating updates based on events or simulated input.

Chapter 8, *Service and Factory Testing with Mocks and Spies*, will show recipes on how to use this module to test common HTTP requests. Services form a communication channel with outside services requesting data and feeding it into the application. They are singleton objects and relatively easy to test due to their segregation from core application logic. Isolation of components to remove dependencies is facilitated using mocks and spies that are documented in this chapter.

Chapter 9, *A Brief Look at Testing Animations* shows you how to test basic animations and delays synchronously and asynchronously. This is just an introduction into animations as AngularJS did a major overhaul on the animation aspect of their framework and the result included the mock.animate module.
In this chapter, you will learn the following recipes:

- Creating a basic AngularJS application
- Running a simple test using Jasmine
- Installing Protractor
- Running a simple test using Protractor
- Installing Karma
- Running tests using Karma
- Installing Testem
- Running tests using Testem
- Automating test runners using Grunt
- Automating test runners using Gulp

**Introduction**

In this chapter, you will learn various approaches available to configure and test an AngularJS application. These include simple unit tests using Jasmine, integration tests using Protractor, running tests using Karma/Testem, and finally using the Grunt/Gulp build tools to automate tasks.

It's important not to get too overwhelmed by automation of the various libraries available for testing. Each recipe within this book can simply be written and tested using Jasmine and Protractor. This cookbook's overall intention is to make each recipe as accessible as possible using the minimum number of toolsets.

The first half of this chapter is crucial to understand testing throughout this cookbook and we recommend that anyone looking to get started testing AngularJS applications reads it carefully.
Setup and Configuration

There are, however, some great advantages to be gained from using some of the tools and libraries available out there to configure and automate running tests. The second half of this chapter will provide instructions on how to get started with these tools and the information learned from these recipes can be used throughout the book if you choose to.

Creating a basic AngularJS application

This recipe will form the structure for the majority of recipes throughout this cookbook. Additionally, it will provide a code base that can be used and tested in this chapter's recipes, sparing you the task of recreating your own code base throughout these initial configuration steps. The intention is for you, as a reader, to run recipes with minimal configuration and a limited set of libraries. The justification behind this simplicity is to maintain accessibility to each recipe independently. After completing this recipe, you will have a clear idea of the basic setup of an AngularJS application and can build on this foundation as the recipes advance.

Getting ready

Nothing specific is required for this recipe except your usual text editor and browser. Alternatively, you can use an online editor such as http://jsfiddle.net or http://plnkr.co.

How to do it...

1. You should first create a new directory to store the application files named basic_example.
2. Following this, create a new JavaScript file named cookbook.js and within this file, create an application module named cookbook:
   ```javascript
   angular.module('cookbook', [])
   ```
3. Next, add the controller's constructor function to the module using the `.controller()` method. Assign a value to the emcee property on the scope instance:
   ```javascript
   .controller('MainCtrl', ['$scope', function($scope) {
     $scope.emcee = 'Kool G Rap';
   }])
   ```
4. You now need to create an index.html file and add script references to both the angular.js source code (via their content delivery network) and the cookbook.js files either between the `<head>` tags or just before the closing `<body>` tag:
   ```html
   <script type="text/javascript" src="http://code.angularjs.org/1.2.8/angula.min.js"></script>
   <script type="text/javascript" src="cookbook.js"></script>
   ```
5. Following this, bootstrap the application on a document level naming the module `cookbook`:

```html
<html ng-app="cookbook">
```

6. Declare a controller named `MainCtrl` on an HTML `div` tag and using AngularJS's binding syntax, declare the `emcee` expression as follows:

```html
<div ng-controller="MainCtrl">
  <span>Favourite member of the Juice Crew: {{emcee}}</span>
</div>
```

This will be part of the HTML code, which you can see in its entirety, as follows:

```html
<!DOCTYPE html>
<html ng-app="cookbook">
  <head>
    <script type="text/javascript"
      src="http://code.angularjs.org/1.2.28/angular.min.js">
    </script>
    <script type="text/javascript"
      src="cookbook.js"></script>
  </head>
  <body>
    <div ng-controller="MainCtrl">
      <span>Favourite member of the Juice Crew: {{emcee}}</span>
    </div>
  </body>
</html>
```

By this point, you may have created your application. You can see this by opening `index.html` in a browser. Also, Kool G Rap will be displayed as you can see in the following screenshot:

Favourite member of the Juice Crew: Kool G Rap
How it works…

The preceding process demonstrates the basic steps to configure a simple AngularJS application and should be familiar to you. The view component is automatically synchronized with the model data using AngularJS data binding. AngularJS takes data binding one step further by offering two-way data binding, resulting in changes to the model propagated to the view and vice versa.

See also

- For more information, specific to data binding, please consult the official documentation at [http://docs.angularjs.org/guide/databinding](http://docs.angularjs.org/guide/databinding).
- The AngularJS website offers a step-by-step tutorial available at [https://docs.angularjs.org/tutorial](https://docs.angularjs.org/tutorial) that offers a quick overview to getting started.

Running a simple test using Jasmine

This book is focused on testing AngularJS with the behavior-driven development framework Jasmine. Jasmine is currently the framework used for testing the AngularJS code base. With the increased popularity of test runners and generators, the core basics of using Jasmine without these additional tools is sometimes overlooked. In this recipe, you will learn how to set up a basic specification, load an AngularJS module, instantiate a controller, and finally, run a spec testing a value on scope. The series of steps within this recipe are the basis for all your AngularJS tests using Jasmine. The general structure is as follows:

- Injecting dependencies
- Defining a context
- Writing specs

To solidify your base of comprehension regarding Jasmine, I recommend that you read through the documentation at [http://jasmine.github.io/2.0/introduction.html](http://jasmine.github.io/2.0/introduction.html). Within this recipe and throughout the rest of this book, we will follow the same directory structure as the AngularJS team ([https://github.com/angular/angular.js/tree/v1.2.28](https://github.com/angular/angular.js/tree/v1.2.28)). Their approach places application code within a directory named `src` and test-related code in a directory named `test`. You may have your own preferences on naming conventions and directory structure, and can choose to adopt those as you feel comfortable within the recipes throughout this book.
Getting ready

In this recipe, we will build upon the basic project created in this chapter's first recipe. To get ready, perform the following steps to install Jasmine and prepare your project for testing:

1. First, create a new directory called src in your project directory.
2. Next, move the cookbook.js file we wrote earlier in this chapter to the src directory.
4. Finally, copy the unzipped jasmine-2.0.0 directory to a folder named lib within your project directory.

How to do it…

1. First, copy the SpecRunner.html file from the jasmine-2.x.x directory to a new directory named test in the project root folder.
2. Next, update the SpecRunner.html file with the correct paths to the Jasmine files:

   ```html
   <link rel="stylesheet" type="text/css"
   href="../lib/jasmine-2.0.0/jasmine.css">
   <script type="text/javascript" src="../lib/jasmine-2.0.0/jasmine.js"></script>
   <script type="text/javascript" src="../lib/jasmine-2.0.0/jasmine-html.js"></script>
   <script type="text/javascript" src="../lib/jasmine-2.0.0/boot.js"></script>
   ```

3. Now, update the SpecRunner.html file to include AngularJS and AngularJS mock libraries. Order is important here, the mocks.js library must be below the AngularJS script include:

   ```html
   <script type="text/javascript" src="http://code.angularjs.org/1.2.28/angular.js"></script>
   <script type="text/javascript" src="http://code.angularjs.org/1.2.28/angular-mocks.js"></script>
   ```

4. Edit the SpecRunner.html file replacing the source file paths with our main cookbook.js file path:

   ```html
   <script type="text/javascript" src="../src/cookbook.js"></script>
   ```
Setup and Configuration

5. Next, create a new file named `cookbookSpec.js` within the test directory and add a describe block with a beforeEach() function that loads our module:

   ```javascript
   describe('MainCtrl', function () {
       beforeEach(module('cookbook'));
   });
   ```

6. Still within the describe block, create a basic test that injects the $controller service and $rootscope. We can then create a new $scope object and instantiate a new instance of our controller providing the $scope object:

   ```javascript
   it('should assign the correct rapper to scope',
       inject(function ($controller, $rootScope) {
           var $scope = $rootScope.$new();
           $controller('MainCtrl', {
               $scope: $scope
           });
       }));
   ```

7. Create an expectation asserting that the value on scope is as expected:

   ```javascript
   expect($scope.emcee).toEqual('Kool G Rap');
   ```

8. Update `SpecRunner.html` to include `cookbookSpec.js`:

   ```html
   <script type="text/javascript" src="cookbookSpec.js"></script>
   ```

9. Finally, open `SpecRunner.html` in a browser and you should see your first test passing:
How it works...

In step 1, we use the SpecRunner.html file that Jasmine provides as a basis to build on for our test. Once we move the file into our directory structure, we need to ensure that all the paths correlate to the Jasmine files correctly; this we do in step 2.

The idea behind unit testing is to test the functionality of a piece of code in isolation. The AngularJS team helps to ease this process by offering replicas of objects called mocks that make unit testing easier by decoupling the components. The angular-mocks.js file loads the ngMock module (https://docs.angularjs.org/api/ngMock) so we can inject and mock AngularJS services. In step 3, we ensure that the AngularJS library is loaded in addition to the angular-mocks.js file.

In step 4, we ensure that our main cookbook.js application code is loaded and ready to test. Then, in step 5, we define a context based on our MainCtrl controller and a beforeEach() function. The beforeEach() function is passed a string alias cookbook as an argument to the mock module function that configures information ready for the Angular injector.

In step 6, we define our test starting with the it() function that requires a string detailing our test intention, plus a second argument that makes use of the mock module's inject function. The mock inject function basically creates an injectable wrapper around this second argument creating a new instance of $injector for each test. The $injector instance is used to resolve all references for this recipe; this includes the $controller service function, which instantiates controllers and $rootScope. We use the injected $rootScope service to create a new child scope using the $new() method (https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$new). This new child scope object is then passed to our newly instantiated controller that's constructed using the $controller service. Please refer to the introduction part of Chapter 2, Getting Started with Testing and AngularJS for further explanation about the Jasmine API.

Our expectation in step 7 is that when our MainCtrl controller is created, the scope's emcee property is expected to match the value of Kool G Rap (https://github.com/pivotal/jasmine/wiki/Matchers). This is asserted in step 7 by comparing the $scope.emcee property to Kool G Rap.

For a more in-depth overview of each of the steps involved in writing a test, including expectations and matchers, please read Chapter 2, Getting Started with Testing and AngularJS.
**Setup and Configuration**

**There's more...**

An alternative approach to injecting the controller reference within the spec would be to move this logic and assign it to a variable reference outside of the spec. We can do this in another beforeEach() function with the refactored code looking like this:

```javascript
describe('MainCtrl', function () {
  var mainCtrl, $scope;
  beforeEach(module('cookbook')));
  beforeEach(inject(function ($controller, $rootScope) {
    $scope = $rootScope.$new();
    $controller('MainCtrl', {
      $scope: $scope
    });
  }));
  it('should assign the correct rapper to scope', function () {
    expect($scope.emcee).toEqual('Kool G Rap');
  });
});
```

**See also**

- The Creating a basic AngularJS application recipe
- Chapter 2, Getting Started with Testing and AngularJS

**Installing Protractor**

During the first half of 2014, there was an abundance of renewed debate regarding Test Driven Development (TDD) taking place on the Web. David Heinemeier Hansson (creator of Rubber on Rails) challenged the fundamentals behind TDD and suggested placing more emphasis on system/browser tests. There are many arguments, for and against, as you can imagine, but what we are interested in is how we can implement the type of testing being advocated in an AngularJS application.

Luckily, to test application logic with browser tests (as opposed to the more granular unit tests), the AngularJS team developed a solution named Protractor (https://github.com/angular/protractor). This is a Node.js (http://nodejs.org/) program to run end-to-end tests against an actual browser using WebDriver (which you can access at https://code.google.com/p/selenium/wiki/GettingStarted). Webdriver is used to control browsers and simulate user actions. Protractor is quite an active project and therefore subject to change; this recipe is based on the current version at the time of writing this, which is 0.23.1. This recipe will focus on installing and configuring Protractor and Webdriver.
Getting ready

You will need to have Node.js (http://nodejs.org/) installed on your machine. For the purpose of this recipe, we use the Chrome browser that you can download at http://www.google.co.uk/chrome/. Alternative browsers can be configured and I recommend that you read the Protractor browser setup guide at https://github.com/angular/protractor/blob/master/docs/browser-setup.md.

How to do it...

1. You will first need to install Protractor using the npm install -g protractor command.

2. You will then need to install the necessary WebDrivers using the webdriver-manager update command.

3. Following this, start the Selenium server using the webdriver-manager start command.

4. Protractor will then accept a configuration file. Create a new file named protractor.conf.js with the following content:

```javascript
exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  jasmineNodeOpts: {
    showColors: true, // use colors in the command line
    report
    defaultTimeoutInterval: 30000
  }
};
```

There's more...

Protractor can start an instance of the Selenium standalone server if you provide the path to the Selenium server standalone JAR as follows. This will slow down the overall time it takes to run your tests due to Protractor having to start the server as opposed to an instance already running:

```javascript
seleniumServerJar: './node_modules/protractor/selenium/selenium-server-standalone-2.41.0.jar'
```

The path may be vary based on OS or if you installed Protractor globally.
Running a simple test using Protractor

In the *Installing Protractor* recipe in this chapter, you learned how to configure Protractor and WebDriver. This recipe will introduce you to running a basic end-to-end test with Protractor. You will learn how to add test files to the Protractor configuration, add a basic end-to-end test, find DOM elements based on AngularJS attributes, and simulate user interaction. The latter has the potential to save a considerable amount of time testing your application and allows you to test expected user interaction. This sort of testing can prove indispensable and Protractor provides a simplified API to simulate user actions, as you will see within this recipe. Please refer to [https://github.com/angular/protractor/blob/master/docs/api.md](https://github.com/angular/protractor/blob/master/docs/api.md) for a comprehensive list and explanation of the API methods available to you using Protractor.

Getting ready

Protractor will connect to a browser to run its tests. Preferably, we would want to run our application against an HTTP server and the quickest way to do this is using Python's (Version 2.4 or greater) *SimpleHTTPServer* by visiting [https://docs.python.org/2/library/simplehttpserver.html](https://docs.python.org/2/library/simplehttpserver.html).

This is a cross-platform solution that requires Python to be installed on your machine. OS X, for example, will have this preinstalled while Windows users can install ActivePython or another Python installation route of your choice.

How to do it...

1. First, create a new directory and add a new file named `index.html` with the following content:

   ```html
   <!DOCTYPE html>
   <html ng-app="cookbook">
   <head>
     <script src="https://code.angularjs.org/1.2.16/angular.js"></script>
     <script src="cookbook.js"></script>
   </head>
   ```
2. You then need to copy the *cookbook.js* file from the *Creating a basic AngularJS application* project recipe into the directory.

3. Following this, add a new file named *protractor.conf.js* with the following content:

```javascript
exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['cookbookSpec.js'],
  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 30000
  }
};
```

4. You should then create a new file named *cookbookSpec.js* with the following content:

```javascript
describe('favourite rapper', function () {
  it('should bind to input', function () {
    browser.get('');
    var emceeInput = element(by.model('emcee'));
    var emceeOutput = element(by.binding('emcee'));
    expect(emceeOutput.getText()).toBe('Kool G Rap');
    emceeInput.clear();
    emceeInput.sendKeys('Aesop Rock');
    expect(emceeOutput.getText()).toBe('Aesop Rock');
  });
});
```

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Setup and Configuration

5. Now, start the local HTTP server by running the `python -m SimpleHTTPServer` command (this command serves HTTP service on IP address 0.0.0.0 using port 8000 by default; you need to use `python -m http.server` if you are a windows user.) and the Selenium server with the `webdriver-manager start` command.

6. Once the server has started, in the command-line console of your choice, run the `protractor protractor.conf.js` command. As a result, you should see the following output:

```

Finished in 3.204 seconds
1 test, 2 assertions, 0 failures
```

How it works...

In step 1, we create an HTML file, which is a basic AngularJS binding example. Step 3 shows the addition of a specs option that expects an array of relative filenames or a glob pattern. Step 4 executes the following sequence of events:

- The `should bind to input` spec begins using the Protractor global browser variable that is a wrapper around a WebDriver instance. The `get()` function expects a URL or path to a running web page for the AngularJS application. The web page can be run over a local web server, which we start in step 5.
- Search for elements on the page using the `element()` method and its additional AngularJS-specific strategies, for example binding names (ng-bind or `{{}}`) and elements by input using `ng-model`. The element method does not return a DOM element, but an ElementFinder. Calls to element can be chained to find child elements within a parent.
- Define an assertion based on the input field value. The initial value will be the default value assigned to the `$scope.emcee` property in the controller.
- Clear the input field value using the `clear()` method on the ElementFinder.
- Populate the input field using the `sendKeys()` method on the ElementFinder.
- Reassert, based on the updated input field, that the output binding has worked as expected.

The final steps run the local HTTP server and the Protractor tests displaying the results.
There's more...

A `baseUrl` option is available within the Protractor configuration, for example `baseUrl (http://0.0.0.0:8000)`. This enables the use of relative paths using the `get()` method that's available on Protractor's browser function, for example `browser.get()`. Please read the Protractor API documentation for more information on available functions (http://angular.github.io/protractor/#/api?view=Protractor).

There are some additional Chrome-specific options that can prove quite useful within your development toolkit. These can be added using a `chromeOptions` object; using this object will display the frame rate within your browser. Monitoring frame rate for consistency and a value between 30 to 60 frames per second (fps) can ensure that your application doesn't appear jerky and animations are smoothly run, for example:

```javascript
  capabilities: {
    'browserName': 'chrome',
    'chromeOptions': {
      'args': ['show-fps-counter=true']
    }
  }
```

To target multiple spec files within your Protractor configuration, you can use a regular expression as opposed to an array of file names:

```javascript
  specs: ['test/*_spec.js']
```

See also

- The Installing Protractor recipe in this chapter

Installing Karma

This recipe will introduce you to the Karma test runner, which you can get at http://karma-runner.github.io/, for test-driven development and continuous integration. You will run through the basic process to install Karma version, which at time of writing this is 0.12.16.

Karma is another fantastic tool from the AngularJS team allowing you to execute JavaScript code in browsers. Earlier in this chapter, (in the Running a simple test using Protractor recipe in this chapter), you learned about Protractor, a tool for integration testing; what Karma offers is configuration for unit testing. It facilitates testing against multiple browsers, whether locally or on a continuous integration server. It can also be configured to run unit tests when application code changes via a watch feature. There are many plugins available to extend Karma and in this recipe we will use two of them. Firstly, we will use Jasmine throughout the cookbook so we will need the plugin from https://github.com/karma-runner/karma-jasmine. Secondly, we will run the tests in the Chrome browser using https://github.com/karma-runner/karma-chrome-launcher. Once you have completed this recipe, you'll have Karma installed with the Chrome and Jasmine plugins and ready to run some tests.
Getting ready

All you need is Node.js (http://nodejs.org/; ideally version 0.10.* or above) installed on your machine.

How to do it...

1. First, install Karma and the required plugins using the following command:
   
   ```
   npm install karma karma-jasmine karma-chrome-launcher --save-dev
   ```

2. Install the karma-cli to use Karma in your command line:
   
   ```
   npm install -g karma-cli.
   ```

3. Next, we use an option that some Node.js libraries provide to run initialization steps, which is typically called `init`. It guides us through the steps required to configure Karma on our command line:
   
   ```
   karma init
   ```

4. Now, at the prompt for testing framework, press `Enter` to accept the default option of `jasmine`.

5. Next, at the prompt to use Require.js, press `Enter` to accept the default option of `no`.

6. When you are prompted to automatically capture any browsers, press `Enter` twice to accept the default option of `Chrome`.

7. Next, at the prompt to define source file location, press `Enter` to accept the default option of an empty string.

8. At the prompt to define file patterns to exclude, press `Enter` to accept the default option of an empty string.

9. Finally, you will be prompted for Karma to watch for file changes so press `Enter` to accept the default option of `yes`. 
The following screenshot shows you the steps discussed earlier:

```
simons-mbp:cookbook newtriks$ karma init  
Which testing framework do you want to use ?  
Press tab to list possible options. Enter to move to the next question .  
> jasmine  

Do you want to use Require . js ?  
This will add Require . js plugin .  
Press tab to list possible options. Enter to move to the next question .  
> no  

Do you want to capture any browsers automatically ?  
Press tab to list possible options. Enter empty string to move to the next question .  
> Chrome  

What is the location of your source and test files ?  
You can use glob patterns, eg. "js/*.js" or "test/**/spec.js".  
Enter empty string to move to the next question .  
>  

Should any of the files included by the previous patterns be excluded ?  
You can use glob patterns, eg. "**/swp".  
Enter empty string to move to the next question .  
>  

Do you want Karma to watch all the files and run the tests on change ?  
Press tab to list possible options.  
> yes  

Config file generated at "/Users/newtriks/src/cookbook/karma.conf.js".  
simons-mbp:cookbook newtriks$ _
```

**There's more...**

If you prefer not to install karma-cli globally, allowing for different versions and different projects, you can remove the `-g` syntax. This will install the package to a `node_modules` folder within the root directory of the project:

```
npm install karma-cli
```

You can then run the commands from `node_modules`, as shown here:

```
./node_modules/.bin/karma init
```

**See also**

- The *Running a simple test using Karma* recipe
Running tests using Karma

To save time and effort when manually running your tests, you can use a test runner. The recipe, Installing Karma, gets you ready to run tests with Karma. This recipe will introduce you to automatically run a Jasmine test with Karma (which you can visit at http://karma-runner.github.io/). You will learn how to set up a basic configuration file and automatically run your tests with Karma.

Getting ready

You can either implement this as an initial step to an existing project, or build upon the basic project created in the first recipe. You can do this as follows:

1. Karma will need access to the angular.js and angular-mocks.js files, which can be downloaded from https://code.angularjs.org/1.2.28/. Ensure these are included in a lib/angular folder in your project root directory.
2. Copy the cookbook.js file from the recipe Creating a basic AngularJS application, into the src directory.
3. Finally, copy the cookbookSpec.js file from the Running a simple test using Jasmine recipe in this chapter, into a test directory.

How to do it...

Firstly, you need to create a Karma configuration file named karma.conf.js with the following code:

```javascript
module.exports = function(config) {
  config.set({
    frameworks: ['jasmine'],
    files: [
      'lib/angular/angular.js',
      'lib/angular/angular-mocks.js',
      'src/cookbook.js',
      'test/cookbookSpec.js'
    ],
    autoWatch: true,
    browsers: ['Chrome']
  });
};
```
Once this has been created, you can run the following command:

```bash
code karma start
```

As a result of this, you should see that Karma launches the Chrome browser and produces the following in the console window:

```
INFO [karma]: Karma v0.12.16 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 35.0.1916 (Mac OS X 10.9.3)]: Connected on socket e2ApyGrjWV7p6A_4u8-3 with id 43420351
Chrome 35.0.1916 (Mac OS X 10.9.3): Executed 1 of 1 SUCCESS (0.025 secs / 0.023 secs)
```

### How it works...

Karma requires a configuration file for it to run our AngularJS application tests. Let’s step through the configuration options:

- **frameworks**: Framework adaptors that must be installed via the `karma init` process or manually, for example `npm install karma-jasmine --save-dev`.
- **files**: These are the file patterns specifying applications and test files.
- **autoWatch**: This option watches for changes to applications/test files and re-run tests.
- **browsers**: These are the launchers that must be installed via the `karma init` process or manually, for example `npm install karma-chrome-launcher --save-dev`.

The `angular.js` file is a dependency for `angular-mocks.js`, therefore `angular.js` must be declared before `angular-mocks.js`.

For a more comprehensive list of configuration options, please refer to Karma’s configuration file documents at [http://karma-runner.github.io/0.12/config/configuration-file.html](http://karma-runner.github.io/0.12/config/configuration-file.html).

### There's more...

Use a glob pattern for files, as opposed to declaring each file explicitly, by inserting the following code:

```javascript
files: [
  "lib/angular/angular.js",
  "lib/angular/angular-mocks.js",
  "src/**/*.*",
  "test/**/*.*",
]
```
Installing Testem

This recipe will introduce you to the Testem test runner (https://github.com/airportyh/testem) for test-driven-development and continuous integration. Testem is an alternative to Karma and although similar, it is favoured by some developers in the community including the Lineman (https://github.com/linemanjs/lineman) team. Karma and Testem have similar feature sets but Karma is developed with regular updates by the AngularJS team. As with Karma, Testem is framework-agnostic and configurable to run tests within a multitude of browsers, whether locally or on a Continuous Integration server. This recipe is based on the current version; at time of writing this it is 0.6.15 and will guide you on installing Testem.

Getting ready

You will need to have Node.js (http://nodejs.org/, version 0.10.* or greater) installed on your machine.

How to do it...

All you need to do is install Testem using the following command:

```
npm install testem --save-dev
```

You can then run the following command:

```
testem
```

You will see that Testem produces the following:

```
TEST'EM 'SCRIPTS!
Open the URL below in a browser to connect.
http://localhost:7357/

Chrome 35.0
0/0 ✓

No tests were run :(  
```

See also

- The Installing Karma recipe in this chapter

- The Running a simple test using Testem recipe
Running tests using Testem

To save time and effort manually running your tests, you can use a test runner. In the Installing Testem recipe you learned how to install and configure the runner. This recipe will introduce you to automatically running a Jasmine test with Testem within the Chrome browser. You will learn how to set up a basic configuration file and automatically run your tests.

Getting ready

To begin using Testem in this recipe you can either perform the following steps within an existing AngularJS project or build upon the basic project created in the first recipe:

1. Firstly, the angular.js and angular-mocks.js files need to be included. They can be downloaded from https://code.angularjs.org/1.2.28/. Ensure these are included in a lib/angular folder in your project root folder.

2. Copy the cookbook.js file from the Creating a basic AngularJS application recipe in this chapter, into the src directory.

3. Finally, copy the cookbookSpec.js file, from the Running a simple test using Jasmine recipe, into the test directory.

How to do it...

To start with, create a Testem configuration file named testem.json with the following JSON:

```json
{
    "framework": "jasmine",
    "src_files": [
        "lib/angular/angular.js",
        "lib/angular/angular-mocks.js",
        "src/cookbook.js",
        "test/cookbookSpec.js"
    ],
    "launch_in_dev": ["Chrome"]
}
```
Next, you need to run the `testem` command. As a result, you'll see that Testem launches the Chrome browser and our single test passing as shown in the following screenshot.

How it works...

A Testem configuration file allows finer control over what is included by the test runner. Let's step through the configuration options:

- **framework**: This is the test framework of choice. Testem already includes our preference of Jasmine.

- **src_files**: These are the file patterns specifying application and test files.

- **launch_in_dev**: This is the list of launchers to use for development runs.

The `angular.js` file is a dependency for `angular-mocks.js`, therefore `angular.js` must be declared before `angular-mocks.js`. Testem will automatically detect our files and changes made to the application or tests and rerun the tests.

For a more comprehensive list of configuration options, please refer to Testem's configuration file documents by visiting [https://github.com/airportyh/testem/blob/master/docs/config_file.md](https://github.com/airportyh/testem/blob/master/docs/config_file.md).
There's more...

Use a glob pattern for files, as opposed to declaring each file explicitly, by inserting this:

```
"src_files": [  
  "lib/angular/angular.js",
  "lib/angular/angular-mocks.js",
  "src/**/*",
  "test/**/*
]
```

See also

- The Installing Testem recipe

Automating test runners using Grunt

The amount of tasks within an AngularJS-based project can rapidly increase, such as running an HTTP server, unit test runner, end-to-end test runner, or automating testing. These repetitive tasks can be automated using task runners such as Grunt (which you can download at http://gruntjs.com/), and Gulp (which can be downloaded at http://gulpjs.com/).

Grunt has been at the forefront of task runners for quite some time now; the community is vast with a great selection of available plugins. Grunt advises a global installation of its Command Line Interface (CLI) to access the grunt command anywhere on your system. Project and task configuration is contained within a Gruntfile, including the loading of plugins to extend the range of tasks available. A great starting point is on the Grunt website itself at http://gruntjs.com/getting-started. Armed with this basic knowledge, this recipe will focus on how to set up and configure Grunt to run the following tasks:

- HTTP web server
- Karma test runner
- WebDriver
- Protractor

Getting ready

You can either perform the following steps within an existing AngularJS project, or build upon the basic project created in the first recipe:

1. Firstly, the angular.js and angular-mocks.js files need to be included. They can be downloaded from https://code.angularjs.org/1.2.28/. Ensure these are included in a lib/angular folder in your project root folder.
Setup and Configuration

2. Next, you then need to copy the cookbook.js file from the Creating a basic AngularJS application recipe into the src directory.

3. Next, copy the cookbookSpec.js file from the Running a simple test using Jasmine recipe into the test/unit directory.

4. Next, copy the cookbookSpec.js file from the Running a simple test using Protractor recipe into the test/e2e directory.

5. Finally, you need to ensure that you have installed Protractor globally, that is using the -g tag (please refer to the Installing Protractor recipe in this chapter to see how to do this). Also ensure that the Selenium standalone server has been downloaded and installed.

Once Protractor and the Selenium server have both been installed, you are now ready to begin.

How to do it...

1. First, let’s install the Grunt CLI globally using this command:
   ```plaintext
npm install -g grunt-cli
   ```

2. Next, run the following command to interactively create a package.json file and either add your specific configuration or accept the defaults when prompted:
   ```plaintext
   npm init
   ```

3. Following this, install the rest of the dependencies required for this recipe locally:
   ```plaintext
   npm install grunt grunt-contrib-connect grunt-karma grunt-protractor-runner grunt-protractor-webdriver karma karma-chrome-launcher karma-jasmine load-grunt-tasks 0.4.0—save-dev
   ```

4. Now, install the WebDrivers using Protractor by running this command:
   ```plaintext
   /node_modules/protractor/bin/webdriver-manager update
   ```

5. A Karma configuration file named karma.conf.js can then be created with the following code:
   ```javascript
   module.exports = function(config) {
     config.set({
       frameworks: ['jasmine'],
       files: [
         "lib/angular/angular.js",
         "lib/angular/angular-mocks.js",
         "src/cookbook.js",
         "test/unit/cookbookSpec.js"
       ],
       autoWatch: true,
       browsers: ['Chrome']
     });
   };
   ```
6. You should then create a Protractor configuration file named `protractor.conf.js`, which contains the following code:

```javascript
exports.config = {
    seleniumAddress: 'http://localhost:4444/wd/hub',
    jasmineNodeOpts: {
        showColors: true,
        defaultTimeoutInterval: 30000
    }
};
```

7. Next, another file named `Gruntfile.js` needs to be created, which will be in the root of the project directory with the following code:

```javascript
module.exports = function (grunt) {
    require('load-grunt-tasks')(grunt);
    grunt.initConfig({
        connect: {
            server: {
                options: {
                    port: 8000,
                    base: 'src'
                }
            }
        },
        karma: {
            unit: {
                configFile: 'karma.conf.js'
            }
        },
        protractor: {
            e2e: {
                options: {
                    args: {
                        specs: ['test/e2e/cookbookSpec.js']
                    },
                    configFile: 'protractor.conf.js',
                    keepAlive: true
                }
            },
            protractor_webdriver: {
                start: {
                    options: {
                        path: './node_modules/protractor/bin/',
                        command: 'webdriver-manager start',
                    }
                }
            }
        }
    });
};
```
Setup and Configuration

```javascript

};

}));

grunt.registerTask('test', [
  'karma:unit'
]);

grunt.registerTask('e2e', [
  'connect:server',
  'protractor_webdriver:start',
  'protractor:e2e'
]);

});

8. Finally, you can run the unit tests of Karma using `grunt test`. You can then run the end-to-end tests of Protractor using `grunt e2e`.

How it works…

Step 1 simply runs the global installation for `grunt-cli` (which you can download at https://github.com/gruntjs/grunt-cli). This enables us to access the Grunt command anywhere on our system. Step 2 and step 3 install all of the dependencies required to successfully run all the Grunt tasks.

After this, step 4 and step 5 show how to set up the configuration files for both Karma and Protractor defining our unit and end-to-end specs.

Using the `grunt-karma` command, you can eliminate the need for a `karma.conf` configuration file and define the configuration in the Gruntfile instead. Visit https://github.com/karma-runner/grunt-karma#heres-an-example-that-puts-the-config-in-the-gruntfile to discover more.

In step 6, we see the Gruntfile.js file where we define all the necessary tasks to automate the unit and end-to-end tests. Both work by:

- **Unit tests**: Running the unit tests with Karma is relatively straightforward and simply requires the `grunt-karma` plugin (https://github.com/karma-runner/grunt-karma). We then add a karma target and provide the config file created in step 4. Finally, we register a task with Grunt named `test`, which calls `karma:unit` and runs the unit tests.
The grunt-karma module supports configuring options for multiple targets, for example `dev` (development) and `continuous` (Continuous Integration). To use the continuous integration mode, use the `singleRun` option. In development, we can use the `autoWatch` option.

For Continuous Integration option, use the following code:
```javascript
{
  singleRun: true,
  browsers: ['PhantomJS']
},
```

For Development option, use the following code:
```javascript
{
  autoWatch: true
}
```

- **End-to-end tests**: Running the end-to-end tests with Protractor is slightly more complex due to the additional requirement of running HTTP and Selenium servers. We can run Protractor using the grunt-protractor-runner plugin (https://github.com/teerapap/grunt-protractor-runner) and adding a protractor target. Within the protractor target, we create an `e2e` target and a single option configuration file pointing to our `protractor.conf.js` file that we created in step 5. The grunt-protractor webdriver plugin (which you can download at https://github.com/seckardt/grunt-protractor-webdriver) enables us to start the WebDriver that we define as a sole command in the `protractor_webdriver` `e2e` target. To start a local HTTP server, we use the grunt-contrib-connect plugin (https://github.com/gruntjs/grunt-contrib-connect) with three options, mainly port, hostname, and the base path, which is the `src` directory containing our AngularJS application. Finally, we register a task with Grunt named `e2e` that connects to our web server, starts the Selenium standalone server, and runs Protractor, phew!

Finally, to run all of these tasks, we call either the `test` or `e2e` task.

**See also**

- The Automating test runners using Gulp recipe
- Brunch (http://brunch.io/)
Automating test runners using Gulp

An additional task runner you can use, other than Grunt, is Gulp (you can download it at http://gulpjs.com/). Gulp is a stream-based build system with a simple yet powerful API focusing on code over configuration. Gulp builds use the Node.js streams that do not need to write temporary files or folders to disk, which results in faster builds. I advise you to read this excellent article on streams at https://github.com/substack/stream-handbook. The Gulp API is small and extremely simple:

- `gulp.src(globs[, options])`: This returns a readable stream
- `gulp.dest(path)`: This can receive an input stream and output to a destination, for example, writing files
- `gulp.task(name[, deps], fn)`: This defines tasks using orchestrator (https://github.com/orchestrator/orchestrator)
- `gulp.watch(glob [, opts], tasks) or gulp.watch(glob [, opts, cb])`: This enables watching for file changes and then running a task or tasks when changes are detected

Please visit https://github.com/gulpjs/gulp/blob/master/docs/API.md for an explanation of the API.

Gulp has its own configuration file called a Gulpfile, focused on code over configuration. It offers a coherent read with a clear visualization of flow. Gulp discourages the use of writing plugins for tasks that don't require a dedicated plugin focusing on providing streams and basic tasks. This recipe will focus on how to set up and configure Gulp to run the following tasks:

- HTTP web server
- Karma test runner
- WebDriver
- Protractor

Getting ready

You can either implement this as an initial step to an existing project or build upon the basic project created in the first recipe. If integrating into an existing project, ensure that you have specified the correct source and test files with their corresponding paths. Once configured, Gulp will run your tests as expected. If using the cookbook example project, then follow these steps:

1. Firstly, angular.js and angular-mocks.js files need to be included. They can be downloaded from https://code.angularjs.org/1.2.28/. Ensure these are included in a lib/angular folder in your project root folder.

2. You then need to copy the cookbook.js file created in the Creating a basic AngularJS application recipe into the src directory.
3. Finally, copy the `cookbookSpec.js` file from the Running a simple test using Jasmine recipe in this chapter into a test/unit directory.

4. To prepare yourself completely for this recipe, you can then copy the `cookbookSpec.js` file from the Running a simple test using Protractor recipe into a test/e2e directory.

**How to do it...**

1. First, let's install Gulp and Protractor globally using this command:

   ```shell
   npm install -g gulp protractor
   ```

2. Next, run the following command to interactively create a `package.json` file and either add your specific configuration or accept the defaults when prompted:

   ```shell
   npm init
   ```

3. Following this, install the rest of the dependencies required for this recipe locally:

   ```shell
   npm install connect gulp-load-plugins gulp-protractor karma karma-chrome-launcher karma-jasmine --save-dev
   ```

4. You then need to create a Karma configuration file named `karma.conf.js` with the following code:

   ```javascript
   module.exports = function () {
     return {
       frameworks: ['jasmine'],
       files: [
         "lib/angular/angular.js",
         "lib/angular/angular-mocks.js",
         "src/cookbook.js",
         "test/unit/cookbookSpec.js"
       ],
       autoWatch: true,
       browsers: ['Chrome']
     };
   };
   ```

5. Create a Protractor configuration file named `protractor.conf.js` with the following code:

   ```javascript
   exports.config = {
     seleniumServerJar: './node_modules/protractor/selenium/selenium-server-standalone-2.41.0.jar',
     specs: ['test/e2e/cookbookSpec.js'],
     jasmineNodeOpts: {
       showColors: true,
   ```
6. Following this, create a file named gulpfile.js in the root of the project directory with the following code:

```javascript
var gulp = require('gulp');
var $ = require('gulp-load-plugins')();

gulp.task('webdriver_update', $_.protractor.webdriver_update);

gulp.task('connect', function () {
  var connect = require('connect');
  var app = connect()
    .use(connect.static('src'));

  $_.server = require('http').createServer(app)
    .listen(8000);
});

gulp.task('test', function (done) {
  var karma = require('karma').server;
  var karmaConf = require('./karma.conf.js')();
  karma.start(karmaConf, done);
});

gulp.task('e2e', ['connect', 'webdriver_update'], function (done) {
  gulp.src(['test/e2e/cookbookSpec.js'])
    .pipe($_.protractor.protractor(
      {configFile: './protractor.conf.js',
      })
    .on('end', function () {
      $_.server.close();
      done();
    });
});

7. Finally, you can start the testing! You can run unit tests with Karma using gulp test and run the end-to-end tests with Protractor using gulp e2e.
How it works...

Step 1 and step 2 install all the dependencies required to successfully run all of the Gulp tasks. Step 3 and step 4 show how we can set up the configuration files for both Karma and Protractor defining our unit and end-to-end specifications.

Step 5 has the gulpfile.js file where we define all the necessary tasks to automate the unit and end-to-end tests:

- **Unit tests:** As opposed to the approach taken in the *Automating test runners using Grunt* recipe, a dedicated plugin to run Karma is not really necessary. We simply call the start() function, providing the configFile created in step 4 as the first argument, and the Gulp callback to cleanly exit the task as necessary. Finally, we register a task with Gulp named test that calls the Karma test runner to the unit tests.

  ```
  To set a default task as opposed to defining the task name explicitly, add the following line of code to your gulp file:
  gulp.task('default', ['test']);
  ```

- **End-to-end tests:** End-to-end tests need to run Protractor but also have the additional tasks of running HTTP and Selenium servers. We can run Protractor using the gulp-protractor plugin (https://github.com/mllrsohn/gulp-protractor). The connect task sets up a static HTTP server on port 8000 using the files within the src directory. Also, the webdriver_update task uses the webdriver-manager script included with Protractor to ensure that necessary drivers are installed; if not, then they will automatically get installed. This is a slight overhead each time the e2e task is run. However, it can easily be removed and you can call the gulp task independently if required. Within the actual e2e task, we pass the path to the test files to gulp's src function. We can then stream the file structure to the gulp-protractor protractor() function, including the protractor.conf.js file created in step 5 as an additional option. Listening for the end event, a handler will close the static HTTP server and then ensure the gulp callback is called to complete the set of events.

Finally, to run these tasks, we call either the test task or the e2e target.

See also

- The *Automating test runners using Grunt* recipe
- An exceptional resource for learning about streams is the Streams Handbook (which you can access at https://github.com/substack/stream-handbook)
Where to buy this book

You can buy AngularJS Testing Cookbook from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.