Dependency Injection with AngularJS

Design, control, and manage your dependencies with AngularJS dependency injection

Alex Knol

Dependency Injection with AngularJS

Design, control, and manage your dependencies with AngularJS dependency injection

Alex Knol

Dependency Injection with AngularJS

Credits

Author
Alex Knol

Reviewers
Iwan van Staveren

Ruoyu Sun

Acquisition Editor
James Jones

Commissioning Editor
Neil Alexander

Technical Editors
Sharvari H. Baet

Dennis John

Copy Editors
Alisha Aranha

Roshni Banerjee

Tanvi Gaitonde

Gladson Monteiro

Shambhavi Pai

Adithi Shetty

Project Coordinator
Sherin Padayatty

Proofreader
Simran Bhogal

Indexer
Rekha Nair

Production Coordinator
Kyle Albuquerque

Cover Work
Kyle Albuquerque

About the Author

Alex Knol is a lifelong tech geek with a passion for automation. After spending some years away from software development, around the beginning of this century, he took up PHP development based on his early experiences with C and Pascal. Surprisingly, he has never really used web tools, but applications instead, to make websites, such as the platform that's driving `kaizegine.com`. Having built various applications using web technologies and frameworks, such as Symfony, he discovered AngularJS at the beginning of 2008, while searching for a way to structure frontend application code and make development easy. He used AngularJS, among other technologies, for a job-matching project in the Netherlands and, more recently, for an online website designer named `Risingtool.com`.

About the Reviewers

Iwan van Staveren is a software architect. He has over 14 years of experience in developing all kinds of web applications. He loves working with the Symfony2 and AngularJs frameworks. He is owner of the privately-owned E-one Software.

Ruoyu Sun is a designer and developer living in Hong Kong. He is passionate about programming and has been contributing to several open source projects. He has founded several tech startups, using a variety of technologies, before going into the IT industry. He is the author of the book *Designing for XOOPS*, *O'Reilly Media*, *July 2011*.

> I would like to thank all my friends and family who have always supported me.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On-demand and accessible via web browsers

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Dependency Injection is a term often used in the world of object-oriented software design. AngularJS depends on it at its core. This book teaches you why and how to use Dependency Injection with AngularJS.

What this book covers

Chapter 1, *Learning to Fly*, will take you through the basics of an Angular application. This chapter prepares a sample app that will be used throughout the examples in the book.

Chapter 2, *Better Code*, takes you from bad coding practices to maintainable and testable software using Dependency Injection. It also shows you the theory of Dependency Injection and some of its positive effects.

Chapter 3, *The Magic*, is a technical explanation of how Dependency Injection can be used with AngularJS and the caveats to watch out for.

Chapter 4, *Testing*, is a chapter that will show you the advantages that Dependency Injection brings when testing your application. Integration testing and unit testing are covered. The set up and use of recommended AngularJS testing frameworks are covered as well.

Chapter 5, *Large Applications*, will show you ways to implement the theory and techniques used in large applications. The result will be less code and better maintainability.

What you need for this book

To play along with the examples in this book, you just need a working installation of NodeJS and a text editor. To view the output, a web browser is required.

Who this book is for

If you are a developer or software engineer, this book is ideal for you. You should have a basic understanding of JavaScript and web development.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Earlier, you saw how to use the `ng-app` attribute to bootstrap the application."

A block of code is set as follows:

```
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.
min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/raphael/2.1.0/
raphael-min.js"></script>
    <script src="http://cdn.oesmith.co.uk/morris-0.4.1.min.js"></
script>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.
min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/raphael/2.1.0/
raphael-min.js"></script>
    <script src="http://cdn.oesmith.co.uk/morris-0.4.1.min.js"></
script>
```

New terms and **important words** are shown in bold.

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

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.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Learning to Fly

When you code larger applications in JavaScript, you soon run into a need for some kind of organization for your code. There are a great number of frameworks out there that can help you; the sheer number is enough to just keep you coding as you have done for years. Although there is a learning curve involved like in any framework or tool library, it will help you deliver a better end result with fewer headaches. A handy guide for choosing a framework is `http://addyosmani.github.com/todomvc/`. It shows you the same app using different tools and libraries.

Let's get going

The traditional way of getting started with any kind of JavaScript library is downloading it from its website or repository and including it in your HTML file. The first thing you see when you go to `http://angularjs.org/` is a big **Download** button. It will take you right to a pop up to download AngularJS. The defaults will be for the latest stable version. Unless you are interested in the source, the minified version will be fine, as shown in the following screenshot:

Download AngularJS

Branch	Stable	Unstable	?
Build	Minified	Uncompressed	Zip ?
CDN	https://ajax.googleapis.com/ajax/libs/angularjs/1.0.7/angular.min.js		?
Bower	bower install angular		?

Extras   Previous Versions   ⬇ Download

This will download just a minified JavaScript file that you have to include in your HTML file. To get started, your HTML could look like the following:

```
<!doctype html>
<html lang="en" ng-app="myApp">
<head>
  <meta charset="utf-8">
  <title>My First html page with AngularJS</title>
</head>
<body>
  <script src="lib/angular. min. js"></script>
</body>
</html>
```

In our first code excerpt we have used a local version of the Angular JavaScript files. It is recommended to use the CDN version for production applications:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.8/
angular.min.js"></script>
```

While AngularJS does a good job of helping you get organized, along the way, it also makes you think a bit differently about DOM and how to manipulate it. When you are used to working just with tools such as **jQuery** or **Prototype**, a new dimension and a model, will be added to your world. When you use other frameworks such as **KnockoutJS** or **BackBoneJS**, you will feel right at home, but will experience a more opinionated way of writing substantially less code than you were used to.

Let's dive into some code! Let's say you want to ask the user to type a temperature, and depending on the number, show a different response. A simple HTML file with this content would be enough to do that for you, shown as follows:

```
//index.html

<!doctype html>
<html lang="en" ng-app>
<head>
  <meta charset="utf-8">
  <title>My AngularJS App</title>
  <link rel="stylesheet" href="css/app.css"/>
</head>
<body>
  Current temperature: <input ng-model='temp' type='number'/> Celcius
  <p ng-show="temp>=17">Not too bad! {{ temp }} degrees, {{ temp - 10
}} would be a little cold</p>
  <p ng-show="temp<17">{{ temp }} degrees, is a little chilly, {{ temp
+ 10 }} would be a nicer</p>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.4/
angular.min.js"></script>
</body>
</html>
```

Valid HTML

If you like valid HTML, Angular offers ways to specify instructions so that your pages will be completely valid HTML. For instance, you can prefix all the `ng-*` notations with `x-` or `data-`.

Now what exactly is happening here? Just below the doctype declaration, you see `ng-app`. This is a general bootstrapping directive for Angular. In this case, it means that anything between the `<HTML>` tags can contain instructions for Angular to add its magic to your page. Further down you see some more `ng-` attributes. The attribute `ng-model="temp"` binds the value of the input to the variable named `temp`. Now that the input value is bound, setting the value of the input in the HTML file with `value="17"` won't work anymore because Angular has bound it to an empty variable. The `ng-show` attribute triggers a function to show or hide the node it is applied to, based on the evaluation of the expression. All values of the Angular `name="value"` pairs can contain something that resembles regular JavaScript, although there are differences. Check the documentation to read about the differences at `http://docs.angularjs.org/guide/expression`. In general, you should keep the amount of "code" in your templates to a minimum to avoid mixing presentation with logic. Now that we have the input, we need something to display the `temp` variable. Angular, by default, uses double curly braces for interpolating variables in a template, for example, `{{ temp }}`. As you can see, the interpolation can also contain any valid JavaScript expression; for example, `{{ temp + 10 }}`.

Delimiters {{ ... }}

When you serve your Angular application from a framework like Symfony, the double curly braces `{{ }}` will conflict with the Twig templating engine, because it also uses the same characters as delimiters. To stay compatible with any plugins that rely on these delimiters on the server side, you should not change them. Angular offers a way to change the delimiters into something else:

```
var myAppModule = angular.module('myApp',[],
  function ($interpolateProvider) {
    $interpolateProvider.startSymbol('<[');
    $interpolateProvider.endSymbol(']>');
  });
```

This will change the stock `{{ }}` to `<[ ]>` for the current Angular app.

As you saw earlier, the input field was bound to an empty variable `temp` to begin with. What if we wanted to start with a temperature of, say, 17 degrees? Technically, we could set the value of `temp` to 17 in the template, but because one of our goals here is to have better code organization, we'll place this nicely inside a JavaScript file.

Adding a controller

Earlier in the chapter, you saw how to use the `ng-app` attribute to bootstrap the application. We can specialize this bootstrapping to apply a specific piece of JavaScript to the containing nodes. So we could have the `<BODY>` tag enriched by a bit of Angular. To do this, the file should look like the following:

```
// index.html
…...
</head>
<body ng-controller="TempCtrl">
Current temperature: <input ng-model='temp' type='number'/> Celcius
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.4/
angular.min.js">
<script src="app/js/app.js">
…...
```

This will tell Angular to look for a function named `TempCtrl` inside the included JavaScript files. The JavaScript file that will contain the code will be `app.js` so, we will need to include that in the HTML file.

To start off with a temperature of 17 degrees, the controller would look like the following:

```
//app/js/app.js

'use strict';

function TempCtrl($scope) {
  $scope.temp = 17;
}
```

The controller function is called with the `$scope` variable. This variable represents the scope the controller is responsible for. All properties of the `$scope` variable are magically bound to variables of the same name in the HTML template. As you can see, the temperature is simply set to 17 and the manipulation done inside this controller gets synchronized to the template by Angular. The reverse is also true; when a variable is changed in the template, it also gets synced to the controller. Angular has added a two-way binding between the JavaScript code and the HTML template, as shown in the following screenshot:

Current | History

Current temperature: 17   ▲▼ Celcius

Not too bad! 17 degrees, 7 would be a little cold

> `use strict` is optional but helps you, as a developer, to create ECMAScript 5-compatible JavaScript and provides more feedback in supported browsers. For more information, refer to the following link, which is a good read among others `http://ejohn.org/blog/ecmascript-5-strict-mode-json-and-more/`.

What about routes?

Next to seeing the actual temperature in the current page, it would be nice to have a page showing some history of what the temperature has been in the past. To do this, we need some kind of routing to navigate between the pages. Along with the routing feature, you will see the Angular module system to organize the various components, shown as follows:

```
// index.html

<!doctype html>
<html lang="en" ng-app="tempApp">
  <head>
    <meta charset="utf-8">
    <title>My AngularJS App</title>
    <link rel="stylesheet" href="css/app.css"/>
  </head>
  <body>
    <ul class="menu">
      <li><a href="#/current">Current</a></li>
      <li><a href="#/history">History</a></li>
    </ul>

    <div ng-view></div>

    <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.0.4/
angular.min.js"></script>
```

```
      <script src="app/js/app.js"></script>

   </body>
</html>

// partials/current.html
Current temperature: <input ng-model='temp' type='number'/> Celcius
<p ng-show="temp>=17">Not too bad! {{ temp }} degrees, {{ temp - 10 }}
would be a little cold</p>
<p ng-show="temp<17">{{ temp }} degrees, is a little chilly, {{ temp +
10 }} would be a nicer</p>

// partials/history.html
<p>This will show some history</p>
```

The observant reader will have noticed that the `ng-app` attribute has been extended with `="tempApp"`. This is to tell Angular to use a specific module to bootstrap this part of the HTML page. The code that was inside the `<BODY>` tag has been moved to the `partial` folder and has been replaced by a navigation menu. The navigation menu just refers to routes using hashbangs (`#!`). The following manual can help you use the HTML5 History API and allow correct indexing using search engine bots: `http://docs.angularjs.org/guide/dev_guide.services.$location`. Our module is defined in the following JavaScript file:

```
// app/js/app.js
'use strict';

var tempApp = angular.module('tempApp', []).
  config(['$routeProvider', function($routeProvider) {
    $routeProvider.when('/current', {templateUrl: 'partials/current.
html', controller: 'CurrentCtrl'});
    $routeProvider.when('/history', {templateUrl: 'partials/history.
html', controller: 'HistoryCtrl'});
    $routeProvider.otherwise({redirectTo: '/current'});
  }]);

// app/js/controllers.js
'use strict';

tempApp.controller('CurrentCtrl', ['$scope',
  function($scope) {
    $scope.temp = 17;
  }]);

tempApp.controller('HistoryCtrl', ['$scope',
  function($scope) {

  }]);
```

First the module is initialized with the name from the template `tempApp`. The empty array after the module name can contain dependencies for the module. Then the module is configured using `$routeProvider` as a dependency, which is used to redirect a URI to a partial HTML template using a specific controller. The notation of the controllers has also significantly changed. They are now contained in the module as a property. While defining the controllers as a function in the global scope will still work fine, `.controller` is the recommended way of writing Angular modules. As before, the `temp` controller is still depending on `$scope` to be present inside the function.

Showing a list

The History tab is only showing a placeholder, so let's go ahead and change that. Let's assume we have stored some temperature data in a database and we have read that data into a variable. For demonstration purposes we have just defined some data inside the controller. Fetching data from a backend is beyond the scope of this book:

```
tempApp.controller('HistoryCtrl', ['$scope',
  function($scope) {
    $scope.historyData = [
      { day: 'saturday', temp: 8},
      { day: 'sunday', temp: 13},
      { day: 'monday',temp: 15},
      { day: 'tuesday', temp: 11},
      { day: 'wednesday',temp: 15},
      { day: 'thursday',temp: 17},
      { day: 'friday',temp: 21}
    ]
  }]);
```

The controller just depends on `$scope` and has assigned some data inside an array to the property `historyData`. The template looks like the following:

```
  <ul ng-repeat="value in historyData">
    <li>{{ value.day }} : {{ value.temp }}</li>
</ul>
```

`ng-repeat` is an Angular function to do something similar to a `for-each` loop. Because it is declared on the `<ul>` tag, it will repeat the elements inside the list for every element it finds in the array `historyData`. `value.day` just refers to the `day` property `day` inside the hash.

Adding a filter

Let's say our users are only interested in temperatures above 15 degrees. We could modify the data in the controller. If we need this same data elsewhere, we could create a copy of the data and modify that for display. Angular has a neat feature called **filters**. There are several filters included with Angular, but it is simple to create your own filters. First, we'll use a standard Angular filter to convert the names of all days to uppercase:

```
<ul ng-repeat="value in historyData">
  <li>{{ value.day | uppercase}} : {{ value.temp }}</li>
</ul>
```

Angular follows the Unix-style pipe to transfer data down the line using a | symbol. To just show temperatures above 15, we need to filter the `historyData` array:

```
tempApp.filter('plusFifteen', [ function() {
    return function(arrTemp) {
      var arrReturn = new Array();
      angular.forEach(arrTemp, function(value, key){
        if(value.temp>=15) arrReturn.push(value);
      });
      return arrReturn;
    }
  }]);
```

This filter takes an array as its argument and returns only those elements with a higher temperature than 15. This piece of code is tightly coupled with the example we have here. The filter is used in a similar way as the uppercase filter we used before:

```
<ul ng-repeat="value in historyData | plusFifteen">
  <li>{{ value.day | uppercase}} : {{ value.temp }}</li>
</ul>
```
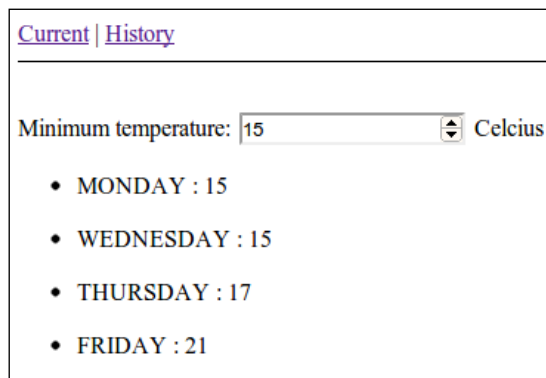
Now we could make our filter a bit more configurable by making the minimum temperature configurable in the template. That way we can also reuse this filter for lists where we want to see other minimum temperatures:

```
tempApp.filter('minimum', [ function() {
    return function(arrTemp, minimum) {
      var arrReturn = new Array();
      var min = minimum ? minimum : 15;
      angular.forEach(arrTemp, function(value, key){
        if(value.temp>=min) arrReturn.push(value);
      });
      return arrReturn;
    }
  }]);
```

The filter now takes an optional argument as the minimum value to display. When you run the code without changing the template, it works exactly like before. To reflect the new functionality of our filter, we now also have changed the filter's name to `minimum`. To take advantage of the new feature, we have to specify the minimum temperature as a filter argument. While we are at it, we will let our users decide for themselves what minimum temperature they wish to see:

```
Minimum temperature: <input ng-model='tempMin' type='number'/> Celcius
<ul ng-repeat="value in historyData | minimum:tempMin">
  <li>{{ value.day | uppercase}} : {{ value.temp }}</li>
</ul>
```

Our template now has an input box in which the value `tempMin` is bound to the filter as an argument, shown as follows:

Current | History

Minimum temperature: 15 ⬍ Celcius

- MONDAY : 15
- WEDNESDAY : 15
- THURSDAY : 17
- FRIDAY : 21

As the user changes the value of the input box, the list is dynamically filtered. To start the input box with the value 15, all we have to do is add `$scope.tempMin = 15` to our history controller.

Chart directives

It is not bad to see our data as an unordered list, but showing it in a chart would really make our users happy. We will be using a third-party library to render our chart and wrap it inside an Angular directive. The goal of this exercise is to use our directive as a new HTML tag `<chart ...></chart>`. This is, by far, one of the coolest features of Angular. For a while the subtitle of the Angular website was after all "teaching HTML new tricks". Let's go ahead and change our template, so it uses our new directive:

```
Minimum temperature: <input ng-model='tempMin' type='number'/> Celcius
<chart historyData | minimum:tempMin"></chart>
```

The real work is now delegated to our new directive, leaving the template clean and concise:
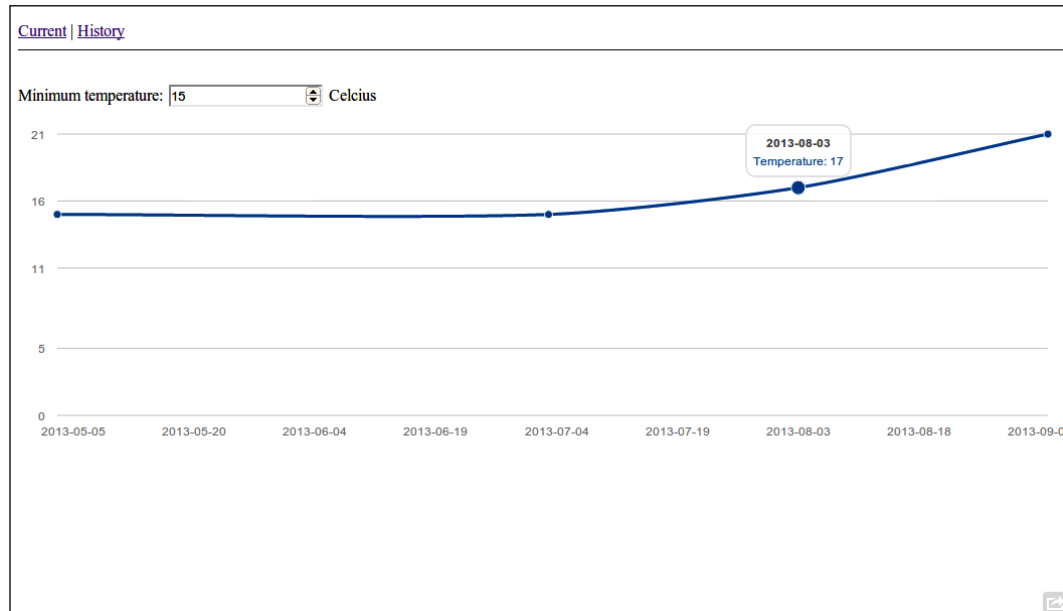
```
tempApp.
        directive('tempChart', [function(version) {
    return {
      template: '<div id="container"></div>',
      link: function(scope, element, attrs) {
        var chart = new Morris.Line({
          // ID of the element in which to draw the chart.
          element: 'container',
          // The name of the data record attribute that contains
x-values.
          xkey: 'date',
          // A list of names of data record attributes that contain
y-values.
          ykeys: ['temp'],
          // Labels for the ykeys -- will be displayed when you hover
over the
          // chart.
          labels: ['Temperature']
        });
        scope.$watch(function() {
          chart.setData(angular.copy(JSON.parse(attrs.data)));
        });
      }
    }
  }]);
```

The `tempChart` directive encapsulates a charting library by taking data from the template and returning the rendered chart. One of the key elements in the code snippet is the `scope.$watch` statement without a specific `watchExpression`. It simply waits for a `$digest` run by Angular and will update the data for the chart. `$digest` is run anytime a key is pressed and Angular calls `$apply` internally. A good explanation of these concepts is in the Angular manual at `http://docs.angularjs.org/guide/concepts`.

The chart library we used is **Morris.js** (`http://www.oesmith.co.uk/morris.js`) and the chart-specific code is annotated because the details of that library are beyond the scope of this book. To get Morris to work correctly, add the following lines to the `index.html` file:

```
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.
min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/raphael/2.1.0/
raphael-min.js"></script>
    <script src="http://cdn.oesmith.co.uk/morris-0.4.1.min.js"></
script>
```

The output is shown as follows:

Current | History

Minimum temperature: 15 ⊟ Celcius

The result is a page where the user can now use the number input to control the data visualized in the chart.

Using services

One last element of Angular that deserves attention is the concept of services. One of the built-in services is the location service. The **location** service provides read and write accesses to URLs and lets the developer use it for navigating around the application. It can be used to read and change the URL. Changes can then be picked up by the route service, as we have already seen before. Services are injected into controllers or other services that depend on them.

These are the basic building blocks to create single-page applications using AngularJS. In the following chapters you will learn more about injecting dependencies and why this will benefit you as a developer and your clients.

> **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.

Summary

In this chapter you have seen how to get started with AngularJS. First, we looked at the setup of an Angular project. Then we created a simple page with some angular bindings. We added a controller and another page. The second page showed just a list at first, but got upgraded to a nice chart. We immediately encapsulated the chart in a directive. Lastly, we mentioned services since we had already used them.

2
Better Code

When we created our `HistoryController`, we put the `historyData` object containing days and their corresponding temperatures inside it. For demonstration purposes, this is not the end of the world, but the controller is useless without it. So the controller depends on this object to function correctly. The best thing to do here is to take the hash object out of our controller and replace it with something that will retrieve the data for us. After all, we are not building a static application. Let us refactor that to make it a bit more lifelike by using an external source. For this example, we will use `Parse.com`, a **Backend as a Service** (**BaaS**) that functions like an API. This service can handle all the tasks that we would normally have to handcode in the backend of our applications. Since it is beyond the scope of this book, using a Baas, let us use a backend without coding one up.

Wiring up the backend

We have created a free developer account and imported our sample data into the Parse class `Reading` using the Parse dashboard.

```
// /path/to/file/history.json
[
   { "date": "2013-04-01T17:01:22.634Z", "temp": 8},
   { "date": "2013-04-02T17:01:22.634Z", "temp": 13},
   { "date": "2013-04-03T17:01:22.634Z", "temp": 15},
   { "date": "2013-04-04T17:01:22.634Z", "temp": 11},
   { "date": "2013-04-05T17:01:22.634Z", "temp": 15},
   { "date": "2013-04-06T17:01:22.634Z", "temp": 17},
   { "date": "2013-04-07T17:01:22.634Z", "temp": 21}
]
```

First the data is moved out of the controller into a file somewhere on the filesystem. Notice the difference in the notation of JSON compared to the previous JavaScript notation. JSON doesn't allow keys to be unquoted. Next, we have to somehow get our controller to use the Parse SDK to get data from the backend. There are a few design patterns that could help us here. The first one is as suggested by Parse the documentation. Just load the following script before the Angular `script` tag:

```
// index.html
…
<script type="text/javascript
src="http://www.parsecdn.com/js/parse-1.2.3.min.js"></script>
<script src="//ajax.googleapis.com/ajax/libs/angularjs/
1.0.4/angular.min.js"></script>
<script src="app/js/app.js"></script>
<script src="app/js/controllers_factory.js"></script>
…
```

After loading the script, you will have to update the controller to get data from `Parse.com`, using the Parse object from the global scope:

```
//app/controllers.js
tempApp.controller('HistoryCtrl', ['$scope', function ($scope) {
    Parse.initialize("wNpkWu0OBGAAajJlnnqYPW3wsOT3T43LMn0e3VFb",
"04wqQGq62frJjEWzDOhISMrmtWDRFPjGuCoD4zWi");

    var Reading = Parse.Object.extend("Reading");
    var query = new Parse.Query(Reading);
    query.find({
        success: function (results) {
            $scope.historyData = [];
            angular.forEach(results, function (i, e, a) {
                $scope.historyData.push({
                    date: i.get('date'),
                    temp: i.get('temp')
                })
            })
            $scope.$apply();
        },
        error: function (error) {
            alert("Error: " + error.code + " " + error.message);
        }
    });
    //below unchanged
    $scope.tempMin = 15;
    $scope.minimum = function (value) {
        if (value.temp >= $scope.tempMin) return value;
    }
}]);
```

First, the Parse library needs to know our credentials in order to access the service. This is done in the `Parse.initialize` call. After that is done a local `Reading` object is created by extending `Parse.Object`. This connects our local object to a remote entity named `reading`. Then, a query is created on the collection of the `Reading` entities that is used by the Parse service to return all objects in the collection. Upon success, the result is then iterated and pushed into the `$scope.historyData` property. The end result is the same as before; the template renders our chart directive with the correct `historyData` object.

Duplicating code

Duplicating code seems like a good approach. We're using a cool service to get our data and insert it in the scope variable. It also looks very clean and concise. There are, however, several problems here. The first is that the credentials are hardcoded and there is a lot of code specific to Parse in our controller now. To illustrate one of the difficulties, we will change our `current` page so that it will allow users to save the temperature they selected, using the Parse service, shown as follows:

```
// partials/current.html
Current temperature: <input ng-model='temp' type='number'/> Celcius
<p ng-show="temp>=17">Not too bad! {{ temp }} degrees, {{ temp - 10 }}
would be a little cold</p>
<p ng-show="temp<17">{{ temp }} degrees, is a little chilly, {{ temp +
10 }} would be a nicer</p>
<input type="button" ng-click="save()" value="save"/>
```

First, we have to change the template because we want the user to decide which values get saved to the database using a button. We just added this button as an Angular attribute `ng-click="save()"`. This means that it will execute the function that `$scope.save` returns when the button is clicked.

```
// app/controllers.js
tempApp.controller('CurrentCtrl', ['$scope', function ($scope) {
    $scope.temp = 17;
    Parse.initialize("wNpkWu0OBGAAajJlnnqYPW3wsOT3T43LMn0e3VFb",
    "04wqQGq62frJjEWzDOhISMrmtWDRFPjGuCoD4zWi");

    var Reading = Parse.Object.extend("Reading");
    var reading = new Reading();

    $scope.save = function () {
        reading.set("date", new Date().toISOString());
        reading.set("temp", $scope.temp);
        reading.save(null, {
            success: function (gameScore) {
                alert('reading saved');
```

```
            },
            error: function (gameScore, error) {
                // The save failed.
                // error is a Parse.Error with an error code and
                    description.
            }
        });
    }
}]);
```

The first few lines are copied straight from our `HistoryController`. Next, the `$scope.save` function gets invoked when the user presses the button. We use the Parse SDK syntax to create a new `Reading` object, fill it with the current date and the selected temperature in it and persist it in the Parse backend. The two-way binding takes care of propagating the value that the user selected from the input element to the `$scope.temp` variable. To confirm that the save action worked, an alert will be shown with a success message. The same could be done in case of an error. So now when we have an application that works well and performs all that we have asked of it.

Current | History

Current temperature: 19 Celcius

Not too bad! 19 degrees, 9 would be a little cold

save
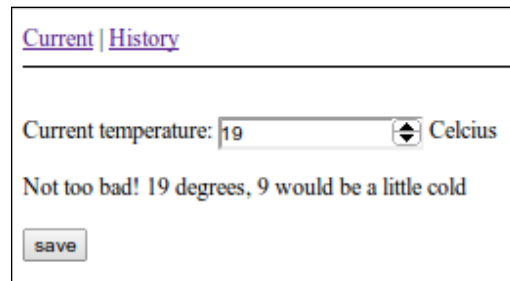
However, we have duplicated code which is not very **DRY** (**Don't Repeat Yourself**). This means that every time something changes, we have to keep it all in sync with our credentials or the Parse API every time something changes. Secondly, the way to test this service is a bit tricky since we need to have a live Internet connection and a valid Parse account with known data at all times.

When developing in a test-driven manner, you would create your tests at the same time or even before you create the actual code you are testing. We have dedicated a chapter to testing and the advantages of Dependency Injection for testing. So don't worry; we won't forget to create the tests!

Angular service to the rescue

To circumvent all these problems, Angular has a concept of services that we mentioned at the end of *Chapter 1*, *Learning to Fly*. Talking to the Parse API or service can be viewed as a service to the application. The service is responsible for initializing Parse and instantiating the `Reading` object that is ready for manipulation. The service can then be injected in the two controllers and the shared code is nicely centralized:

```
// app/services.js
angular.module('serviceModule', [function ($provide) {
    $provide.factory('parse', [function () {
        // inti Parse SDK
        Parse.initialize("wNpkWu0OBGAAajJlnnqYPW3wsOT3T43LMn0e3VFb",
        "04wqQGq62frJjEWzDOhISMrmtWDRFPjGuCoD4zWi");
    }]);
}]);
```

To make our service completely separate from our application, we create a new module named `serviceModule`. You can read more about organizing your application in the *Chapter 5*, *Large Applications*. The service simply initializes the Parse service with the required credentials and returns the instance. When you look closely, you see that the factory registers a function instead of the Parse instance. This means that the service is only instantiated when actually called. Creating our service inside a new module means our `tempApp` has no knowledge of our service yet. It means we need to inject this new module in our application.

```
// app/app.js
var tempApp = angular.module('tempApp', ['serviceModule']).
  config(['$routeProvider', function($routeProvider) {
    $routeProvider.when('/current', {templateUrl: 'partials/current.
html', controller: 'CurrentCtrl'});
  ...
```

Let's take our new `serviceModule` one step further and extend it with a second service on top of our generic Parse service that will use the Parse instance to expose methods for `query` and `save`. This allows us to take some more code out of our controllers.

```
angular.module('serviceModule', [function ($provide) {
    $provide.factory('parse', [function () {
        // init Parse SDK
        Parse.initialize("wNpoWu0OBGAAajJlnnqYPW3wsOT3T43
LMn0e3VFb", "04wqQGq62frJjEWzDOhISMZmtWDRFPjGuCoD4zWi");
    }]);
    $provide.factory('reading', ['parse', function () {
        // our reading object
        var Reading = Parse.Object.extend("Reading");
        // the service that will be returned
        var serviceInstance = {
            save: function (temp) {
                reading = new Reading();
                reading.set("date", new Date().toISOString());

                reading.set("temp", temp);
                reading.save(null, {
                    success: function (reading) {
                        alert('reading saved');
                    },
                    error: function (reading, error) {
                        // The save failed.
                        alert("Error: " + error.code + " " +
                                error.message);
                    }
                });
            },

            query: function (callback) {
                var query = new Parse.Query(Reading);
                query.find({
                    success: function (results) {
                        var historyData = [];
                        angular.forEach(results,
                        function (i, e, a) {
                            historyData.push({
                                date: i.get('date'),
                                temp: i.get('temp')
                            })
                        })
                        callback(historyData);
                    },
                    error: function (error) {
```

```
                    alert("Error: " + error.code + " " + error.
                        message);
                }
            });
        }
    };

    return serviceInstance;
    }]);
}]);
```

Using the array notation that have seen before, we inject the `Parse` service into our `Reading` service and use it to return `serviceInstance`. The `Reading` object is instantiated and exposed locally. Then, a public method for querying the `Reading` objects is exposed through `query`. Lastly, we created a method for creating and persisting an object by the `save` method. That's all we need for now, but we can freely extend the functionality if we need to. As you can see, we have wrapped the native Parse methods with our own methods. This means that if we should ever want to swap Parse for something else, all we have to change is the service and nothing else.

```
tempApp.controller('CurrentCtrl', ['$scope', 'reading', function
($scope, reading) {
    $scope.temp = 17;

    $scope.save = function () {
        reading.save($scope.temp);
    }
}]);

tempApp.controller('HistoryCtrl', ['$scope', 'reading', function
($scope, reading) {
    reading.query(function (data) {
        $scope.historyData = data;
        $scope.$apply();
    });

    //below unchanged
    $scope.tempMin = 15;
    $scope.minimum = function (value) {
        if (value.temp >= $scope.tempMin) return value;
    }
}]);
```

First of all, the Parse-specific code has been centralized and is extracted from the controllers. We can very easily use the reading service in many more classes with minimal lines of code. Reducing lines of code is not a goal of Dependency Injection but a side effect. Another advantage of separating the responsibilities is that the separate parts are now smaller chunks of code. This means it will be easier for a new developer coming to your team or (open source) project, to learn the purpose and the functionality of the code. Fewer lines of code also means that there are less possibilities for errors. The way we have now separated the responsibilities naturally benefits testing because we can test individual functions and we are not forced to test a bunch of different functions in one single body of code.

We have now abstracted our service into a module. This means it has been separated totally from our application and our shiny new service has been injected into our controllers. Not only can we share the Parse module in our own application, but we can also share it with others. A good place to look for existing modules, or share your modules for Angular is `http://www.ngmodules.org`. This is a registry for public modules to be used for your Angular applications. It includes modules for using jQuery UI or Twitter Bootstrap in your project. Have a look around; there are many useful modules available. After all, the first rule of software development is to make sure that you are not going to make something that is already available.

The theory behind Dependency Injection

We have now used Dependency Injection and seen it in action. By now you should have a decent grasp of why it is useful, but what about the theory behind it? Let's start with **SOLID**. This is a basic principle by *Robert C. Martin* that was introduced in the year 2000. It is an acronym that describes the five basic principles of object-oriented software design. It advocates a method of development that allows you to produce software that can easily be extended and is also easier to read. The following table lists the five SOLID principles:

Initial	Stands for	Concept
S	**Single responsibility principle (SRP)**	This principle states that a class should have only a single responsibility.
O	**Open/closed principle (OCP)**	This principle states that software entities should be open for extension, but closed for modification.
L	**Liskov substitution principle (LSP)**	This principle states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Initial	Stands for	Concept
I	**Interface segregation principle (ISP)**	This principle states that many client-specific interfaces are better than one general-purpose interface.
D	**Dependency inversion principle (DIP)**	This principal states that one should depend upon abstractions. Do not depend upon concretions. Dependency injection is one method following this principle.

Source: `http://en.wikipedia.org/wiki/SOLID`

Let's see the principles of object-oriented software design in detail, shown as follows:

- **Single responsibility principle**: This principle dictates that a class can only have one responsibility, just as our services are divided into one that configures the Parse service and the other that uses the configured Parse service for providing a `Reading` object with some basic methods. This seems like a very straightforward principle but can very easily be overlooked when you're in a hurry. The benefit is that your code will be easy to test and less likely to break when something in the code needs change.

- **Open/closed principle**: This principle states that code should be designed in such a way that they are open to extension but closed for modifications. Code written in this way will never change unless there is a bug in it. This means that everything that depends on the code will always understand it, and tests written for it will not have to be changed either because they keep passing.

- **Liskov's principle**: This principle reminds us of something that should already be a common practice for any object-oriented developer: subclasses should be behaviorally compatible with their superclass. A simple example would be a new object named `highestReading` that is based on the `Reading` object. This new object may not change `temp` to, say, `highestTemp`. This principle really works together with the open/close principle. It allows you to re-use your subclasses just like the superclass.

- **Interface Segregation principle**: This principle tells us we should not force users of a class to depend on methods they do not use. So our `Reading` service should not force the caller to call unnecessary methods that are not relevant for performing either a query or a save operation.

- **Dependency Inversion principle**: This principle resembles the subject of this book quite closely and is, in fact, related to it, but not as closely as you might think. Dependency inversion is about having high- and low-level components in a software project that depends on abstractions. Abstractions can be thought of as standard interfaces. In traditional software development, lower-level code was consumed by higher-level components to create more complex systems. This resulted in a close coupling of the different levels of code. By depending on the different levels on abstractions, this coupling has been largely reduced and code can be maintained and reused in an easier way. Dependency Injection is one of the ways these dependencies can be made available to the code. Others are plugins or a service locator.

- **Law of Demeter (LaD)**: This is another design principle that has some significance in the context of Dependency Injection. It roughly states that when calling classes or functions, you should not *reach through* the callee and use functions of another object or class inside the one you called. This causes tight coupling and makes maintenance and adaptability more difficult. When applied to JavaScript, breaking the Law of Demeter could look like this:

```
// service
angular.module('serviceModule', [function ($provide) {
  $provide.factory('parse', [function () {
    // init Parse SDK
    Parse.initialize("wNpoWu0OBGAAajJlnnqYPW3......");
    return Parse;
  }]);
  $provide.factory('reading', ['parse', function
  (backend) {
    // our reading object
    var Reading = backend.Object.extend("Reading");
    // the service that will be returned
    var serviceInstance = {
      save: function (temp) {…..
      },
      ….....
      backend: backend
      return serviceInstance;
      // return the Parse object tool
    }]);
  }]);

//controller
var currentCtrl = function ($scope, reading) {
    $scope.temp = 17;

    $scope.bad_idea = reading.backend.VERSION
    //reach through the 'reading' service
```

```
  $scope.save = function () {
    reading.save($scope.temp);
  }
};
```

In the preceding example, we exposed the Parse service through the "reading" service. In the controller, we reached through the reading service and called a method on the Parse service with `reading.backend.VERSION`. This is a bad idea, because when something changes in the Parse API, all its uses have to be changed throughout your code. Also, replacing Parse with another service will mean a bigger search and replacement of tasks. It means we should have written wrappers for the Parse methods that we want to expose through the reading class. This would be the correct way to get the version of the parse backend in the controller.

```
// service
 angular.module('serviceModule', [function ($provide) {
   $provide.factory('parse', [function () {
     // init Parse SDK
     Parse.initialize("wNpoWu0OBGAAajJlnnqYPW3......");
     return Parse;
   }]);
   $provide.factory('reading', ['parse', function
    (backend) {
     // our reading object
     var Reading = backend.Object.extend("Reading");
     // the service that will be returned
     var serviceInstance = {
       save: function (temp) {…..
       }….....
       version: backend.VERSION
       return serviceInstance;
     }]);
   }]);

 //controller
 var currentCtrl = function ($scope, reading) {
     $scope.temp = 17;

     $sopce.good_idea = reading.version

     $scope.save = function () {
       reading.save($scope.temp);
     }
   };
```

Summary

In this chapter we have looked at simplifying our application by removing duplicate code and abstracting that functionality away into a service and then injecting that into different parts of the application. Afterwards, we looked at the SOLID principles and how they play a role in Dependency Injection. The Law of Demeter was the closing piece of to back up the Dependency Injection paradigm.

3
The Magic

It is time we explain some of the inner workings of Angular. Angular does some neat things for us. It saves us from having to write a bunch of boilerplate code. Dependency Injection is baked into AngularJS and heavily used throughout. Another feature is a built-in subset of a jQuery functionality called jQLite.
It contains all the necessary functions to make AngularJS run without jQuery and has the exact same interface. If jQuery is available in your application, it will be used instead. Angular also takes the burden of bootstrapping your application, which will be covered later in this chapter.

Application flow

In the following diagram, from the Angular manual, you find a comprehensive schematic depiction of the program flow inside Angular:
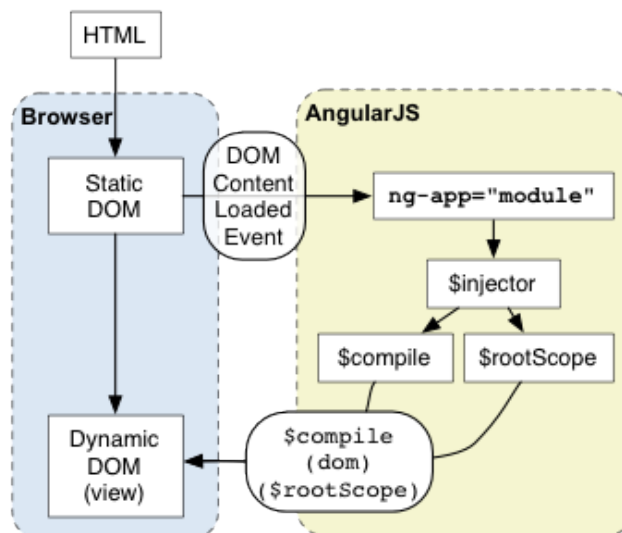
After the browser loads the HTML and parses it into a DOM, the `angular.js` script file is loaded. This can be added before or at the bottom of the `<body>` tag, although adding it at the bottom is preferred. Angular waits for the browser to fire the `DOMContentLoaded` event. This is similar to the way jQuery is bootstrapped, as illustrated in the following code:

```
$(document).ready(function(){
    // do jQuery
})
```

In the `Angular.js` file, towards the end, after the entire code has been parsed by the browser, you will find the following code:

```
jqLite(document).ready(function() {
angularInit(document, bootstrap);
  });
```

The preceding code calls the function that looks for various flavors of the `ng-app` directive that you can use to bootstrap your Angular application.

```
['ng:app', 'ng-app', 'x-ng-app', 'data-ng-app']
```

Typically, the `ng-app` directive will be the HTML tag, but in theory, it could be any tag as long as there is only one of them. The module specification is optional and can tell the `$injector` service which of the defined modules to load.

```
//index.html
<!doctype html>
<html lang="en" ng-app="tempApp">
<head>
  …...
// app.js
…..
angular.module('tempApp', ['serviceModule'])
…..
```

In turn, the `$injector` service will create `$rootscope`, the parent scope of all Angular scopes, as the name suggests. This `$rootscope` is linked to DOM itself as a parent to all other Angular scopes. The `$injector` service will also create the `$compile` service that will traverse the DOM and look for directives. These directives are searched for within the complete list of declared Angular internal directives and custom directives at hand. This way, it can recognize directives declared as an element, as attributes, inside the class definition, or as a comment. Now that Angular is properly Bootstrapped, we can actually start executing some application code. This can be done in a variety of ways, shown as follows:

- In the initial examples, we started creating some Angular code with curly braces using some built-in Angular functions

- It is also possible to define a controller to control a specific part of the HTML page, as we have shown in the first `tempCtrl` code snippet
- We have also shown you how to use Angular's built-in router to manage your application using client-side routing

As you can see, Angular extends the capabilities of HTML by providing a clever way to add new directives. The key ingredient here is the `$injector` service, which provides a way to look up for dependencies and create `$rootscope`.

Different ways of injecting

Let's look a bit more at how `$injector` does its work. Throughout all the examples in this book, we have used the array-style notation to define our controllers, modules, services, and directives.

```
// app/ controllers.js
tempApp.controller('CurrentCtrl', ['$scope', 'reading',
  function ($scope, reading) {
      $scope.temp = 17;
...
```

This style is commonly referred to as **annotation**. Each injected value is annotated in the same order inside an array. You may have looked through the AngularJs website and may have seen different ways of defining functions.

```
// angularJs home page JavaScript Projects example
functionListCtrl($scope, Project) {
  $scope.projects = Project.query();
}
```

So, what is the difference and why are we using another way of defining functions? The first difference you may notice is the definition of all the functions in the global scope. For reference, let's call this the **simple injection method**. The documentation states that this is a concise notation that really is only suited for demo applications because it is nothing but a potential clash waiting to happen. Any other JS library or framework you may have included could potentially have a function with the same name and cause your software to malfunction by executing this function instead of yours. After assigning the Angular module to a variable such as `tempApp`, we will chain the methods to that variable like we have done in this book so far; you could also just chain them directly as follows:

```
angular.module('tempApp').controller('CurrentCtrl', function($scope)
{})
```

These are essentially the same definitions and don't cause pollution in the global scope. The second difference that you may have noticed is in the way the dependencies are injected in the function. At the time of writing this book, most, if not all of the examples on the AngularJs website use the simple injection method. The dependencies are just parameters in the function definitions. Magically, Angular is able to figure out which parameter is what by the name because the order does not matter. So the preceding example could be rewritten as follows, and it would still function correctly:

```
// reversedangularJs home page JavaScript Projects example
functionListCtrl( Project, Scope ) {
  $scope.projects = Project.query();
}
```

This is not a feature of the JavaScript language, so it must have been added by those smart Angular engineers. The magic behind this can be found in the injector. The parameters of the function are scanned, and Angular extracts the names of the parameters to be able to resolve them.

The problem with this approach is that when you deploy a wonderful new application to production, it will probably be minified and even obfuscated. This will rename `$scope` and `Project` to something like `a` and `b`. Even Angular will then be unable to resolve the dependencies. There are two ways to solve this problem in Angular. You have seen one of them already, but we will explain it further. You can wrap the function in an array and type the names of the dependencies as strings before the function definition in the order in which you supplied them as arguments to the function.

```
// app/ controllers.js
tempApp.controller('CurrentCtrl', ['$scope', 'reading',
  function ($scope, reading) {
      $scope.temp = 17;
.......
```

The corresponding order of the strings and the function arguments is significant here. Also, the strings should appear before the function arguments.

If you prefer the definition without the array notation, there is still some hope. Angular provides a way to inform the injector service of the dependencies you are trying to inject.

```
varCurrentCtrl = function($scope, reading) {
  $scope.temp = 17;

  $scope.save = function() {
reading.save($scope.temp);
  }
};

CurrentCtrl.$inject = ['$scope', 'reading'];

tempApp.controller('CurrentCtrl', CurrentCtrl);
```

As you can see, the definition is a bit more sizable, but essentially the same thing is happening here. The injector is informed by filling the `$inject` property of the function with an array of the injected dependencies. This is where Angular will then pick them up from.

> To understand how Angular accomplishes all of this, you should read this excellent blog post by *Alex Rothenberg*. Here, he explains how all of this works internally. The link to his blog is as follows:
>
> `http://www.alexrothenberg.com/2013/02/11/the-magic-behind-angularjs-dependency-injection.html`.

Angular cleverly uses the `toString()` function of objects to be able to examine in which order the arguments were specified and what their names are.

> There is actually a third way to specify dependencies called ngmin, which is not native to Angular. It lets you use the simple injection method and parses and translates it to avoid minification problems.
>
> `https://github.com/btford/ngmin`
>
> Consider the following code:
>
> ```
> angular.module('whatever').controller('MyCtrl',
>   function ($scope, $http) { ... });
> ```
>
> ngmin will turn the preceding code into the following:
>
> ```
> angular.module('whatever').controller('MyCtrl',
>   ['$scope', '$http', function ($scope, $http) { ... }]);
> ```

Summary

In this chapter, we started by looking at how AngularJS is bootstrapped. Then, we looked at how the injector works and why minification might ruin your plans there. We also saw that there are ways to avoid these problems by specifying dependencies differently.

4
Testing

During the course of this book we have spoken about testing a few times, but never actually got into it. Now is the time to get our (testing) hands dirty. The reason we reserved a section for testing is because it is the second most important part of the development process, after your awesome code, of course.

Any modern software development process should include testing, and all developers test their code by executing it to see if the correct behavior is delivered by the code. For web developers this means looking at their work in a browser and interacting with it. For many developers and even teams, testing is something that comes at the end of programming what they made. While this is fine for most small projects up to a few hundred lines of code for a single developer, it will quickly become a royal pain if there are more developers involved or the code base grows.

Test automation

Automated testing to the rescue! Automated testing will test your code without you having to go to the browser, and click on things and fill out forms. The tests will do this for you, as if they were a real person visiting your application. The advantage is that once you have defined your tests, they will be able to run exactly like that every time, unlike your manual tests, where you have to take care to remember to test every case every time. They are also able to run much faster than you could inside a browser window. For those who haven't found the need for automated testing, imagine having an Angular application behind an Ajax login form. As on every test you will need to fill out the login form to test the functionality behind it, this will get very tedious later.

With the rise of open source software, the adoption of **test driven development** has taken flight. When you want to contribute to a project, most of the time you will be required to accompany your code with tests. Writing decent tests for your code involves testing the definition of the functionality. You basically write down your requirements in a special test language and your test is defined. A test could read something like the following:

```
// pseudo code
describe "home page"
  it "should show the 10 latest tweets"
   expect("tweets").count to_equal(10)
```

The output of this test reads almost like a story: "Home page should show the 10 latest tweets", and "expect the tweets count to equal 10;" as you can see, this natural language resembles the actual testing syntax.

One could even argue that if you'd write down your thoughts about how to construct a piece of code, you would have a large part of your tests already written. The thing that is missing for now is the negative testing. We are now expecting things to be there, but it gets more interesting when we start testing for edge negatives. A scenario that describes typing a name in an e-mail input field could look like the following:

```
// pseudo code
describe "login form"
   it "should require an email address"
     element('#submit').click()
     expect(input#email('name')).to_raise(validation_exception)
…
```

This will test if an exception was raised when the input was filled with some unacceptable value. This test implies a rendered form in a web browser, but it could just as easily have been testing a validation on a model class or a controller. Writing your tests before you actually start coding will make you think about your code in a way that you might not directly do so when you would have tested in the browser afterwards. Test-driven development is built on the premise that when you write your tests before you code, you will have thought about more edge cases and caught more anomalies. When the tests are written first, they will naturally fail. This is called the **red**. After you start coding to meet the requirements in your tests, they will start turning **green**. This change from red to green creates eagerness in the developer's mind, which is called **The Flow** in psychological terms.

Test your code, not the framework

This eagerness is also a danger, because testing is a means to an end, not a purpose! You should only test your code and not the tools you embedded. An example is a test that checks if a scope variable was correctly shown on the page using the curly braces {{ }}, shown as follows:

```
// pseudo code for our tempApp project
describe('tempApp', function() {

  it('should automatically redirect to /view1 when location hash/
fragment is empty', function() {
browser().navigateTo('../../app/index.html#/current');
    expect(element('[ng-view] p:first').text()).
        toMatch(/17/);
  });
```

This test checks if the Angular assignment `$scope.temp = 17;` was correctly bound to the {{ `temp` }} value in the template. Here, we are actually testing the Angular framework and not the code we wrote. When are interested in the state of the Angular tests, you should check out their Travis CI state at `https://travis-ci.org/angular/angular.js`.

Testing the parts

Angular projects can be tested with two types of tests. To test your entire application front to back, end-to-end tests should be used. To test the full stack of an Angular application, the best tool for the job would be to use **Protractor**. To test individual components, unit testing is the way to go. Before we get our hands dirty, we need to set up the testing environment. **Jasmine** will be used in this book to run AngularJS unit tests using a behavior-driven approach seen in many server-side testing frameworks. There are certainly other options out there, such as Mocha or QUnit, and they integrate quite nicely with AngularJS as well. The AngularJS integration of Jasmine solves dependency resolving quite nicely, whereas, others leave more to the developer. Jasmine is the default framework in the documentation and seems to be the most widely adopted option for AngularJS projects. This means that you can easily find help on the web for Jasmine. The **behavior-driven approach** means that the tests have some resemblance to human language. Have a look at the following sample taken from the Jasmine website:

```
describe("Adding function", function() {
  it("should add two arguments", function() {
    expect(add(1,2)).toBe(3);
  });
});
```

When the test is executed, it will read like human language: "Adding a function should add two arguments". The `describe` function is used to put a description on your group of tests, or assertions, as they are sometimes called. This string has no functionality. The `it()` function starts a new test, which is also named with a string. Be sure to group and name your tests in such a way that when you have a few hundred, they are still descriptive enough to know what they are about. The actual test is as follows:
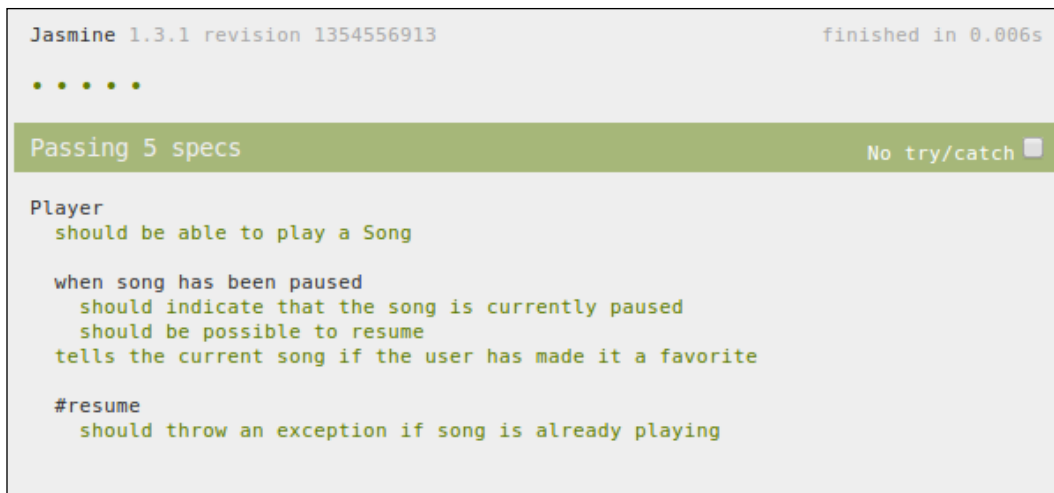
```
expect(add(1,2)).toBe(3);
```

We are comparing whether the function `add(a,b)` produces `3` when fed with `1` and `2` as arguments. This seems silly to test, and maybe it is, because you made the `add()` function to add numbers and return the result. But what about negative numbers, or what happens if I feed it a string or a binary number? The fun part about testing is that as you are writing your tests and thinking of ways to "break" your code, you are actually shaping the way your function should look.

Most developers enter the testing arena by creating some tests after the code has been created, and most of the time, has even been stabilized. When you take the step to start creating tests while you are contemplating your awesome new code, you will find that the code you produce will be better from the start and remain "watched" by the tests you created.

To get started with Jasmine is very easy. Just head over to `http://tryjasmine.com/` and try some tests to get used to the syntax. Another thing to familiarize yourself with is the matchers. **Matchers** take the output of the `expect()` function and compare it to something for success or failure.

After you have gotten familiar with the Jasmine syntax, head over to `https://github.com/pivotal/jasmine/downloads` and download the latest version. Inside the ZIP file, you will find the Jasmine files in the `/lib` directory. The most interesting file is `specRunner.html`, because this will actually run the tests. There are some example functions and accompanying tests to get you started. The source code is in `src` and the tests are in `spec`. When you extract this ZIP file somewhere in your local web root, you can navigate to the `specRunner.html` file and see the Jasmine tests in action. Your screen should look something like the following:

```
Jasmine 1.3.1 revision 1354556913                      finished in 0.006s

. . . . .

Passing 5 specs                                          No try/catch ☐

Player
  should be able to play a Song

  when song has been paused
    should indicate that the song is currently paused
    should be possible to resume
  tells the current song if the user has made it a favorite

  #resume
    should throw an exception if song is already playing
```

Before we look at the tests, let's examine the `specRunner.html` file to find out how Jasmine gets all the files together. The header of the file looks like the following:

```html
<title>Jasmine Spec Runner</title>


<link rel="shortcut icon" type="image/png" href="lib/jasmine-1.3.1/
jasmine_favicon.png">
<link rel="stylesheet" type="text/css" href="lib/jasmine-1.3.1/
jasmine.css">
<script type="text/javascript" src="lib/jasmine-1.3.1/jasmine.js"></
script>
<script type="text/javascript" src="lib/jasmine-1.3.1/jasmine-html.
js"></script>


<!-- include source files here... -->
<script type="text/javascript" src="src/Player.js"></script>
<script type="text/javascript" src="src/Song.js"></script>


<!-- include spec files here... -->
<script type="text/javascript" src="spec/SpecHelper.js"></script>
<script type="text/javascript" src="spec/PlayerSpec.js"></script>
```

After loading the Jasmine styles and libraries, the source files are loaded first and then the tests under `/spec`; although the order in which the files are loaded doesn't matter. By the way, the term "spec" comes from "behavior specification", which is what drives behavior-driven tests. The inline JavaScript in the following code snippet is the configuration of Jasmine and the reporter that is used to output the test results. The default is the HTML reporter inside the `jasmine-hmtl.js` file, but there are options available on the Web to produce output in other formats. At the bottom, the Jasmine suite is bootstrapped.

The following snippet from the `/spec/PlayerSpec.js` file shows us an actual behavior specification, so let's take a look:

```
describe("Player", function() {
  var player;
  var song;

  beforeEach(function() {
    player = new Player();
    song = new Song();
  });
…..
```

This file specifies the behavior of the `Player()` function (class). `BeforeEach` will be executed before every player test, defined by `it()`. So, every test will have an instantiated player and song available, shown as follows:

```
it("should be able to play a Song", function() {
    player.play(song);
    expect(player.currentlyPlayingSong).toEqual(song);

    //demonstrates use of custom matcher
    expect(player).toBePlaying(song);
});
```

The first test specifies the ability to play a song by expecting the player's current song to be the song that was asked to play. The last line in `spec` uses a custom matcher. There is no magic here. The `toBePlaying()` assertion has been defined in the `specHelper.js` file:

```
beforeEach(function() {
  this.addMatchers({
    toBePlaying: function(expectedSong) {
      var player = this.actual;
      return player.currentlyPlayingSong === expectedSong &&
             player.isPlaying;
```

```
    }
  });
});
```

As this `beforeEach` block has not been included in a `describe()` block, it will be executed before any Jasmine test. The value passed to the custom matcher is `this.actual`. An explanation of how to add a custom failure message can be found at `https://github.com/pivotal/jasmine/wiki/Matchers#writing-new-matchers`.

Now let's change this to test the code that we have made so far. We will start by configuring Jasmine correctly in the `SpecRunner.html` file. We will need to let Jasmine know about Angular and the code we have made:

```
// test/SpecRunner.html
…...
    <link rel="stylesheet" type="text/css" href="lib/jasmine-1.3.1/
jasmine.css">
    <script type="text/javascript" src="lib/jasmine-1.3.1/jasmine.
js"></script>
    <script type="text/javascript" src="lib/jasmine-1.3.1/jasmine-
html.js"></script>
    <!-- include angular specific files here... -->
    <script type="text/javascript" src="lib/angular.js"></script>
    <script type="text/javascript" src="lib/angular-mocks.js"></
script>

    <!-- include source files here... -->
    <script type="text/javascript" src="../app/js/app.js"></script>
    <script type="text/javascript" src="../app/js/controllers.js"></
script>
    <script type="text/javascript" src="../app/js/services.js"></
script>
    <script type="text/javascript" src="../app/js/directives.js"></
script>
    <script type="text/javascript" src="../app/js/filters.js"></
script>

    <!-- include spec files here... -->
    <script type="text/javascript" src="/test/spec/app/js/filterSpec.
js"></script>
…....
```

We started by adding a section for Angular that includes the `Angular.js` file. The next inclusion is `angular-mocks.js`. This file contains all kinds of functions that help us, or even better, help our testing framework test our Angular code after we simply include all our application code. Our first actual test specification is a file that doesn't exist yet — `filterCtrlSpec.js`. In this file we will test the filter we have written. We will create the file inside the `spec/app/js` directory and add the following content:

```
describe("filters", function () {
   beforeEach(
         module('tempApp')
   );

   describe("minimum filter", function() {
    it('should be available', function () {
      inject(function ($filter) {
        expect($filter('minimum')).toBeDefined();
      });
    });
   });
});
```

This will perform a simple test to see if the filter exists. Before we look into how the test is constructed, let's take a look at the output in the browser. When the `Node.js` web server is running, the output can be viewed at `http://localhost:8000/test/SpecRunner.html`.

```
Jasmine 1.3.1 revision 1354556913

   •

   Passing 1 spec

HistoryController
   minimum filter
      should be available
```

Every time you refresh the browser, the tests are run again and the result is shown. Now for the test code there are two `describe` blocks: the outer one tests filters and the inner one specifies the minimum filter. In the outer loop of the `beforeEach` block, the module for our application is loaded:

```
beforeEach(
    module('tempApp')
);
```

This happens before any of the inner blocks that follow inside the outer block. Like in our application, this exposes all filters, controllers, directives, and so on, inside this module. Inside the actual test, the `$filter` object is injected, which should contain all the filters of this module:

```
 it('should filter an array to be above a minimum value', function ()
{
    inject(function ($filter) {
        expect($filter('minimum')).toBeDefined();
    });
});
```

This simple filter just checks if the `minimum` filter is registered inside the module. Of course this is not a real test, because we are actually testing the Angular framework. Let's refactor and test some conditions to test our minimum filter:

```
it('should filter an array to be above or equal to the default 15',
function () {
        inject(function ($filter) {
            var unfiltered = [
            { "date": "2013-04-01T17:01:22.634Z", "temp": 8},
            { "date": "2013-04-02T17:01:22.634Z", "temp": 13},
            { "date": "2013-04-03T17:01:22.634Z", "temp": 15},
            { "date": "2013-04-04T17:01:22.634Z", "temp": 11},
            { "date": "2013-04-05T17:01:22.634Z", "temp": 15},
            { "date": "2013-04-06T17:01:22.634Z", "temp": 17},
            { "date": "2013-04-07T17:01:22.634Z", "temp": 21}
            ];
            var filtered = $filter('minimum')(unfiltered);
            console.log(filtered)
            expect(filtered.length).toBe(4);
        });
    });
```

This new test prepares an array with data from the `history.json` file that was used in the first chapter. It then runs the array through the filter and checks if the correct number of items are returned. A more thorough test would be to iterate over the resulting array and check if there really aren't any values below `15`. Feel free to play with this example a bit to get the hang of it. The important thing to note here is that we have isolated the filter from our template and controller and are able to test it with a predefined input value and test our expectations:

```
Jasmine 1.3.1 revision 1354556913

• •

Passing 2 specs

filters
  minimum filter
    should be available

  minimum filter
    should filter an array to be above the default 15
```

This is just a simple example of unit testing with Angular. Next, we will do something more advanced—test our `historyController`. The controller has one function and one property. To test these, our test could look something like the following:

```
beforeEach(function () {
    module('tempApp');
});

beforeEach(inject(function ($rootScope, $controller, currentUser) {
    scope = $rootScope.$new();
    $controller('HistoryCtrl', {
        $scope: scope
    });
}));

describe('tempMin', function () {
    it('should be defined', function () {
        expect(scope.tempMin).toBeDefined();
    });
})
```

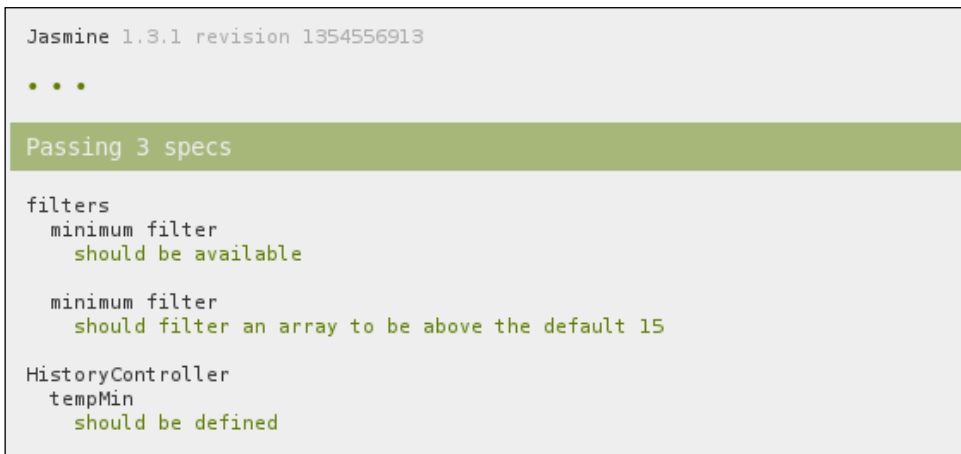This simple start of a test will fail, as shown in the following screenshot:

```
Jasmine 1.3.1 revision 1354556913                                    finished in 0.065s

• X X

Failing 2 specs                                                      No try/catch ☐

3 specs | 2 failing

HistoryController tempMin should be defined.

ReferenceError: Parse is not defined

ReferenceError: Parse is not defined                                              ▲
    at Object.<anonymous> (http://localhost:8000/app/js/services.js:4:3)
    at Object.invoke (http://localhost:8000/test/lib/angular.js:2864:28)
    at http://localhost:8000/test/lib/angular.js:2702:37
    at getService (http://localhost:8000/test/lib/angular.js:2824:39)
    at Object.invoke (http://localhost:8000/test/lib/angular.js:2842:13)
    at http://localhost:8000/test/lib/angular.js:2702:37
    at getService (http://localhost:8000/test/lib/angular.js:2824:39)
    at invoke (http://localhost:8000/test/lib/angular.js:2842:13)
    at Object.instantiate (http://localhost:8000/test/lib/angular.js:2874:23)
    at http://localhost:8000/test/lib/angular.js:4759:24                           ≡
Error: Declaration Location
    at window inject angular mock inject (http://localhost:8000/test/lib/angular_mocks_js:1744:25)  ▼

Expected undefined to be defined.
```

As seen before, we have loaded the module `tempApp`. After which we instantiated the controller, as stated in the documentation:

```
beforeEach(inject(function ($rootScope, $controller) {
    scope = $rootScope.$new();
    $controller('HistoryCtrl', {
        $scope: scope
    });
}));
```

This is where it went wrong! The controller not only depends on the `$scope` variable, but also on our `reading` service. Since the controller depends on it, we can feed the controller a special version of this `reading` service that will supply static data. The benefit is that the outcome can be predefined and we don't need an Internet connection to run the test. The test will also run a lot faster. We will update the controller test to instantiate a mock version of the `reading` service:

```
beforeEach(module(function ($provide) {
    $provide.service('reading', [ function (Project) {
        this.query = function () {
            return [
                { "date": "2013-04-01T17:01:22.634Z", "temp": 8},
                { "date": "2013-04-02T17:01:22.634Z", "temp": 13},
                { "date": "2013-04-03T17:01:22.634Z", "temp": 15},
                { "date": "2013-04-04T17:01:22.634Z", "temp": 11}
            ];
        };
    }]);
}));
```

The instantiation uses the `$provide` function of Angular and it just returns a static version of some history data. We have injected this dependency so our controller can function and the outcome is predictable. When we visit the `SpecRunner.html` file again, you will see that the test will now pass. The Jasmine output will now look something like the following:

```
Jasmine 1.3.1 revision 1354556913

• • •

Passing 3 specs

filters
  minimum filter
    should be available

  minimum filter
    should filter an array to be above the default 15

HistoryController
  tempMin
    should be defined
```

In these simple examples, the power of dependency injection really shows it's strength. We would have had a hard time testing the controller without it. If you want to see how the unit tests for other parts of the application are constructed, have a look in the Git repository for this book `<packpub git url>`.

The Karma test runner

The Angular team has made a test runner that will watch your files and re-run your tests when needed, so you don't have to refresh your browser to find out if your tests pass or not. It is a small inconvenience, but having it automated will lower the barrier to writing tests. The Karma test runner runs your unit tests inside real browsers and reports the results back to you. You can run several browsers in parallel and verify that your "Angularized" JavaScript code actually runs in those browsers. The setup is very easy when you have node, and thereby npm, installed. Just run `npm install -g karma` in a command prompt window.

After Karma is installed, we need to create a new file in the `config` directory for the unit tests. In the following code snippet you will find an example that will run our test suite:

```
// Karma configuration
// Generated on Fri Aug 16 2013 16:33:21 GMT+0200 (CEST)

module.exports = function (config) {
  config.set({

    // base path, that will be used to resolve files and exclude
    basePath: './',

    // frameworks to use
    frameworks: ['jasmine'],

    // list of files / patterns to load in the browser
    files: [
      '../test/lib/angular.js',
      '../test/lib/angular-mocks.js',

      '../app/js/*.js',
      '../test/spec/app/**/*.js'
    ],

    // list of files to exclude
```

```
exclude: [

],

// test results reporter to use
// possible values: 'dots', 'progress', 'junit', 'growl',
   'coverage'
reporters: ['progress'],


// web server port
port: 9876,


// enable / disable colors in the output (reporters and logs)
colors: true,


// level of logging
// possible values: config.LOG_DISABLE || config.LOG_ERROR ||
   config.LOG_WARN || config.LOG_INFO || config.LOG_DEBUG
logLevel: config.LOG_INFO,


// enable / disable watching file and executing tests whenever any
   file changes
autoWatch: true,


// Start these browsers, currently available:
// - Chrome
// - ChromeCanary
// - Firefox
// - Opera
// - Safari (only Mac)
// - PhantomJS
// - IE (only Windows)
browsers: ['Chrome'],


// If browser does not capture in given timeout [ms], kill it
captureTimeout: 60000,


// Continuous Integration mode
// if true, it capture browsers, run tests and exit
```

```
        singleRun: false,

        preprocessors: {

        }

    });
};
```

The config will load the same files as the Jasmine `SpecRunner.html` file we created before. There are some additional configuration parameters for the Karma test runner that are explained in the comments of the preceding code snippet. Additional information on the configuration can be found on the projects homepage at `http://karma-runner.github.io`.

The second thing to do is create a script to run Karma with this configuration. This script will go in the `/scripts` folder and looks like the following:

```
#!/bin/bash

BASE_DIR=`dirname $0`

echo ""
echo "Starting Testacular Server (https://github.com/karma-runner/
karma)"
echo "-----------------------------------------------------------------
----"

karma start $BASE_DIR/../config/karma.conf.js $*
```

It just prints a startup message and then starts Karma with our configuration file. On Windows, this file just needs to be saved with the `.bat` extension and contain the following:
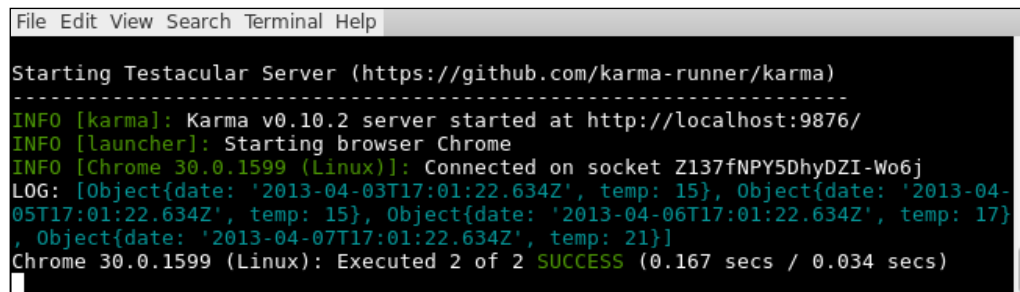
```
@ECHO OFF

ECHO
ECHO Starting Testacular Server (https://github.com/karma-runner/
karma)
ECHO ----------------------------------------------------------------
---

node_modules/.bin/testacular start test/karma.conf.js
```

When we fire up Karma by executing this script, the output should look like the following:

```
File Edit View Search Terminal Help

Starting Testacular Server (https://github.com/karma-runner/karma)
--------------------------------------------------------------
INFO [karma]: Karma v0.10.2 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 30.0.1599 (Linux)]: Connected on socket Z137fNPY5DhyDZI-Wo6j
LOG: [Object{date: '2013-04-03T17:01:22.634Z', temp: 15}, Object{date: '2013-04-
05T17:01:22.634Z', temp: 15}, Object{date: '2013-04-06T17:01:22.634Z', temp: 17}
, Object{date: '2013-04-07T17:01:22.634Z', temp: 21}]
Chrome 30.0.1599 (Linux): Executed 2 of 2 SUCCESS (0.167 secs / 0.034 secs)
```

As long as you keep this window open, Karma will keep watching your files and rerunning all the unit tests. Notice the time it took to run the tests? The entire test suite for the Angular framework itself takes less than a second to run. To keep Karma running while you develop is very useful, because it will give you almost instant feedback!

End-to-end testing

End-to-end testing should actually be the start of your test-driven development cycle when you adhere to the BDD standard. While **behavior-driven development** (**BDD**) is much more than just testing, one of the pillars is that you start by defining a test for a behavior at the highest possible level. This test will obviously fail if the parts underneath aren't ready yet. It will allow a well-defined behavior and all the different parts that will power this feature will get their own special unit test. We have seen how these unit tests are constructed and how dependency injection plays a role in there. After all the underlying unit tests are made to pass, the end-to-end test should pass as well. In the end, the end-to-end tests should test the entire stack including all the code tested by the unit tests.

The following is a quote from an article by Dan North on BDD: `http://dannorth.net/introducing-bdd/`:

> "*At first, the fragments are implemented using mocks to set an account to be in credit or a card to be valid. These form the starting points for implementing behavior. As you implement the application, the givens and outcomes are changed to use the actual classes you have implemented, so that by the time the scenario is completed, they have become proper end-to-end functional tests.*"

The end-to-end tests require us to have a running application server because this type of testing will act like a "real user," clicking and interacting with your application in other ways. We need to have a way to navigate pages and simulate this user interaction. The tool for this job is Protractor—a brand new testing framework based on Selenium and WebDriver. Here we will explain how to use some of the specific additions that Protractor adds to the WebDriver syntax.

Setting up the Protractor

This setup is similar to the one we did for Karma; it also needs a configuration file and a startup script. However, the first thing to do is install Protractor with `npm install protractor -g`. This will make Protractor available for all of your projects. The following is the content for those two files to get started:

```
// config/selenium-conf.js
// protractor configuration file.
// fully annotated config file can be found here:
// https://github.com/angular/protractor/blob/master/referenceConf.js
exports.config = {
   // The address of a running selenium server. If this is specified,
   // seleniumServerJar and seleniumPort will be ignored.
   seleniumAddress: 'http://localhost:4444/wd/hub',

   // A base URL for your application under test. Calls to protractor.
get()
   // with relative paths will be prepended with this.
   baseUrl: 'http://localhost:8000',

   // Capabilities to be passed to the webdriver instance.
   capabilities: {
      'browserName': 'firefox'
   },

   // Spec patterns are relative to the current working directly when
   // protractor is called.
   specs: ['../test/e2e/**/*.js'],

   // Options to be passed to Jasmine-node.
   jasmineNodeOpts: {
      isVerbose: true,
      showColors: true,
      includeStackTrace: true
   }
};
// scripts/test-e2e.sh
#!/bin/bash
```

```
BASE_DIR=`dirname $0`

echo ""
echo "Starting Protractor Server (https://github.com/angular/
protractor)"
echo "-------------------------------------------------------------
----"

protractor config/selenium-conf.js
```

Since Protractor runs on top of Selenium, there are many different setups and options available that go too far for our book. This setup will read end-to-end tests from the `/test/e2e/` folder and execute them inside a Firefox browser. The tests can be started by simply executing `/scripts/test-e2e.sh`.

The difference is that this time as well the application server has to be started along with the Selenium server. Starting the server is, the same as before, to be able to access the application in a browser:

`node scripts/web-server.js`

Installing Selenium is as easy as downloading the appropriate JAR files from `https://code.google.com/p/selenium/downloads/list`. Since it is a Java executable, you do need to have Java installed on your machine. Most Mac and Linux systems will have it by default, but any platform can be outfitted with Java by visiting `http://www.java.com/en/download/index.jsp`.

Once all is in place, execute the JAR file by typing `java -jar <path to selenium server file>`. This will start the Selenium server, which can be verified by opening this `http://localhost:4444/wd/hub` link in your browser. You should see something like the following if all is well:

```
┌─ Sessions ─────────────────────────────────────────────────────┐
│  [ Create Session ]  |  [ Refresh Sessions ]                    │
│  ┌───────────────────────────────────────────────────────────┐ │
│  │ No Sessions                                                 │ │
│  │                                                             │ │
│  │                                                             │ │
│  │                                                             │ │
│  └───────────────────────────────────────────────────────────┘ │
│                         Linux 3.10-2-amd64  |  v2.35.0  |  rc916b9d │
└─────────────────────────────────────────────────────────────────┘
```

Now that we are all set up, we can start coding our first test. This will be done in a new file in the `test/e2e` folder:

```
// /test/e2e/app.js
'use strict';

describe('tempApp', function() {
  var ptor;

  describe('current page', function() {
    ptor = protractor.getInstance();

    beforeEach(function() {
      ptor.get('/index.html');
    });

    it('should find an element by binding', function() {
      var binding = ptor.findElement(protractor.By.binding('{{ temp
}}'));
      expect(binding.getText()).toContain(17);
    });
  });
});
```
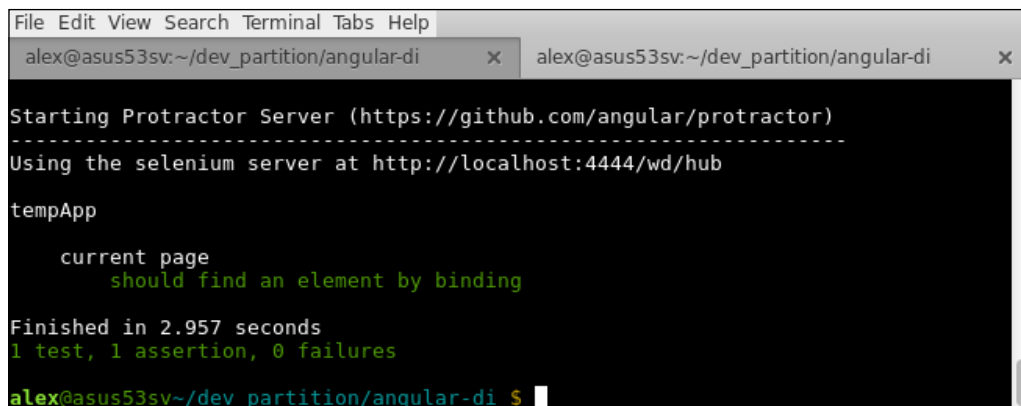
The syntax is similar to the unit tests we created with Jasmine, so it follows the same patterns. First, describe what you are testing and then write your test in the `it()` function. This test just tests to see if `17` is present in the text node that contains the `{{ temp }}` binding. If you started both, the NodeJS server and the Selenium server, you can run the tests. The output of this test should be similar to the following:

```
File Edit View Search Terminal Tabs Help
alex@asus53sv:~/dev_partition/angular-di      ×    alex@asus53sv:~/dev_partition/angular-di      ×

Starting Protractor Server (https://github.com/angular/protractor)
----------------------------------------------------------------
Using the selenium server at http://localhost:4444/wd/hub

tempApp

    current page
        should find an element by binding

Finished in 2.957 seconds
1 test, 1 assertion, 0 failures

alex@asus53sv~/dev_partition/angular-di $ ▮
```

Even the output looks like the Karma output! More importantly, the browser opened your application and "saw" in the page that string 17 was present.

In the next test we will try to have our test runner change the input box to something else and then test the outcome. The second test is appended to the test file within the current `describe` page function:

```
it('should reflect a change in the input field', function () {
    var binding = ptor.findElement(protractor.By.binding('{{ temp
}}'));
    expect(binding.getText()).toContain(17);

    ptor.findElement(protractor.By.tagName('input')).clear();
    ptor.findElement(protractor.By.tagName('input')).sendKeys('20');

    var binding = ptor.findElement(protractor.By.binding('{{ temp
}}'));
    expect(binding.getText()).toContain(20);
})
```

First we check if `{{ temp }}` contains 17 before we change it. Then the `temp` input is cleared and filled with the new value 20. Afterwards, there is a check to see if the value has changed to 20. When you run the test, you will see all of this happening on the screen. Also, you will notice that these tests take a lot longer than the unit tests. A few simple tests only take a few seconds on a fast system to complete. That's one of the reasons you want to run these tests manually, unlike the Karma unit tests. Ideally, you'd run these and the unit tests on different browsers on a continuous integration system. This is, however, outside the scope of this book.

> To get the most out of Angular e2e testing with Protractor, you are strongly encouraged to refer to `https://github.com/angular/protractor` and all the linked documentation in that page. At the time of writing this book, Protractor is still very much in development, but the Angular team has already adopted the framework.

Summary

In this chapter we have covered some of the reasons behind testing after which we dove right into the testing, starting with unit tests in Jasmine, and then automating them so they are re-run after each file change. Lastly we introduced end-to-end testing with the Angular-specific Protractor test framework.

5

Large Applications

Writing a large, single-page application using traditional methods that are based on DOM manipulation has turned out to be difficult to manage and maintain. As we have seen in this book, Angular brings a new paradigm to the table for creating more maintainable applications by using dependency injection. Even when using Angular, it is still entirely possible to create unmaintainable and untestable code. In this chapter, we will look at the patterns that can be used with Angular to further ease your work as a developer.

Organizing your application

Most newcomers to Angular start off by cloning the Angular seed project on GitHub. This roughly resembles the setup that you have seen in this book so far `https://github.com/angular/angular-seed.git`.

This basic project contains most of the elements that we have discussed so far in this book. There is some sample code that uses routes, controllers, service, directive, and a filter. All of the code has accompanying tests, and testing has been set up with Karma and Jasmine. Although the Angular team themselves have titled the repository as a basic skeleton, many people find the setup sufficient for sizable applications that have lengths of up to several hundred lines of code. The project layout on the GitHub page explains the different parts quite well:

```
app/                 --> all of the files to be used in production
  css/               --> css files
    app.css          --> default stylesheet
  img/               --> image files
  index.html         --> app layout file (the main html template file
of the app)
  index-async.html   --> just like index.html, but loads js files
asynchronously
  js/                --> javascript files
```

```
     app.js            --> application
     controllers.js  --> application controllers
     directives.js   --> application directives
     filters.js      --> custom angular filters
     services.js     --> custom angular services
   lib/              --> angular and 3rd party javascript libraries
     angular/
       angular.js        --> the latest angular js
       angular.min.js    --> the latest minified angular js
       angular-*.js      --> angular add-on modules
       version.txt       --> version number
   partials/           --> angular view partials (partial html
templates)
     partial1.html
     partial2.html

config/karma.conf.js        --> config file for running unit tests
with Karma
config/karma-e2e.conf.js    --> config file for running e2e tests with
Karma

scripts/            --> handy shell/js/ruby scripts
  e2e-test.sh       --> runs end-to-end tests with Karma (*nix)
  e2e-test.bat      --> runs end-to-end tests with Karma (windows)
  test.bat          --> autotests unit tests with Karma (windows)
  test.sh           --> autotests unit tests with Karma (*nix)
  web-server.js     --> simple development webserver based on node.js

test/               --> test source files and libraries
  e2e/              -->
    runner.html     --> end-to-end test runner (open in your browser
to run)
    scenarios.js    --> end-to-end specs
  lib/
    angular/              --> angular testing libraries
      angular-mocks.js    --> mocks that replace certain angular
services in tests
      angular-scenario.js --> angular's scenario (end-to-end) test
runner library
      version.txt         --> version file
  unit/                   --> unit level specs/tests
    controllersSpec.js    --> specs for controllers
    directivessSpec.js    --> specs for directives
    filtersSpec.js        --> specs for filters
    servicesSpec.js       --> specs for services
```

Thanks to *Igor Minar* for the annotated directory structure. This being a "seed" application, it's tailored for simple projects. This setup puts all the Angular application code in single files, which will work fine for small to medium sized projects. The repository comes with a complete test setup with end-to-end unit tests to get you started. The `scripts` directory houses a simple NodeJS server script next to the test startup scripts. Although, strictly speaking, you don't need a server to serve the static files, this makes it really easy to get started—without any limitations some browsers put on running scripts from the local filesystem.

This setup has its limitations. When your app grows, the number of controllers, directives, filters, and services will probably grow too. This implies that you'd have to search through large files for your code. When you get to the point where you have to use your editor's search function to find a specific piece of code in a file, it's time to reorganize.

Going a bit larger

The next step is to organize your application by separating all similar code. For example, all controllers can be separated into single files under a specific directory. It would look something like the following:

```
app/                  --> all of the files to be used in production
  …...
  js/                 --> javascript files
    app.js            --> application
    controllers/
   main.js  --> main application controller
   sub1.js  --> another application controller
    directives/
   chart.js   --> the chart directive
   another.js   --> another directive
    filters/
   translate.js      --> a translation filter filters
    services/
   parse.js      --> parse data backend service
    …..
```

Naturally, this setup will have to be mirrored in the test directory, so that things are easily locatable. It should serve you quite well until the number of files becomes so big that you can't find what you're looking for anymore. In a well-built application, you will see many directives. Grouping related functionality into a directory may help organize things a little more.

> **The changing filesystem**
>
> You would have to manually change files that are included in the main HTML file to stay in sync with your filesystem changes. When your JavaScript application is served from within a framework such as Ruby on Rails, Symfony (PHP), or NodeJS using Brunch.io, organizing your app is simplified considerably. They, and others, provide tools to automatically select, compile, and minify your files and more. This means that you only configure this once and changes in your files are noticed, and the files served to the browser are updated.
>
> Check out the following sites for reference: `http://rubyonrails.org/`, `http://brunch.io/`, and `http://symfony.com/`.

Organizing using dynamic teams

Organizing your code according to the type hierarchy doesn't help new developers find the code related to a certain function or application area. A new developer or a reviewer would have to look through all four top level `js/` directories to find what they are looking for.

So, when the project gets to a level where the team has to grow and will most likely have a more dynamic nature, organizing according to functional area is a good alternative. This is also where the Angular module system comes into play.

Using modules

Angular modules are a way to organize your code in a functional manner. Modules can contain any kind of Angular code. They are injected into each other to make the Angular code available. It makes sense to do this in a functional manner because a group of functionalities such as charting functions is not needed in areas where they are not shown.

In the filesystem, the result of our functional approach could look like the following:

```
App/                        →  all of the files to be used in production
  …...
  js/                       →  javascript files
    app.js                  →  application
    common/                 → common module
  filters/
      translation.js        → translation filter
  directives/
```

```
        checkbox.js          → checkbox directive
      services/
        logging.js           → central application logging service
        parse.js             → parse data backend service
        config.js            → central configuration service
      admin/                 → admin module
    controller.js            →  the main admin controller
    sidebar/
        controller.js        → sidebars controller
        tree.js              →  tree directive specific for sidebar
      main/                  → main module
    controller.js            →  the main user facing application
  controller
      directives/
        chart.js             → a directive for a specific chart widget
        twitter.js           → a directive for a twitter widget


      …..
```

Then, the various modules can be easily injected in the following manner:

```
var App = angular.module('App', ['commonModule', 'adminModule',
'mainModule'])
```

There are different ways to bundle functionality into modules, but it makes sense to have submodules independent of any other modules, so that they can be tested and shared more easily.

Testing with module separation becomes easy because you only need to bootstrap the module you are testing. This means you are sure there are no hidden dependencies, such as some property attached to the application global variable. For instance, testing your chart directive in the previous example would require you to just write the following:

```
'use strict';

describe('chart', function () {
  var scope;

  beforeEach(module('mainModule'));

  describe("chart", function () {
   it(….
 }
….
```

Organizing using directives

Directives have been mentioned a few times already in this chapter. So far, we have used them to encapsulate JavaScript code which is not Angular aware, such as the third-party charting library from the previous chapters. Another way to use directives is to abstract pieces of functionality, so that their complexity is encapsulated inside the directive. Even when we don't need the possibilities that are not available outside a directive, such as DOM manipulation, code clarity and legibility can be served by this kind of encapsulation. Directives can only be "used" and not injected, although the module they belong to has to be injected.

Nesting controllers

An interesting topic, which is not covered in the documentation, is **nested modules**. However, they offer a great way to organize code. Besides that, they also fit well with the functional organization because all code is grouped in a layered structure like the parts that the user sees on the screen. Nested controllers can be used at any level of an Angular application and all the properties and methods of their parents are available.

> Nested controllers copies the scope variables in the inheritance chain. Updating a parent property in a child controller affects only its own child level and the lower levels. If you want to update the property at the top level, use a setter. Using the `$scope.$parent` notation to update the property will work, but it makes unit testing the subcontroller problematic.

With this setup, a page that is navigated using an Angular route will have one main controller and several subcontrollers, which in turn can have more sub controllers. The file structure could look something like the following:

```
…....
main/            → main module
    controller.js            → the main user facing application
controller
    toptoolbar/
        controller.js        → controls the top toolbar
    leftsidebar/
        controller.js        → controller for entire sidebar
        navigator/
            controller.js    → controller for the navigator
        properties/
            controller.js    → controller for the properties
….
```

When testing nested controllers, it is very easy to stub scope variables of the parent controllers so that they are available inside the sub controller. When you used a setter to set a parent scope variable, it should also be stubbed in the test:

```
describe('nested controller', function() {

    var $scope, ctrl;
    it('should have a value', inject(function($rootScope, $controller)
{

        $rootScope.myValue = 1;

        $rootScope.setMyValue = function(val) {
        $rootScope.myValue = val;
        }

        $scope = $rootScope.$new();

        ctrl = $controller('nestedController', {
            $scope: $scope
        });

        expect($scope.myValue).toEqual(1);
    }));
});
```

More powerful nesting

While the previous approach with nested controllers is useful, it does have its limitations. For instance, there is no natural way to open a page in your routing in a certain state; we could have a list view in our main controller and a detail view based on the user interaction. In the default AngularJS routing setup, you would have to use query parameters to achieve this and then still re-render the client-side view. The UI-router project from the Angular-UI team has a great solution for this at `https://github.com/angular-ui/ui-router`.

The following is taken from the `readme.md` file:

Main Goal

To evolve the concept of an AngularJS route into a more general concept of a state for managing complex application UI states.

We won't even try to replace their excellent documentation, but it centers around state instead of URL and makes it possible to access pages *in a certain state* using a URL. So, the state of a page becomes a part of the routing possibilities, and as such, will allow you to step over the limitation of the default routing system. So, hop on over to the GitHub page and have a look.

Application communication

Even though we have options to organize our application in different ways, the parts still need to communicate with each other. Imagine a protected part of your application has a feature implemented that a session times out when there is no user interaction. This requires the user to log in again.

Events

The traditional way of handling this is to set up events and handlers to take of this. This approach is very acceptable and can also be implemented in Angular. Angular supports scope events out of the box using the `scope.$broadcast`, `scope.$emit` and `scope.$on` methods. `$broadcast` is used to send an event down the scope, and it is mostly used from the rootScope to notify listeners. `$emit` sends events upwards so it is used to notify the rootScope. A listener is defined with `$on` on any event.

In the example, the expired session would send an event up the scope chain and the rootScope would catch that and initiate route change to the login page. This is a familiar way for many JavaScript applications that do not use Angular.

Let the model speak

With Angular following a model-driven approach, there is an alternative. For example, in a situation where a dataset needs to be rendered in a specific way based on its content. An example of this is a model describing a DOM element that's being moved by the user by dragging inside a workspace:

```
//pseudo code
{
  id: "box1",
  style: {
    left: 10px,
    top:10px
  }
}
```

In Angular, it makes sense to capture the dragging events and update the position in the model so the renderer can update its position on the screen. When the element is moving over the screen, we might need to show the actual coordinates of the object, the `left` and `top` CSS properties, in the sidebar. This can then easily be done by binding scope values to the `top` and `left` properties of the model and showing these in the view template. This approach requires that the model should be separated from the original controller and should be encapsulated inside a "model service." This is done so that it can be injected in both the workspace and the sidebar. As the model is updated by the JS drag events, the dependent scope variables will change along with the models' values.

In contrast, we could have just updated the DOM to reflect the position changes, but that would have left us unable to share or store the data in a consistent manner.

Summary

This chapter discussed the different ways to organize larger Angular applications, from the Angular Seed repository, to more complex projects layouts. Also, the way different parts of your application communicate was covered. This can be done using inheritance, events, or simply by using the model. After reading this chapter, you now have a larger set of tools to organize your application in a more structured way.

Index

Thank you for buying
Dependency Injection with AngularJS

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
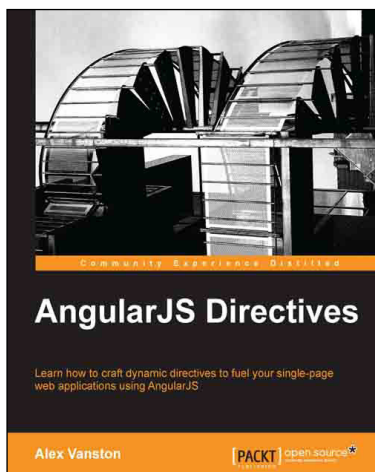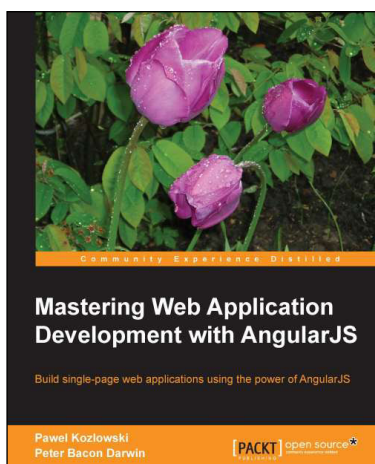
Angular JS Directives

ISBN: 978-1-78328-033-9   Paperback: 110 pages

Learn how to craft dynamics directives to fuel your single-page web applications using AngularJS

1. Learn how to build an AngularJS directive

2. Create extendable modules for plug-and-play usability

3. Build apps that react in real-time to changes in your data model

Mastering Web Application Development with AngularJS

ISBN: 978-1-78216-182-0   Paperback: 372 pages

Build single-page web applications using the power of AngularJS

1. Make the most out of AngularJS by understanding the AngularJS philosophy and applying it to real life development tasks

2. Effectively structure, write, test, and finally deploy your application

3. Add security and optimization features to your AngularJS applications

4. Harness the full power of AngularJS by creating your own directives

Please check **www.PacktPub.com** for information on our titles
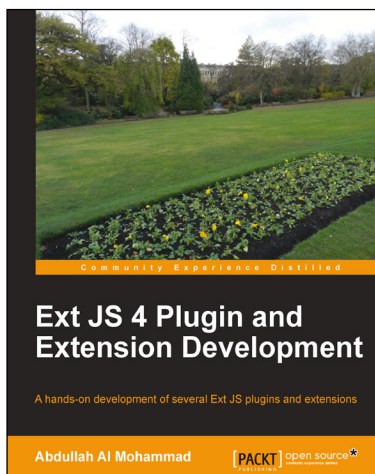
Ext JS 4 Plugin and Extension Development

ISBN: 978-1-78216-372-5   Paperback: 116 pages

A hands-on development of several Ext JS plugins
and extensions

1. Easy-to-follow examples on ExtJS plugins
   and extensions

2. Step-by-step instructions on developing ExtJS
   plugins and extensions

3. Provides a walkthrough of several useful ExtJS
   libraries and communities
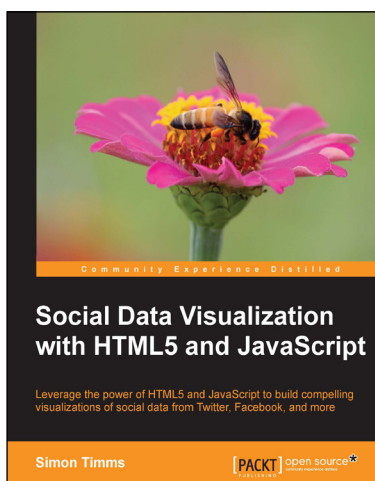
Social Data Visualization with HTML5 and JavaScript

ISBN: 9781782166542   Paperback: 104 pages

Leverage the power of HTML5 and JavaScript to
build compelling visualizations of social data from
Twitter, Facebook, and more

1. Learn how to use JavaScript to create
   compelling visualizations of social data

2. Use the d3 library to create impressive SVGs

3. Master OAuth and how to authenticate with
   social media sites

Please check **www.PacktPub.com** for information on our titles