Community Experience Distilled

Learning Web Development with Bootstrap and AngularJS

Build your own web app with Bootstrap and AngularJS, utilizing the latest web technologies

Stephen Radford

[PACKT] PUBLISHING open source community experience distilled

Learning Web Development with Bootstrap and AngularJS

Build your own web app with Bootstrap and AngularJS, utilizing the latest web technologies

Stephen Radford

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Learning Web Development with Bootstrap and AngularJS

First published: May 2015

Production reference: 1260515

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-755-0

www.packtpub.com

Credits

Author
Stephen Radford

Reviewers
Tasos Bekos
Jack Hsu
Ole B. Michelsen
Jurgen Van de Moere

Commissioning Editor
Rebecca Youé

Acquisition Editor
Rebecca Youé

Content Development Editor
Manasi Pandire

Technical Editor
Namrata Patil

Copy Editors
Puja Lalwani
Laxmi Subramanian

Project Coordinator
Shipra Chawhan

Proofreaders
Stephen Copestake
Safis Editing

Indexer
Tejal Soni

Graphics
Jason Monteiro

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Author

Stephen Radford is a full-stack web developer based in the heart of Leicester, England. Originally from Bristol, Stephen moved to Leicester after studying graphic design in college to accept a job at one of UK's largest online marketing companies.

While working at a number of agencies, Stephen developed several side projects, including FTPloy, a SaaS designed to make continuous deployment available to everyone. The project was subsequently a finalist in the .Net Awards Side Project of the Year category.

He and his business partner now run Cocoon, a web development company that builds and maintains web apps such as FTPloy and Former. Cocoon also works closely with a handful of startups and businesses to develop ideas into websites and apps.

I'd like to thank a few people who supported me during the writing of this book. First of all, my partner, Declan. He's been incredibly supportive and I couldn't ask for anyone better in my life. Paul Mckay was the first person I showed the book to and he even helped me write my bio, because for some reason I find writing about myself extremely difficult. And of course, I'd like to thank my parents. My dad has been patiently awaiting his print copy of the book, so hopefully, it's now pride of place on their coffee table.

About the Reviewers

Tasos Bekos is a software engineer and has been working with web technologies for over a decade. He has worked as a freelance developer and consultant for some of the biggest international telecom and financial companies. He is currently working as a frontend architect for ZuluTrade, where he makes use of the latest in frontend technologies. He has deep knowledge of AngularJS and is active in the open source community as a major contributor to the AngularUI Bootstrap project. When not coding, Tasos spends time playing with his two sons.

Jack Hsu is a web developer specializing in frontend tools and technologies. He is the lead frontend developer at Nulogy, bringing his JavaScript and AngularJS expertise to the team. Prior to joining Nulogy, he worked at a variety of companies, including The Globe & Mail, Ontario Institute of Cancer Research, and Wave Accounting. During his spare time, Jack can be found playing video games, experiencing the diverse neighborhoods of Toronto, or travelling the world. You can find an assortment of programming-related posts on his personal blog.

Ole B. Michelsen has been working with full-stack web development for more than 12 years, and has completed his BSc in computer science from DIKU, University of Copenhagen. In recent years, he has specialized in frontend JavaScript development, with particular emphasis on WebRTC and single-page app frameworks.

Jurgen Van de Moere was born in 1978, grew up in Evergem, Belgium, with his parents, sister, and pets. At the age of 6, he started helping his dad, who owned a computer shop, with assembling computers for clients. While his friends were playing computer games, Jurgen was rather fascinated by writing custom scripts and programs to solve problems that his dad's clients were dealing with. After graduating in Latin-Mathematics from Sint-Lievens college in Ghent, Jurgen continued his education at University of Ghent, where he studied computer science. His Unix username at the university was "jvandemo," the nickname he still goes by on the Internet today. In 1999, Jurgen started his professional career at Infoworld. After years of hard work and dedication as a developer and network engineer, he was awarded different management positions in 2005 and 2006. Being a true developer at heart, he missed writing code, and in 2007, he decided to end his management career to pursue his true passion again—development. Since then, he has been studying and working from his home office in Belgium, where he currently lives with his girlfriend, son, and dogs. In a rapidly evolving world of data-intensive, real-time applications, he now focuses on JavaScript-related technologies with a heavy emphasis on AngularJS and Node.js. His many private and public contributions have helped form the foundation of numerous successful projects around the world. If you need help with your project, Jurgen can be reached at `hire@jvandemo.com`. You can follow him on Twitter at `@jvandemo`. You can go through his blog at `http://www.jvandemo.com`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a web browser

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 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

I've worked on projects of various sizes over the course of my career, ranging from small brochure sites to building entire social networks. One thing in common with all of them was the need for well-structured JavaScript and CSS.

This book covers two fantastic open source projects that stemmed from this need—Bootstrap and AngularJS.

What this book covers

Chapter 1, *Hello, {{name}}*, looks at the basics of AngularJS and Bootstrap whilst building a simple "Hello, World" app.

Chapter 2, *Let's Build with AngularJS and Bootstrap*, introduces to the main app we'll be building over the course of the book, a look at Bootstrap's grid system, and some of the components that make up AngularJS.

Chapter 3, *Filters*, takes a look at some of AngularJS's built-in filters and also build our own.

Chapter 4, *Routing*, uses AngularJS's built-in router, and we'll learn how to utilize partials to create a multiview web app.

Chapter 5, *Building Views*, covers Bootstrap's grid system, and we'll flesh out the partials.

Chapter 6, *CRUD*, shows that our views are in place we can implement create, read, update, and delete functions.

Chapter 7, *AngularStrap*, covers the third-party module, which will allow us to use all of Bootstrap's plugins via AngularJS.

Chapter 8, *Connecting to the Server*, looks at the two official ways of connecting to a server.

Chapter 9, *Using Task Runners*, minifies all of our JS and Less files using Grunt and gulp.

Chapter 10, *Customizing Bootstrap*, allows you to easily customize Bootstrap now that Grunt.js is setup.

Chapter 11, *Validation*, includes validation out of the box; we'll implement that and manage server errors.

Chapter 12, *Community Tools*, takes a look at some of the tools built by the AngularJS community.

Appendix A, *People and Projects to Watch*, covers some key people in the AngularJS and Bootstrap worlds as well as the projects to watch.

Appendix B, *Where to Go for Help*, provides answers to the questions you might have.

Appendix C, *Self-test Answers*, provides all the answers enlisted in the self-test questions sections of the book.

What you need for this book

AngularJS and Bootstrap have no dependencies at all, so you will not need a lot for this book. Really, all you need is a web browser and a text editor. I recommend you use Chrome and Atom.

Who this book is for

If you're interested in modern web development at all, you'll no doubt have come across Bootstrap and AngularJS. This book is aimed at people with a little bit of JavaScript experience who want to dive head first into building web apps.

However, one thing that's definitely required is an understanding of JavaScript. If you're not sure what the difference is between a string and an object, you'll need to pick that up beforehand.

Of course, if you've used AngularJS or Bootstrap earlier, and want to learn more, then you'll feel right at home here.

Conventions

In this book, you will find a number of text styles 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: "Pop this script tag with in the `<head>` of your page."

A block of code is set as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title></title>
</head>
<body>

</body>
</html>
```

Any command-line input or output is written as follows:

```
open -a 'Google Chrome' --args -allow-file-access-from-files
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "We're going to need to display all the same information we entered in the **Add Contact** view as well as our Gravatar."

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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in 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 at `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 from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. 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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Hello, {{name}}

The best way to learn code is to write code, so we're going to jump right in. To show you just how easy it is to get up and running with Bootstrap and AngularJS, we're going to make a super simple application that will allow us to enter a name and have it displayed on the page in real time. It's going to demonstrate the power of Angular's two-way data binding and the included templating language. We'll use Bootstrap to give the app a bit of style and structure.

Before we install our frameworks, we need to create our folder structure and the `index.html` file that will be the basis of our web app.

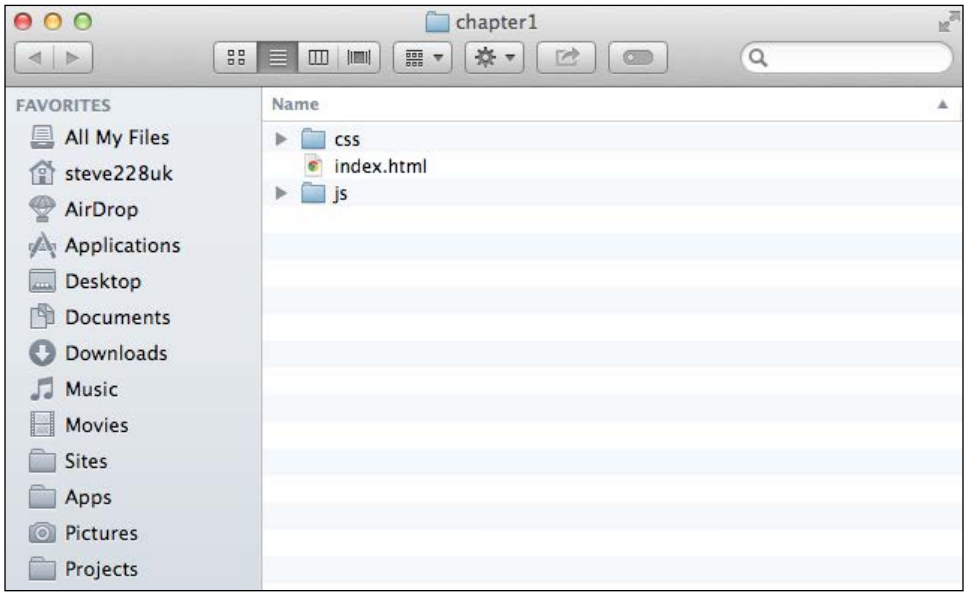
Setting up

In order to create our Angular and Bootstrap application, we need to do a little bit of setting up, which just involves creating an HTML page and including a few files. First, create a new directory called `chapter1` and open it up in your editor. Create a new file called `index.html` directly inside it and pop in this boilerplate code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title></title>
</head>
<body>

</body>
</html>
```

This is just a standard HTML page that we can do something with once we've included Angular and Bootstrap.

Now, create a couple of folders inside `chapter1` folder: `css` and `js`. Your completed folder structure should look something like this:



Installing AngularJS and Bootstrap

Installing both of our frameworks is as simple as including a CSS or JavaScript file on our page. We can do this via a **content delivery network** (**CDN**) like Google Code or MaxCDN, but we're going to fetch the files manually for now. Let's take a look at what steps you should be aware of when including AngularJS and Bootstrap in your project.

Installing Bootstrap

Head to `http://getbootstrap.com` and hit the **Download Bootstrap** button. This will give you a zip of the latest version of Bootstrap that includes CSS, fonts, and JavaScript files. Previous versions of Bootstrap included an images directory but Version 3 brings the change to icon fonts.

For our app, we're only interested in one file at the moment: `bootstrap.min.css` from the `css` directory. The stylesheet provides the entire structure and all of the lovely elements, such as buttons and alerts, that Bootstrap is famous for. Copy it over to your project's `css` directory and open up the `index.html` file in your text editor.

Including Bootstrap is as easy as linking that CSS file we just copied over. You just need to add the following within your `<head>` tag. Pop this script tag within the `<head>` of your page:

```
<link rel="stylesheet" href="css/bootstrap.min.css">
```

Installing AngularJS

Okay, now that we've got Bootstrap included in our web app, we need to install Angular. Visit `https://angularjs.org/` and click on the **Download** button. You'll be presented with a few options; we want the minified stable version.

Copy the downloaded file over to your project's `js` directory and open up your `index.html` file. Angular can be included in your app just like any other JavaScript file.

It's recommended that Angular is included in the `<head>` tag of your page or certain functions we'll be taking advantage of throughout the course of the book won't work. While it's not necessary, there will be extra steps you'll need to take if you choose to load Angular further down your HTML file.

Pop this `<script>` tag within the `<head>` of your page.

```
<script src="js/angular.min.js"></script>
```

Ready to go? Well, almost. We need to tell Angular that we want to utilize it in our app. Angular calls this bootstrapping and the framework makes this extremely simple for us. All we need to do is include an additional attribute in our opening `<html>` tag:

```
<html lang="en" ng-app>
```

That's it! Angular now knows we want to take advantage of it.

Angular also allows us to prefix these attributes with `data-` (for example, `data-ng-app`) should we be concerned about writing valid HTML5.

Using AngularJS

So we've got a lot of the theory behind Angular down; it's time to actually put it in place. Once we've got our application working, we'll take a look at how we can make it shine with Bootstrap.

Let's open that `index.html` file again, but this time also open it up in your browser so we can see what we're working with. This is what we've got so far:

```
<html lang="en" ng-app>
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <title></title>
  <script type="text/javascript" src="js/angular.min.js"></script>
</head>
<body>

</body>
</html>
```

So, we've got Bootstrap and Angular there and we've initialized our app with the `ng-app` attribute in the opening `<html>` tag; let's get cracking.

We're going to have a Hello, World app with a bit of a difference. Instead of saying hello to the world, we're going to have an input field that will bind the data and echo it out in our view automatically, and we're going to do all of this without writing a line of JavaScript.

Let's start out by getting an `<h1>` tag in our `<body>` tag:

```
<h1>Hello, World</h1>
```

If you view this in your browser, you should notice that Bootstrap has tidied up the default. We no longer have Times New Roman but instead Helvetica and those excess margins around the edge have been removed:

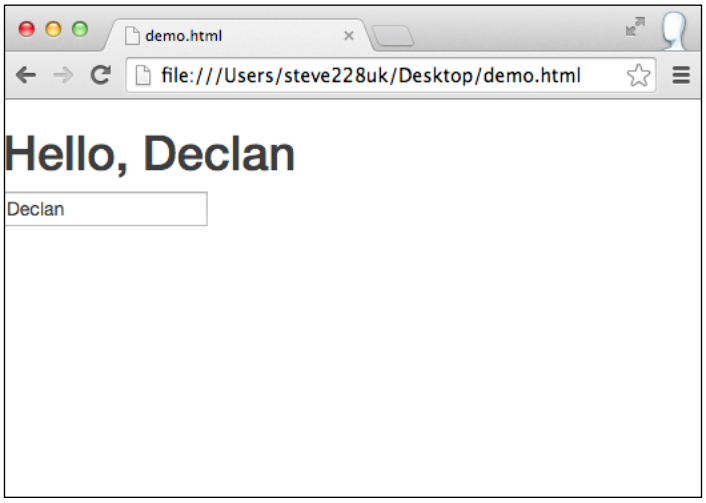
We now need to include our text input and also specify the model we want to use. Remember, a model can be any type, but in this case it will be a string that the input will return:

```
<input type="text" ng-model="name">
```

The `ng-model` attribute declares a model binding on that element, and anything we type into the input box will be automatically bound to it by Angular. Obviously this isn't going to be displayed on our page by magic; we need to tell the framework where we want it echoed. To display our model on the page, we just need to wrap the name of it in double curly braces:

```
{{name}}
```

Pop this in place of `World` in your `<h1>` tag and refresh the page in your browser. If you pop your name in the input field, you'll notice that it's automatically displayed in your heading in real time. Angular is doing all of this for us and we haven't written a single line of JavaScript.



Now, while that's great, it would be nice if we could have a default in place so it doesn't look broken before a user has entered their name. What's awesome is that everything in between those curly braces is parsed as an AngularJS expression, so we can check and see if the model has a value, and if not, it can echo `World`. Angular calls this an expression and it's just a case of adding two pipe symbols as we would in JS:

```
{{name || 'World'}}
```

Angular describes an expression as the following: "JavaScript-like code snippets that are usually placed in bindings such as `{{ expression }}`."

It's good to remember that this is JavaScript, and that's why we need to include the quotation marks here, to let it know that this is a string and not the name of a model. Remove them and you'll notice that Angular displays nothing again. That's because both the `name` and `World` models are undefined.

These models can be defined directly from within our HTML using an attribute as we've seen, but we can also assign a value to them from a controller. To do this, we're going to need to create a new JS file called `controller.js` and include it in our app:

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

Pop this in after you've included Angular on your page to avoid any errors being thrown.

Controllers are just functions that Angular can utilize; let's take a look at one:

```
function AppCtrl($scope){
}
```

Here, we've declared our controller (essentially just a plain JavaScript constructor function) and have injected the scope service into it. The scope is what we can access from within our view. There can be multiple controllers and multiple scopes on a single page. It's essentially a JavaScript object of our models and functions that Angular works its magic with, for example, the scope of our application so far looks like this:

```
{
 name: "Stephen"
}
```

The scope changes depending upon what's entered into the input field. This can then be accessed from our view as well as the controller.

Now that we've created our controller, we need to tell Angular we want to use it. For our application we only need a single controller, so let's add a second attribute to the `<html>` tag again:

```
ng-controller="AppCtrl"
```

This attribute tells Angular we want to use the `AppCtrl` function we've just created as our controller for the page. We could of course add this to any element on the page including the body if we so wish.

To check everything's working okay, we're going to specify an initial value for our model. This is as easy as setting a property on any object:

```
function AppCtrl($scope) {
  $scope.name = "World";
}
```

If you refresh your app in your browser, you'll notice that `World` is now pre-filled as the model's value. This is a great example of Angular's powerful **two-way data binding**. It allows us to use pre-defined data perhaps from an API or database and then change this in the view directly before picking it back up in the controller.

Angular describes data binding as "the automatic synchronization of data between the model and view components". Two-way data binding means that if you change the value of a model in your view or in your JavaScript controller, everything will be kept up-to-date.

Bootstrap

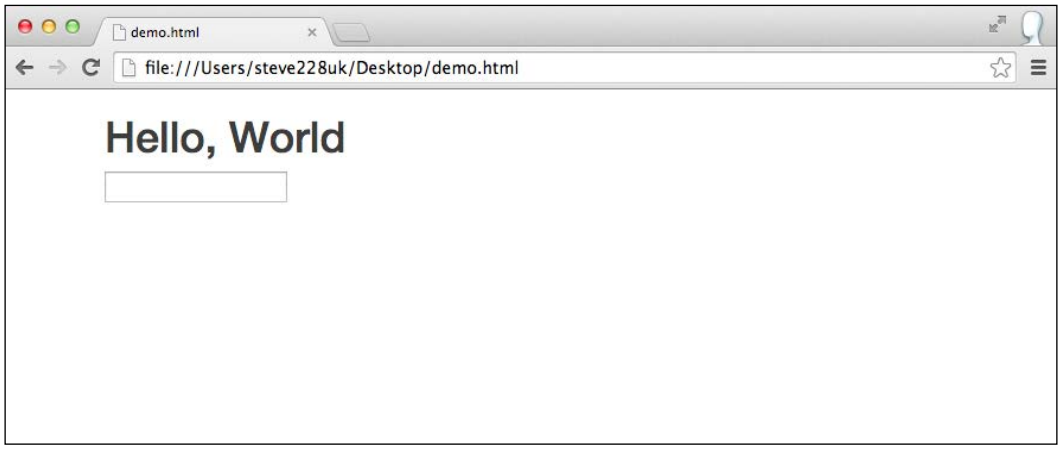
Now that we've created our Hello World application and everything is working as expected, it's time to get involved with Bootstrap and add a bit of style and structure to our app.

The application is currently misaligned to the left, and everything is looking cramped so let's sort that out first with a bit of scaffolding. Bootstrap comes with a great **mobile first** responsive grid system that we can utilize with the inclusion of a few divs and classes. First though, let's get a container around our content to clean it up immediately:

Mobile first is a way of designing/developing for the smallest screens first and adding to the design rather than taking elements away.

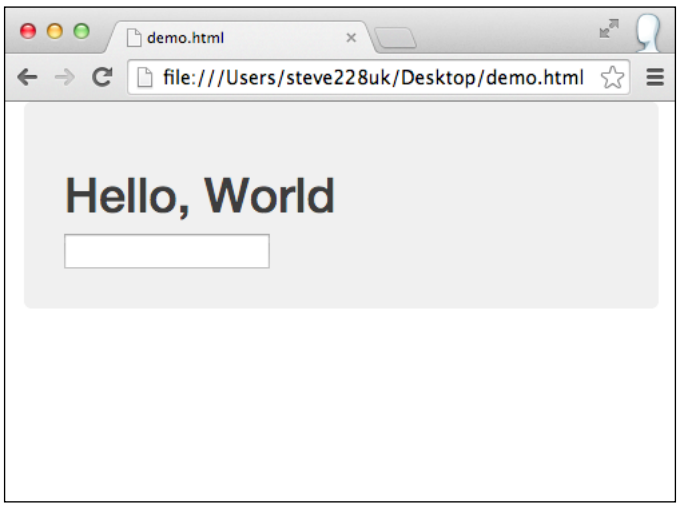
```
<div class="container">
  <h1>Hello, {{name || 'World'}}</h1>
  <input type="text" ng-model="name">
</div>
```

If you resize your browser window, you should start to notice some of the responsiveness of the framework coming through and see it collapsing:



Now, I think it's a good idea to wrap this in what Bootstrap calls a **Jumbotron** (in previous versions of Bootstrap this was a Hero Unit). It'll make our headline stand out a lot more. We can do this by wrapping our `<h1>` and `<input>` tags in a new div with the `jumbotron` class:

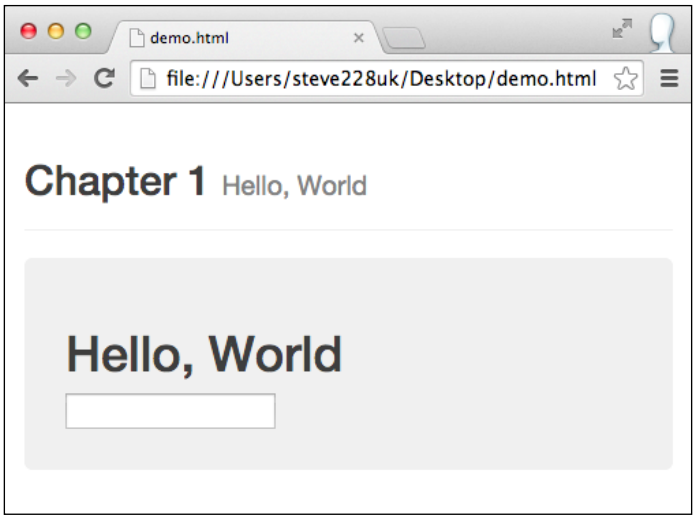
```
<div class="container">
  <div class="jumbotron">
    <h1>Hello, {{name || 'World'}}</h1>
    <input type="text" ng-model="name">
</div>
</div>
```



It's starting to look a lot better but I'm not too happy about our content touching the top of the browser like that. We can make it look a lot nicer with a page header but that input field still looks out of place to me.

First, let's sort out that page header:

```
<div class="container">
  <div class="page-header">
    <h2>Chapter 1 <small>Hello, World</small></h2>
  </div>
  <div class="jumbotron">
    <h1>Hello, {{name || 'World'}}</h1>
    <input type="text" ng-model="name">
</div>
</div>
```



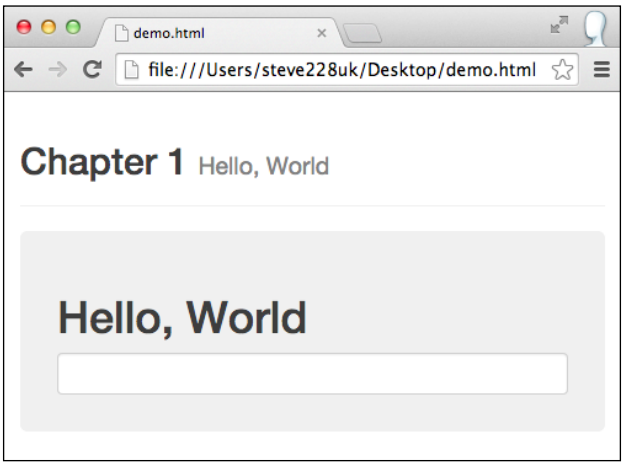
I've included the chapter number and title here. The `<small>` tag within our `<h2>` tag gives us a nice differentiation between the chapter number and the title. The `page-header` class itself just gives us some additional margin and padding as well as a subtle border along the bottom.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. 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.

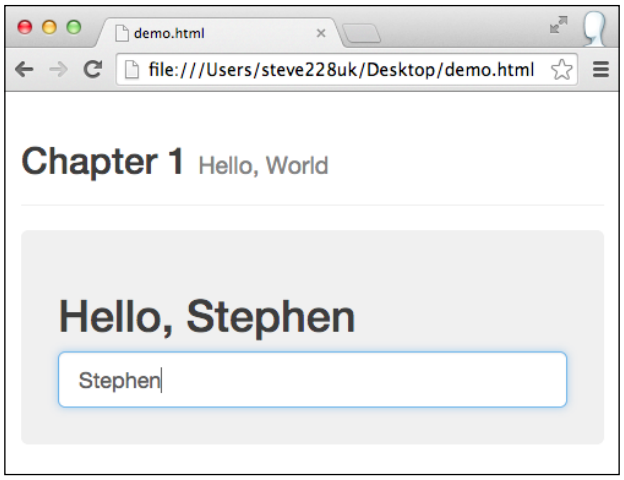
The last thing I think we could improve upon is that input box. Bootstrap comes with some great input styles so let's include those. First, we need to add the class of `form-control` to the text input. This will set the width to 100% and also bring out some nice styling such as rounded corners and a glow when we focus on the element:

```
<input type="text" ng-model="name" class="form-control">
```



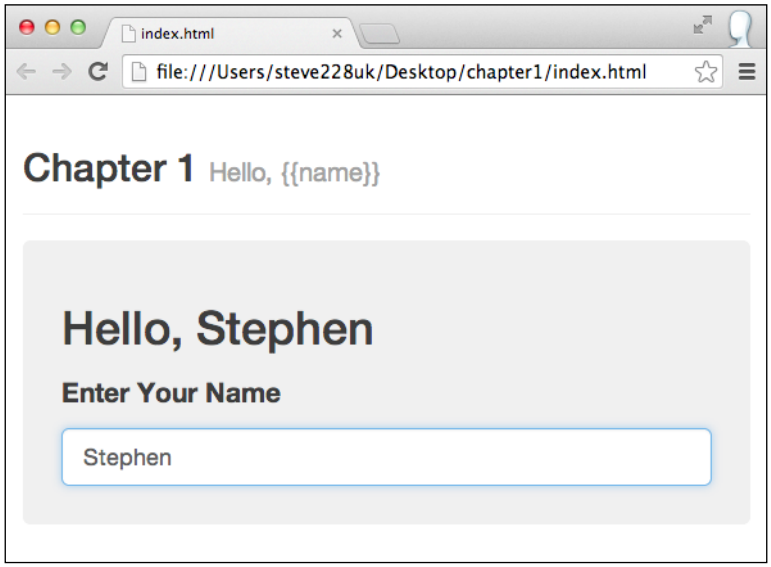
Much better, but to me it looks a little small when you compare it with the heading. Bootstrap provides two additional classes we can include that will either make the element smaller or larger: `input-lg` and `input-sm` respectively. In our case, the `input-lg` class is the one we want, so go ahead and add that to the input.

```
<input type="text" ng-model="name" class="form-control input-lg">
```



That's better but we still need to sort the spacing out, as it looks a bit snug against our `<h1>` tag. It's probably also a good idea that we add a label in so it's clear what the user should be entering in the box. Bootstrap allows us to kill two birds with one stone as it includes a margin on the label:

```
<label for="name">Enter Your Name</label>
<input type="text" ng-model="name" class="form-control input-lg"
id="name">
```



Self-test questions

1. How is Angular initialized on the page?
2. What is used to display a model's value on the page?
3. What does MVC stand for?
4. How do we create a controller and tell Angular we want to use it?
5. In Bootstrap 3, what's the new name for a Hero Unit?

Summary

Our app's looking great and working exactly how it should, so let's recap what we've learnt in the first chapter.

To begin with, we saw just how easy it is to get AngularJS and Bootstrap installed with the inclusion of a single JavaScript file and stylesheet. We also looked at how an Angular application is initialized and started building our first application.

The Hello, World app we've created, while being very basic, demonstrates some of Angular's core features:

- Expressions
- Scopes
- Models
- Two-way data binding

All of this was possible without writing a single line of JavaScript, as the controller we created was just to demonstrate two-way binding and wasn't a required component of our app.

With Bootstrap, we utilized a few of the many available components such as the `jumbotron` and the `page-header` classes to give our application some style and substance. We also saw the framework's new mobile first responsive design in action without cluttering up our markup with unnecessary classes or elements.

In *Chapter 2*, *Let's Build with AngularJS and Bootstrap* we're going to explore some more AngularJS and Bootstrap fundamentals and introduce the project we're going to be building over the course of this book.

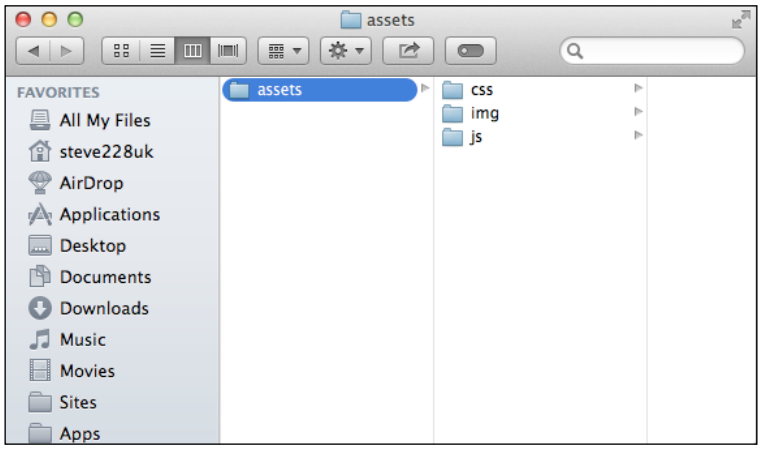
2
Let's Build with AngularJS and Bootstrap

Now that you've officially built your first web app using AngularJS and Bootstrap, it's time to up the ante. Over the course of the book we're going to be using both frameworks to build a contacts manager complete with full text search, creation, editing, and deletion. We'll look at building a maintainable code base as well as exploring the full potential of both frameworks. So, let's build!

Setting up

Let's quickly create a new directory for our app and set up a similar structure to our Hello, World app we made in *Chapter 1*, *Hello, {{name}}*.

The following folder structure is perfect for now:



You'll notice I've popped our directories into an `assets` directory to keep things tidy. Copy Angular and Bootstrap from *Chapter 1*, *Hello, {{name}}* into the relevant directories and create an `index.html` file in the root, which will become the basis of the contacts manager. The following code snippet is just a base HTML page with Bootstrap and Angular included. I've also initialized Angular on the page with the `ng-app` attribute on the `<html>` tag. Here's what our page should look like at this stage:

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
  <meta charset="utf-8">
  <title>Contacts Manager</title>
  <link rel="stylesheet" href="assets/css/bootstrap.min.css">
  <script type="text/javascript"
    src="assets/js/angular.min.js"></script>
</head>
<body>

</body>
</html>
```

Scaffolding

Okay, now that we've got our base file and folder structure sorted we can begin to scaffold out our app using Bootstrap. Apart from including a collection of components, such as navigation and buttons, that we can use throughout our contacts manager, Bootstrap also includes an extremely powerful and responsive grid system that we're going to harness the power of.

Navigation

We're going to need a **navbar** to switch between each of our views. Naturally, this will be placed at the top of the screen.

Let's take a look at our completed navigation before we break it down:

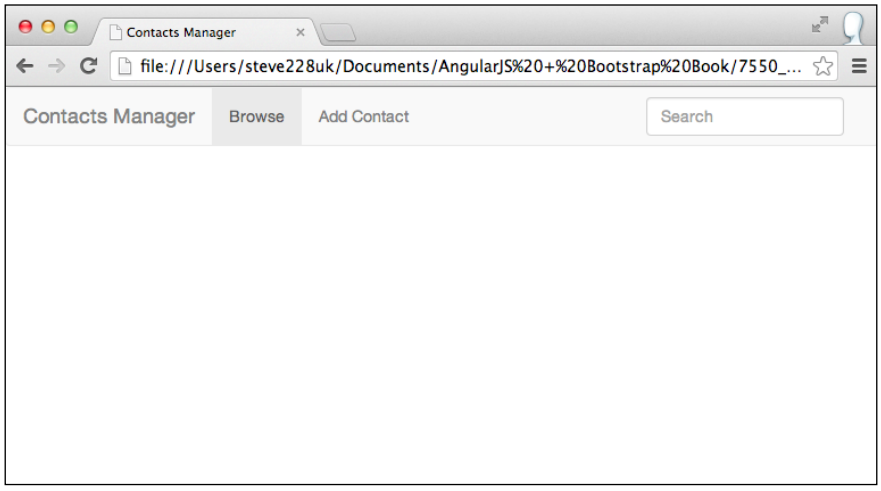
```
<nav class="navbar navbar-default"role="navigation">
    <div class="navbar-header">
        <button type="button" class="navbar-toggle"
          data-toggle="collapse" data-target="#nav-toggle">
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
        </button>
```

```
            <a class="navbar-brand" href="/">Contacts Manager</a>
        </div>

        <div class="collapse navbar-collapse" id="nav-toggle">
            <ul class="nav navbar-nav">
                <li class="active"><a href="/">Browse</a></li>
                <li><a href="/add">Add Contact</a></li>
            </ul>
            <form class="navbar-form navbar-right" role="search">
                <input type="text" class="form-control"
                placeholder="Search">
            </form>
        </div>
    </nav>
```

It can look quite intimidating for what is a very simple component of our page, but if we break it down, it becomes clear that everything here is completely necessary.

The `<nav>` tag holds everything within our navbar. Inside of this, the navigation is split into two sections: the `navbar-header` and `navbar-collapse`. These elements are exclusively for mobile navigation and control what is shown and what is hidden under the toggle button.

The `data-target` attribute on the button directly corresponds with the `id` attribute of the `navbar-collapse` element so Bootstrap knows what it should be toggling. The following screenshot is what our navigation will look like on devices bigger than a tablet.



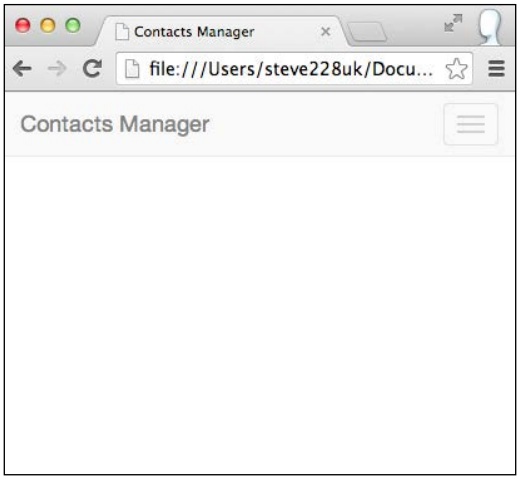
We're going to include our navigation directly within our `<body>` tag. This will allow it to span across the full width of the browser.

If you scale the browser down, you'll notice Bootstrap displays the mobile header with the toggle button below 768px—the size of an iPad screen in portrait. However, if you click the button to toggle the navigation, you'll notice nothing happens. That's because we haven't included Bootstrap's JavaScript file that was included in the `ZIP` file we downloaded earlier.

Copy it across to your app's `js` directory and reference it in your `index.html` file. You also need to include jQuery in the application as Bootstrap's JS depends on this. You can fetch the latest version from `http://jquery.com/`—again, add this to your directory and include it on your page before `bootstrap.js`. Ensure your JavaScript files are included in the following order:

```
<script src="assets/js/jquery.min.js"></script>
<script src="assets/js/bootstrap.min.js"></script>
<script src="assets/js/angular.min.js"></script>
```

If you reload the browser you should now be able to click the toggle button to display the mobile navigation.



Getting to grips with Bootstrap's grid

Bootstrap's 12-column grid system is very powerful and allows us to scaffold our responsive web app with very few elements, taking advantage of modular CSS along the way. The grid is composed of rows and columns that can be adapted using a series of classes. Before we begin, we need to include a container for our rows or the framework won't respond as expected. This is just a `<div>` tag that we can place below our navbar:

```
<div class="container"></div>
```

This will center our grid as well as add a `max-width` property to keep things nice and tidy.

There are four class prefixes, which define the behavior of the columns. For the most part, we'll be utilizing the `col-sm-` prefix. This collapses our columns down so they appear atop one another when the container is less than 750px wide.

The other classes all relate to different device screen sizes and react in a similar way. The following table from `http://getbootstrap.com/` shows the variations between all four classes:

	Phones (<768px)	**Tablets (≥768px)**	**Desktops (≥992px)**	**Desktops (≥1200px)**
Grid behavior	Horizontal at all times	Collapsed to start, horizontal above breakpoints		
Max container width	None (auto)	750px	970px	1170px
Class prefix	`.col-xs-`	`.col-sm-`	`.col-md-`	`.col-lg-`
Max column width	Auto	60px	78px	95px
Offsets	N/A	Yes		
Column Ordering	N/A	Yes		

Let's quickly make a two-column layout with a main content area and a sidebar. As the grid is made up of 12 columns, we're going to need to ensure our content area adds up to this or we'll end up with some empty space.

I think eight columns for our content area and four for our sidebar sounds perfect, so how would we go about implementing that?

Inside our container we first need to create a new `<div>` tag with the `row` class. We can have as many rows as we like, which can each house up to twelve columns.

```
<div class="container">
    <div class="row">

    </div>
</div>
```

As we only want our columns to stack on mobile devices, we're going to be using the `col-sm-` prefix. Creating a column is as simple as taking the desired prefix and appending the number of columns you wish for it to span. Let's take a look at how our basic two-column layout will look:

```
<div class="container">
    <div class="row">
        <div class="col-sm-8">
            This is our content area
        </div>
        <div class="col-sm-4">
            Here is our sidebar
        </div>
    </div>
</div>
```

When displayed on a screen larger than a mobile device, Bootstrap will automatically add 30px of guttering between each column (15px on either side). However, there will be times when you're going to want to create additional space between your columns or pull them in a little. Bootstrap provides a way to do this by adding an additional class to the column.

Once again, take the required prefix, but this time you need to include the keyword `offset`:

```
<div class="col-sm-4 col-sm-offset-1"></div>
```

This time, the number on the end controls the number of columns you wish to offset over. The additional class does this by adding an additional margin to the left.

Remember: Offset columns count toward your total of 12 columns in a row.

Inside our columns, we can nest additional rows and columns to create a more complex layout. Let's take a look:

```
<div class="container">
    <div class="row">
        <div class="col-sm-8">
            <div class="row">
                <div class="col-sm-6">
                    <p>Lorem ipsum dolor...<p>
                </div>
                <div class="col-sm-6">
                    <p>Class aptent taciti...</p>
```

```
                </div>
            </div>
        </div>
    </div>
</div>
```

This will create two columns within our main content container we created earlier. I've popped in some dummy text to demonstrate this.

If you open it up in your browser, you'll notice there are now three columns. However, as our grid is nested, we can create a new row and have a single column, three columns, or whatever our layout requires.

Helper classes

Bootstrap also includes a few helper classes that we can use to adapt our layout. These are generally utilitarian and are designed to serve a single purpose. Let's take a look at some examples.

Floats

Floating is often essential to creating a decent layout on the Web and Bootstrap gives us two classes to pull elements left or right:

```
<div class="pull-left">...</div>
<div class="pull-right">...</div>
```

In order to use floats effectively, we need to wrap our floated elements in a `clearfix` class. This will clear the elements and keep the flow of the document as expected:

```
<div class="clearfix">
  <div class="pull-left">...</div>
  <div class="pull-right">...</div>
</div>
```

If the float classes are directly within an element with the `row` class, then our floats are cleared automatically by Bootstrap and the `clearfix` class does not need to be applied manually.

Center elements

Alongside floats, there's often cause to center block-level elements. Bootstrap allows us to do this with the `center-block` class:

```
<div class="center-block">...</div>
```

This sets the margin-left and margin-right properties to auto, which will center the element.

Show and hide

You may wish to show and hide elements with CSS, and Bootstrap gives you a couple of classes to do this:

```
<div class="show">...</div>
<div class="hidden">...</div>
```

It's important to note that the show class sets the display property to block, so only apply this to block-level elements and not elements you wish to be displayed inline or inline-block.

Bootstrap also includes numerous classes to enable elements to be shown or hidden at specific screen sizes. The classes use the same pre-defined sizes as Bootstrap's grid.

For example, the following will hide an element at a specific screen size:

```
<div class="hidden-md"></div>
```

This will hide the element on medium devices but it will still be visible on mobiles, tablets, and large desktops. To hide an element on multiple devices, we need to use multiple classes:

```
<div class="hidden-md hidden-lg"></div>
```

Likewise, the visible classes work in reverse, showing elements at specific sizes. However, unlike the hidden classes, they also require us to set the display value. This can be block, inline, or inline-block:

```
<div class="visible-md-block"></div>
<div class="visible-md-inline"></div>
<div class="visible-md-inline-block"></div>
```

Of course, we can use the various classes in tandem. If, for example, we wanted a block-level element on a smaller screen but have it become inline-block later, we would use the following code:

```
<div class="visible-sm-block visible-md-inline-block"></div>
```

If you can't remember the various class sizes, be sure to take another look at the *Getting to grips with Bootstrap's grid* section.

Working with directives

Something we've been using already without knowing it is what Angular calls **directives**. These are essentially powerful functions that can be called from an attribute or even its own element, and Angular is full of them. Whether we want to loop data, handle clicks, or submit forms, Angular will speed everything up for us.

We first used a directive to initialize Angular on the page using `ng-app`, and all of the directives we're going to look at in this chapter are used in the same way—by adding an attribute to an element.

Before we take a look at some more of the built-in directives, we need to quickly make a controller. Create a new file and call it `controller.js`. Save this to your `js` directory within your project and open it up in your editor.

As we learnt in *Chapter 1*, *Hello, {{name}}*, controllers are just standard JS constructor functions that we can inject Angular's services such as `$scope` into. These functions are instantiated when Angular detects the `ng-controller` attribute. As such, we can have multiple instances of the same controller within our application, allowing us to reuse a lot of code. This familiar function declaration is all we need for our controller.

```
function AppCtrl(){
}
```

To let the framework know this is the controller we want to use, we need to include this on the page after Angular is loaded and also attach the `ng-controller` directive to our opening `<html>` tag:

```
<html ng-controller="AppCtl">
…
<script type="text/javascript"
  src="assets/js/controller.js"></script>
```

ng-click and ng-mouseover

One of the most basic things you'll have ever done with JavaScript is listened for a click event. This could have been using the `onclick` attribute on an element, using jQuery, or even with an event listener. In Angular, we use a directive.

To demonstrate this, we'll create a button that will launch an alert box—simple stuff. First, let's add the button to our content area we created earlier:

```
<div class="col-sm-8">
    <button>Click Me</button>
</div>
```

If you open this up in your browser, you'll see a standard HTML button created—no surprises there. Before we attach the directive to this element, we need to create a handler in our controller. This is just a function within our controller that is attached to the scope. It's very important we attach our function to the scope or we won't be able to access it from our view at all:

```
function AppCtl($scope){
    $scope.clickHandler = function(){
        window.alert('Clicked!');
    };
}
```

As we already know, we can have multiple scopes on a page and these are just objects that Angular allows the view and the controller to have access to. In order for the controller to have access, we've injected the `$scope` service into our controller. This service provides us with the scope Angular creates on the element we added the `ng-controller` attribute to.

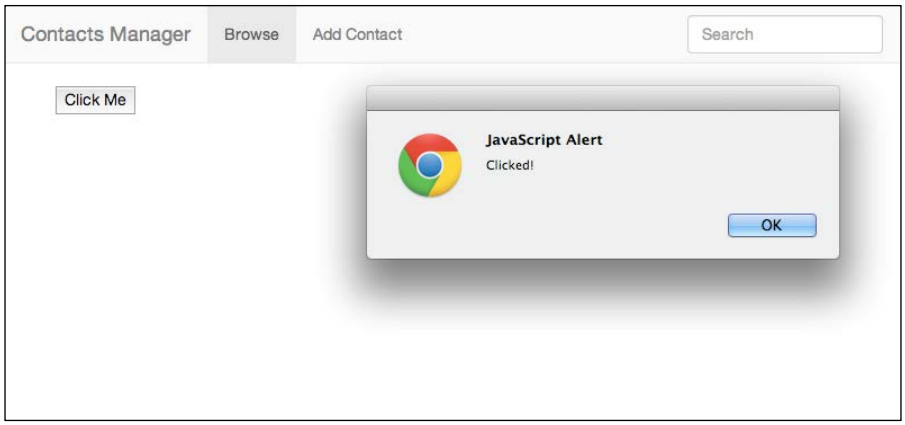
Angular relies heavily on dependency injection, which you may or may not be familiar with. As we've seen, Angular is split into modules and services. Each of these modules and services depend upon one another and dependency injection provides referential transparency. When unit testing, we can also mock objects that will be injected to confirm our test results. DI allows us to tell Angular what services our controller depends upon, and the framework will resolve these for us.

An in-depth explanation of AngularJS' dependency injection can be found in the official documentation at `https://docs.angularjs.org/guide/di`.

Okay, so our handler is set up; now we just need to add our directive to the button. Just like before, we need to add it as an additional attribute. This time, we're going to pass through the name of the function we're looking to execute, which in this case is `clickHandler`. Angular will evaluate anything we put within our directive as an AngularJS expression, so we need to be sure to include two parentheses indicating that this is a function we're calling:

```
<button ng-click="clickHandler()">Click Me</button>
```

If you load this up in your browser, you'll be presented with an alert box when you click the button. You'll also notice that we don't need to include the `$scope` variable when calling the function in our view. Functions and variables that can be accessed from the view live within the current scope or any ancestor scope.



Should we wish to display our alert box on hover instead of click, it's just a case of changing the name of the directive to `ng-mouseover`, as they both function in the exact same way.

ng-init

The `ng-init` directive is designed to evaluate an expression on the current scope and can be used on its own or in conjunction with other directives. It's executed at a higher priority than other directives to ensure the expression is evaluated in time.

Here's a basic example of `ng-init` in action:

```
<div ng-init="test = 'Hello, World'"></div>
{{test}}
```

This will display Hello, World onscreen when the application is loaded in your browser. Above, we've set the value of the test model and then used the double curly-brace syntax to display it.

ng-show and ng-hide

There will be times when you'll need to control whether an element is displayed programmatically. Both `ng-show` and `ng-hide` can be controlled by the value returned from a function or a model.

We can extend upon our `clickHandler` function we created to demonstrate the `ng-click` directive to toggle the visibility of our element. We'll do this by creating a new model and toggling the value between true or false.

First of all, let's create the element we're going to be showing or hiding. Pop this below your button:

```
<div ng-hide="isHidden">
    Click the button above to toggle.
</div>
```

The value within the `ng-hide` attribute is our model. Because this is within our scope, we can easily modify it within our controller:

```
$scope.clickHandler = function(){
    $scope.isHidden = !$scope.isHidden;
};
```

Here we're just reversing the value of our model, which in turn toggles the visibility of our `<div>`.

If you open up your browser, you'll notice that the element isn't hidden by default. There are a few ways we could tackle this. Firstly, we could set the value of `$scope.hidden` to true within our controller. We could also set the value of hidden to true using the `ng-init` directive. Alternatively, we could switch to the `ng-show` directive, which functions in reverse to `ng-hide` and will only make an element visible if a model's value is set to true.

Ensure Angular is loaded within your header or `ng-hide` and `ng-show` won't function correctly. This is because Angular uses its own classes to hide elements and these need to be loaded on page render.

ng-if

Angular also includes an `ng-if` directive that works in a similar fashion to `ng-show` and `ng-hide`. However, `ng-if` actually removes the element from the DOM whereas `ng-show` and `ng-hide` just toggles the elements' visibility.

Let's take a quick look at how we'd use `ng-if` with the preceding code:

```
<div ng-if="isHidden">
    Click the button above to toggle.
</div>
```

If we wanted to reverse the statement's meaning, we'd simply just need to add an exclamation point before our expression:

```
<div ng-if="!isHidden">
    Click the button above to toggle.
</div>
```

ng-repeat

Something you'll come across very quickly when building a web app is the need to render an array of items. For example, in our contacts manager, this would be a list of contacts, but it could be anything. Angular allows us to do this with the `ng-repeat` directive.

Here's an example of some data we may come across. It's array of objects with multiple properties within it. To display the data, we're going to need to be able to access each of the properties. Thankfully, `ng-repeat` allows us to do just that.

Here's our controller with an array of contact objects assigned to the contacts model:

```
function AppCtrl($scope){

    $scope.contacts = [
        {
            name: 'John Doe',
            phone: '01234567890',
            email: 'john@example.com'
        },
        {
            name: 'Karan Bromwich',
            phone: '09876543210',
            email: 'karan@email.com'
        }
    ];

}
```

We have just a couple of contacts here, but as you can imagine, this could be hundreds of contacts served from an API that just wouldn't be feasible to work with without `ng-repeat`.

First, add an array of contacts to your controller and assign it to `$scope.contacts`. Next, open up your `index.html` file and create a `<ul>` tag. We're going to be repeating a list item within this unordered list so this is the element we need to add our directive to:

```
<ul>
    <li ng-repeat="contact in contacts"></li>
</ul>
```

If you're familiar with how loops work in PHP or Ruby, then you'll feel right at home here. We create a variable that we can access within the current element being looped. The variable after the `in` keyword references the model we created on `$scope` within our controller. This now gives us the ability to access any of the properties set on that object with each iteration or item repeated gaining a new scope. We can display these on the page using Angular's double curly-brace syntax as we discovered in *Chapter 1*, *Hello, {{name}}*:

```
<ul>
    <li ng-repeat="contact in contacts">
        {{contact.name}}
    </li>
</ul>
```

You'll notice that this outputs the name within our list item as expected, and we can easily access any property on our contact object by referencing it using the standard dot syntax.

ng-class

Often there are times where you'll want to change or add a class to an element programmatically. We can use the `ng-class` directive to achieve this. It will let us define a class to add or remove based on the value of a model.

There are a couple of ways we can utilize `ng-class`. In its most simple form, Angular will apply the value of the model as a CSS class to the element:

```
<div ng-class="exampleClass"></div>
```

Should the model referenced be undefined or false, Angular won't apply a class. This is great for single classes, but what if you want a little more control or want to apply multiple classes to a single element? Try this:

```
<div ng-class="{className: model, class2: model2}"></div>
```

Here, the expression is a little different. We've got a map of class names with the model we wish to check against. If the model returns true, then the class will be added to the element.

Let's take a look at this in action. We'll use checkboxes with the `ng-model` attribute we've already seen in *Chapter 1*, *Hello, {{name}}*, to apply some classes to a paragraph:

```
<p ng-class="{'text-center': center, 'text-danger': error}">
    Lorem ipsum dolor sit amet
</p>
```

I've added two Bootstrap classes: `text-center` and `text-danger`. These observe a couple of models, which we can quickly change with some checkboxes:

The single quotations around the class names within the expression are only required when using hyphens, or an error will be thrown by Angular.

```
<label><input type="checkbox" ng-model="center"> text-
  center</label>
<label><input type="checkbox" ng-model="error"> text-
  danger</label>
```

When these checkboxes are checked, the relevant classes will be applied to our element.

ng-style

In a similar way to `ng-class`, this directive is designed to allow us to dynamically style an element with Angular. To demonstrate this, we'll create a third checkbox that will apply some additional styles to our paragraph element.

The `ng-style` directive uses a standard JavaScript object, with the keys being the property we wish to change (for example, color and background). This can be applied from a model or a value returned from a function.

Let's take a look at hooking it up to a function that will check a model. We can then add this to our checkbox to turn the styles off and on.

First, open up your `controller.js` file and create a new function attached to the scope. I'm calling mine `styleDemo`:

```
$scope.styleDemo = function(){
    if(!$scope.styler){
        return;
    }

    return {
        background: 'red',
        fontWeight: 'bold'
    };
};
```

Inside the function, we need to check the value of a model; in this example, it's called `styler`. If it's false, we don't need to return anything, otherwise we're returning an object with our CSS properties. You'll notice that we used `fontWeight` rather than `font-weight` in our returned object. Either is fine, and Angular will automatically switch the CamelCase over to the correct CSS property. Just remember than when using hyphens in JavaScript object keys, you'll need to wrap them in quotation marks.

This model is going to be attached to a checkbox, just like we did with `ng-class`:

```
<label><input type="checkbox" ng-model="styler"> ng-style</label>
```

The last thing we need to do is add the ng-style directive to our paragraph element:

```
<p .. ng-style="styleDemo()">
    Lorem ipsum dolor sit amet
</p>
```

Angular is clever enough to recall this function every time the scope changes. This means that as soon as our model's value changes from false to true, our styles will be applied and vice versa.

ng-cloak

The final directive we're going to look at is `ng-cloak`. When using Angular's templates within our HTML page, the double curly braces are temporarily displayed before AngularJS has finished loading and compiling everything on our page. To get around this, we need to temporarily hide our template before it's finished rendering.

Angular allows us to do this with the `ng-cloak` directive. This sets an additional style on our element whilst it's being loaded: `display: none !important;`.

To ensure there's no flashing while content is being loaded, it's important that Angular is loaded in the head section of our HTML page.

Self-test questions

1. What did we add to the top of our page to allow us to switch views?
2. How many columns a Bootstrap's grid system comprises of?
3. What is a directive and how are most of them used?
4. Which directive would we use to loop data?

Summary

We've covered a lot in this chapter, so before we continue onto the next chapter, let's recap it all.

Bootstrap allowed us to quickly create a responsive navigation. We needed to include the JavaScript file included with our Bootstrap download to enable the toggle on the mobile navigation.

We also looked at the powerful responsive grid system included with Bootstrap and created a simple two-column layout. While we were doing this, we learnt about the four different column class prefixes as well as nesting our grid. To adapt our layout, we discovered some of the helper classes included with the framework to allow us to float, center, and hide elements.

In this chapter, we saw in detail Angular's built-in directives: functions Angular allows us to use from within our view. Before we could look at them, we needed to create a controller, which is just a function that we can pass Angular's services into using dependency injection.

The directives we looked at here are ones that will be essential as we build our contact manager throughout the course of the book. Directives such as `ng-click` and `ng-mouseover` are essentially just new ways of handling events that you will have no doubt done using either jQuery or vanilla JavaScript. However, directives such as `ng-repeat` will probably be a completely new way of working. It brings some logic directly within our view to loop through data and display it on the page.

We also looked at directives that observe models on our scope and perform different actions based on their values. Directives like `ng-show` and `ng-hide` will show or hide an element based on a model's value. We also saw this in action in `ng-class`, which allowed us to add some classes to our elements based on our models' values.

3
Filters

In the previous chapter, we looked at one of the core components of AngularJS: directives. As with many frameworks, Angular also has other paradigms to help us build our web app. Filters allow us to easily manipulate and sort data from either the view or controller, and just like with directives, there are a good few filters included out of the box.

There are many use cases for filters, and we'll take a look at a few of them over the course of the chapter. For example, you may simply want to manipulate a string. This could be by converting, localizing, or even truncating. Of course, filters also allow you to work with other JavaScript types, such as arrays and objects. Perhaps you'd want to create a live search to filter through a dataset you've looped using `ng-repeat`. All of that is possible with filters.

Before we take a look at some of the pre-included filters, we should probably see how a filter is applied from the view.

Applying a filter from the view

Filters can be applied directly to expressions within our templates. Remember, an expression is anything within the double curly-brace syntax or a directive:

```
{{expression | filter}}
```

It's easy to apply a filter; we just add a pipe symbol followed by the name of the filter we want to place on the expression. We can follow the same idea to apply multiple filters to a single expression. Multiple filters are chained and applied in succession. In the following example, `filter2` will be applied to the output of `filter1` and so forth:

```
{{expression | filter1 | filter2 | filter3}}
```

Some filters may have arguments, and these can be applied using a similar syntax:

```
{{expression | filter:argument1:argument2}}
```

Throughout the chapter, we'll demonstrate a number of the filters included with Angular directly from the view using the syntax we've just looked at. We'll then take a look at how we can apply the same filters from the controller and also how we can create our own.

Currency and numbers

The first filter we're going to look at is one that formats numbers into currency. In the en-US locale, it adds a comma to separate thousands and a decimal point in the right place. It also prepends the relevant symbol:

```
{{12345 | currency}}
```

The currency symbol will depend on locale. As we're using en-US, by default, Angular prepends a dollar symbol (`$`), but we can pass through the symbol of our choosing as an argument:

```
{{12345 | currency:'£'}}
```

It's important to remember to wrap the symbol in quotation marks, as this is a string.

Angular also includes a second filter to format numbers, which gives us a little more control. It allows us to specify the number of decimal places we wish the number to be rounded to:

```
{{12345.225 | number:2}}
```

The output of this filter will be `12,345.23`. You'll notice that the number has been rounded up to two decimal places and a comma has been added to separate thousands.

Lowercase and uppercase

These two filters are perhaps the simplest ones included with Angular. They simply convert the provided string to lowercase or uppercase:

```
{{'Stephen' | lowercase}}

{{'Declan' | uppercase}}
```

These filters output the following:

```
stephen
DECLAN
```

limitTo

There are times when you need to limit a string or an array, and this can easily be achieved in AngularJS using the `limitTo` filter:

```
{{'Lorem ipsum dolor sit amet' | limitTo:15}}
```

You'll notice that this filter takes a single argument, which is the number to which the input should be limited. Here we've limited a string, but this could quite easily be an array in an `ng-repeat` directive, for example:

```
<div ng-repeat="array | limitTo:2"></div>
```

Date

When working with data from an API, it's often the case that the date will be given as a UNIX time or a full timestamp. This isn't the friendliest thing to work with, and thankfully, Angular includes an easy way to format dates with a filter:

```
{{expression | date:format}}
```

The filter takes one argument: format. For example, if we wanted to take a timestamp and just output the year, we could easily do that with the following expression:

```
{{725508723000 | date:'yyyy'}}
```

We can combine this with the input for day and month and output a standard date string easily:

```
{{725508723000 | date:'dd/MM/yyyy'}}
```

Here's a list of some of the most useful elements the format string can be comprised of. A full list can be found on the AngularJS website:

Element	Output	Example
yyyy	4-digit year	2013
yy	2-digit year	13
MMMM	Full text month	December
MMM	Short text month	Dec
MM	Padded numerical month	01
M	Numerical month	1
dd	Padded day	01
d	Day	1
EEEE	Day in week	Monday

Element	Output	Example
EEE	Short day in week	Mon
HH	Padded 24-hour	01
H	24-hour	1
hh	Padded 12-hour	01
h	12-hour	1
mm	Padded minute	05
m	Minute	5
ss	Padded second	09
s	Second	9
a	AM/PM	AM or PM
Z	Timezone	+0100
ww (1.3+ Only)	Week of the year, padded.	03
w (1.3+ Only)	Week of the year	3

There are also a number of predefined formats we can use; let's take a look at one:

```
{{725508723000 | date:'medium'}}
```

The medium keyword is just one of the predefined formats this filter recognizes and outputs Dec 28, 1992 2:12:03 AM.

Here's a full list of predefined formats that the date filter will accept:

Keyword	Equivalent	Example
medium	MMM d, y h:mm:ss a	Sep 3, 2010 12:05:08 pm
short	M/d/yy h:mm a	9/3/10 12:05 pm
fullDate	EEEE, MMMM d,y	Friday, September 3, 2010
longDate	MMMM d, y	September 3, 2010
mediumDate	MMM d, y	Sep 3, 2010
shortDate	M/d/yy	9/3/10
mediumTime	h:mm:ss a	12:05:08 pm
shortTime	h:mm a	12:05 pm

We can also include literal values within our format string; for example:

```
{{725508723000 | date:"h 'in the morning'"}}
```

Literal values must be wrapped in single quotations. In order for this to happen, we need to swap our single quotations used around the argument for double quotes. Should you wish to include a single quotation mark within the string, you simply need to use two single quotes:

```
{{725508723000 | date:"h 'o''clock'"}}
```

Filter

This confusingly named filter allows us to select a subset of items from an array easily. Within our view, this can be used in combination with the `ng-repeat` directive we looked at in the previous chapter.

We can use this to build a pretty powerful search tool that will filter through our array. Let's take a look at the `ng-repeat` example we used in the previous chapter:

```
<ul>
    <li ng-repeat="contact in contacts">
        {{contact.name}} - {{contact.phone}}
    </li>
</ul>
```

Before we can add our filter, we just need to add the pattern object that will be used for selection from our array. This can be a model, string, pattern object, or function. As we're creating a search, let's just hook a model up to a text input:

```
<input type="text" ng-model="search">
```

The last thing to do is to attach our model to the `ng-repeat` directive. This is done exactly the same as any other filter: a pipe symbol followed by the name of the filter. In this case, we also need to add one argument telling the filter which model, string, object, or function we wish to use:

```
<li ng-repeat="contact in contacts | filter:search">
```

This will allow us to use the input field we create to search through everything within our array, which includes names, phone numbers, and email addresses. However, what happens if we want to limit our search to only the name property on our objects? We simply just need to change our model:

```
<input type="text" ng-model="search.name">
```

It's important that we leave the name of the model on our `ng-repeat` as `search` or the filtering won't be limited to our desired property. Alternatively, we could use the following syntax on our `ng-repeat` directive to limit our filtering to specific properties. This would allow us to leave the name of our model as `search`:

```
<li ng-repeat="contact in contacts | filter:{'name': search}">
```

orderBy

Apart from filtering our object from within the `ng-repeat` directive, we can also order it. This is great if the data you're given from an array isn't sorted already or doesn't provide an option to do so.

Currently, our object is all jumbled up and there's no apparent order to it. Let's take a look at how we could go about sorting this by name:

```
<li ng-repeat="contact in contacts | filter:search |
  orderBy:'name'">
```

The first argument we can pass through is a string with the name of the property we want to sort our array by. Should we want to filter by phone number or email address instead, we could pass those values through here.

We can also pass a second argument through a Boolean which controls whether the filter should reverse the order or not:

```
<li ng-repeat=" .. | orderBy:'name':true">
```

JSON

The last included filter is mainly for debugging purposes. It will output any JavaScript object into a JSON string for output onto the page.

Let's take our array of contacts that we used in the last chapter to demonstrate `ng-repeat` and apply the `json` filter to it:

```
{{contacts | json}}
```

The following is the output to our view:

```
[
  {
     "name":"John Doe",
     "phone":"01234567890",
     "email":"john@example.com"
  },
```

```
    {
        "name":"Karan Bromwich",
        "phone":"09876543210",
        "email":"karan@email.com"
    },
    {
        "name":"Declan Proud",
        "phone":"2341234231",
        "email":"declan@email.com"
    },
    {
        "name":"Paul McKay",
        "phone":"912345678",
        "email":"p.mckay@domain.com"
    }
]
```

As you can see, this is just a JSON representation of the array of objects we created earlier.

Applying filters from JavaScript

There are times when you'll want to apply a filter using JavaScript, usually from your controller, so it's important we take a look at how to do this; there are a couple of options.

We can either inject the `$filter` service into our controller and utilize any filter included within our application. Alternatively, we can inject the filter as its own dedicated service and use it on its own. Both methods are perfectly valid, and it's down to you, whichever you prefer.

Let's first take a look at using the `$filter` service. We'll take the `json` filter we've just looked at and `console.log` the very same array. To begin, let's inject that service into our controller:

```
function AppCtl($scope, $filter){
…
}
```

Great! We can now utilize this just as we can `$scope`. To use it, we simply need to call it as a function and pass through the name of the filter we wish to use, which in our case is `json`:

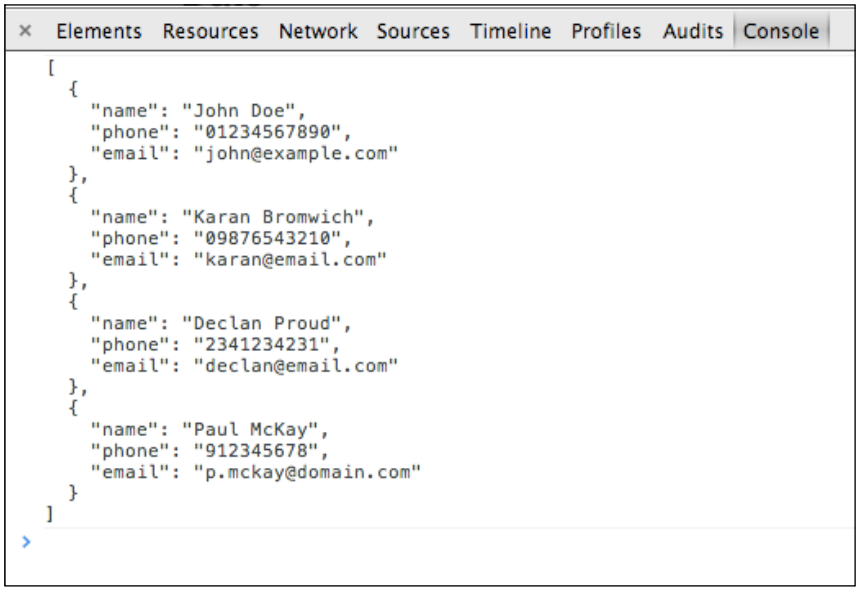
```
$filter('json');
```

This actually returns the filter itself, and we can see this in the output if we `console.log` it directly. This means we can call the function immediately by adding a second set of parenthesis straight after:

```
$filter('json')($scope.contacts);
```

As we know, the `json` filter doesn't accept any arguments. However, the first argument of all filter functions is actually the input. We don't see this when we're calling them from the view as Angular does its magic behind the scenes to simplify things.

If you wrap the preceding expression in a `console.log`, you'll see that the output is identical to the output in our view using the same filter:



Alternatively, if you don't want to use the `$filter` service, you can inject each filter separately as a service. These are named in the pattern `filternameFilter`. So for our example, we need to inject `jsonFilter`:

```
function AppCtl($scope, jsonFilter){
…

}
```

This can then be used identically as the function returned by the `$filter` service, allowing us to pass through our object to filter:

```
jsonFilter($scope.contacts);
```

Now that we know how we can use filters from within our controller, let's take a look at how we could create our own.

Building your own filter

As we've seen, the `limitTo` filter is great for truncating strings of text. I've always felt that the filter could do with appending an ellipsis should the string cross the limit. Thankfully, Angular lets us expand upon the included filters and build our own.

Modules

In order to create our filter, we first need to create what's called a **module**. This will enable us to attach a filter and utilize it in our views or controllers. I think the AngularJS documentation explains what a module is perfectly:

> *"You can think of a module as a container for the different parts of your app: controllers, services, filters, directives, and so on."*

Okay great, but why would we or do we need to use one? There are a couple of reasons why you might want to use a module. Primarily, the most popular reason for their existence is the ease of creation of reusable code.

Imagine you are working on a blogging platform. You might build a module to allow for a media browser/uploader. This would be a collection of controllers, services, and filters all bundled up nicely. Should you wish to use this media browser in another project, then you'd just need to copy the module over.

There are also other reasons why you'd want to use modules. If you were unit testing, the tests only need to load relevant modules to keep them quick, and code becomes easier to understand and follow as each component is neatly packaged amongst other things.

Creating a module

It's very easy to create a module, and it allows us to extend upon the core much more, as Angular won't allow custom filters without one. Here, we've made a new module called `contactsMgr`. The second argument is just an empty array. We can have as many modules as we like and include them as dependencies, but for now we'll just leave it empty:

```
angular.module('contactsMgr', []);
```

We do, however, need to make a slight adjustment to how our controller is added. Currently it's just a function, but we need to add this to our module for Angular to be able to pick it up:

```
angular.module('contactsMgr', [])

.controller('AppCtl', function($scope, jsonFilter){

…

});
```

We can chain our controllers onto our module. You'll notice we need to use the `controller` method. The first argument is our controller name, and the second is our callback function with our injected services.

If you load up your app now, you'll see that nothing is working as expected and that the controller function cannot be found. That's because we haven't told Angular which module we wish to use. To do this, we just need to add the name of our module to the `ng-app directive`:

```
<html lang="en" ng-app="contactsMgr" ng-controller="AppCtl">
```

Once that's in, everything should start to work just as it did previously. You're now utilizing the module we just created.

Creating a filter

Now that our module is created and working, we can get to work on our improved `limitTo` filter. It's wise to work out exactly what we want our filter to do before we dive in. We can break down the functions we want to perform into a just a few short steps:

- Take our input with a single argument for our limit
- Check the length of the input against the limit
- If the input is greater than the limit, truncate and add an ellipsis
- Otherwise, just return the input

When working with modules, creating a filter follows a very similar pattern to creating a controller:

```
.filter('truncate', function(){

});
```

Just like we did when we moved our controller over to our new module, we use a new method, which accepts two arguments: the filter name and a callback function. As we learnt when we applied filters from the controller, when a filter is called, it actually returns a second function, so we need to add that here:

```
.filter('truncate', function(){
    return function(){
    };
})
```

We also discovered that the first argument of a filter is always the input or data that we're going to be filtering. Within this function, we can also include additional arguments. For our truncate filter, we only need one argument to tell the filter how many characters it should limit the string by:

```
.filter('truncate', function(){
    return function(input, limit){
    };
})
```

The construction of our filter is complete and we can now actually use the filter in exactly the same way as the filters we looked at before. Of course, we don't have any of our logic in place here and nothing is being returned from the filter function, so we'd actually end up displaying nothing on our page.

All we need to do now is check the length, truncate the string, and append an ellipsis. All of this can be done in one string with the help of a ternary statement:

```
return (input.length > limit) ? input.substr(0, limit)+'…' :
  input;
```

We check the length of the string, and if it's greater than the limit, we truncate it and append an ellipsis. If it doesn't match our condition, we return the original input. This is important because Angular won't display anything if nothing's returned from our filter.

Okay, let's put everything together and take a look at our completed function:

```
.filter('truncate', function(){
  return function(input, limit){
    return (input.length > limit) ? input.substr(0, limit)+'…'
      : input;
  };
})
```

Our new filter can now be used in the exact same way as the built-in `limitTo` filter, so let's swap the filter out and take a look:

```
{{'Lorem ipsum dolor sit amet' | truncate:15}}
```

As expected, the output now includes an ellipsis, whereas previously the string was just chopped off after the limit.

Self-test questions

1. How do we apply a filter from the view?
2. How do we pass through arguments to our filter from the view?
3. Which filter would we use to create a live search?
4. How can we use a filter from the controller?
5. What do we need to create before we can create our own filter?
6. What three arguments does the filter method accept?

Summary

By now you should definitely know what a filter does and why it is so helpful, but let's recap everything we've covered in this chapter.

We started off by looking at how a filter is applied directly from our view using the pipe symbol syntax and separating any arguments with a colon. Once we had the basics covered, it was time to look at the numerous included filters.

A few filters were basic, not requiring any arguments at all, but we also looked at some of the more advanced filters that allow us to order or filter an array of objects.

Apart from applying filters from the view, we also looked at two methods of filtering from our controller. We could either use the included `$filter` service or choose to inject our filters separately.

Finally, we looked at extending Angular to create our own filter to truncate text. Before we could do this, we had to look at creating a module to contain our filters and controllers. Once our module was up and running, we were able to create our filter and use it identically to the ones included.

We've now covered many of the core paradigms and ideologies of Angular. In the next chapter, we're going to look at setting up routing to handle multiple views and controllers for our contacts manager.

4
Routing

All web apps will require more than one page or view, and Angular is well-equipped to handle this with its router. You may be familiar with routing in server-side frameworks, such as Ruby on Rails or Laravel. Angular's is, of course, entirely on the client side and all the magic happens within the HTML file our browser is pointed to.

In this chapter, we'll take a look at how we can create static routes as well as routes containing parameters. We'll also discover some of the pitfalls you might face.

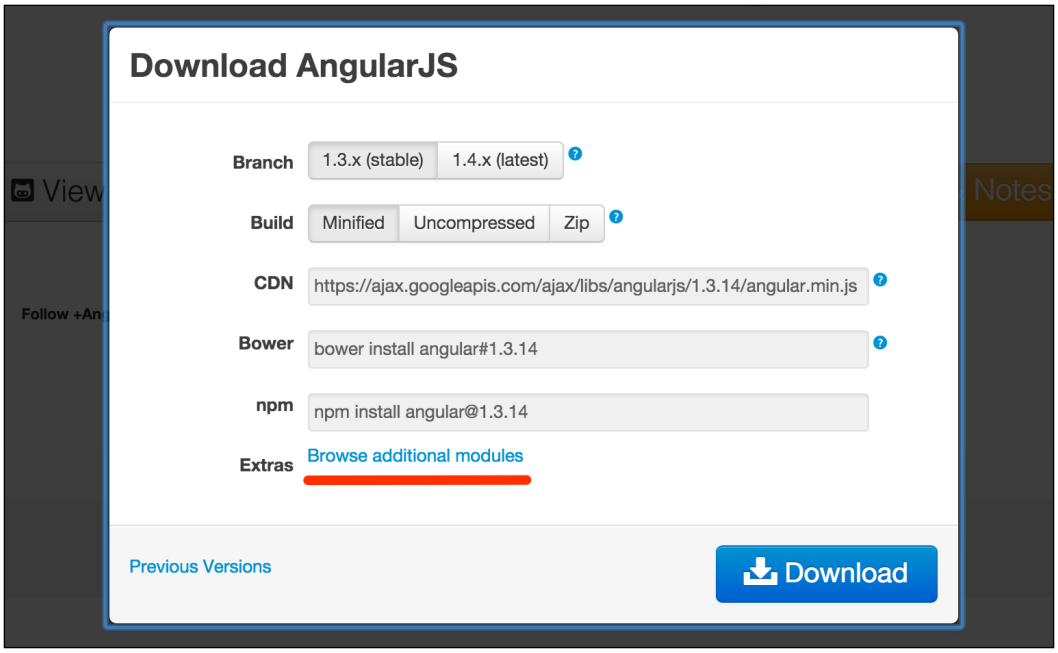
Before we begin, let's plan out exactly what routes we're going to need for our contacts manager:

- **Index**: This is going to be our main page, which will list all of our contacts in a table
- **View Contact**: Here, we'll be able to see the contact in more detail and edit any of the information it presents
- **Add Contact**: This will include a form that will allow us to add a contact to the manager

These are all of our essential routes; so let's take a look at how we can create them.

Installing ngRoute

Since AngularJS 1.2, the router has been packaged as a separate module outside of Angular's core. The file we're looking for—`angular-route.min.js`—can be downloaded from Angular's website below the **Extras** section within the download window.



Once you've got the download, drag it into your project's `js` directory and include it in the page after AngularJS:

```
<script src="assets/js/angular-route.min.js"></script>
```

We also need to let our module know that we want to utilize the router. We can do this by adding it to the module's dependency list. We can have as many dependencies as we like; currently all we need to include is `ngRoute`:

```
angular.module('contactsMgr', ['ngRoute'])
```

Creating basic routes

As we've already discovered, in order to configure the router within AngularJS, a module is required. In *Chapter 3*, *Filters*, we created one to allow us to build a custom filter. We can utilize this same module to build our routes.

Routes are created within the `config` method of our application's module:

```
angular.module('contactsMgr', ['ngRoute'])
.config(function($routeProvider){

})
```

The method accepts an anonymous function that we can inject our required `$routeProvider` service into. This service has just two methods: `when` and `otherwise`. To add a route, we use the `when` method, which accepts two parameters: the path as a string and options for the route as an object:

```
angular.module('contactsMgr', ['ngRoute'])
.config(function($routeProvider){
    $routeProvider.when('/', {});
})
```

There are two properties within our route options object that we're particularly interested in: `controller` and `templateUrl`. The controller property calls an existing `controller` constructor or defines a new one using an anonymous function. Meanwhile, the `templateUrl` property allows us to define the path to an HTML file that will house our entire markup for that view. Alternatively, we could define the template directly within the route object. However, things can get messy fairly quickly that way and are only really recommended for one- or two-line templates.

Let's take a look at the route we're going to define for our index page:

```
$routeProvider.when('/', {
    controller: 'indexCtl',
    templateUrl: 'assets/partials/index.html'
});
```

The path to the template is relative to our base HTML file; hence, it includes the `assets` directory in the path. We can now go ahead and create that HTML template. Angular refers to these as partials and we'll be using them for all of our views.

The controller argument within our route is optional, but we've included it as we're going to need one for our application. Let's create that `controller` to allow us to build models and functions exclusively for our `index` view.

Within our `controller.js` file, let's can chain this onto the end:

```
.controller('indexCtrl', function($scope){

});
```

Let's quickly add our second route with our `config` method. This will house our add-contact form:

```
$routeProvider.when('/', {
    controller: 'indexCtl',
    templateUrl: 'assets/partials/index.html'
})

.when('/add-contact', {
    controller: 'addCtl',
    templateUrl: 'assets/partials/add.html'
});
```

Just as we can with controllers, we can chain our routes. Now just create the relevant `controller` and `partial`:

```
.controller('addCtl', function($scope){

});
```

The last thing we need to do before Angular kicks the router into action is include the `ng-view` directive on our page. This pulls in the partial we've defined in the route.

Note: You can only include `ng-view` once on per page.

```
<div class="container">
    <ng-view></ng-view>
</div>
```

This directive can be included as its own element. I've opted to include the directive as an element in my root `index.html` file. If you have anything in your container already, clear it out and replace it with `ng-view` instead.

If you open the project in your browser, you'll notice that the route has been appended to the URL with the # symbol preceding it. Unfortunately, if you're using Chrome, it's likely that the partials will fail to load. If you open up the console, you'll probably see a similar error to the following:

```
Cross origin requests are only supported for HTTP.
```

There are a couple of ways to fix this. We can either load the code up on a web server, or if we're using Chrome, we can run the browser using a flag to enable cross-origin requests over the `file://` protocol on OS X or over `c:/` on Windows.

On OS X, run the following in Terminal:

```
open -a 'Google Chrome' --args -allow-file-access-from-files
```

On other *nix-based systems run the following:

```
google-chrome  --allow-file-access-from-files
```

On Windows, you need to edit the desktop shortcut to add a flag at the end of the Target:

```
C:\ ... \Application\chrome.exe --allow-file-access-from-files
```

If you don't want to run Chrome with a flag, you can run the contact manager on a web server. You could use the web server built into Python or PHP, or a full-stack app like MAMP or WAMP.

Change directory into your project and run the following command to server your application using Python's web server:

```
python -m SimpleHTTPServer 8000
```

You can now navigate to `localhost:8000` in your browser to view your app. Alternatively, if you would prefer to run PHP's web server, you can do that with the following:

```
php -S localhost:8000
```

Routes with parameters

Okay, we've set up multiple routes but we still need to look at how we can include parameters within them. This is important to allow a level of dynamism within our contacts manager. For example, we're going to be using them to view a specific contact by referencing an ID number or index.

Adding a parameter is easy; we just need to add a placeholder in. This is done with a colon followed by the name of the parameter we wish to create. Let's take a look at the route we're going to make to view our contact. Once more, we can chain this onto our existing routes:

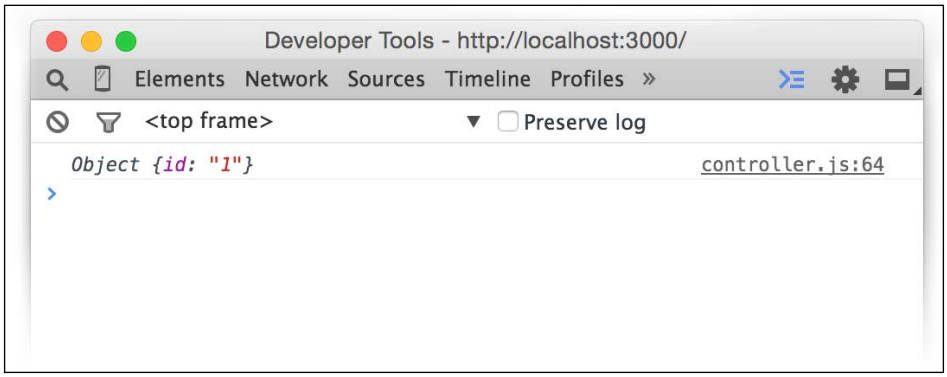
```
    .when('contact/:id', {
        controller: 'contactCtl',
```

```
        templateUrl: 'assets/partials/contact.html'
    });
```

We can add as many parameters as required, and it's easy to pull these out in our controller. It's just a case of injecting a service into the controller and we'll have access to all route parameters as objects:

```
    .controller('contactCtl', function($scope, $routeParams){
        console.log($routeParams);
    });
```

If you navigate to `localhost:8000/#/contact/1` and open up your console, you'll see the route parameters logged as a JS object:



That means we can access any of the properties on the object using the standard syntax:

```
$routeParams.id;
```

The fallback route

The last route we need to configure is the one that will show when no route is matched. You could create a 404 page for this, but let's take a look at how we can redirect a route instead of displaying a template.

To create our fallback route, we use the second method that the `$routeProvider` service gives us—`otherwise`:

```
.otherwise({
    redirectTo: '/'
});
```

Now, if the requested route doesn't match any of the ones defined in our router, Angular will redirect us back to our index page.

HTML5 routing or removing

All of our essential routes are configured and we now have access to separate partials for all of them. That's great, but I'm not really happy with the routes following the # symbol in the URL. Thankfully, there's an easy way to eradicate that, by enabling what Angular calls `html5Mode`.

The mode enables Angular to take advantage of pushState in modern browsers while still providing a fallback for legacy browsers, such as IE 8.

Enabling HTML5Mode

To enable the new mode, we need to look at our `config` method again. Like before, we're going to need to inject a service into it:

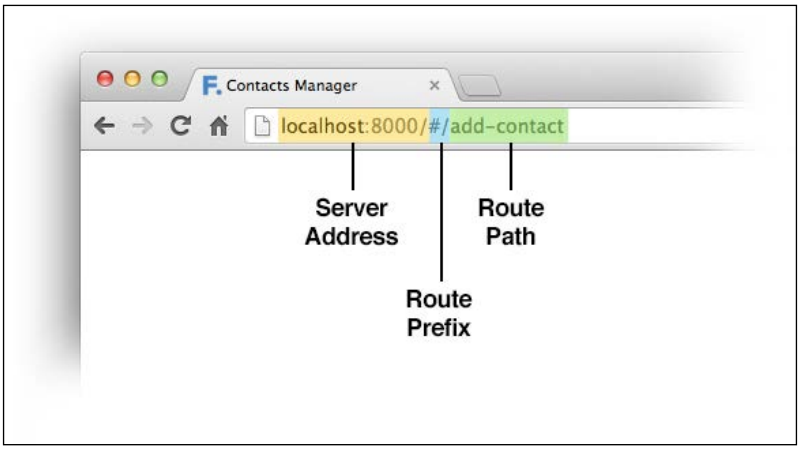
```
.config(function($routeProvider, $locationProvider){

    ...

    $locationProvider.html5Mode(true);

})
```

You'll notice that we've now injected a second service: `$locationProvider`. This allows us to take advantage of the `html5Mode` method, which accepts a Boolean to turn it off or on.

The service also provides us with a second method, and though we won't be taking advantage of it during our build, it's still good to know. The `hashPrefix` method allows us to add a prefix after the # symbol in the URL. For example, we could add an exclamation mark and turn the prefix into a hashbang (`#!`):

```
$locationProvider.hashPrefix('!');
```

The following diagram shows our application's URL and splits the address down into the sections of our route:



Linking routes

Linking routes is no different than linking to pages on a website. We still use an anchor tag and in place of the link to the page we want to link the route.

For example, if we wanted to link up the **Add Contact** button in our navbar, we would do the following:

```
<a href="/add-contact">Add Contact</a>
```

Angular will automatically display the correct partial when we click the link and also change the URL. If you've opted not to use html5Mode, we can still link using an anchor tag, but the `href` attribute is a little different—we need to add the hash:

```
<a href="#/add-contact">Add Contact</a>
```

Self-test questions

1. What file/module do we need to include to enable routing?
2. Which method is used to create our routes?
3. What needs to be injected into the method for us to be able to create a route?
4. How do we create a route?
5. What can we use when none of our routes match the current path?
6. How can we remove the # symbol from the URL?

Summary

In this chapter, we transformed our application from a single page into a multi-page view and multi-route app that we can build our contacts manager upon. We started by planning out the essential routes in our application before installing the requisite module.

We then looked at how we can use the `config` method on our own module to set up our routes. This was done by injecting the `$routeProvider` service and using the when and other methods provided. This allowed us to set up static and dynamic routes containing parameters.

Finally, we looked at how we can remove the `#` symbol from the URL using HTML5's `pushState` and how we can link both types of routes. In the next chapter, we'll populate our partials with layouts we'll be building using Bootstrap.

5
Building Views

In *Chapter 4, Routing*, we took a look at how we could turn our application into a multi-route and multi-view web app. We took advantage of Angular's router and set up partials for all of our core views. Now it's time to build up our views using Bootstrap so that we're ready to populate our app with data. Let's break down each of the partials one by one.

Populating the Index view

Our Index view is what's displayed when we first open the app. It is probably a good idea to list all of our contacts here, as we're going to need quick access to the information stored.

A table seems like it would be a good option, but first we need to think about what's going to be stored in our contact manager. Here's a list of possible items:

- Name
- Email address
- Phone number
- Address
- Website
- Notes
- Gravatar (A global avatar service from the creators of WordPress)

Not all of this information will need to be displayed in our Index view. Don't forget that we also have the option to click through to the contact so we can display more information there.

A sensible option seems to be name, email address, and phone number displayed in our table with a link to click through.

Open up your Index's partial, which is located at `assets/partials/index.html`. Currently, this file is completely blank, so let's add a page header to begin with:

```
<div class="page-header">
  <h1>All Contacts</h1>
</div>
```

Remember, we don't need to include a container around this, as our partial is nested within our app's main `index.html` file on the route and we've already included the container there.

```
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Email Address</th>
      <th>Phone Number</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Karan Bromwich</td>
      <td>karan@example.com</td>
      <td>01234 56734</td>
      <td><a href="#">View</a></td>
    </tr>
    <tr>
      <td>Declan Proud</td>
      <td>declan@example.com</td>
      <td>01234 567890</td>
      <td><a href="#">View</a></td>
    </tr>
  </tbody>
</table>
```

That looks like a pretty good structure to me, but it's not looking too great on our page. Like most components, Bootstrap does include styles for tables, but we need to include an extra class to activate them. Simply add the `table` class to our opening table tag and Bootstrap will tidy it up immediately by adding some much-needed borders and making it span the full width.

There are also some secondary classes we can include to add a bit of extra pizzazz to our table:

- `table-bordered`: includes a border around all sides of the table and all cells.
- `table-striped`: adds a grey background to alternating rows to make it easier to read.
- `table-hover`: changes the background of the row when hovered upon.
- `table-condensed`: removes some of the top and bottom padding, making it take up less vertical height.

Apart from these classes, there are also some classes that can be applied to rows or cells specifically, which color the background of the rows to give it some context:

- `active`: adds the hover state to the row
- `success`: colors the background green, indicating a successful action
- `info`: uses below to draw attention to the row or cell
- `warning`: indicates that an action may be required and colors the cell yellow
- `danger`: demonstrates an error or problem

For now, I'm just going to add the `table-striped` class, but it's up to you if you want to experiment with some of the other included classes.

Our table is beginning to look great. You'll notice, however, that on smaller screen sizes the table is cut off horizontally. To combat this, we need to wrap our table in another element that will allow it to scroll at smaller sizes:

```
<div class="table-responsive">
…
</div>
```

That's much better, as our content is no longer getting cut off or breaking our responsive layout. The last thing I want to do to our table is turn that view link into a button. Bootstrap comes with a plethora of button styles that we can take advantage of.

All buttons are combinations of the following classes:

- Default button class
- Context class
- Size class

Together, these give us all the control we need to pick the right button for the right occasion. Let's put these together to create a button that works within our table:

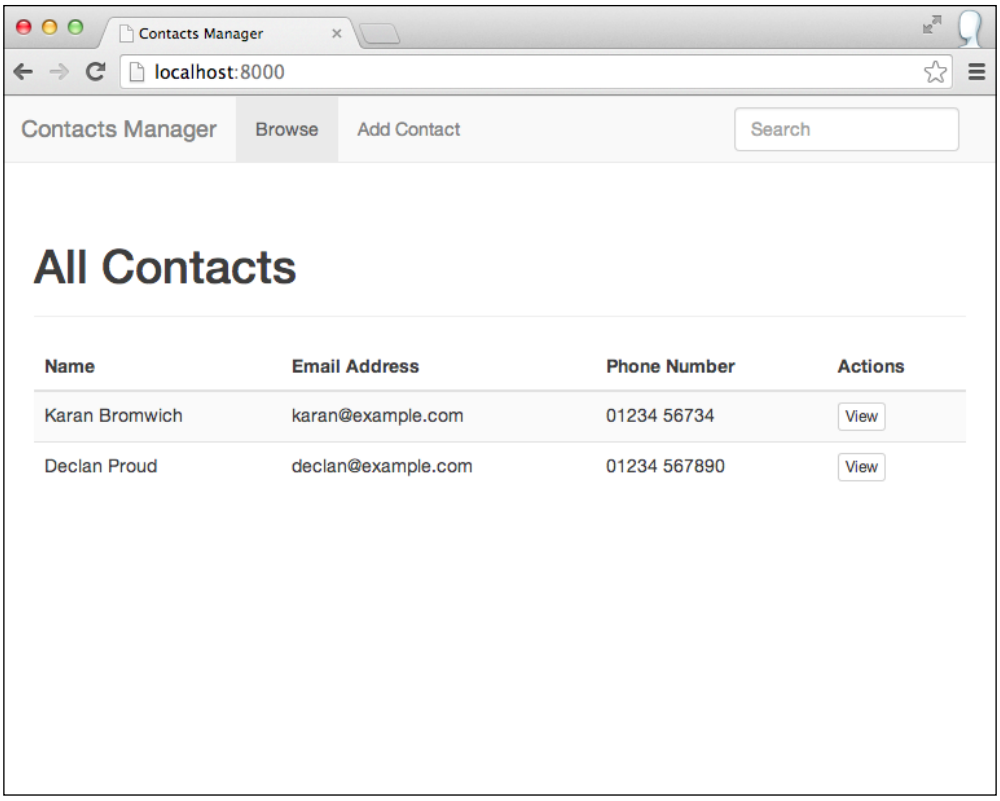
```
<a href="#" class="btn btn-default btn-xs">View</a>
```

Our first class here provides some default button styles; the second gives it color (in this case, the default is white), and the final class defines the size of the button. Alternatively, we could have used one of the following classes to change the color of our button:

Class Name	**Description**
`btn-default`	White button with a grey border
`btn-primary`	Blue button
`btn-success`	Green button
`btn-info`	Light blue button
`btn-warning`	Orange button
`btn-danger`	Red button
`btn-link`	Styled to look like a link

Apart from providing the default size, there are also three classes we can utilize to change the size of our buttons. In the preceding code, we've already used `btn-xs` to make our button really small, but we could have also used `btn-sm` to make it a little smaller than the default or `btn-lg` to make it larger.

Our Index view is looking pretty complete to me now, and it's ready to be populated when we're ready. Let's take a look at the finished product, as seen in the following image:



Populating the Add Contact view

It's quite clear what we're going to need in our Add Contact view—a form to allow us to enter the required information. Thankfully, Bootstrap provides us with a lot of control when arranging our fields. We've already worked out what data we're going to be storing, so it's just a case of working out what type of field is best:

- **Name**: Text field
- **Email address**: Email field
- **Phone number**: Tel field
- **Address**: Textarea
- **Website**: Text field
- **Notes**: Textarea
- **Gravatar**: N/A

As Gravatar uses an email address to serve images, we don't need to request any additional information here. We've got a total of six fields, so I think two columns would be great here.

The first thing we need to do is open up our form. Once we've done that, we can add our columns inside. We've already got our container class, so we just need to open up a new row and add our two columns. As we learnt in *Chapter 2*, *Let's Build with AngularJS and Bootstrap*, Bootstrap's grid-system is 12-columns wide, so we need to keep that in mind when creating our layout:

```
<form>
  <div class="row">
    <div class="col-sm-6">

    </div>
    <div class="col-sm-6">

    </div>
  </div>
</form>
```

Just as before, we're using the `col-sm` prefix to allow our columns to collapse down on smaller tablets and mobile devices.

We could just pop our labels and inputs directly within our columns, but for optimum spacing, we need to wrap our elements in a `div` tag with the `form-group` class:

```
<div class="form-group">
  <label for="name">Name</label>
  <input type="text" id="name">
</div>
```

To take advantage of Bootstrap's styles in our inputs, we do to add the `form-control` class to them. If we add the `control-label` class to our label, it will also give us a bit of extra padding:

```
<div class="form-group">
  <label for="name" class="control-label">Name</label>
  <input type="text" id="name" class="form-control">
</div>
```

Let's quickly add the rest of our elements in. I'm going to add name, phone number, and address to the left column and email address, website, and notes to the right.

Horizontal forms

Okay, that's looking great. If you're not a fan of the labels up top, we can position them on the left with a bit of tweaking. By adding the `form-horizontal` class to our opening form tag, our `form-group` classes behave as grid rows, meaning we can use column classes in the elements within them. Let me show you what all this means:

```
<div class="form-group">
  <label for="name" class="col-sm-4 control-label">Name</label>
  <div class="col-sm-8">
    <input type="text" id="name" class="form-control">
  </div>
</div>
```

After including form-horizontal, you'll notice we can now add Bootstrap's column classes to our label. As `form-control` sets the width to 100%, it matches the parent we need to wrap it in an additional element. As we've also included the `control-label` class, the label is centered vertically.

Our form looks a lot less cluttered using the `form-horizontal` class, so let's go ahead and wrap all of our inputs in that `form-control` element.

There might be times where you need to give the user a bit more information about what's required. This can be included underneath the related input by using a `span` tag with the `help-block` class:

```
<div class="form-group">
  <label for="notes" class="col-sm-4 control-label">Notes</label>
  <div class="col-sm-8">
    <textarea id="notes" class="form-control"></textarea>
    <span class="help-block">Any additional information about the
  contact.</span>
  </div>
</div>
```

The last thing we need to do is add a submit button. In previous versions of Bootstrap, this would usually have been wrapped in an element with the `form-actions` class. However, in Bootstrap 3, we just need to use the same form-group we've been using all along. If you're using the form-horizontal style, you'll need to offset your columns:

```
<div class="form-group">
  <div class="col-sm-offset-4 col-sm-8">
    <button class="btn btn-primary">Add Contact</button>
  </div>
</div>
```

As our label spans four columns, we need to offset our button the same amount here so it doesn't look misaligned.

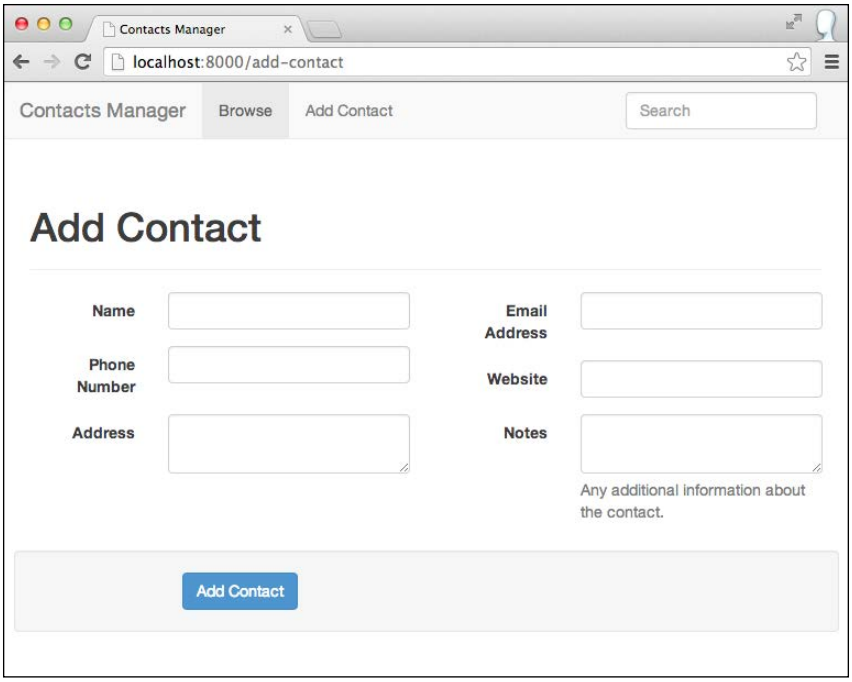
I quite liked the contrast the old form-actions class used to provide. Thankfully, we can achieve a similar result using Bootstrap's well component. Here, I've moved our `form-group` class containing the submit button to be directly below our existing row (remember, `form-horizontal` makes all groups act like rows) and have also added a `well` class:

```
<div class="form-group well">
  <div class="col-sm-offset-2 col-sm-10">
    <input type="submit" class="btn btn-primary" value="Add
      Contact">
  </div>
</div>
```

Finally, to complete the page, I'm going to give it the same `page-header` element we added to our Index view and position it at the very top of our markup:

```
<div class="page-header">
  <h1>Add Contact</h1>
</div>
```

The end result will look as seen in the following screenshot:



Populating the View Contact view

The final partial we need to populate is the screen where we can view our contact. I was tempted to just add this in as a form but I quite like the idea of having static text, which we can choose to edit certain sections of.

We're going to need to display all the same information we entered in the **Add Contact** view, as well as our Gravatar.

Title and Gravatar

To begin, we're going to include a `page-header` class. This is going to house an `h1` tag with our contact's name within:

```
<div class="page-header">
  <h1>Declan Proud</h1>
</div>
```

I also want to include our Gravatar here, so let's take a look at how we can achieve that. For now, we're just going to use some placeholder images from `http://placehold.it`. If you've not used this website before, it just serves up placeholders of any size. I think we just need a 50px x 50px image, and we can pull that in with the following:

```
<img src="http://placehold.it/50x50">
```

Feel free to tweak the size to something of your liking. We can slot this directly before the name of our contact within the `h1` tag. I'm also opting to add the `img-circle` class:

```
<div class="page-header row">
  <h1><img src="http://placehold.it/50x50" class="img-circle">
  Declan Proud</h1>
</div>
```

The class is one of three available to give some style to images and adds a 50% border radius to the image to create a circle. Also available is `img-rounded`, which rounds off the corners a little, as well as `img-thumbnail`, which adds a nice double border.

The form-horizontal class

We're quite lucky in that we can recycle quite a lot of what we did in the **Add Contact** view. The `form-horizontal` class will work just as well here with static content instead of fields. This page will later become our editing screen as well as our contact card, so it's handy that we can use the class for both views.

This time, however, we're going to use a `div` tag rather than a form element to wrap around our two-column layout:

```
<div class="form-horizontal">
  <div class="row">
    <div class="col-sm-6">
      ...
    </div>
    <div class="col-sm-6">
    ...
    </div>
  </div>
</div>
```

Other than that small change, you'll notice that the layout is identical. We've got our same row and 6-width columns from the view we created previously.

We can now also take advantage of form groups as well as control labels, which allow us to structure our label really nicely without the use of a list or table, as seen here:

```
<div class="form-group">
  <label for="name" class="col-sm-4 control-label">Name</label>
  <div class="col-sm-8">
    <p>Declan Proud</p>
  </div>
</div>
```

However, if you load this up in your browser, you'll notice that the label and our contact are misaligned. To prevent this, we can include an additional class in our `paragraph` tag to even things out:

```
<p class="form-control-static">Declan Proud</p>
```

Quickly add this in for the all of the fields. I'm following the same layout as before with name, phone number, and address in the left column, and our contact's email, website, and notes on the right.

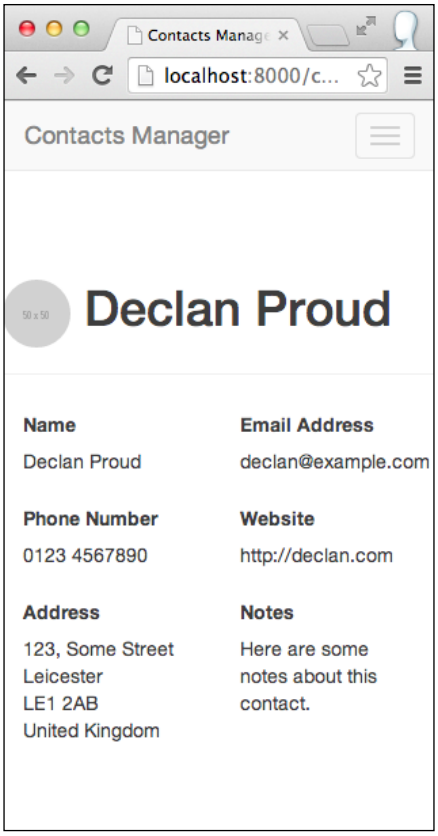
I'm pretty happy with that layout. However, when we scale it down, we seem to be left with a lot of white space on the right. It might be better if we change our main columns so that they stay in that layout all the way down to mobile.

I'm going to switch out our `col-sm-6` columns inside our row for the following:

```
<div class="col-xs-6">
…
</div>
```

This means we now get two columns on smaller devices and we don't have the issue with excessive white space.

You can see the same in the following screenshot:



Self-test questions

1. Why don't we need to include a container within our partials?
2. Besides table, what classes can be added to a table to give it some style?
3. How would we create a large, light blue button?
4. What do our labels and inputs need to be wrapped in?
5. How does `form-horizontal` change our form?
6. What would we use to display an additional help message by a form input?
7. What three classes can we apply to images and what do they do?

Summary

Okay, so we've got our three main layouts complete. We've achieved all of this without writing a single line of CSS by harnessing Bootstrap's core styles and components.

In our Index view, we saw how Bootstrap includes default styles and then uses secondary classes to add some extra flare. This is a pattern Bootstrap uses throughout, and we also saw it in action with our buttons and inputs.

With our **Add Contact** view, we had a look at the best ways to lay out a form and settled on using `form-horizontal`. This was something we could then recycle when creating our contact card.

In the next chapter, we're going to take a look at hooking up our views to dynamic data. We'll need to utilize AngularJS to create contacts, loop them through and display them in our table, edit them, and delete them.

6
CRUD

Until now, everything we've done has either been exploring Angular's paradigms or building the structure of our web app using Bootstrap. In this chapter, we're going to use the ideas and concepts we've studied over the course of the book to make our app work.

CRUD stands for Create, Read, Update, and Delete—everything we need to build a functional contact manager. We'll look at each letter of this acronym in detail and see how we can take full advantage of Angular.

Read

For the time being, we're going to ignore create and skip straight to read. We're going to use dummy data in the form of an array of objects in our controller. Let's first look at how this data is going to be formatted, and exactly what we need to include:

```
.controller('indexCtl', function($scope){

    $scope.contacts = [
        {
            name: 'Stephen Radford',
            phone: '0123456789',
            address: '123, Some Street\nLeicester\nLE1 2AB',
            email: 'stephen@email.com',
            website: 'stephenradford.me',
            notes: ''
        },
        {
            name: 'Declan Proud',
            phone: '91234859',
            address: '234, Some Street\nLeicester\nLE1 2AB',
```

```
            email: 'declan@declan.com',
            website: 'declanproud.me',
            notes: 'Some notes about the contact.'
        }
    ];

});
```

You'll notice we've attached the array directly to the scope, which, as we know, means we can now access it directly within our view. Using the `ng-repeat` directive we studied in *Chapter 2*, *Let's Build with AngularJS and Bootstrap*, we can loop through our array to display the contacts in our table:

```
<tr ng-repeat="contact in contacts">
  <td>{{contact.name}}</td>
  <td>{{contact.email}}</td>
  <td>{{contact.phone}}</td>
  <td><a href="/contact/{{$index}}" class="btn btn-default btn-
    xs">View</a></td>
</tr>
```

By now, I'm sure that this is looking very familiar to you. We've attached our directive to the element we wish to repeat (in this case, our table row), and have used the double curly brace syntax to display our contact data.

In our link, you'll notice that we've got something a little different. The `ng-repeat` directive allows us to grab the index of the current object using `$index`. You'll remember our single contact route accepts an ID number. We're using the index of our array as this ID so we can easily access the contact when we need to.

Sharing data between views

There's a slight problem with what we've got so far. Whilst it works perfectly for our Index view, it's completely useless when we want to display a single contact. This is because our contacts array is currently contained within our index controller and, therefore, cannot be shared.

Sharing data using $rootScope

There are a couple of ways we can share data between views, the first of them being `$rootScope`. Just as we have a scope for each of our views, the application itself also has one called root scope, and it works in exactly the same way.

To utilize it, we need to inject a new service, which is helpfully named `$rootScope`:

```
.controller('indexCtl', function($scope, $rootScope){

    $rootScope.contacts = [
        {
            name: 'Stephen Radford',
            phone: '0123456789',
            address: '123, Some Street\nLeicester\nLE1 2AB',
            email: 'stephen@email.com',
            website: 'stephenradford.me',
            notes: ''
        },
        {
            name: 'Declan Proud',
            phone: '91234859',
            address: '234, Some Street\nLeicester\nLE1 2AB',
            email: 'declan@declan.com',
            website: 'declanproud.me',
            notes: 'Some notes about the contact.'
        }
    ];

    $scope.contacts = $rootScope.contacts;

})
```

You'll notice that we can attach things to our root scope object in exactly the same way. Here, I've also added it to our view's scope, but I could have easily changed the view to access the root scope directly:

```
<tr ng-repeat="contact in $root.contacts">
```

To access anything on the root scope from the view, it's just a case of prefixing the model name with `$root` and a dot.

This method of sharing data does not really take advantage of all of the tools Angular gives us and creates quite a lot of noise in our application.

Using `$rootScope`, and in particular accessing it from the view, is bad practice and can make your project fairly unmaintainable; an application-wide controller or custom service should be used to share data as a preference.

Creating a custom service

A better way of sharing data across views is to build a custom service. A service is essentially a little class that we can access once it's injected into our controllers—just as we have seen with `$scope`.

There are three kinds of services within AngularJS: `.service()`, `.factory()`, and `.value()`. All of them act as singletons—design patterns that restrict it to only be instantiated to one object. We'll touch upon all of them before we build our own.

Value

The most basic of these three services is the `value` method. Let's add one to our module:

```
.value('demoService', 'abc123');
```

As you can see, it's a very simple service and accepts two parameters: the name of the service we wish to create and the value that service should hold. This data can then be shared across our application by injecting it into our controllers:

```
.controller('indexCtl', function($scope, demoService){
    $scope.demo = demoService;
});
```

It is very simple, but it provides a quick and easy way to share data. We might need to share an API key across multiple controllers, for example. Value would be perfect for that.

Factory

While value is nice and simple, it lacks a lot of features. Angular's factory service lets us call other services through DI, and it also provides service initialization and lazy initialization. Let's quickly rewrite that value example in factory:

```
.factory('demoService', function demoServiceFactory(){
  return 'abc123';
});
```

Notice that we've named the function here, `[serviceName]Factory`. While not necessary, it is the best practice, as it allows us to debug things a lot more easily in stack traces.

This will work, but for something so primitive, it is overkill, and value should be used instead. As we've discovered, factory can call other services, so let's create another and inject our initial `demoService`:

```
.factory('anotherService', function ['demoService',
  anotherServiceFactory(demoService){
  return demoService;
});
```

Our factory service can also modify the value provided to us by `demoService`, but a more likely use case would be using it to connect to an API. As with value, factory can return any JavaScript type. Let's take a look at returning an object:

```
.factory('anotherService', function ['demoService',
  anotherServiceFactory(demoService){
    return {
      connect: function(){
      }
    };
  });
```

The `connect` method defined in the returned object will be directly accessible when we inject our service into our controller:

```
.controller('indexCtl', function($scope, anotherService){
    anotherService.connect();
});
```

Service

The final service type in AngularJS is called service (quite confusing, just like the `Filter` filter). It produces a singleton just like value and factory, but it does this by invoking a constructor using the `new` operator. This all sounds confusing, so let's try to clear things up.

Here, we have a constructor function called `Notifier` that is just an alias for the browser's `window.alert` method:

```
function Notifier(){
  this.send = function(msg){
    window.alert(msg);
  }
}
```

We could add this directly within our controller and call it like so:

```
var notifier = new Notifier();
notifier.send('Hello, World');
```

While this would work, it's not ideal. What if we wanted to use our `Notifier` function again in another controller? As we know, we can share things like this using a service in AngularJS. With factory, we'd have to do something like this:

```
.factory('notifierService', function notifierFactoryService(){
  return new Notifier();
});
```

Not bad, but this is exactly the kind of thing service is designed for. It instantiates and returns our object for us. Let's take a look:

```
.service('notifierService', Notifier);
```

That's it! Angular will take our constructor, instantiate it, and return our object. We can inject this into our controller and use it as expected:

```
.controller('indexCtl', function($scope, notifierService){
    notifierService.send('Hello, World');
});
```

Rolling our own service

We could use service or factory to share our contacts across controllers, but as we're not instantiating anything, let's use factory:

```
.factory('contacts', function contactsFactory(){

})
```

As we've seen, it's only the returned object that contains our public methods and properties. We can take advantage of this by including the contacts array privately, only allowing it to be returned an accessed by our public methods:

```
var contacts = [
    …
];
```

Our service is going to include two methods to read contacts. The first is going to return our entire array, and the second, just a single contact object dependent on the index given to it. Let's quickly return our object containing those two methods and take a look at what they consist of:

```
return {
    get: function(){
        return contacts;
    },
    find: function(index){
        return contacts[index];
    }
};
```

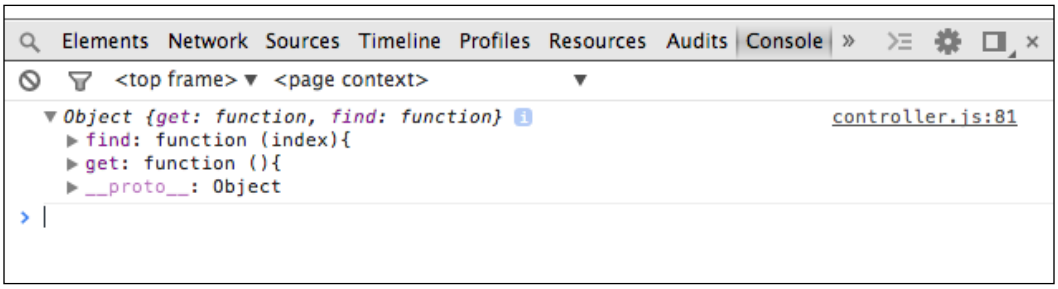
As you can see, both are very basic functions. The `get` method returns the entire array, and the `find` method accepts an index just returning the one contact requested. Let's piece all that together and see what we've got:

```
.factory('contacts', function(){
    var contacts = [
        {
            name: 'Stephen Radford',
            phone: '0123456789',
            address: '123, Some Street\nLeicester\nLE1 2AB',
            email: 'stephen@email.com',
            website: 'stephenradford.me',
            notes: ''
        },
        {
            name: 'Declan Proud',
            phone: '91234859',
            address: '234, Some Street\nLeicester\nLE1 2AB',
            email: 'declan@declan.com',
            website: 'declanproud.me',
            notes: 'Some notes about the contact.'
        }
    ];
    return {
        get: function(){
            return contacts;
        },
        find: function(index){
            return contacts[index];
        }
    };
})
```

We can now inject this service into our controller, just like we did with `$scope` and `$rootScope`:

```
.controller('indexCtl', function($scope, contacts){
    $scope.contacts = contacts.get();
})
```

I've also swapped our contacts in the scope over to the new method, which has really cleaned up our controller. If we add a `console.log` to our contacts service, we'll notice that we can't see the raw data, but only the two methods we've given access to.



Using route parameters

Now that we've got our service working successfully, we can begin to populate our single-contact view. Of course, we'll need to pull that ID out of the route in order to do this.

We briefly looked at how we access route parameters when we set up the route in *Chapter 4*, *Routing*, but let's quickly refresh on that. First of all, we need to inject another service into a single-contact controller: `$routeParams`. This service just returns an object with all of the parameters in our route as separate properties:

```
.controller('contactCtl', function($scope, $routeParams,
  contacts){
    $scope.contact = contacts.find($routeParams.id);
});
```

Here, we've accessed the `id` parameter and are using it to find the correct contact using the service we created earlier. We pass it to the view by creating a new model on the scope called `contact`.

Let's quickly pull all of the relevant information out in our `contact.html` partial. Remember all of our data is a property of the model contact, so we can access it like so:

```
{{contact.name}}
```

Everything's looking great, except for a couple of minor details. The address information really could do with respecting those line-endings, and I'd like to pull out the Gravatar dynamically. To accomplish both of these things, we're going to need to create a filter and a directive.

Creating a custom directive

We've seen the power of directives, and until now, we've had no reason to write our own. However, in order to include our `gravatar`, building a custom directive is the solution.

Just like we've done with our controllers, filters, and services, our new directive needs to be attached to the module by using the relevant method:

```
.directive('gravatar', function(){

})
```

Just like controllers, filters, and services, the `directive` method requires two parameters. The first is the name of our directive, and the second is a function. A directive must return an object, and the properties of the returned object define how the directive behaves.

The first property we're going to set is `restrict`. This defines how the directive can be used. We've already seen how most directives can be used as attributes or custom elements, but Angular also allows us to use them in a couple of other ways. The following values can be set to the `restrict` property:

- `A`: This restricts the directive to be attached using an attribute, `<div gravatar></div>`
- E: This allows the directive to be used as a custom element, `<gravatar></gravatar>`
- `C`: This allows the directive to be used by adding it as a class to the element, `<div class="gravatar"></div>`
- `M`: Allows the directive to be executed via an HTML comment, `<!-- directive: gravatar -->`

It's recommended that attributes and elements be used in favor of classes and comments for directives.

By default, Angular sets this to be an attribute only. However, we can use combinations of the aforementioned values to fine-tune how we want our directive to be used. Let's set it as `AE` so we can call it via an attribute or custom element:

```
.directive('gravatar', function(){
    return {
        restrict: 'AE'
    }
})
```

We can also opt to create a template for our directive if required. This can be included directly within our object using the `template` property, or, by using the `templateUrl` property, we can load an external template file from the specified URI. As we're only creating an `img` tag, we might as well add it directly into our object:

```
.directive('gravatar', function(){
    return {
        restrict: 'AE',
        template: '<img ng-src="{{img}}" class="{{class}}">'
    }
})
```

This template behaves just as our views do. I've added in two placeholders for our image's URI and also any classes we want to include.

To hook everything up, we just need to attach a function to the `link` property in our object. From here, we can access the scope, the element the directive is attached to, as well as any attributes on that element:

```
.directive('gravatar', function(){
    return {
        restrict: 'AE',
        template: '<img src="{{img}}" class="{{class}}">',
        link: function(scope, elem, attrs){

        }
    }
})
```

In order to fetch our Gravatar image, we need to hash our contact's email address using `md5`. Unfortunately, this isn't a method native to JavaScript, so we'll need to include a separate library. I've included one in the downloadable assets that accompany this chapter, which can be included as a single-line variable:

```
.directive('gravatar', function(){
    return {
```

```
        restrict: 'AE',
        template: '<img src="{{img}}" class="{{class}}">',
        replace: true,
        link: function(scope, elem, attrs){
            var md5=function(s){function
L(k,d){return(k<<d)|(k>>>(32-d))}function K(G,k){var
I,d,F,H,x;F=(G&2147483648);H=(k&2147483648);I=(G&1073741824);d=(k&
1073741824);x=(G&1073741823)+(k&1073741823);if(I&d){return(x^21474
83648^F^H)}if(I|d){if(x&1073741824){return(x^3221225472^F^H)}else{
return(x^1073741824^F^H)}}else{return(x^F^H)}}function
r(d,F,k){return(d&F)|((~d)&k)}function
q(d,F,k){return(d&k)|(F&(~k))}function
p(d,F,k){return(d^F^k)}function
n(d,F,k){return(F^(d|(~k)))}function
u(G,F,aa,Z,k,H,I){G=K(G,K(K(r(F,aa,Z),k),I));return
K(L(G,H),F)}function
f(G,F,aa,Z,k,H,I){G=K(G,K(K(q(F,aa,Z),k),I));return
K(L(G,H),F)}function
D(G,F,aa,Z,k,H,I){G=K(G,K(K(p(F,aa,Z),k),I));return
K(L(G,H),F)}function
t(G,F,aa,Z,k,H,I){G=K(G,K(K(n(F,aa,Z),k),I));return
K(L(G,H),F)}function e(G){var Z;var F=G.length;var x=F+8;var k=(x-
(x%64))/64;var I=(k+1)*16;var aa=Array(I-1);var d=0;var
H=0;while(H<F){Z=(H-
(H%4))/4;d=(H%4)*8;aa[Z]=(aa[Z]|(G.charCodeAt(H)<<d));H++}Z=(H-
(H%4))/4;d=(H%4)*8;aa[Z]=aa[Z]|(128<<d);aa[I-2]=F<<3;aa[I-
1]=F>>>29;return aa}function B(x){var
k="",F="",G,d;for(d=0;d<=3;d++){G=(x>>>(d*8))&255;F="0"+G.toString
(16);k=k+F.substr(F.length-2,2)}return k}function
J(k){k=k.replace(/rn/g,"n");var d="";for(var
F=0;F<k.length;F++){var
x=k.charCodeAt(F);if(x<128){d+=String.fromCharCode(x)}else{if((x>1
27)&&(x<2048)){d+=String.fromCharCode((x>>6)|192);d+=String.fromCh
arCode((x&63)|128)}else{d+=String.fromCharCode((x>>12)|224);d+=Str
ing.fromCharCode(((x>>6)&63)|128);d+=String.fromCharCode((x&63)|12
8)}}}return d}var C=Array();var P,h,E,v,g,Y,X,W,V;var
S=7,Q=12,N=17,M=22;var A=5,z=9,y=14,w=20;var
o=4,m=11,l=16,j=23;var
U=6,T=10,R=15,O=21;s=J(s);C=e(s);Y=1732584193;X=4023233417;W=25623
83102;V=271733878;for(P=0;P<C.length;P+=16){h=Y;E=X;v=W;g=V;Y=u(Y,
X,W,V,C[P+0],S,3614090360);V=u(V,Y,X,W,C[P+1],Q,3905402710);W=u(W,
V,Y,X,C[P+2],N,606105819);X=u(X,W,V,Y,C[P+3],M,3250441966);Y=u(Y,X
,W,V,C[P+4],S,4118548399);V=u(V,Y,X,W,C[P+5],Q,1200080426);W=u(W,V
,Y,X,C[P+6],N,2821735955);X=u(X,W,V,Y,C[P+7],M,4249261313);Y=u(Y,X
,W,V,C[P+8],S,1770035416);V=u(V,Y,X,W,C[P+9],Q,2336552879);W=u(W,V
,Y,X,C[P+10],N,4294925233);X=u(X,W,V,Y,C[P+11],M,2304563134);Y=u(Y
,X,W,V,C[P+12],S,1804603682);V=u(V,Y,X,W,C[P+13],Q,4254626195);W=u
(W,V,Y,X,C[P+14],N,2792965006);X=u(X,W,V,Y,C[P+15],M,1236535329);Y
=f(Y,X,W,V,C[P+1],A,4129170786);V=f(V,Y,X,W,C[P+6],z,3225465664);W
```

```
=f(W,V,Y,X,C[P+11],y,643717713);X=f(X,W,V,Y,C[P+0],w,3921069994);Y
=f(Y,X,W,V,C[P+5],A,3593408605);V=f(V,Y,X,W,C[P+10],z,38016083);W=
f(W,V,Y,X,C[P+15],y,3634488961);X=f(X,W,V,Y,C[P+4],w,3889429448);Y
=f(Y,X,W,V,C[P+9],A,568446438);V=f(V,Y,X,W,C[P+14],z,3275163606);W
=f(W,V,Y,X,C[P+3],y,4107603335);X=f(X,W,V,Y,C[P+8],w,1163531501);Y
=f(Y,X,W,V,C[P+13],A,2850285829);V=f(V,Y,X,W,C[P+2],z,4243563512);
W=f(W,V,Y,X,C[P+7],y,1735328473);X=f(X,W,V,Y,C[P+12],w,2368359562)
;Y=D(Y,X,W,V,C[P+5],o,4294588738);V=D(V,Y,X,W,C[P+8],m,2272392833)
;W=D(W,V,Y,X,C[P+11],l,1839030562);X=D(X,W,V,Y,C[P+14],j,425965774
0);Y=D(Y,X,W,V,C[P+1],o,2763975236);V=D(V,Y,X,W,C[P+4],m,127289335
3);W=D(W,V,Y,X,C[P+7],l,4139469664);X=D(X,W,V,Y,C[P+10],j,32002366
56);Y=D(Y,X,W,V,C[P+13],o,681279174);V=D(V,Y,X,W,C[P+0],m,39364300
74);W=D(W,V,Y,X,C[P+3],l,3572445317);X=D(X,W,V,Y,C[P+6],j,76029189
);Y=D(Y,X,W,V,C[P+9],o,3654602809);V=D(V,Y,X,W,C[P+12],m,387315146
1);W=D(W,V,Y,X,C[P+15],l,530742520);X=D(X,W,V,Y,C[P+2],j,329962864
5);Y=t(Y,X,W,V,C[P+0],U,4096336452);V=t(V,Y,X,W,C[P+7],T,112689141
5);W=t(W,V,Y,X,C[P+14],R,2878612391);X=t(X,W,V,Y,C[P+5],O,42375332
41);Y=t(Y,X,W,V,C[P+12],U,1700485571);V=t(V,Y,X,W,C[P+3],T,2399980
690);W=t(W,V,Y,X,C[P+10],R,4293915773);X=t(X,W,V,Y,C[P+1],O,224004
4497);Y=t(Y,X,W,V,C[P+8],U,1873313359);V=t(V,Y,X,W,C[P+15],T,42643
55552);W=t(W,V,Y,X,C[P+6],R,2734768916);X=t(X,W,V,Y,C[P+13],O,1309
151649);Y=t(Y,X,W,V,C[P+4],U,4149444226);V=t(V,Y,X,W,C[P+11],T,317
4756917);W=t(W,V,Y,X,C[P+2],R,718787259);X=t(X,W,V,Y,C[P+9],O,3951
481745);Y=K(Y,h);X=K(X,E);W=K(W,v);V=K(V,g)}var
i=B(Y)+B(X)+B(W)+B(V);return i.toLowerCase()};
        }
    }
})
```

Now that we've included our `md5` function, we can access our contact's image from Gravatar. We're going to pass through two things to our link function. The first will be the email address, and the second optional parameter will be the size of the image we want to fetch.

The attributes passed to our function are just objects we can access. For example, if we wanted to fetch the value of the email attribute, we could access it using `attrs.email`.

We've also defined the `replace` property. This will replace the element we've bound the directive to with the template specified. By default, Angular will append the template as a child element.

Let's quickly finish up here and try our new directive out:

```
.directive('gravatar', function(){
    return {
        restrict: 'AE',
```

```
        template: '<img src="{{img}}" class="{{class}}">',
        link: function(scope, elem, attrs){
            var md5 = function(s){ ... };
            var size = (attrs.size) ? attrs.size : 64;
            scope.img = 'http://gravatar.com/avatar/'+md5(attrs.
email)+'?s='+size;
            scope.class = attrs.class;
        }
    }
})
```

We've used a ternary operator to optionally set the size of our image. I've also attached the classes assigned to our element, as well as pieced together the Gravatar URL to our scope.

Our directive is ready. Let's try using it via an attribute first:

```
<div gravatar email="{{contact.email}}" size="50" class="img-
  circle"></div>
```

Great, everything seems to be working. The Gravatar is being displayed and our class is being included to give us a nice circular image. Unfortunately, it looks like our image is being wrapped in the `div` tag we attached our directive to and is, therefore, being pushed onto a new line. That's because we haven't told Angular we want to replace the existing element with our compiled template. We can do this by setting the `replace` property on our directive's object to true:

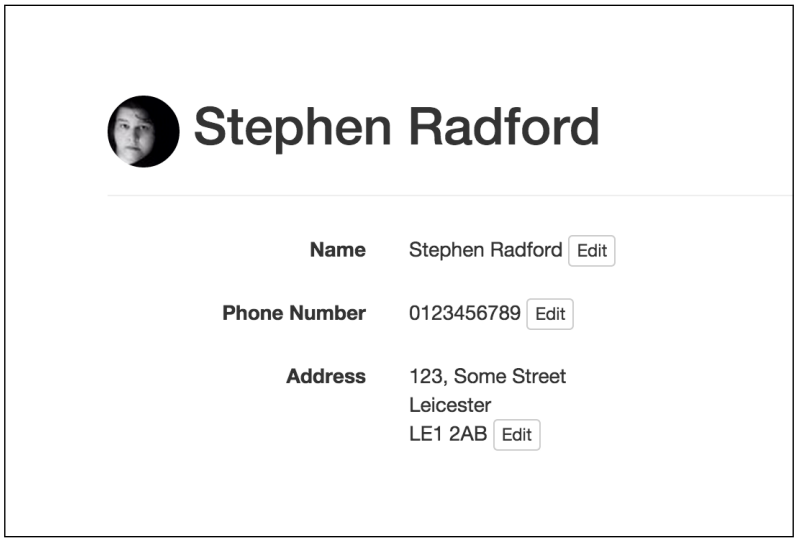
```
.directive('gravatar', function(){
    return {
        restrict: 'AE',
        template: '<img src="{{img}}" class="{{class}}">',
        replace: true,
        link: function(scope, elem, attrs){
            var md5=function(s){ … };
            var size = (attrs.size) ? attrs.size : 64;
            scope.img = 'http://gravatar.com/avatar/'+md5(attrs.
email)+'?s='+size;
            scope.class = attrs.class;
        }
    }
})
```

If we refresh our browser, we'll notice that instead of the image being wrapped in the original element, it's now replaced with our image. Alternatively, we could have called our directive through a custom element:

```
<gravatar email="{{contact.email}}" size="50" class="img-
  circle"></gravatar>
```

Unless we specifically need to support IE8 (no longer supported in AngularJS 1.3+), directives that insert new elements through the use of a template should be called with a custom element as seen in the preceding example. Directives that manipulate existing elements, perhaps calling a jQuery plugin, for example, should be restricted to just being called via an attribute.

Here's what our new directive looks like when viewing one of the contacts:



Respecting line-endings

Currently our address and notes fields aren't respecting line-endings, and this is because the new lines need to be converted into HTML line-breaks. Fortunately, Angular makes this super easy for us through a custom filter.

As we've already covered creating a filter in *Chapter 3*, *Filters*, I'm just going to quickly show you the `paragraph` filter we need to create to convert the `\n` line-ending into `<br />`:

```
.filter('paragraph', function(){
    return function(input){
        return (input) ? input.replace(/\n/g, '<br />') : input;
    };
})
```

If we add this filter to our address using the pipe-symbol syntax, you'll notice something a little odd. The page breaks are actually being converted to HTML entities and displayed on the page. For security, Angular will automatically escape our HTML to stop things like cross site scripting.

As we clearly don't want these displayed, we need to use an included directive to bind our model to the page. The `ng-bind-html` directive will achieve exactly what we like. Here it is in action in our `paragraph` tag:

```
<p class="form-control-static" ng-bind-html="contact.address |
  paragraph"></p>
```

Wait a second, that's still not working. If we check our console, Angular is throwing the following error:

```
Error: [$sce:unsafe] Attempting to use an unsafe value in a safe
  context.
```

This is because Angular requires the `ngSanitize` module, which can be downloaded from the **Extras** section at `https://angularjs.org/`. The module filters out dangerous snippets of code, such as scripts, and leaves behind a clean, sanitized output.

Grab this from Angular's website, add it to your `js` directory, and include it in your main `index.html` file:

```
<script type="text/javascript" src="/assets/js/angular-
  sanitize.min.js"></script>
```

Once the module is included on the page, we need to include it as a dependency of our module, just as we did with `ngRoute`:

```
angular.module('contactsMgr', ['ngRoute', 'ngSanitize'])
```

If you refresh your page, you'll notice that the address is now split over multiple lines, as expected.

Search and adding the active page class

The last section we need to cover under our *Read* heading is the search. As our search is just going to filter our table on the Index view, we need to redirect it when we start typing. The model we use also needs to be accessible from anywhere.

We also need to take a look at sorting out that active page class. Currently, it's stuck on *Browse*, and it would be nice to make this dynamic.

Both of these things are outside of `ng-view` so we're going to need to create a controller for our entire application. This means we'll also be able to access that search model from anywhere.

```
.controller('appCtl', function($scope, $location){

});
```

As we need to redirect the page, I've injected in the `$location` service. This gives us access to the `path` method, which we can use to do exactly what we need. Unlike our route controllers, we need to call this one on the page by adding it to our opening `HTML` tag:

```
<html lang="en" ng-app="contactsMgr" ng-controller="appCtl">
```

Search

Now that our controller is initialized, we can get to work. The `ng-keyup` directive will fire as soon as we start typing in the search box, redirecting us to the Index view.

First, let's add our handler to the `controller`:

```
.controller('appCtl', function($scope, $location){

    $scope.startSearch = function(){
        $location.path('/');
    };

});
```

It's a pretty simple and self-explanatory function. Once we add our directive to the search box, it's going to change the current path to the Index view. So, let's go ahead and hook that up now. If you haven't already, now's a good time to assign a model to the search input as well:

```
<form class="navbar-form navbar-right" role="search">
    <input type="text" class="form-control" placeholder="Search" ng-
      model="search" ng-keyup="startSearch()">
</form>
```

With our redirect sorted, we just need to filter `ng-repeat` from earlier:

```
<tr ng-repeat="contact in contacts | filter:search">
```

Remember, if you want to restrict this to just name, then you'll need to change the `model` in your search box to the following:

```
<input type="text" class="form-control" placeholder="Search" ng-
  model="search.name" ng-keyup="startSearch()">
```

The active page class

Finally, we need to sort those active page classes out. We just need to check the current path and add a class if necessary. All of this can be achieved with `ng-class` and a function in our app controller.

Our function is just going to check if the current path is the same as the one passed through to it:

```
$scope.pageClass = function(path){
    return (path == $location.path()) ? 'active' : '';
};
```

If it's a match, we're returning our `active` class; otherwise, we're returning nothing. We can now add that to both of our navigation elements:

```
<li ng-class="pageClass('/')"><a href="/">Browse</a></li>
<li ng-class="pageClass('/add-contact')"><a
  href="/add-contact">Add Contact</a></li>
```

Once the `ng-class` directive is added to our list element, the current page being shown by the `active` class becomes completely dynamic and correct.

Create

We skipped past the first letter in the CRUD acronym, but now it's time to jump back and hook up our add contact form. The first thing we need to do is ensure our form's inputs all have the relevant model attached:

```
<input type="text" id="name" class="form-control" ng-
  model="contact.name">
```

As we've created a service to handle our contacts, it makes sense that we extend upon this to allow us to create contacts using it as well. Let's make a `create` method that pushes the contact into the array and add it to our service's object:

```
create: function(contact){
    contacts.push(contact);
}
```

We can now think about exactly what we want to happen when we submit our form. We're going to want to insert one contact into our array using the method we just created in our service. Apart from that, we're also going to need to provide some feedback that this has actually worked, and finally, clear the form:

```
$scope.submit = function(){
    contacts.create($scope.contact);
    $scope.contact = null;
    $scope.added = true;
};
```

Here, we've done it all: pushed the contact over to our service, reset the contact model, and set a model that we can observe to provide some feedback.

There are a couple of ways we can call the function we've just created. We could add it as an `ng-click` directive to the submit button, but it's probably wiser and more accessible to add it to an `ng-submit` directive in the form itself:

```
<form class="form-horizontal" ng-submit="submit()">
```

It's now just a case of including that alert box to let the user know that their contact was added successfully. We want it hidden by default, so by using `ng-show` and watching our added `model`, we can choose when we want to display it:

```
<div class="alert alert-success" ng-show="added">
  The contact was added successfully.
</div>
```

Update

As you'll recall, we didn't create a view exclusively to edit our contacts. We can either use the same partial we used to add contacts, or alternatively do something a little more fun with the single-contact view and create our own directive to edit the data.

It's unlikely that you'll ever want to edit all aspects of a contact at once; often, it's only a phone number or email address that's changed. I like the idea of displaying our text alongside a little **Edit** button. When clicked, this will allow us to edit this section of the contact.

Let's call the directive `editable` and start it up exactly as we did before:

```
.directive('editable', function(){
    return {

    };
})
```

Again, I think it's wise to allow the choice of including this as a custom element or using the directive via an attribute. We're going to be using it exclusively as an attribute for now, but you never know what a project may require in the future:

```
.directive('editable', function(){
    return {
        restrict: 'AE',
        templateUrl: '/assets/partials/editable.html'
    };
})
```

We're going to need a lot more than just one line of markup this time, so I've gone ahead and switched over to using the `templateUrl` property instead. Utilizing this is just a case of creating the reference partial, as we do with our routes.

Scope

As well as switching over to using a URL for our template, we're going to take a look at a new property: `scope`. This property gives us more control of the scope we want to use with our directive.

Setting this as a hash here will create a new isolate scope. This doesn't prototypically inherit from its parent, which means we don't have to worry about accidentally reading or editing unwanted data from our view's scope. Here, I've added two values to our `scope` hash.

```
.directive('editable', function(){
    return {
        restrict: 'AE',
        templateUrl: '/assets/partials/editable.html',
        scope: {
            value: '=editable',
            field: '@fieldType'
        }
    };
})
```

The key is the name we're assigning to the new scope, and the value is an attribute in our element. You'll also notice that we've prefixed both the values differently, which means both will behave in very different ways.

Note that attributes separated with a hyphen are converted to CamelCase by Angular.

When we prefix it with an `=` sign, we can directly bind a model to our parent scope to our directive's scope. This means we don't need to use the `{{}}` syntax, and we can take advantage of that two-way data binding. We'll need this as we're going to be editing the value of the bound model within our directive.

If prefixed with the `@` symbol, our directive will use the literal value of that attribute. We can use the `{{}}` syntax to pass through a model's value or just enter a string. No model is bound when we use the `@` symbol.

Controller

We've seen previously how we can use the `link` method in our directive, but we also have another one at our disposal. The `controller` method acts and behaves just like a controller directly attached to your module. We can inject any services required and it will feel very familiar.

The difference with `link` is the order in which it's processed. Our controller is run before our application has finished compiling; the `link` method after. We should always opt to use controller unless we're creating a wrapper directive for a jQuery plugin or something that needs to run after everything has finished loading. We used `link` for our Gravatar directive earlier, as I wanted to highlight the differences between the two ways of doing things inside our directive. Add the `controller` method to your directive. At this stage is should look like the following:

```
.directive('editable', function(){
    return {
        restrict: 'AE',
        templateUrl: '/assets/partials/editable.html',
        scope: {
            value: '=editable',
            field: '@fieldType'
        },
        controller: function($scope){

        }
    };
})
```

Using the `controller` property in our directive also acts as a sort of API, allowing other directives to communicate with one another, whereas `link` does not.

Piecing it together

Okay, we've set up our scope and controller. Now it's time to populate that partial and figure out our functionality. Let's first of all display that model's value and include our **Edit** button that's going to activate our editor:

```
<span ng-bind-html="value | paragraph"></span> <button class="btn
  btn-default btn-xs">Edit</button>
```

Remember, we aliased the name of our editable attribute to be `value`, so that's what we're using here. We also need to ensure that we cater to both single-line and multi-line field types; hence, we're using `ng-bind-html` and our paragraph filter.

We're going to use `ng-show` and `ng-hide` to watch a model in our directive's scope. I also think it's a good idea to allow the user to cancel their changes, so we're not going to want to edit that value we passed through directly. Adding the following to our controller creates a new model we can edit, as well as giving us something `ng-show` and `ng-hide` can keep an eye on:

```
$scope.editor = {
    showing: false,
    value: $scope.value
};
```

We can use the `editor.showing` model to create two sections in our template. One section will be displayed before we click **Edit**, and one after:

```
<div ng-hide="editor.showing">
  <span ng-bind-html="value | paragraph"></span> <button class="btn
    btn-default btn-xs">Edit</button>
</div>
<div ng-show="editor.showing">

</div>
```

Let's create the function we're going to use to show or hide the editor, which we'll call from `ng-click` once we're done:

```
$scope.toggleEditor = function(){
    $scope.editor.showing = !$scope.editor.showing;
};
```

Now, let's hook that up to our Edit button in the partial:

```
<span ng-bind-html="value | paragraph"></span> <button class="btn
  btn-default btn-xs" ng-click="toggleEditor()">Edit</button>
```

Our directive is going to allow us to choose the type of input we require (such as text, email, textarea, and so on) but I'd like the default to be a single-line text box as we're using that most frequently. Here, we've used a ternary operator to check if a value has been set or if the default should be used:

```
$scope.field = ($scope.field) ? $scope.field : 'text';
```

Recently added in AngularJS 1.2 is the `ng-if` directive. Unlike `ng-show` and `ng-hide`, it actually attaches or detaches elements from the **Document Object Model** (**DOM**) if they don't match our condition; so it's perfect to check the type of field here:

```
<div ng-show="editor.showing">
  <div ng-if="field == 'textarea'">
    <textarea ng-model="editor.value" class="form-
      control"></textarea>
  </div>
  <div ng-if="field != 'textarea'">
    <input type="{{field}}" ng-model="editor.value" class="form-
      control">
  </div>
</div>
```

As you can see, it's equally simple to use and gives us complete control over whether we want to use a textarea or input element. I've also gone ahead and populated our input tag's `type` attribute with the value passed through and hooked both elements up to our new model.

The last thing we need to include in our template is a couple of buttons to save or cancel. I think it'll be nice to separate these with a horizontal rule tool:

```
<div ng-hide="editor.showing">
  <span ng-bind-html="value | paragraph"></span> <button class="btn
    btn-default btn-xs" ng-click="toggleEditor()">Edit</button>
</div>
<div ng-show="editor.showing">
  <div ng-if="field == 'textarea'">
    <textarea ng-model="editor.value" class="form-
      control"></textarea>
  </div>
  <div ng-if="field != 'textarea'">
```

```
      <input type="{{field}}" ng-model="editor.value" class="form-
        control">
    </div>
    <hr>
    <button class="btn btn-success btn-xs" ng-
      click="save()">Save</button>
    <button class="btn btn-default btn-xs" ng-
      click="toggleEditor()">Cancel</button>
  </div>
```

I've hooked the save button up to a new save function we've yet to create, and that cancel button uses the same `toggleEditor` function from earlier.

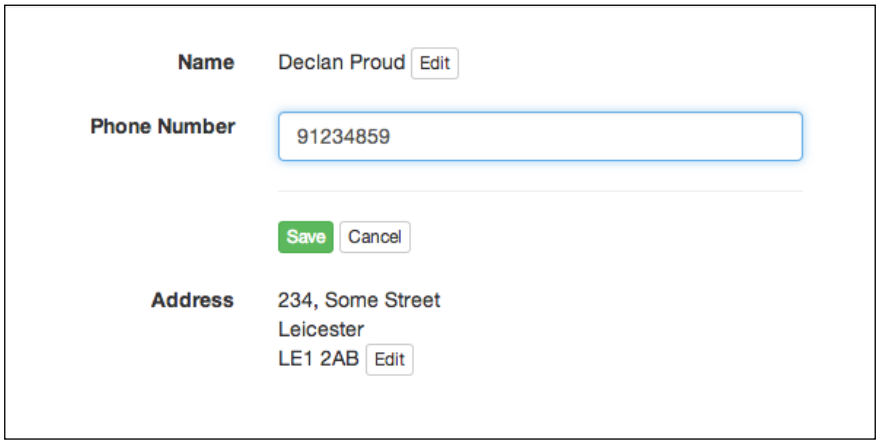
The save function is simple. It just assigns the new model we created with the one we bound to our directive earlier and then calls the `toggleEditor` function to hide everything away:

```
$scope.save = function(){
    $scope.value = $scope.editor.value;
    $scope.toggleEditor();
};
```

That finishes up our editable directive nicely, but how do we use it? In our `contact.html` partial, we earlier displayed all the models within our paragraph tags using the double curly brace syntax. Now that our directive is doing all that for us, we can replace the contents of our `<p>` tag with and add a couple of attributes to it instead:

```
<p class="form-control-static" editable="contact.email" field-
  type="email"></p>
```

Once you've replaced all of your models, you should now have an easy-to-use, visual editor for each section of your contact.



Delete

Deleting our contacts can once again be achieved through our service. We can create a final method that will accept an index of our array and remove it. Since delete is a keyword in JavaScript, let's use `destroy` as the name for our method:

```
destroy: function(index){
    contacts.splice(index, 1);
}
```

It's a very simple method. We're just taking the index and using the native `splice` method to remove it from the array. Now, we need to create a function on our index view's scope that can call this method from our service:

```
$scope.delete = function(index){
    contacts.destroy(index);
};
```

Finally, we can add a button to our actions column in our table to delete the required contact on click:

```
<button class="btn btn-danger btn-xs" ng-
  click="delete($index)">Delete</button>
```

Self-test questions

1. Name two ways we can share data between views.
2. What are the three types of services available?
3. What do we need to include in order to utilize `ng-bind-html`?
4. What's the difference between the link and controller methods of a directive?
5. When using isolate scope, what do the prefixes `=` and `@` mean?
6. How would we restrict a directive to an element and a comment?
7. Why did we need to make a controller for our entire application?
8. How would we get the index of an object from `ng-repeat`?

Summary

We covered a huge amount in this chapter, so it's definitely a good idea to summarize it. In the grand scheme of things, we transformed what were static templates into a fully functioning web app that allows us to Create, Read, Update, and Delete.

In doing this, we explored a lot more. We discovered the best way to share data between views by creating a custom service to handle our contacts. The completed service allows us to fetch all contacts, find a single contact, add a new contact, and delete a contact from anywhere in our application.

Along with creating a custom service, we also took a look at how we could make our own directives. The first one we looked at allowed us to display a Gravatar image based on an email address. We discovered the many different ways a directive can be used, be it via an attribute or even an HTML comment.

The second directive we created was more detailed. We identified isolate scopes and the difference between a directive's controller and link methods; we also built a super-awesome way to edit our contact's data.

In the next chapter, we're going to look at the third-party AngularStrap library that will enable us to take advantage of Bootstrap's plugins within Angular.

7
AngularStrap

We've already taken a look at the huge number of components that Bootstrap offers, but we've yet to look at how we can utilize the JavaScript plugins available to us. We could create our own directives for each of the plugins we wish to use, but the Angular community already has a module full of them called AngularStrap.

In this chapter, we'll take a look at AngularStrap, the Bootstrap plugins it allows us to use, and how we can utilize these in our application.

Installing AngularStrap

First of all, you need to download AngularStrap. You can fetch it from `http://mgcrea.github.io/angular-strap/`. Click on the download button on the top right and download the latest version as a ZIP file.

Within the ZIP file are all of the single modules, as well as everything in a handy minified file. You'll find the two files we're looking for inside the `dist` directory. Copy `angular-strap.min.js` and `angular-strap.tpl.min.js` into your project's `js` directory. Once you've done that, include them in your root `index.html` file after Angular and before your project module:

```
<script type="text/javascript" src="/assets/js/angular-
strap.min.js"></script>
<script type="text/javascript" src="/assets/js/angular-
strap.tpl.min.js"></script>
```

As with every module, we need to inject this into our application. Our declaration is located in the first line of `controller.js`; the name of the AngularStrap module is `mgcrea.ngStrap`. Below we've added AngularStrap as a dependency on our `contactsMgr` module:

```
angular.module('contactsMgr', ['ngRoute', 'ngSanitize',
  'mgcrea.ngStrap'])
```

That's not all we need to do here, unfortunately. AngularStrap depends on the `ngAnimate` module that we're yet to include in our project. We can find this under the **Extras** link in the download modal window at `https://angularjs.org/`.

Add the minified version to your project's `js` directory and include it before AngularStrap:

```
<script type="text/javascript" src="/assets/js/angular-
  animate.min.js"></script>
```

The `ngAnimate` module doesn't need to be injected into our project unless we want to use it outside of the Bootstrap directives. It allows us to use CSS animations on things such as `ngShow` and `ngHide` so that we can fade things rather than having them just appear.

If we wanted, we could build our own animations to use alongside AngularStrap. However, the AngularStrap creator also maintains AngularMotion as a perfect companion. AngularMotion is simply a style sheet that contains pre-made animation ready for use with the `ngAnimate` module.

We can download the latest release from `http://mgcrea.github.io/angular-motion/` using the download button at the top right. Again, the ZIP file we're provided gives us the source files as well as the minified production-ready version, which we can find in the `dist` directory. Copy this over to your project and include it as a secondary style sheet on the page:

```
<link rel="stylesheet" href="/assets/css/angular-motion.min.css">
```

We'll now be able to use the fade, slide, scale, and flip animations provided by AngularMotion in tandem with the AngularStrap directives.

You may find it strange that we don't need to include Bootstrap's plugins script at all. This is because the directives we've included with AngularStrap aren't just wrapper functions that execute jQuery, but instead complete rewrites that fully take advantage of Angular.

Using AngularStrap

Now that we've installed AngularStrap, let's take a look at some of the plugins it offers and how we use them.

Before we begin, let's quickly set up a demo environment. Let's duplicate our `index.html` file and rename it `demo.html`. We will also change our controller to `demoCtrl` and add that to our `controller.js` file. This gives us a nice clean canvas we can play with.

The modal window

A modal window is an extremely common UI paradigm within web apps. It's a great way to display a small amount of information without taking the user out to a new page.

The AngularStrap modal can be called on the click of a button with the `bs-modal` directive applied to it:

```
<button class="btn btn-primary" bs-modal="modal">Show
  Modal</button>
```

The value passed to it is a model on our scope. It's a hash that contains two values: `title` and `content`. Here it is within our controller:

```
$scope.modal = {
    title: 'Modal Title',
    content: 'Modal content'
};
```

There are also a number of options that can be used with the modal directive. These are applied to our element as attributes and prepended with `data-`. For example, if we wanted to change the animation, we could do that with the following:

```
<button class="btn btn-primary" bs-modal="modal" data-
  animation="am-fade-and-scale">Show Modal</button>
```

This will now use the fade and scale animation from AngularMotion. Be sure to check out the AngularMotion website for a full list of available animations.

The following table from the AngularStrap website shows the full list of options available for the modal directive:

Name	Type	Default	Description
animation	string	am-fade	Applies a CSS animation.
backdropAnimation	string	am-fade	Applies a CSS animation to backdrop.
placement	string	'top'	Positions the modal—top/bottom/center.
title	string	''	Default title value.
content	string	''	Default content value.
html	boolean	false	Replace `ng-bind` with `ng-bind-html`.

Name	Type	Default	Description
backdrop	boolean or 'static'	true	Includes a modal-backdrop element. Use `static` for a backdrop that doesn't close the modal on click.
keyboard	boolean	true	Closes the modal when escape key is pressed.
container	string/ false	false	Appends the modal to a specific element. Example, `container: 'body'`.
template	path	false	If provided, overrides the default template.
contentTemplate	path	false	If provided, fetches the partial and includes it as the inner content.

Tooltip

Tooltips are a great way to provide hints and tips without being intrusive. AngularStrap makes them super easy to include, and we trigger them using click, hover, or even focus:

```
<button class="btn btn-link" bs-tooltip="tooltip">what's
  this?</button>
```

Here, we've got a button (styled to look like a link using Bootstrap classes), and have included the `bsTooltip` directive. Just as we did with the modal directive, we can pass the directive a model. This time we only need to include the `title` property in our object:

```
$scope.tooltip = {
    title: 'Tooltip Title'
};
```

By default, our tooltip will show when we hover on our button, but that's easily changed using those data attributes we saw earlier:

```
<button class="btn btn-link" bs-tooltip="tooltip" data-
  trigger="click">what's this?</button>
```

The directive also allows us to bind it to an input and show the tooltip on focus. Positioning can also be determined by a data attribute:

```
<input type="text" bs-tooltip="tooltip" data-trigger="focus" data-
  placement="right">
```

This will show the tooltip on the right when the input is focused. Following is the full list of options taken from the AngularStrap documentation:

Name	Type	Default	Description
animation	string	am-fade	Applies a CSS animation.
placement	string	'top'	Positions the tooltip—top/bottom/left/right, or any combination such as bottom-left.
trigger	string	'hover'	Defines how the tooltip is triggered—click/hover/focus.
title	string	''	Default title value.
html	boolean	false	Replaces `ng-bind` with `ng-bind-html`.
delay	number/object	0	Delay showing and hiding the tooltip (ms)—does not apply to manual trigger type. If a number is supplied, delay is applied to both hide/show. Object structure is: `delay: { show: 500, hide: 100 }`.
container	string/false	false	Appends the modal to a specific element. Example: `container: 'body'`
template	path	false	If provided, overrides the default template.
contentTemplate	path	false	If provided, fetches the partial and includes it as the inner content.

Popover

Popovers are a sort of like an extended tooltip and provide a `title` and `content` area. Similar to tooltips, they can be triggered by click, hover, or focus:

```
<button class="btn btn-primary" bs-popover="popover">Show
  Popover</button>
```

The model bound is identical in format to the one used for our modal window in that it contains a title and content property:

```
$scope.popover = {
    title: 'Title',
    content: 'Popover content'
};
```

Of course, everything can be tweaked with data attributes. Following is a full list of options:

Name	Type	Default	Description
animation	string	am-fade	Applies a CSS animation.
placement	string	'top'	Positions the tooltip—top/bottom/left/right, or any combination such as bottom-left.
trigger	string	'hover'	Defines how the tooltip is triggered—click/hover/focus.
title	string	''	Default title value.
content	string	''	Default content value.
html	boolean	false	Replaces `ng-bind` with `ng-bind-html`.
delay	number/object	0	Delay showing and hiding the tooltip (ms)—does not apply to manual trigger type. If a number is supplied, delay is applied to both hide/show. Object structure is: delay: { show: 500, hide: 100 }
container	string/false	false	Appends the modal to a specific element. Example: `container: 'body'`.
template	path	false	If provided, overrides the default template.
contentTemplate	path	false	If provided, fetches the partial and includes it as the inner content.

Alert

We've already seen how we can use Bootstrap's alerts to provide feedback to users. AngularStrap allows us to pop these on steroids, fading them in, and also allowing users to clear them. We utilize the alert directive by adding the `bs-alert` attribute to an element.

```
<button class="btn btn-primary" bs-alert="alert">Show
  Alert</button>
```

Our model's object defines not only the title and content but also the context class we're going to be using. This can be success, info, warning, or danger and will change the color of the background and text accordingly:

```
$scope.alert = {
    title: 'Title',
    content: 'Alert content',
    type: 'success'
};
```

We're going to want to fine-tune exactly where the alert is added. We can use the data-container attribute to define a specific element where we want to show our alert. Let's create a new element at the top of our page for our container:

```
<div id="alertContainer"></div>
```

Let's add that to our button using the `data-container` attribute:

```
<button class="btn btn-primary" bs-alert="alert"
  data-container="#alertContainer">Show Alert</button>
```

Now when we click our button, the alert appears at the top of the screen. A full list of options available from the AngularStrap website is as follows:

Name	Type	Default	Description
animation	string	am-fade	Applies a CSS animation
placement	string	'top'	Positions the tooltip—top/bottom/left/right, or any combination such as bottom-left
title	string	''	Default title value
content	string	''	Default content value
type	string	'info'	Default type value
keyboard	boolean	true	Closes the alert when the escape key is pressed
container	string/ false	false	Appends the modal to a specific element. Example: `container: 'body'`
template	path	false	If provided, overrides the default template

Utilizing AngularStrap's services

The majority of the modules included within AngularStrap also expose services to our application. We can use these to show things such as modals, alerts, and popovers without having to use a directive.

Let's take a look at how we can use the `$alert` service to show an alert from our controller. We're going to use the `ng-click` directive on a button to trigger it all. First of all, create a button and attach the `ng-click` directive:

```
<button class="btn btn-success" ng-click="showAlert()">Alert via
  Service</button>
```

We'll sort the `showAlert()` function out in our controller in just a second. Before that, we need to create an alert using that service. Inject `$alert` into the controller and create a new instance of an alert using the following:

```
controller('demoCtl', function($scope, $alert){
    var alert = $alert({
        title: 'Alert Title!',
        content: 'Here\'s some content.',
        type: 'danger',
        container: '#alertContainer',
        show: false
    });
});
```

The service constructor accepts a hash following the same pattern as the directive accepts. We can also include any options here, such as the container we want to append the alert to. By default, the alert that's created will automatically be shown. To hide it, we need to include the show property and set it to false.

Finally, it's time to hook up the `showAlert()` handler. The alert instance created by the service gives us three methods we can use: `show()`, `hide()`, and `toggle()`. Let's use `show()`:

```
$scope.showAlert = alert.show;
```

If you click the new button, the alert will now appear at the top of the page (or wherever you've placed your container) and behave exactly as expected.

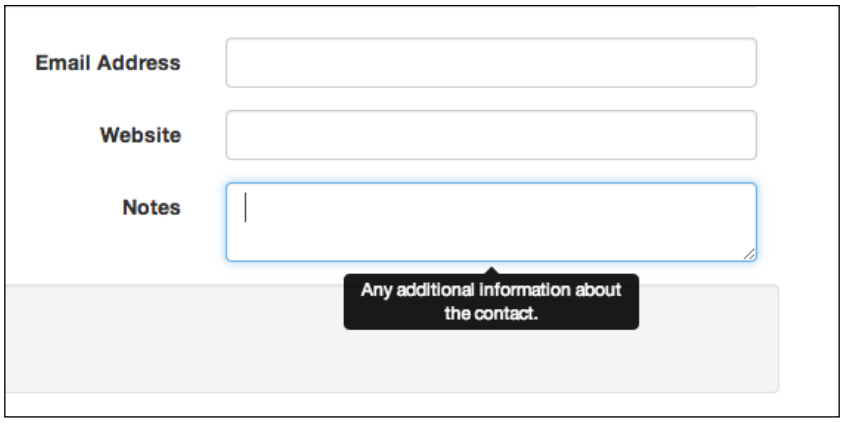
Integrating AngularStrap

Now that we've seen how we can use many of the plugins, it's time to put them into use and jazz up our contacts manager. We're going to use the tooltip and alert plugins to provide some hints and feedback to our users.

First of all, let's replace that hint text underneath the notes box in the Add Contact view with a tooltip:

```
<textarea id="notes" class="form-control" ng-model="contact.notes"
  bs-tooltip data-title="Any additional information about the
    contact." data-trigger="focus"
      data-placement="bottom"></textarea>
```

Rather than creating a model and binding it to the directive, it makes sense to just take advantage of the `data-title` attribute that's available to us. Here, we've opted to place it underneath and trigger it on focus.



There are two places that could benefit from an alert. One is the pre-existing alert after we've added a new contact, and the other is after we've deleted a contact in the Index view.

Let's first tackle that pre-existing alert. We need to replace the alert element with the container we created earlier:

```
<div id="alertContainer"></div>
```

We can now inject that `$alert` service and prepare our alert instance before we display it within our submit function. Our alert is going to have the following configuration:

```
var alert = $alert({
    title: 'Success!',
    content: 'The contact was added successfully.',
    type: 'success',
    container: '#alertContainer',
    show: false
});
```

We're going to append it to `alertContainer` we created earlier. As the context here is a successful message, we've set the type to success.

Now all that's left to do is show our alert after we've successfully created our contact:

```
$scope.submit = function(){
    contacts.add($scope.contact);
    $scope.contact = null;
    alert.show();
};
```

We can do the exact same thing for when we delete a contact to provide a little more feedback to the user. Just as before, place the container where you'd like the alerts to appear in your Index view:

```
<div id="alertContainer"></div>
```

Next, we need to inject the `$alert` service into our controller:

```
.controller('indexCtrl', function($scope, contacts, $alert){
```

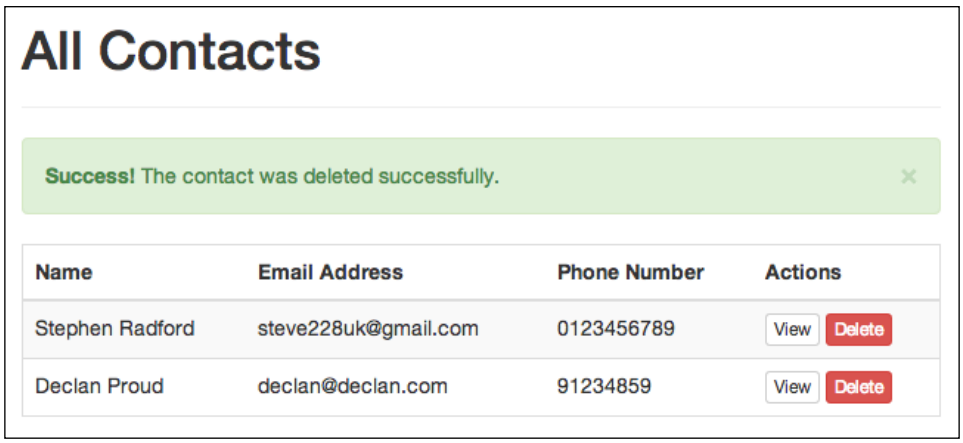
Now we can utilize the newly-injected service to create `deletionAlert`:

```
var deletionAlert = $alert({
    title: 'Success!',
    content: 'The contact was deleted successfully.',
    type: 'success',
    container: '#alertContainer',
    show: false
});
```

The last thing we need to do is show the alert when we hit that delete button:

```
$scope.delete = function(index){
    contacts.destroy(index);
    deletionAlert.show();
};
```

This is how the output would look:



Self-test questions

1. What module does AngularStrap depend upon?
2. What's the name of the project that we can use for our prebuilt CSS animations?
3. What needs to be prepended to our attributes to use them as options within AngularStrap directives?
4. What are the four ways in which a popover or tooltip can be triggered?
5. What three methods are given to us after we create an instance via the alert service?

Summary

In this chapter, we saw just how easy it can be to use the abundance of modules included with AngularStrap. Whilst they may not directly utilize Bootstrap's JavaScript, they're all components straight from Bootstrap and work seamlessly within our application.

We looked at just a few of the plugins available to us and how we can use them via directives. There are, of course, times when a directive isn't the best solution, so we also explored how we can use the services that are bundled with AngularStrap.

In the next chapter, we're going to examine how we can connect our application to the server to retrieve and store our contacts.

8
Connecting to the Server

So far, our application is entirely in the frontend and is, therefore, pretty useless. We need somewhere to store our contacts so that we can fetch them later on. In order to do this, we're going to connect to a server, which is going to house a RESTful API that serves up JSON.

Angular opens up a few different ways we can connect to the server. In this chapter, we're going to take a look at a couple of the solutions as well as touch on some alternatives for you to investigate further.

I'm not going to show you how to build the server-side aspect here, as that's outside of the scope of the book. However, it is included in the downloadable assets that accompany the book.

Here's what we'll be covering in this chapter:

- How to pull data from the server using `$http`
- How to use and where to find `ngResource`
- Community alternatives such as `RestAngular`
- Integration of our new server connection into our application

So, let's get started!

Connecting with $http

Out of the box, Angular includes some low-level methods of fetching and posting our data. If you've ever used `$.ajax`, `$.post` or `$.get` within jQuery, you'll feel right at home here.

I'm sure you've already realized that these methods come in the form of a service that we can inject into our controllers or services. Here you can see the `$http` service is injected into our controller:

```
.controller('indexCtrl', function($scope, contacts, $alert,
  $http){

})
```

The service includes a few methods we can utilize that work with all verbs of the `REST` protocol. The following methods are available within the `$http` service:

- `$http.get()`: Accepts a URL and optional `config` object. Performs an HTTP `GET` request.
- `$http.head()`: Accepts a URL and optional `config` object. Performs an HTTP `HEAD` request.
- `$http.post()`: Accepts a URL, data object, and optional `config` object. Performs an HTTP `POST` request.
- `$http.put()`: Accepts a URL, data object, and optional `config` object. Performs an HTTP `PUT` request.
- `$http.delete()`: Accepts a URL and optional `config` object. Performs an HTTP `DELETE` request.
- `$http.jsonp()`: Accepts a URL and optional `config` object. The callback name should be the string `JSON_CALLBACK`.
- `$http.patch()`: Accepts a URL, data object, and optional `config` object. Performs an HTTP `PUT` request.

All of these methods are shortcuts for the main `$http()` function, which accepts one argument: an object. The aforementioned functions automatically set the verb and/or type of content we're looking at fetching.

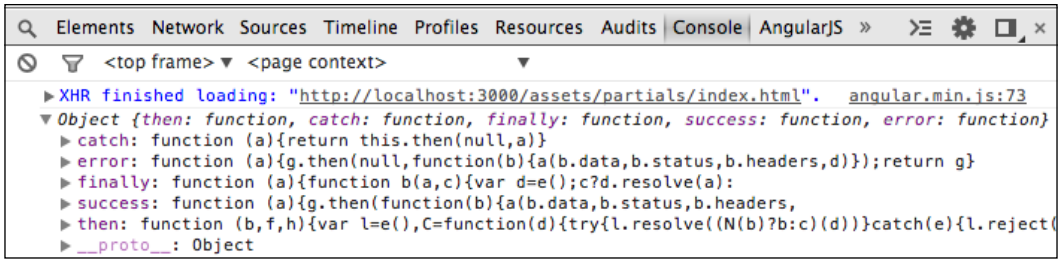
For example, the following two snippets of code are identical, but you'll notice that the second is much more readable:

```
$http({
    method: 'GET',
    url: 'http://localhost:8000'
```

```
});

$http.get('http://localhost:8000');
```

Fetching data is easy, and Angular takes advantage of the promises pattern introduced by the Promises/A+ organization and popularized by jQuery. The pattern allows us to easily determine whether the URL we've accessed has returned a success or error response.

This may sound quite complex, but what it boils down to is a collection of methods we can chain to easily create a try/catch approach to our asynchronous calls. If we wrap a `console.log` around `$http.get()`, you'll see all of the methods available displayed within your console.



All of the methods accept a single callback function, with the exception of `then`, which accepts two methods, one for success and one for error. Let's take a look at how we can take advantage of the success and error methods. Within our index controller, we can quickly swap the `contacts.get()` line with the following:

```
$http.get('http://localhost:8000')
.success(function(data){
    $scope.contacts = data;
})
.error(function(){
    window.alert('There was an error!');
});
```

Angular takes care of the rest. The success method's callback is executed when a status code `2xx` is returned; an error is executed otherwise.

Of course, we could have shortened the preceding code using the `then` method and two callback functions as follows:

```
$http.get('http://localhost:8000')
.then(function(result){
    $scope.contacts = result.data;
```

```
}, function(){
    window.alert('There was an error!');
});
```

This doesn't save much code and makes it less readable to other developers who may not be familiar with AngularJS. Note that data isn't the argument passed to the callbacks within the then method; instead, we get an object containing data, status, and headers.

Posting data

Just like fetching data, posting data using `$http` is very easy and similar to jQuery's implementation. The `$http.post()` function works in the exact same way as `$http.get()` but accepts a second parameter: a hash containing all the data we want to post to the server:

```
$http.post('http://localhost:8000', {
    name: 'Declan Proud',
    email: 'declan@example.com',
    ...
});
```

Likewise, the `post` method also returns a promise with those exact same methods we saw earlier:

```
$http.post('http://localhost:8000', {
    name: 'Declan Proud',
    email: 'declan@example.com',
    ...
})
.success(function(){
    ...
})
.error(function(){
    ...
});
```

Connecting with ngResource

Low-level connection helpers like `$http` are great for single connections, but they quickly become cumbersome when managing a full project. Thankfully, Angular has another way we can access data on the server side in the form of an optional module called `ngResource`.

Including ngResource

Just like `ngRoute`, the `ngResource` module can be found under the Extras link in the download modal at `https://angularjs.org/`. Download it and drag it into your project's `js` directory. You then need to include it after Angular in the root HTML file:

```
<script type=""text/javascript"" src="/assets/js/angular-
  resource.min.js"></script>
```

Lastly, ensure that the `contactsMgr` module knows that the `ngResource` module is a dependency:

```
angular.module('contactsMgr', ['ngRoute', 'ngSanitize',
  'mgcrea.ngStrap', 'ngResource'])
```

Configuring ngResource

The module exposes the `$resource` service that we can inject into our controllers or services. The methods included are much more high level and, in fact, use `$http` to interact with the server.

Let's inject that `$resource` service into our contracts service we created in *Chapter 7*, *AngularStrap* and see how we can connect to the server:

```
.factory('Contact', function ContactFactory($resource){
...
})
```

The service includes one method, which we use to set up our connection. This then returns a number of functions in the form of a resource object. These functions are what will actually fetch or send data to/from our server.

Let's take a look at how we use that single method and what's returned by it:

```
var Resource = $resource('http://localhost:8000/contacts/:id',
  {id: '@id'});
```

This will return the following object, which are actions that can be used to fetch, save, or delete data:

```
{
    'get': {method:'GET'},
    'save': {method:'POST'},
    'query': {method:'GET', isArray:true},
    'remove': {method:'DELETE'},
    'delete': {method:'DELETE'}
};
```

The first parameter is the root of our resource on the server. For example, if we were writing a blogging system, we might have a number of resources, such as posts, tags, and authors. We can also add placeholders here just like we would when creating a route.

The second parameter is a hash including defaults for those placeholders. Should a placeholder's default value be prefixed with the `@` symbol, that value will be fetched from the data object passed to it when we access the server.

We can also pass through a third parameter to extend upon those default actions that get returned. Let's add an `update` method that will utilize the `PUT` verb to update an existing contact on the server:

```
var Resource = $resource('http://localhost:8000/contacts/:id',
  {id: '@id'}, {
    update: {method: 'PUT'}
});
```

As you can see, this is just a standard JS object where we can define multiple custom actions. There are a number of things we can include within the configuration object attached to the action, but it's likely that you'll only need to set `method` and `isArray`. The `method` property selects which HTTP verb should be used (in our case `PUT`), and `isArray` is a Boolean used to tell `ngResource` whether a single item or array of items is going to be returned by the server.

Getting from the server

We've successfully configured `ngResource`; we now just need to put it into action, which couldn't be easier. It's just a case of using one of those actions returned by the `$resource` object.

We want to fetch all of our contacts, so it looks like the `query` method is going to be the best fit here. It uses the `GET` method and the `isArray` property is checked:

```
.factory('Contact', function ContactFactory($resource){
    var Resource = $resource('http://localhost:8000/:id', {id:
  '@id'}, {
        update: {method: 'PUT'}
    });
    return {
        get: function(){
            return Resource.query();
        },
        ...
    };
})
```

That's it! We don't have to worry about unwrapping promises, as that's handled automatically by `ngResource`. All that's left here is to switch our `$http` call within our `indexCtl` back to the `Contact.get()` method:

```
$scope.contacts = Contact.get();
```

As we're no longer using a hard-coded array, we can't access single contacts using an index. Most APIs will return an ID within their items, and ours is no exception. Let's change that `{{$index}}` from ng-repeat being used in our link, to use the ID instead of our single contacts instead; it should be around line 23 in our `partials/index.html` file:

```
<a href="/contact/{{contact.id}}" class="btn btn-default btn-
  xs">View</a>
```

We now need to change our `find` method within our contacts service to fetch a single contact based on the ID we give to it. We've already set up our resource to allow for an `id` parameter, so we just need to ensure we populate that when using the `get` method of our resource:

```
find: function(id){
    return Resource.get({id: id});
},
```

We've also changed the name of the parameter from `index` to `id` to make it more readable should somebody else work on this project later.

Posting to the server

That's everything fetched from the server, but how would we go about creating a new contact or updating an existing one with `ngResource`? Let's first tackle the issue of creation and see how `ngResource` handles this.

We're going to change our `create` method from the contacts service to return a new instance of our resource:

```
create: function(){
    return new Resource();
},
```

This will become our model within the Add Contact view. It behaves just as you would expect but gives us access to a `$save` method to push it to the server. Let's call our new `create` method and assign it to the contact model within `addCtl`:

```
$scope.contact = Contact.create();
```

Everything should behave as expected when you load up the view. We now need to swap the now defunct `contacts.set` method with our submit handler for that new `$save` function given to us by our resource:

```
$scope.submit = function(){
    $scope.contact.$save();
    $scope.contact = Contact.create();
    alert.show();
};
```

We've also changed our `$scope.contact` from being wiped out completely by switching it to fetch a new instance of the resource from the service.

Similarly, we can use that same update action we created earlier to save an existing contact's changes. To do this effectively, we need to use a custom event so our controller knows when we've saved our changes within the editable directive. Custom events behave exactly like their native JavaScript counterparts, such as `click` and `mouseover`. We can listen to them and perform actions when they're fired.

To create a custom event, we use the `$scope.$emit` method. This accepts two parameters: the name of the event and an array of any parameters we want to pass through to the listener. In our case, we don't need to pass any parameters, so let's just call our saved event and pop it into the `$scope.save` function within our editable directive:

```
$scope.$emit('saved');
```

Listening to the event is equally as easy using the `$scope.$on` method. This again takes two parameters. The first is the name of the event we're listening to, the second is our handler function. Let's add the following to our `contactCtl` controller.

```
$scope.$on('saved', function(){
    ...
});
```

Should we have passed parameters to our event, they'd be accessible as parameters within our event listener, with the first parameter always being our JS event.

Our listener is just going to run that `$update` method on our contact model. However, as our event is emitted before the model has finished being updated, we need to push it to the end of the current stack or queue. If you're familiar with JavaScript, you'll know that we can do this using `setTimeout`. This is exactly what we're going to do here, but rather than using `setTimeout`, let's use Angular's wrapper service: `$timeout`.

To use it, we need to inject it into our controller:

```
.controller('contactCtrl', function($scope, $routeParams, Contact,
  $timeout){
...
})
```

Then, it's just a case of using it in the same way as `setTimeout`:

```
$scope.$on('saved', function(){
    $timeout(function(){
        $scope.contact.$update();
    }, 0);
});
```

Now, if you edit one of your contacts and hit the save button, your data will be saved to the server.

Deleting contacts

The last thing we need to do is hook up our delete button to hit the server. Let's first of all change our method within the contacts service:

```
destroy: function(id){
    resource.delete({id: id});
}
```

Here, we're calling the `delete` method on our resource. We could have chosen to use remove as it does exactly the same thing. Again, we've changed the name of the `index` parameter to make it more readable.

That's not all we need to do in this instance, we need to update our delete function within the `indexCtl` controller. Let's take a look at the finished method and break it down:

```
$scope.delete = function(index){
    Contact.destroy($scope.contacts[index].id);
    $scope.contacts.splice(index, 1);
    alert.show();
};
```

As we need to both ping the server as well as remove it from the local array, we need to continue using the `index` this time. You'll notice that within our `Contact.destroy()` call we're accessing the relevant contact and fetching its ID. We're also removing it directly from our local array using the native JS `splice` method to ensure everything is in sync.

Error handling

We can handle errors in exactly the same way as we would with `$http`. All of the actions we've seen accept two callback functions: one for success and one for error. Here's the `get` method with both of the callback functions in place:

```
return Resource.get({id: id}, function(){
    window.alert('Success!');
}, function(){
    window.alert('Error!');
});
```

From here we can inform the user there's an issue or perform additional actions if necessary. As we're doing this from a service, it's wise to include two parameters to allow for these callbacks to be set from the controller when calling the method in question:

```
find: function(id, success, error){
    return Rgesource.get({id: id}, success, error);
},
```

Alternative ways of connecting

We've already explored a couple of ways we can connect to a server and have configured our web app to take advantage of `ngResource`. There are other modules we can use to connect to a server as well, and we're going to take a quick look at a couple of them.

RestAngular

RestAngular is a community project that provides a service to connect to RESTful APIs—much like `ngResource`. It does have a few significant differences that are worth being aware of.

The most important thing to note is that RestAngular uses promises just like `$http`. This means you get access to that nice pattern to determine whether a call was successful or not, but it does mean there are a few extra steps, and we can't just assign it to a model.

Whilst there's a bit of extra code you need to write due to the use of promises, you don't have to write out those placeholders when following the REST pattern; RestAngular will do it for you.

Personally, I prefer `ngResource`. It feels like there's a few more steps involved with RestAngular, and `ngResource` works perfectly with our services. It's always worth seeing what works for you though, so I would definitely recommend you give RestAngular a try.

Using RestAngular

RestAngular can be downloaded from `https://github.com/mgonto/restangular` and included like any other module. We're going to quickly look at how we can set up RestAngular and how we can grab a list of our contacts.

One thing I do like about RestAngular is the ability to set a base URL. This can be done in our module's global `config` method using the `RestangularProvider` service:

```
.config(function($routeProvider, $locationProvider,
  RestangularProvider){
    RestangularProvider.setBaseUrl('http://localhost:8000/');
})
```

Once the base URL is set, we can use RestAngular by just naming the resource we wish to access and one of RestAngular's methods:

```
Restangular.all('contacts').then(function(contacts){
    $scope.contacts = contacts;
});
```

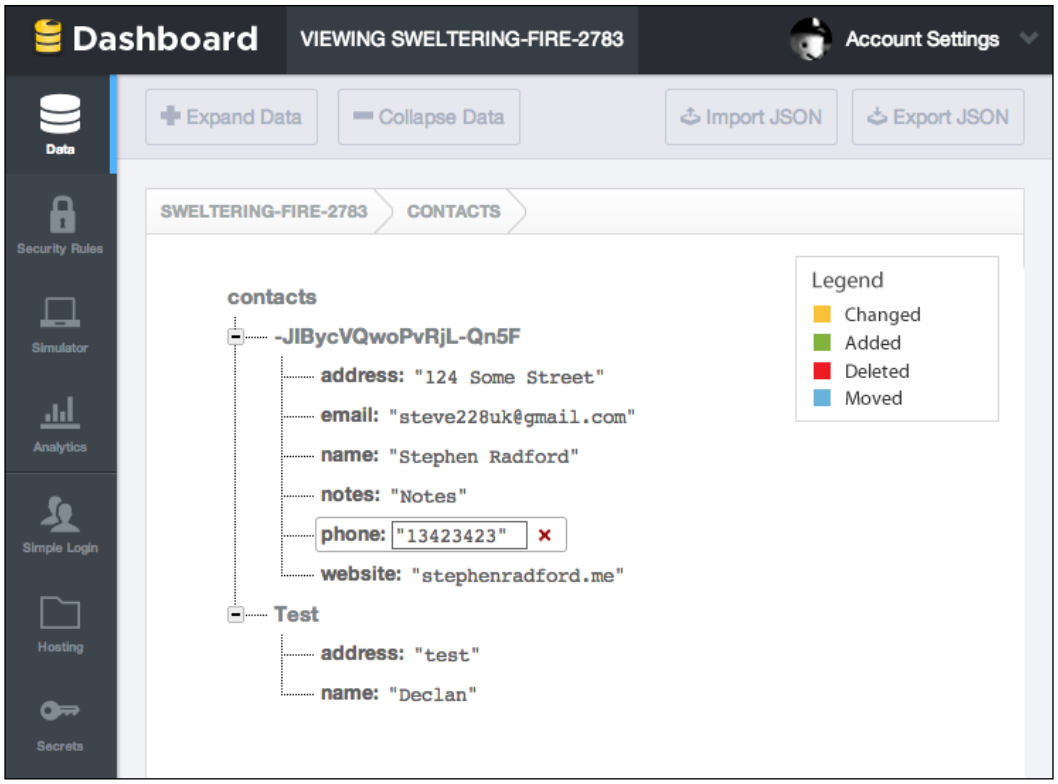
As you can see, RestAngular is using the promises pattern utilized by `$http` and we need to unwrap it to assign the data returned to our model.

Take a look at the RestAngular documentation on GitHub for a full list of methods that can be used.

Firebase

Firebase is a relatively new service that allows you to easily create a real-time application without writing a single line of backend code. When working with Angular, the company provides a handy helper library to sync your data with their service easily.

The Firebase dashboard will allow you to see your data in a collapse tree like the following:



Once you've signed up for your account at `http://firebase.com` and configured your app in the forge, it's time to hook up AngularFire. We need to include the Firebase client and AngularFire, both of which can be found at `http://angularfire.com`.

It's incredibly easy to fetch data from Firebase, especially considering this is all in real time, and any changes made elsewhere are reflected automatically within our application. Like most modules, AngularFire exposes a service, in this case `$firebase`. We can inject this into our controllers, directives, or services as follows:

```
.factory('Contact', function ContactFactory($resource, $firebase){
...
})
```

We can then launch our connection to Firebase using their JS client:

```
var contacts = new
Firebase("https://<yourbase>.firebaseio.com/contacts");
```

The AngularFire service gives us a little sugar to fetch data from Firebase easily. This is what we could do with our service's `get` method:

```
get: function(){
    return $firebase(contacts);
},
```

Adding contacts is super easy as well. Once we've retrieved our data, we gain access to `$add`, `$remove` and `$update` methods:

```
$firebase(contacts).$add({
    name: 'Declan Proud',
    ...
});
```

If you open up your Firebase control panel and add a contact manually, you should notice it pop up on your contacts list automatically. Obviously, this is overkill for something like a contacts manager, but it opens up infinite possibilities for things such as chat clients, notifications, and more without having to write a single line of backend code.

Self-test questions

1. What kind of object does the `$http` method return?
2. How would we get an array of contacts and assign them to a model with `$http`?
3. What three parameters does the `$resource` method accept?
4. What does the `@` symbol within a default parameter configuration mean?
5. Name the two main differences between ngResource and RestAngular.
6. What does Firebase make our application?

Summary

In this chapter, we transformed our application from a frontend app using hardcoded data into one that can interface with an API to store and retrieve information. We saw just how flexible Angular can be by taking a look at four different methods of connecting to a server.

Low-level services like `$http` are great for some things, but when building a full-fledged application, we saw that it's wise to use something more high level. Things like `ngResource` keep our code base maintainable and **DRY** (**Don't Repeat Yourself**).

In the next chapter, we'll expand upon the idea of keeping our code manageable by looking at two code runners: Grunt and gulp.

9
Using Task Runners

Our project is looking great, but so far, it's not all that efficient. We've included 10 JavaScript files, which mean 10 network requests, not even taking into account our style sheet. Of course, this means that the page is going to take longer to load. Once our page has loaded, the browser then has to go and fetch each JavaScript file and compile them.

We could manually take these files and concatenate them into a single file. However, as we're still working on our project, it's likely we're going to continue to make changes, and it would be tedious to continually repeat the process.

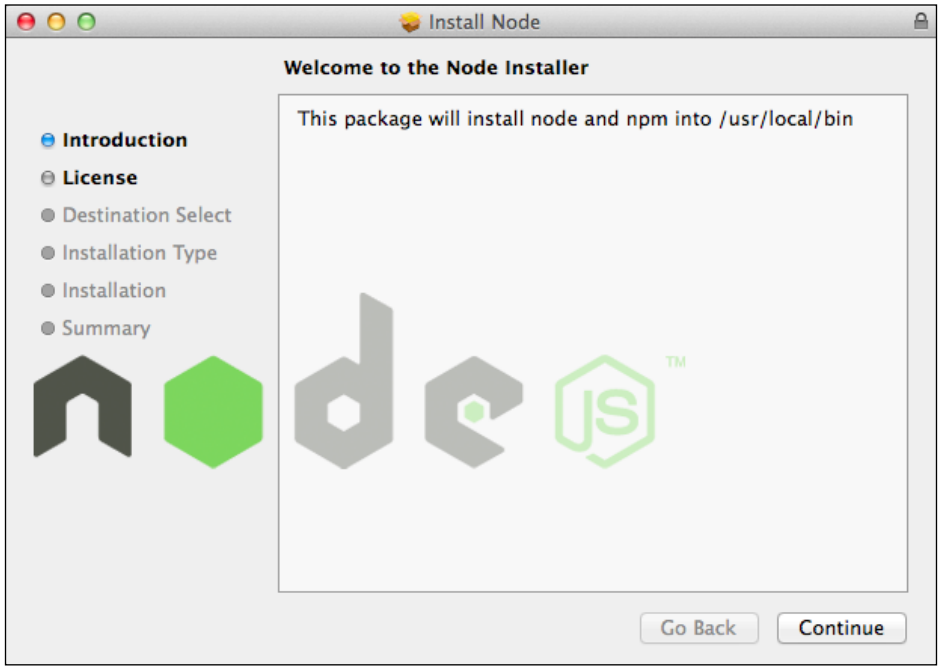
Task runners are a great way of automating boring tasks like this. We'll no longer need to manually concatenate and minify, but instead, we can have a setup watching our files for changes and automatically make them.

You might not have used one, but it's likely you'll have heard of task runners like Grunt and gulp. In this chapter, we're going to take a look at how we can use both of these task runners to concatenate and minify our JavaScript files down to a single file.

Installing Node and NPM

Both Grunt and gulp rely on Node and the bundled **Node Package Manager** (**NPM**). If you've already installed and configured Node, feel free to skip this section. I'm going to cover the Mac installation, but the process is similar for Windows. Linux users will need to compile from the source or see `https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager` to install it via a package manager.

First, head over to `http://nodejs.org/download/` and download the requisite installer for your platform. Open up the installer, agree to the license agreement, and complete the installation.



If everything goes as planned, you should be greeted with a similar message as follows:

Node was installed at

/usr/local/bin/node

npm was installed at

/usr/local/bin/npm

Make sure that `/usr/local/bin` is in your `$PATH`.

If you're not sure whether `/usr/local/bin` is in your path, run the following in your terminal:

```
echo $PATH
```

Look for `/usr/local/bin`. If you can't see it, you'll need to add the following to your `~/.bash_profile` or `~/.zshrc`:

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

Once that's done, Node and NPM is fully installed. You'll have access to the `node` and `npm` commands and will now be able to install Grunt or gulp within your project.

You may need to reload your terminal session in order for everything to work as expected.

Utilizing Grunt

Now that we've installed Node, we can start taking advantage of Grunt. Setting everything up is done in three stages. There's the command line tool, local installation of Grunt within our project, and configuration of the Gruntfile.

Installing the command-line interface

Installing the **command-line interface** (**CLI**) couldn't be easier, thanks to NPM. We just need to run the following in our terminal. The `-g` flag will ensure we're installing Grunt globally.

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

Depending on the permissions, you may need to run this as root. On OS X- and *nix-based systems, this is just a case of running it with the prefix of `sudo`. On Windows, you'll need to open the command shell as an administrator.

Once installed, the `grunt` command will be available and added to your system's path, allowing it to be run from any directory.

Installing Grunt

Using Grunt with our project will require adding two files—`package.json` and `Gruntfile.js`.

The `package.json` file isn't used by Grunt, but by NPM. It tells the package manager which packages our project needs when we run the installer. Our `Gruntfile.js` is what configures Grunt. It tells the task runner multiple things, from what files we want it to look at to what tasks we actually want to run.

Creating a package.json file

First, let's create that `package.json` file so NPM knows what files to fetch. Here's our completed JSON file. The Node Package Manager will let us easily create this by running the `npm init` command:

```
{
  "name": "ContactsMgr",
  "version": "1.0.0",
  "description": " A simple contacts manager in AngularJS +
    Bootstrap",
  "dependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-uglify": "~0.2.0",
    "grunt-contrib-watch": "~0.5.3"
  }
}
```

As you can see, it's a pretty standard JSON object with a few key properties set. The name, version, and description are required but only used if we release the project as a package on NPM. The dependencies property is where the interesting stuff happens.

As you can see, Grunt is at the top of our dependency list. The other packages we've included here are what will do the heavy lifting. The `grunt-contrib-uglify` package is going to concatenate and minify our JS files, and the last package in the list watches our files for changes and runs specific tasks.

It's important to remember that the name of our package cannot contain any spaces or special characters.

Building the Gruntfile.js file

The Gruntfile is very important. Think of it as Grunt's instruction manual. Without it, it simply wouldn't know what to do. Our entire configuration is done within the following Grunt wrapper function:

```
module.exports = function(grunt){

};
```

It's important that the `Gruntfile.js` is saved at the root of where you want to run Grunt and that the file starts with a capital G.

Most plugins will require to use Grunt's `initConfig` method, and this is what we're going to use with the `uglify` plugin:

```
module.exports = function(grunt){

    grunt.initConfig({
       ...
    });

};
```

Within this, we can configure each of our plugins using the name as our key. We can also get information directly from our `package.json` file, such as the name to use within our tasks:

```
grunt.initConfig({
    pkg: grunt.file.readJSON('package.json')
});
```

This will load the `JSON` file and assign it to the `pkg` key, allowing us to access any of the information we set in there earlier using a standard templating style syntax (`<%= %>`).

When configuring the `uglify` task, we're going to set two properties: `options` and `build`. The `options` property allows us to set things like a banner we want to include in our compiled file, whether to create a source map, or if we just want to concatenate for debugging purposes.

The `build` property is our target and can actually be named anything we like. For example, we could have one called `dev` and another called `production` with different options set. This can accept the `src` and `dest` properties to allow us to set what files go in and what comes out. We can also set another `options` object here, which is useful for when we do want to utilize multiple targets. Here's the `uglify` task with the `options` hash:

```
grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
        options: {
            banner: '/*! <%= pkg.name %> <%=
              grunt.template.today("yyyy-mm-dd") %> */\n'
        }
```

```
    }
});
```

Here, we've included the banner in our `options` object. As the `package.json` file has been converted to a JS object, it's just a case of using the standard syntax to access the name. Grunt also comes with a couple of helpers that we can utilize here. You'll notice that we're pulling today's date out here, but we can also use the `grunt.template.date` method to format a JS timestamp. This can be useful for when you want to include the date in a banner or filename.

Now we can set up our target. The `src` property can either be a string or an array. In our case, as we're using a number of JS files, we'll need to use an array. The `dest` property is the relative path to the file you want Grunt to create by default, but can also be changed using the `grunt.file.setBase` method. We've added the `build` object here and have set the `src` and `dest` properties:

```
grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
        options: {
            banner: '/*! <%= pkg.name %> <%=
              grunt.template.today("yyyy-mm-dd") %> */\n'
        },
        build: {
            src: [
                'assets/js/vendor/jquery.js',
                'assets/js/vendor/bootstrap.js',
                'assets/js/vendor/angular.js',
                'assets/js/vendor/angular-animate.js',
                'assets/js/vendor/angular-resource.js',
                'assets/js/vendor/angular-route.js',
                'assets/js/vendor/angular-sanitize.js',
                'assets/js/vendor/angular-strap.js',
                'assets/js/vendor/angular-strap.tpl.js',
                'assets/js/controller.js'
            ],
            dest: 'assets/js/build/<%= pkg.name %>.js'
        }
    }
});
```

You'll notice that we're using the package name for the filename, and also that all the minified files have been swapped for their unminified versions. As we're minifying everything, we need to ensure we use the development versions of the files to avoid any issues during compilation.

The order of the files in the `src` array is the order in which they will be included in the destination file. As we need jQuery to be included before Angular and Angular before our modules, it's important to get this right.

Our plugin is fully compiled, but Grunt doesn't know that we want to use that `uglify` task we downloaded from NPM earlier. To accomplish that, we need to use the `loadNpmTasks` method:

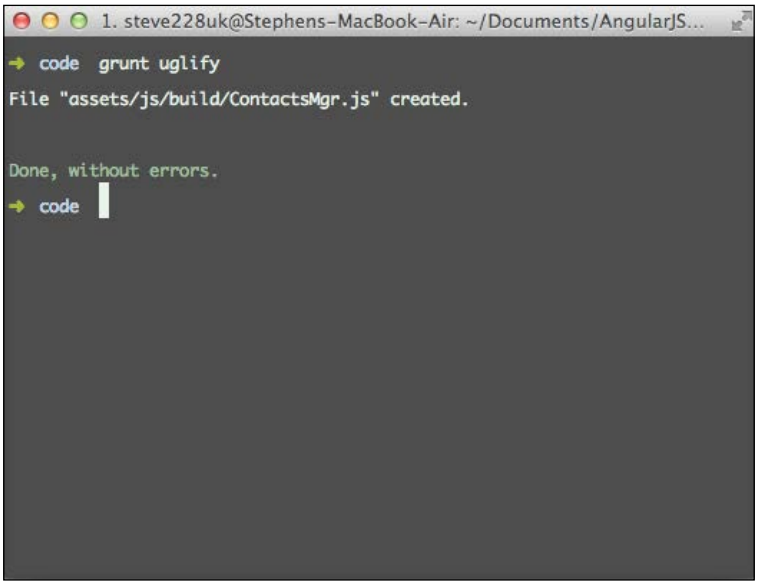
```
grunt.loadNpmTasks('grunt-contrib-uglify');
```

This goes within the `module.exports` function, making the completed Gruntfile look like the following:

```
module.exports = function(grunt){

    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        uglify: {
            options: {
                banner: '/*! <%= pkg.name %> <%=
                  grunt.template.today("yyyy-mm-dd") %> */\n'
            },
            build: {
                src: [
                    'assets/js/vendor/jquery.js',
                    'assets/js/vendor/bootstrap.js',
                    'assets/js/vendor/angular.js',
                    'assets/js/vendor/angular-animate.js',
                    'assets/js/vendor/angular-resource.js',
                    'assets/js/vendor/angular-route.js',
                    'assets/js/vendor/angular-sanitize.js',
                    'assets/js/vendor/angular-strap.js',
                    'assets/js/vendor/angular-strap.tpl.min.js',
                    'assets/js/controller.js'
                ],
                dest: 'assets/js/build/<%= pkg.name %>.js'
            }
        }
    });

    grunt.loadNpmTasks('grunt-contrib-uglify');

};
```

Running Grunt

We can now run the `grunt uglify` task which will generate our completed `ContactsMgr.js` file.



If you swap out the 10 scripts for the new file in the root `index.html` and load the application, you'll notice everything breaks and a console error will be thrown:

```
Error: [$injector:unpr] Unknown provider: a
```

As part of the minification process, variable names are changed for shortened versions. We've learnt that Angular relies heavily on dependency injection, which looks at the name of the variable to inject the correct service into our controllers and directives.

Thankfully, Angular has a quick and easy workaround for this. It involves changing any function we're injecting services into, for arrays with the names of the services we want to inject and the function as its values. Here's how our module's `config` would look:

```
.config(['$routeProvider', '$locationProvider',
  function($routeProvider, $locationProvider){

...

}])
```

As long as our function is last in the array, Angular will look at the variables and use the corresponding service from the array instead. Grunt won't change the value of the items within our array as they're strings and not variable names, so it's important the order in the array matches with what's injected into our function.

You'll only need to add these array wrappers within the `controller.js` file, as all of the libraries and modules we've utilized have already had this process applied to them. Don't forget that even controllers within directives need to use the array notation.

Setting up watch

We've successfully configured Grunt to compile our files and it is working great. However, it kind of defeats the point of automation if we have to run `grunt uglify` every time we make a change to our JavaScript. Grunt can keep an eye on things for us and automatically run some tasks when we make changes to our files.

To do this, we use the `grunt-contrib-watch` package we fetched from NPM earlier. Configuration is very simple and takes just two properties—`files` and `tasks`:

```
watch: {
    files: [
        'assets/js/*.js'
    ],
    tasks: ['uglify']
},
```

As seen here, we can use the asterisk as a wildcard so Grunt detects any `.js` file directly within the `assets/js` directory. We can put as many tasks as we like within that tasks array and they'll all run in order.

Running `grunt watch` within Terminal will keep Grunt running in the background. As soon as a file is changed, it jumps into action and runs the `uglify` task, concatenating and minifying our JavaScript.

Creating the default task

It's often the case that you'll want to run multiple tasks in one go. Grunt enables you to do this by registering your own task:

```
grunt.registerTask('default', ['uglify']);
```

The first parameter is the name of our task and the second is the array of tasks we wish to run. Using the default keyword tells Grunt that this is the task to be run when we don't specify one. For example, we could run the `default` task in either of the following ways:

```
grunt default
grunt
```

Utilizing gulp

Gulp is fairly new and definitely takes its cue from Grunt. Due to its relatively short lifespan, there aren't as many plugins available for it. However, uglify is there and the list is always growing. On the plus side, gulp aims to make configuration simpler and running tasks faster than Grunt—it's up to you to decide which one works best for you.

Just like Grunt, gulp comes in two parts. There's the global command-line tool as well as the local installation that we're going to include within our project.

Installing gulp globally

It's very easy to install gulp globally, and it's all done through a single NPM command again:

```
npm install -g gulp
```

Don't forget, you may need to run this as root either by using `sudo` or running the Windows command prompt as administrator.

Once installed, the `gulp` command will be available for use from the terminal.

Installing gulp dependencies

Exactly as we did with Grunt, we need to create a `package.json` file, which is going to hold all of our project dependencies. For now, let's just install gulp itself:

```
{
  "name": "ContactsMgr",
  "version": "1.0.0",
  "description": "A simple contacts manager in AngularJS +
    Bootstrap",
  "dependencies": {
    "gulp": "~3.6.0"
  }
}
```

We can add `uglify` to our `package.json` file manually, but we can also get NPM to do it for us:

```
npm install --save-dev  gulp-uglify
```

The `save-dev` flag tells NPM we want it to add this to the `package.json` file for us. Alternatively, we could use the `--save` flag, but as gulp is only used during development, we won't need it in production.

Unlike the uglify plugin for Grunt, this won't concatenate the files for us, and we'll need to use another plugin for that:

```
npm install --save-dev gulp-concat
```

Setting up the gulpfile

Unlike Grunt, the gulpfile file does not need to start with a capital G, but it does house our configuration and goes into the root of our project. While the idea of the file is similar, the configuration is very different.

You'll remember that Gruntfile required a `wrapper` function to allow us to access all those Grunt methods. With gulp, it's a little different, and each of the packages we pull down from NPM can be required within our file. Include the following at the top of your gulpfile.

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var concat = require('gulp-concat');
var pkg = require('./package.json');
```

We can also include information from our `package.json` file by requiring it in the same way as an NPM package. The `./` at the beginning just tells Node to look in the same directory as `gulpfile` when we run gulp.

There's also no configuration object here, which really simplifies things. Everything is done within tasks. Here's our finalized `uglify` task for us to break down:

```
gulp.task('uglify', function(){
    gulp.src(paths.js)
    .pipe(concat('ContactsMgr.min.js'))
    .pipe(uglify())
    .pipe(gulp.dest('assets/js/build'));
});
```

The `gulp.task` method takes two parameters: the name of our task and an anonymous function that contains everything our task is going to do.

You'll notice that we've included a variable within our `gulp.src`. This is so we can use it later, and it provides a little more flexibility without having to write it all out each time:

```
var paths = {
    js: [
        'assets/js/vendor/jquery.js',
        'assets/js/vendor/bootstrap.js',
        'assets/js/vendor/angular.js',
        'assets/js/vendor/angular-animate.js',
        'assets/js/vendor/angular-resource.js',
        'assets/js/vendor/angular-route.js',
        'assets/js/vendor/angular-sanitize.js',
        'assets/js/vendor/angular-strap.js',
        'assets/js/vendor/angular-strap.tpl.min.js',
        'assets/js/controller.js'
    ]
};
```

Here's the paths object we're referencing. It's the same array of files we included in our `Gruntfile` earlier.

Gulp uses pipes to process our data. All of the packages we've referenced at the beginning of our `gulpfile` file are functions. The `concat` plugin accepts the name of the file we want to output. We've fetched the name from the `package.json` file and have appended the `.js` extension. Uglify has a number of options we can pass through as a JS hash to help with debugging, and the `gulp.dest` method we're using allows us to enter the name of the directory we want to output to.

With our task setup, we could just run `gulp uglify`, but we might as well get the `watch` setup beforehand. Unlike Grunt, we don't need to include an additional plugin to do this, as it comes as another method within gulp:

```
gulp.task('watch', function(){
    gulp.watch(paths.js, ['uglify']);
});
```

As you can see, it's extremely easy to setup. We just create a new task and use the `gulp.watch` method, passing through the array of files we want to watch and then the array of tasks we want to run when these files change.

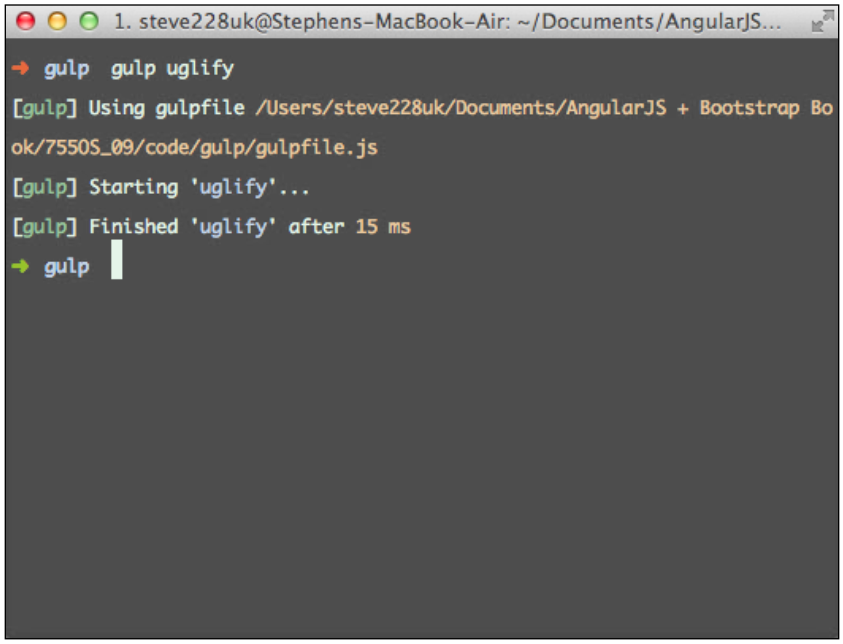
Let's quickly set up a default task to finish off our `gulpfile` file:

```
gulp.task('default', ['uglify']);
```

Here's the completed `gulpfile.js` file that neatly configures our tasks:

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var concat = require('gulp-concat');
var pkg = require('./package.json');

var paths = {
    js: [
        'assets/js/vendor/jquery.js',
        'assets/js/vendor/bootstrap.js',
        'assets/js/vendor/angular.js',
        'assets/js/vendor/angular-animate.js',
        'assets/js/vendor/angular-resource.js',
        'assets/js/vendor/angular-route.js',
        'assets/js/vendor/angular-sanitize.js',
        'assets/js/vendor/angular-strap.js',
        'assets/js/vendor/angular-strap.tpl.min.js',
        'assets/js/controller.js'
    ]
};

gulp.task('uglify', function(){
    gulp.src(paths.js)
    .pipe(concat(pkg.name+'.js'))
    .pipe(uglify())
    .pipe(gulp.dest('assets/js/build'));
});

gulp.task('watch', function(){
    gulp.watch(paths.js, ['uglify']);
});

gulp.task('default', ['uglify']);
```

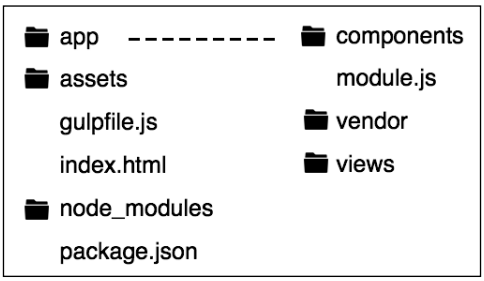
We can now run `gulp uglify` or `gulp` within the terminal to concatenate and minify our JavaScript files. Apart from the one-time option, we can also run `gulp watch` to keep an eye out for any changes and run the `uglify` task automatically.



Restructuring our project

Now we have our task runner set up, it's a great time to restructure our project to give us a more manageable codebase. We're going to separate out our controllers, directives, filters, and services into separate files to keep things tidy.

Let's start by working out a new directory structure as demonstrated here:



Here, we've moved everything out of our `assets/js` directory into its own `app` directory in the root. When we go to production, we're not going to want our source files to be deployed, so it's a good idea to break it out of the assets directory where our compressed files will live.

The new `app` directory has been structured a little differently too. We've got three folders here for components, vendor, and views, as well as a `module.js` file. Components are shared items, so in the case of our app, we'll place our directives and services here. The vendor folder contains all of the third-party JS files and the views folder contains each of the main controllers for our views.

Once we've created the new directories, we can start to separate our controllers into different files for inclusion in the views directory. These still need to be attached to our module, and this can be done by using the `angular.module('contactsMgr')` declaration at the beginning of the file.

Here's the `contactCtl` controller for example; name it `contact.js` and pop it into that views folder:

```
angular.module('contactsMgr').controller('contactCtl', ['$scope',
  '$routeParams', 'contacts', '$timeout', function($scope,
    $routeParams, contacts, $timeout){
    $scope.contact = contacts.find($routeParams.id);

    $scope.$on('saved', function(){
        $timeout(function(){
            $scope.contact.$update();
        }, 0);
    });
}]);
```

Now that all of the controllers have been separated out, let's work on moving over the directives. These are going to go directly into the components folder with that new app directory on the root. Again, just as we did with the controllers, we need to ensure these are still attached to our module.

Copy each of the directives into separate files and name them with the `.directive.js` extension. For example, our Gravatar will go into the components folder as `gravatar.directive.js` and the editable directive as `editable.directive.js`.

As our filters are also shared components, we can place these alongside the directives and name them in a similar fashion. Put the two filters into separate files: `truncate.filter.js` and `newLine.filter.js`. The final component is our contacts service that we use to connect to the server. Create a new `contacts.service.js` file and copy it over.

With our views and components in place, we now need to ensure that the new `module.js` file houses what it should—our module. Copy the contents of the `contactsMgr.js` file from the `assets/js` directory into `module.js` file. Once you're sure all the files have been moved, go ahead and delete the contents of the `js` directory. We'll be adding a fully minified file in here later.

We now need to configure Grunt/gulp to look at our new directories and output to the `js` directory within the assets folder. The revised paths within the Gruntfile or gulpfile files should look like this:

```
'app/vendor/jquery.js',
'app/vendor/bootstrap.js',
'app/vendor/angular.js',
'app/vendor/angular-animate.js',
'app/vendor/angular-resource.js',
'app/vendor/angular-route.js',
'app/vendor/angular-sanitize.js',
'app/vendor/angular-strap.js',
'app/vendor/angular-strap.tpl.js',
'app/module.js',
'app/components/**/*.js',
'app/views/**/*.js'
```

Angular handles the dependency management side of things, but the order here does matter. We're loading `jquery.js` before `angular.js` so Angular knows we want to use that and not the included `jqLite`. All of the `vendor` modules require Angular, so that needs to be included above them. Moreover, our components and views require our module to be loaded or it won't know what to attach to.

Before you run your chosen task runner, change the destination from `assets/js/build` to `assets/js`. Now run the chosen task runner to compile your newly organized application. Finally, change the referenced file in `index.html` now that the destination has changed:

```
<script type="text/javascript"
  src="assets/js/ContactsMgr.js"></script>
```

Open up the browser to double-check that everything went as expected. If all went according to plan, your contacts manager should work perfectly.

If things aren't working as they should, check the console. Chances are you may have left something out or included it in the wrong order.

Self-test questions

1. What environment do both Grunt and gulp rely on?
2. Why do we need a `package.json` file?
3. What plugin is used for minification?
4. What needs to be done to our Angular files before we minify them?

Summary

We've taken a look at two very powerful and very similar tools in this chapter. They've allowed us to not only significantly reduce the number of HTTP requests we were making, but to also completely restructure our app.

Both Grunt and gulp achieve the same result, but it's entirely personal choice which task runner you employ. I personally find gulp to be a little faster and simpler to configure, but there's no denying that Grunt has a lot more plugins at its disposal and is an older, more tested tool.

In the next chapter, we'll look at how we can use these two task runners to take the Less files that Bootstrap uses and compile them down into our own customized version.

10
Customizing Bootstrap

Up until now, our application has looked pretty standard. We're taking full advantage of Bootstrap, but the default look is definitely overused. Bootstrap is designed to be customized, and uses the Less or SASS css preprocessor to make this fast and simple.

Over the course of this chapter, we'll take a look at how we can compile Bootstrap's Less source before moving on to customizing the look to make it our own. We'll cover the following topics:

- The basics of Less
- Customizing Bootstrap
- Compiling Less with Grunt or gulp
- Setting up LiveReload
- Learning about Bootstrap themes

Compiling Less with Grunt or gulp

Before we begin with any kind of customization, it's a good idea to learn how we can turn many Less files into a single cascading style sheet. We've already seen how we can set up Grunt and gulp to concatenate and minify JavaScript files, and we're going to use the very same task runners to compile Less.

Downloading the source

First of all, let's grab the latest version of Bootstrap and pull over those Less files into our project. Head over to `http://getbootstrap.com/` and click **Download Bootstrap**. You'll be presented with three options: **Bootstrap**, **Source code**, and **Sass**. We want the Source code option as this includes the Less files we can customize and compile.

As you can see, the Bootstrap source is around ten times larger than the minified version. To keep things tidy, copy the `less` directory from the download into the `assets` folder within your project. The directory contains the 40 `less` files that make up the Bootstrap styles.

Compiling with Grunt

As we've seen, Grunt is a powerful task runner. We can extend upon the minification of JavaScript to automate the compilation of Less with CSS. Grunt does this via a plugin that can be fetched from NPM. We can include this in our project's `package.json` and then run `npm install` from the terminal. However, it's much simpler to run a single command and let NPM add the dependency into our project's `package.json` file, as follows:

```
npm install grunt-contrib-less --save-dev
```

Now that the plugin is installed, we can configure it within our Grunt file. Everything is again done within the `config` object. We're going to set up two targets: one for development and another for production. This will allow us define options for each scenario. For example, for production, we're going to want to minify our CSS, but this isn't always desirable when we're developing.

Here's the full configuration for the `less` task:

```
less: {
    dev: {
        files: {
            'assets/css/bootstrap.css':
  'assets/less/bootstrap.less'
        }
    },
    production: {
        options: {
            cleancss: true
        },
        files: {
            'assets/css/bootstrap.css':
  'assets/less/bootstrap.less'
        }
    }
}
```

Within our `dev` target, we haven't set any options, but production has the `cleancss` flag set to true to reduce the file size by minifying the output. The `files` object is also shorthand for the `src` and `dest` properties that we saw when configuring Grunt to uglify our JavaScript.

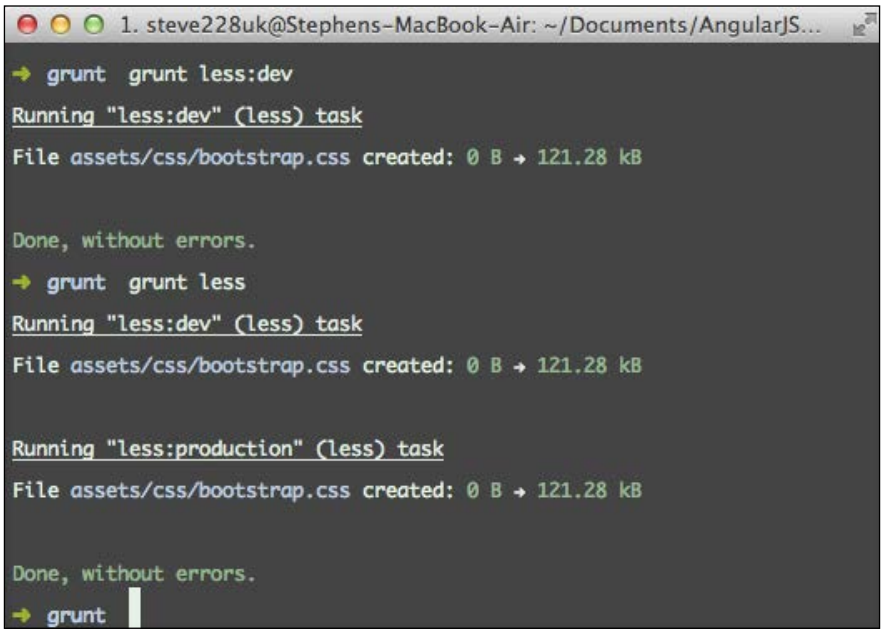
Don't forget that we also need to load that module from NPM or Grunt won't be able to access the `less` task. Pop this within the Grunt wrapper:

```
grunt.loadNpmTasks('grunt-contrib-less');
```

Now we will be able to compile our styles. We can run the following in the terminal to execute the task:

```
grunt less
```

This, however, has a slightly unintentional effect. It runs the task on both targets one after the other.



We can limit it to just a single target by specifying the name of the target after a colon:

```
grunt less:dev
```

Setting up Watch and LiveReload

Of course, the whole point of using a task runner like Grunt or gulp is to automate all of this. We can use the watch plugin just as we did in *Chapter 9*, *Using Task Runners*, to run a task when one of the files changes. This also gives us the ability to live reload our page with the help of a browser plugin.

Configuration is easy as we already have the plugin installed. Here's the current setup we have for our `watch` task:

```
watch: {
    files: [
        'assets/js/*.js'
    ],
    tasks: ['uglify']
}
```

We could add our `less` directory to the `files` array and the `less` task to the `tasks` array. This would mean both tasks are executed whenever a `.js` or `.less` file changes, which isn't what we want to happen. By separating this into two targets, we can have greater control over the tasks that are executed:

```
watch: {
    js: {
        files: [
            'assets/js/*.js'
        ],
        tasks: ['uglify']
    },
    less: {
        files: [
            'assets/less/*.less'
        ],
        tasks: ['less:dev']
    }
}
```

In terms of configuration, everything here is the same. We've just separated the two file types to run our `uglify` and our `less:dev` tasks, respectively.

The plugin also comes with another trick up its sleeve. It acts as a server for the `LiveReload` plugin for multiple browsers. To utilize it with our app, we just need to include an additional `script` tag on the page:

```
<script src="http://localhost:35729/livereload.js"></script>
```

Alternatively, there's an extension for Chrome, Firefox, and Safari, which can be downloaded from `http://livereload.com/`. Once installed, the browser can be pinged by the LiveReload server whenever changes are made.

Setting up Grunt to ping our new browser plugin couldn't be easier; we just need to set the `livereload` property within an `options` object to true. Let's quickly add this to our `less` target:

```
less: {
    files: [
        'assets/less/*.less'
    ],
    tasks: ['less:dev'],
    options: {
        livereload: true
    }
}
```

If you open up your browser and turn LiveReload on, you'll notice the page reloads whenever you make changes to the less files. This is great, but wouldn't it be better if just the CSS refreshed and not the entire page? Grunt will reload the entire page if a file changes. To ensure this is the case, you can add a second target to our configuration that watches for changes to the `bootstrap.css` file:

```
css: {
    files: [
        'assets/css/bootstrap.css'
    ],
    options: {
        livereload: true
    }
}
```

If we turn LiveReload off within our `less` target, the browser will receive a new CSS file and will no longer reload the page.

Compiling with gulp

Now let's take a look at the other task runner: `gulp`. Just like Grunt, gulp uses an additional plugin to enable the compilation of Less. We're also going to need to include a second plugin for LiveReload, as this isn't something that's included with Grunt.

Let's first install and configure the `less` plugin. We can do this through the command line by running the following command:

```
npm install gulp-less --save-dev
```

This will install the plugin to our project and also include it in our `package.json` file for future use. To utilize it within our `gulpfile`, we use Node's require method to include the package. Pop this at the top of the `gulpfile`:

```
var less = require('gulp-less');
```

Let's create a new task called `less` to handle all of our compiling, utilizing the plugin we just included:

```
gulp.task('less', function(){

});
```

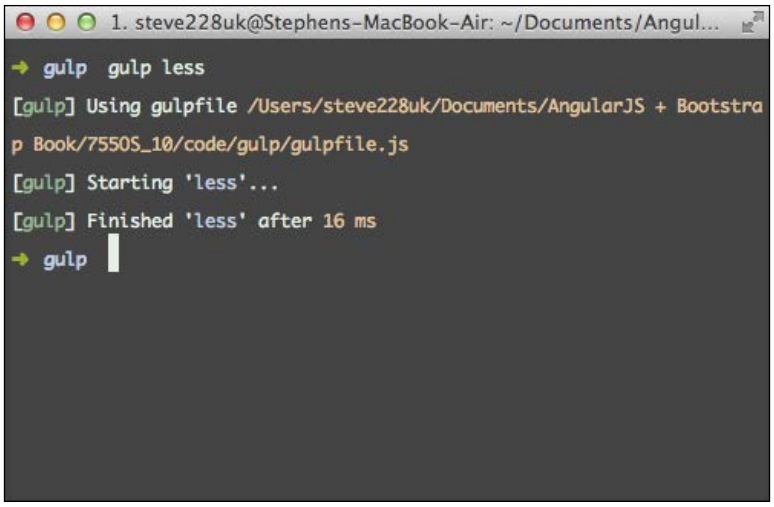
As we did with our JavaScript, we're going to use gulp's `src` method along with a couple of pipes to achieve the desired outcome. Let's take a look at the completed task and then break it down:

```
gulp.task('less', function(){
    gulp.src('assets/less/bootstrap.less')
    .pipe(less({
        filename: 'bootstrap.css'
    }))
    .pipe(gulp.dest('assets/css'));
});
```

We're only including one file here, as all other files are included in it via `@import`. The Less plugin accepts any parameters the official Less compiler does. In our configuration, we're setting a filename, but by default it will use the source filename.

Finally, we're using the `gulp.dest` method to export the compiled file to the CSS directory. Remember, these pipes act as steps that our task executes in order and we can easily add additional stages or reorder here in the future, should we need to.

Gulp is now fully configured to utilize the Less plugin and is ready to compile our styles using the gulp `less` command.



Setting up Watch and LiveReload

Of course, it would defeat the object of automation if we had to run this command manually. We've already seen that `watch` is baked right into gulp, and including `less` in our current configuration is simply a case of including one more line of code:

```
gulp.task('watch', function(){
    gulp.watch(paths.js, ['uglify']);
    gulp.watch(paths.less, ['less'])
});
```

Notice that we're referencing a new property within our paths object. Let's quickly add that so gulp knows where the files we're watching are:

```
less: 'assets/less/*.less'
```

We're using the wildcard to reference every single Less file, so gulp can see exactly when something has changed.

Setting up LiveReload takes a little more work, as it involves installing another plugin. Let's fetch this plugin from NPM with the following command:

```
npm install --save-dev gulp-livereload
```

Once installed, reference the plugin at the top of the `gulpfile`:

```
var livereload = require('gulp-livereload');
```

The function returned is actually the LiveReload service, and we can use this to tell the browser extension which files have changed. To configure this correctly, we have to do a couple of things within our watch task. First, we need to reference the LiveReload server. We can then pass it change files from an event fired by the `gulp.watch` method. Let's take a look:

```
gulp.task('watch', function(){

    var server = livereload();

    gulp.watch(paths.js, ['uglify']);
    gulp.watch(paths.less, ['less']).on('change', function(file){
        server.changed(file.path);
    });

});
```

We've assigned the `livereload()` function to the server variable and have added a listener for the change event to our watcher. The event passes through a file object and we can then pass the path of the file on to the server.

Just like with Grunt, we face the issue of the browser reloading when a non-CSS file is changed. We can resolve this by adding a third watcher into the mix. Following is our completed task with the addition:

```
gulp.task('watch', function(){

    var server = livereload();

    gulp.watch(paths.js, ['uglify']);
    gulp.watch(paths.less, ['less']);
    gulp.watch('assets/css/bootstrap.css').on('change',
  function(file){
        server.changed(file.path);
    });

});
```

Don't forget to swap the minified CSS file for your newly compiled one within your `index.html` file.

Less 101

To get a better understanding of what Less actually provides, let's have a quick Less 101 to get to grips with some of the ideology and syntax behind the pre-processor. We're going to take a look at four of the main features of Less: importing, mixins, nested rules, and variables. A full list of language features can be found at `http://lesscss.org/features/`. What's great about Less is that if you don't want to use any of its new syntax or features, you don't have to. Any valid CSS is also valid Less.

Importing

Just like in CSS, we can include one file within another in Less. It even follows the same syntax as in CSS:

```
@import "file.less"
```

However, unlike CSS, which will make an additional HTTP request for the referenced file, Less will merge the file when it's compiled. If you open the `bootstrap.less` file, you'll see all the required Less files referenced here.

You can omit the .less when including files and compiling with newer versions of Less.

Variables

Bootstrap uses Less variables heavily to enable us to quickly change colors and fonts of variable elements. We can quickly change any of these by opening up the `variables.less` file. A variable is defined with the `@` symbol followed by the name of the variable. Let's take a look at one:

```
@brand-primary:         #428bca;
```

As you can see, these are just references for us to use within our styles. We can call the variable by referencing it within our properties:

```
color: @brand-primary;
```

Upon compilation, Less will swap these references out for the color defined previously. Bootstrap uses these variables throughout all of its elements, so we can quickly change colors and fonts from within this file.

Nested rules

Perhaps one of the most irksome patterns CSS follows is styling children. Rather than just nesting the child within the parent, we have to write a separate rule for it. Here's an example:

```
div {
  background: #ccc;
}

div a {
  color: #000;
}

div a:hover {
  color: #fff;
}
```

Within Less, we can nest these rules to make things a lot tidier:

```
div {
  background: #ccc;
  a {
    color: #000;
    &:hover {
      color: #fff;
    }
  }
}
```

As you can see from the preceding example, we can nest pseudo classes by using the `&` syntax. This is a reference to the parent rule. A secondary class can also be defined in this way. For example, here's a button with two styles for orange and blue:

```
button {
  color: #fff;
  &.orange {
    background: orange;
  }
  &.blue {
    background: blue;
  }
}
```

Mixins

A mixin allows us to include styles from another rule. A mixin can also allow arguments to be passed in to provide flexible control. Let's take a look at an example:

```
.border-radius(@radius: 5px) {
  border-radius: @radius;
}
```

We can then use this mixin within our styles:

```
button {
  .border-radius;
}
```

The default value we've set will automatically be used. However, we can easily override this by setting the value within a parenthesis:

```
button {
  .border-radius(15px);
}
```

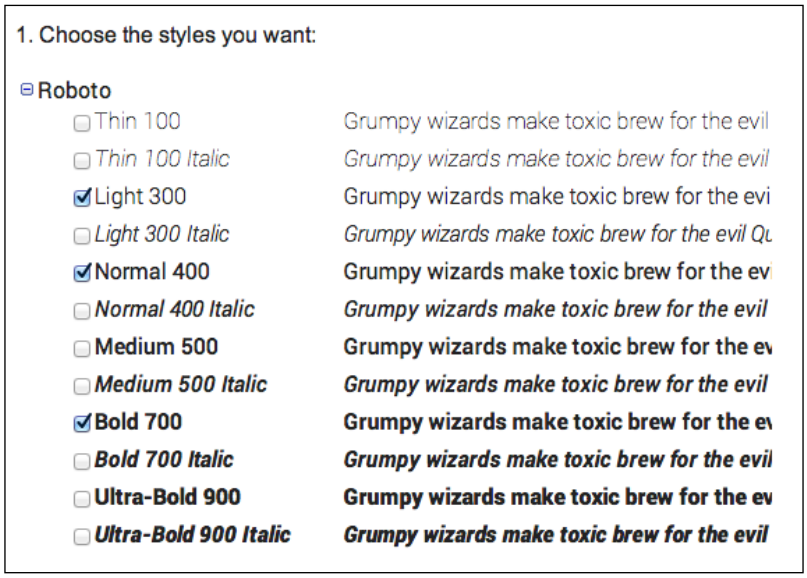
Customizing Bootstrap's styles

Now that we're using the Bootstrap source, we can dive into any file to customize it. Less extends upon CSS and gives us new features that Bootstrap takes full advantage of.

Typography

Bootstrap uses Helvetica, perhaps the most popular typeface in the world. To give our application a bit of character, let's take a look at how we can swap this out for something from the Google Fonts library at `https://www.google.com/fonts`. Take a look through and find something you like. For now, we're going to use Roboto, a humanist sans-serif typeface.

Add the font to your collection and select the **light**, **normal**, and **bold** styles, as seen in the following screenshot:



Copy the `@import` line and include it at the top of the `bootstrap.less` file. This will then give us access to the Roboto font family within our styles. Search the `variables.less` file for Typography. The section begins around line 38. We're going to change the `@font-family-sans-serif` variable that Bootstrap uses by default as its base:

```
@font-family-sans-serif:  Roboto, "Helvetica Neue", Helvetica,
  Arial, sans-serif;
```

In this section, we can also change font sizes, but for now, let's leave that as everything is looking nice and balanced.

navbar

Search for navbar within `variables.less`; the section should begin around line 324. We're going to do some quick tweaks here to make it look a lot less generic. Let's start by changing that dull grey color to something a little more exciting—a nice steel blue:

```
@navbar-default-bg:                #667591;
```

We'll also need to change the color of the text and links to match this darker background color:

```
@navbar-default-color:                  #fff;
@navbar-default-link-color:             #fff;
@navbar-default-link-hover-color:       #ccc;
@navbar-default-link-active-color:      #fff;
```

Looking at the mobile nav, we need to change the color of that toggle button as well:

```
@navbar-default-toggle-hover-bg:        darken(@navbar-default-
  bg, 15%);
@navbar-default-toggle-icon-bar-bg:     #fff;
@navbar-default-toggle-border-color:    #fff;
```

Less includes several helper functions that we can use to modify colors, including saturate, desaturate, and fade. The two Bootstrap functions that are taken advantage of most are darken and lighten. These take a color and a percentage to lighten or darken it—perfect for hover states. Here, we've used the darken variable and have passed in the variable for the navbar background.

Lastly, let's tweak the height and remove the border radius to tidy things up:

```
@navbar-height:                 60px;
@navbar-border-radius:          0;
```

Forms

The forms within Bootstrap are perhaps the most recognizable. Let's make some minor tweaks to make them look a little different. To do this, we're going to make changes both to `variables.less` as well as a couple of other files.

First of all, let's rip that border radius off the inputs within `variables.less`:

```
@input-border-radius:           0;
```

We can also change the color of the border and shadow on focus:

```
@input-border-focus:            #667591;
```

Personally, I think the shadow is a little excessive and it's quite easily removed by modifying one of Bootstrap's mixins. Previously, all of the framework's mixins were contained within a single file: `mixins.less`. However, that's recently changed and the mixins have been split into separate files.

In the `mixins` directory, open `forms.less` and look for `.form-control-focus`. This is the mixin that styles the focus on our form elements, you can see it in the following lines of code:

```
.form-control-focus(@color: @input-border-focus) {
  @color-rgba: rgba(red(@color), green(@color), blue(@color), .6);
  &:focus {
    border-color: @color;
    outline: 0;
    .box-shadow(~"inset 0 1px 1px rgba(0,0,0,.075), 0 0 8px
      @{color-rgba}");
  }
}
```

As you can see, the mixin changes the border color, removes the browser's default outline, and adds its own box shadow. Let's turn that box shadow into a comment for now:

```
//.box-shadow(~"inset 0 1px 1px rgba(0,0,0,.075), 0 0 8px @{color-
  rgba}");
```

Less allows us to use the `//` style comments and these won't be reflected in the output. If we use the standard style CSS comments, (`/* */`) these will be output.

We're also using a well within our Add Contact view, and it looks a little strange with that border radius now that our inputs lack one. We could remove the border radius from everything by setting the `@border-radius-base` variable to 0, but for now, let's open up the `wells.less` file. Within the well base class, set the `border-radius` to 0 to remove those rounded corners.

Buttons

As we've seen, Bootstrap includes many button sizes and colors. We can change all of this within the variables file; you can find the section by searching for `Buttons`—it should be found around line 140. However, Bootstrap uses the same colors for many of its components, and the section on colors starts at or around line 6.

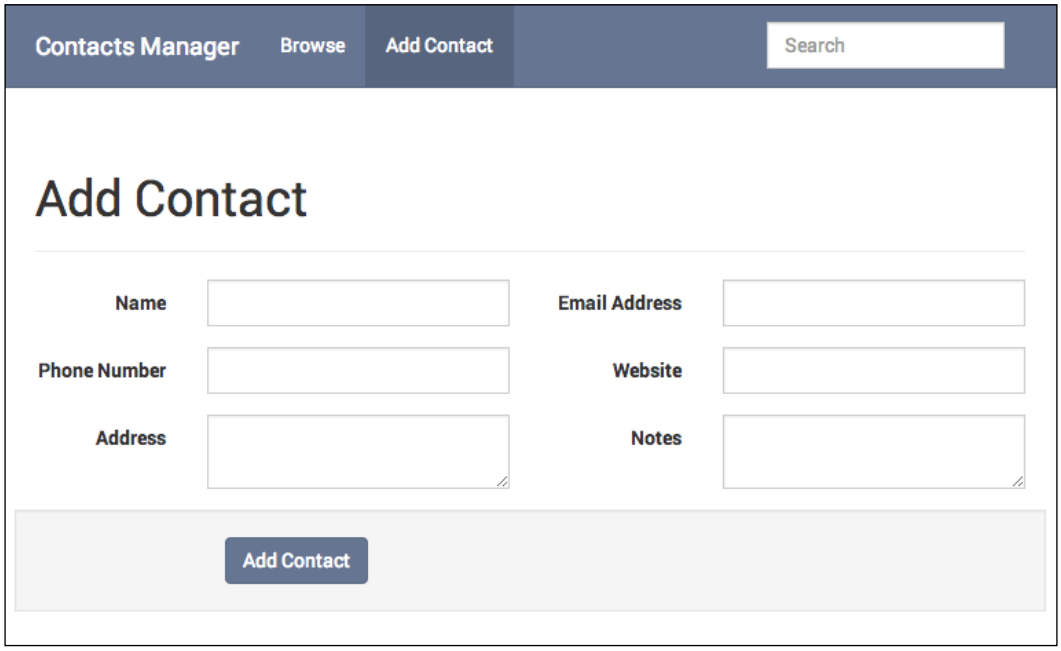
```
@brand-primary:         #428bca;
@brand-success:         #5cb85c;
@brand-info:            #5bc0de;
@brand-warning:         #f0ad4e;
@brand-danger:          #d9534f;
```

Let's change brand-primary to match our navbar background:

```
@brand-primary:         #667591;
```

Bootstrap uses the `darken` function to create the color for the button's border and hover colors.

Of course, we can customize anything within our styles, but for now, I think we'll leave it where it is. Our **Contacts Manager** is looking a lot less bland and generic thanks to its new splash of color and typeface. After all of our customization, our application looks like this:



The Bootstrap themes

One of the big changes in Bootstrap 3 was the removal of a lot of the visual styles. The styles were removed as Bootstrap is designed to be a framework to build upon, and not just be left with the default look.

However, these aren't gone for good and have been moved out to a separate file, which is included in the source. The `theme.less` file brings back the gradients from Bootstrap 2; we just need to import it to our main `Less` file.

Open up `bootstrap.less` and add the following to the bottom of the file to include the theme:

```
@import "theme.less";
```

For an example of what the Bootstrap theme does, check out `getbootstrap.com/examples/theme`.

Where to find additional Bootstrap themes

There are also a number of websites offering Bootstrap themes. These offer a quick and easy way to add a bit of character to the standard Bootstrap look. Take a look at the following sites if you want to try one of these themes on for size:

- `http://www.blacktie.co/`
- `https://wrapbootstrap.com/`
- `http://startbootstrap.com/`
- `http://bootswatch.com/`

Self-test questions

1. What are some of the main features Less adds?
2. How will you reference a pseudo class within a nested rule?
3. What do we need to do to change the font?
4. What does the `theme.less` file do?

Summary

In *Chapter 9*, *Using Task Runners*, we saw how we can use task runners to concatenate and minify our JavaScript into a single file. In this chapter, we saw how we could extend upon this to take Bootstrap's Less source and compile it into CSS.

This opened up the possibility to customize Bootstrap's look using Less' extension upon CSS—nested rules, variables, and mixins. We gave our application some character and also looked at how we can bring back some of the visual style from Bootstrap 2.

In the next chapter, we'll look at validation in AngularJS and how we can integrate it into our app.

11
Validation

Everything is working well and looking great, but currently there is no kind of validation for any of our forms or errors that could be sent back to us by the server. In this chapter, we'll take a look at how validation works in AngularJS and how we can combine it with the styles Bootstrap gives us to provide feedback to the user.

We'll also take a look at how we can expand upon the built-in rules by creating our own custom validator using what we've already learned in the previous chapters.

Form validation

One of the secret ingredients AngularJS brings to the table is native validation. There's basic validation of the most common HTML5 input types, alongside some custom directives such as `required`, `pattern`, and `minlength`, among others. We will look at adding these to our application and extending upon the built-in validation with a custom validator.

In order to utilize AngularJS's validation, we need to add a couple of things to our opening `form` tag:

```
<form name="addForm" novalidate class="form-horizontal" ng-
  submit="submit()">
```

The preceding line of code is the `form` tag from our `add.html` partial. We've added a `name` as well as the `novalidate` attribute. The `name` attribute assigns an object to the current scope, meaning that we can access it from our view and the controller. The `novalidate` attribute switches off the browser's native validation. We want to turn this off as we're going to handle the validation ourselves and don't want the default getting in the way of things or providing unintentional results.

Angular will automatically validate our e-mail input, and if we switch the website field from text to URL, AngularJS will match that for us too. We can quickly add that required attribute to any input we want to be mandatory.

```
<input type="text" id="name" class="form-control" ng-
  model="contact.name" required>
```

Alternatively, we could use `ng-required`. This will set the browser's required attribute to `true` if the AngularJS expression also equals true. For example, on a checkout, you might want to add a checkbox to allow a user to enter a different address for shipping and billing. When the box is checked, we can set the fields to `required` like this:

```
<label><input type="checkbox" ng-model="shippingAddress"> Send this to
  another address</label>
<div ng-show="shippingAddress">
    <input type="text" ng-required="shippingAddress">
</div>
```

As we don't need any conditions within our application, let's stick with the default `required` attribute. Quickly add it to the name, phone number, and e-mail address as these are going to be the most commonly needed fields within our contact.

We now need to stop the form from submitting when everything doesn't validate. We can do this either within our controller or by disabling the submit button.

By adding a name to our form, AngularJS has created a new model, which gives us direct access to it within our view. This model is a little different to the one we'd usually create as it opens up a load of properties to check the validity of not only of our form, but also specific elements should we wish to. Let's quickly see how we'd go about disabling that submit button:

```
<input type="submit" class="btn btn-primary" value="Add Contact"
  ng-disabled="addForm.$invalid">
```

We can use our new model within a directive. Here, it's `ng-disabled` and we're telling AngularJS to disable the button if the form isn't valid. We could have chosen to check for errors instead.

```
<input type="submit" class="btn btn-primary" value="Add Contact"
  ng-disabled="addForm.$error.required || addForm.$error.email">
```

The `$error` property of our model is a hash with the different types of errors the form is throwing. These can be e-mail validation errors, pattern matching errors, or missing fields. Of course, checking for each type of error here gives us greater possibility of making a mistake and uses a lot more code. However, it is more verbose, which can sometimes lead to better clarity.

We also have access to `$dirty` and `$pristine`. These two flags identify whether the user has entered anything in the form or not and can be for various things, including adding additional classes.

As this is just a model, we could also check whether or not our form is valid from within our `submit` function in `addCtl`:

```
$scope.submit = function(){

    if(!$scope.addForm.$valid){
        return window.alert('Error!');
    }

    $scope.contact.$save();
    $scope.contact = contacts.create();
    alert.show();
};
```

If you remove the `ng-disabled` attribute, and then click on the button, you should be presented with a browser alert box. This is great, but we could make it tidier and look like it is part of the application. We already have a successful Bootstrap alert message; let's add it in a second when a validation error occurs.

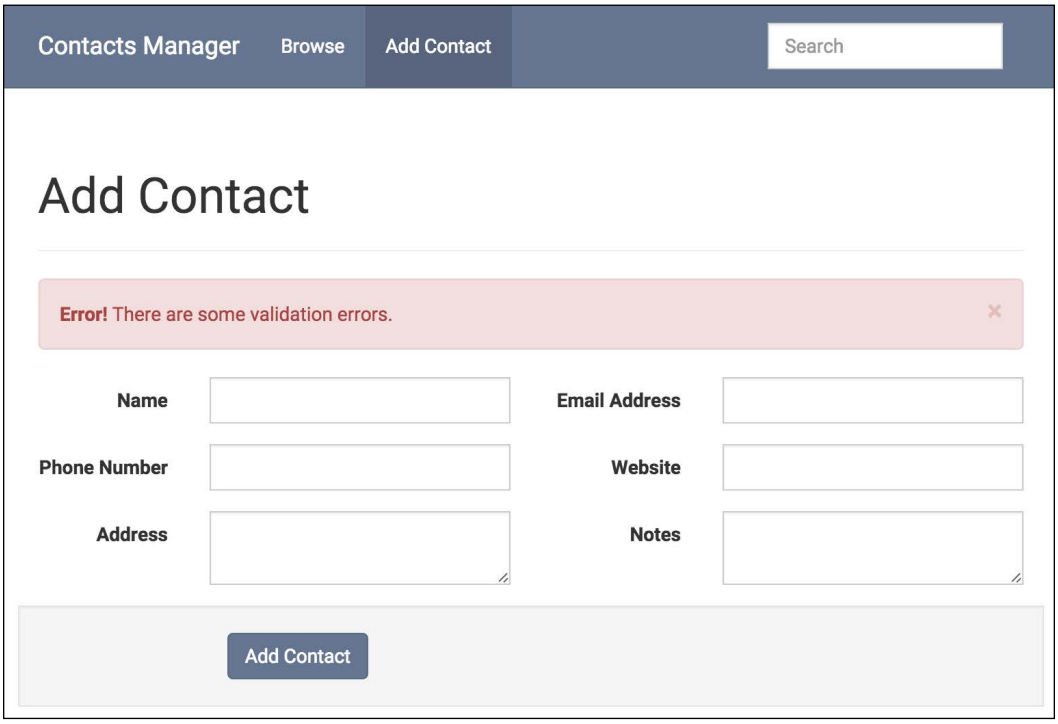
Replace the alert variable at the top of your `add.js` file with a new object containing both our success and error alerts:

```
var alerts = {
    success: $alert({
        title: 'Success!',
        content: 'The contact was added successfully.',
        type: 'success',
        container: '#alertContainer',
        show: false
    }),
    error: $alert({
        title: 'Error!',
        content: 'There are some validation errors.',
        type: 'danger',
        container: '#alertContainer',
        show: false
    })
}
```

Now we can swap out that old alert call and the native browser alert within our controller's `submit` method:

```
$scope.submit = function(){

    if(!$scope.addForm.$valid){
        return alerts.error.show();
    }

    $scope.contact.$save();
    $scope.contact = contacts.create();
    alerts.success.show();
};
```

If you reload your application and hit that submit button, you'll be presented with a Bootstrap alert box informing you that there are some validation errors.



Of course, it would be great to know what these errors were. Thankfully, AngularJS lets us see exactly what models are throwing errors and we can present errors or style accordingly.

We can access each input within our form. However, we do need to define a name for these to be able to validate. For example, if we can add a name of the phone to our phone number field, we can then validate the field by accessing it via the form:

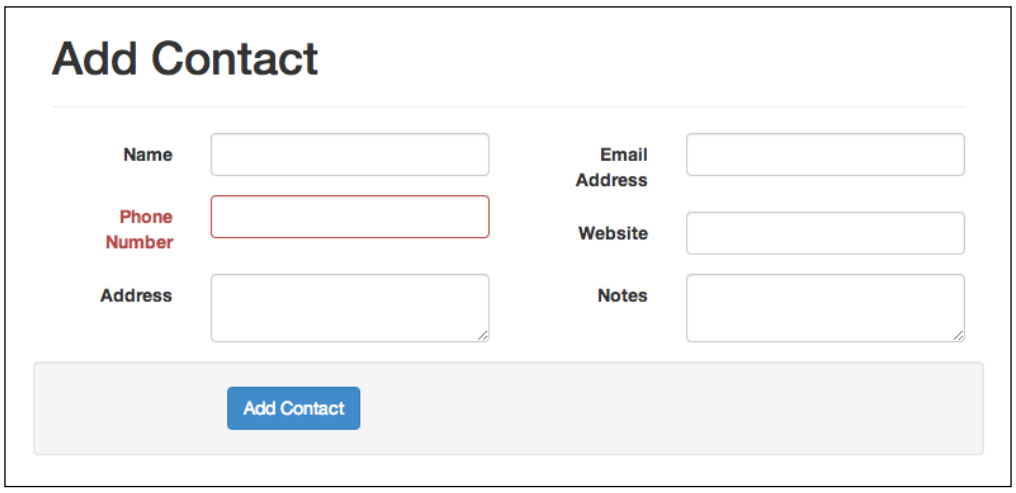
```
addForm.phone.$valid
```

We can use the `ng-class` directive on our form groups to check our input's validity and add the `has-error` class should it fail.

AngularJS does add its own classes to form elements based on validity, but as we want to utilize the Bootstrap `has-error` class, we've opted to use `ng-class`.

```
<div class="form-group" ng-class="{'has-error':
  !addForm.phone.$valid}">
```

Unfortunately, this will add the error class by default, and that probably isn't what we're after.



We can qualify this by setting a second model to `true` if the form is invalid when we submit it.

```
$scope.submit = function(){

    $scope.formErrors = false;

    if(!$scope.addForm.$valid){
        $scope.formErrors = true;
```

```
            return alerts.error.show();
        }

        $scope.contact.$save();
        $scope.contact = contacts.create();
        alerts.success.show();
    };
```

Notice that we set `formErrors` to `false` to begin with, in order to remove the error classes should the form be validated correctly. We can now change our `ng-class` directive to look at both the validations and the new model.

```
<div class="form-group" ng-class="{'has-error': formErrors &&
  !addForm.phone.$valid}">
```

If both inputs are invalid and the `formErrors` model is set to `true`, the class will be added. Let's quickly add the `ng-class` directive to all of the form groups we're validating. Don't forget to change the model we're referencing to what we've put inside the name attribute.

Pattern validation

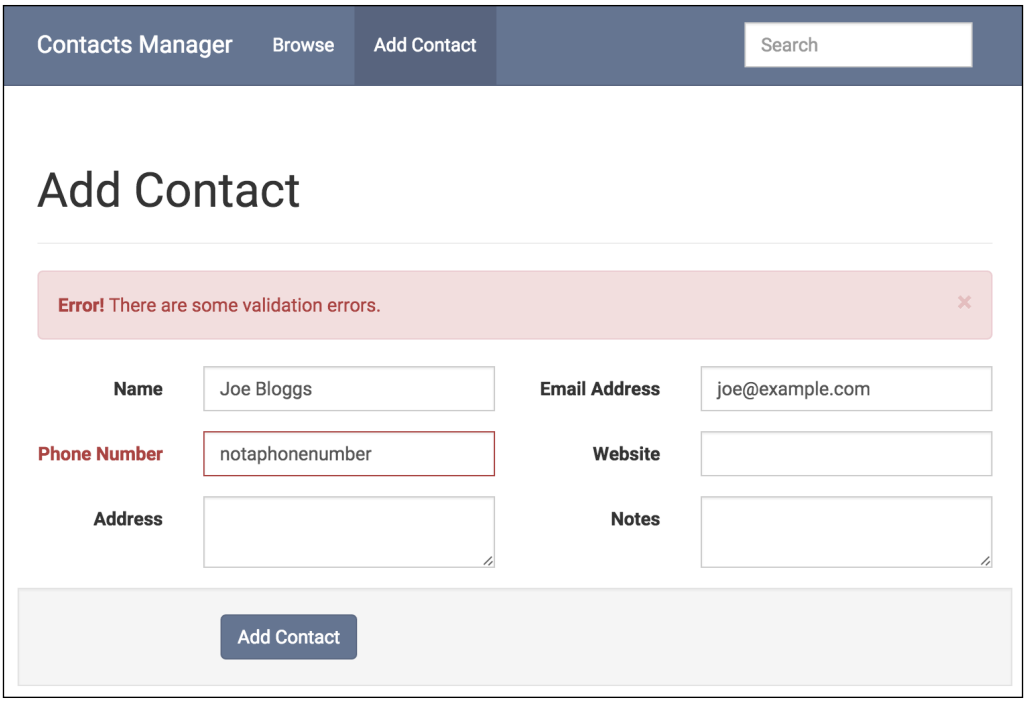
We've got our basic validation underway but, as we've already learnt, Angular gives us a few additional directives that can make our validation much stricter. For example, we can currently type anything we want into that field and it will let us add it. Of course, this isn't ideal, as we want to ensure that we have a phone number for each contact.

The `ng-pattern` directive allows us to define a regular expression (REGEX) pattern to match the input against. For our phone number, we, of course, want to accept digits, but also a plus sign for international numbers and parenthesis for optional numbers or US-formatted phone numbers. We also need to allow spaces and dashes to split up the number.

First of all, let's add that `ng-pattern` directive to the phone input and limit it to just integers for the time being:

```
<input type="tel" name="phone" id="phone" class="form-control" ng-
  model="contact.phone" required="true" ng-pattern="/^[0-9]/">
```

Now, when we enter text into our field and submit the form, you'll notice that AngularJS throws a validation error.



It's only when we enter a number that the form can be submitted. Of course, we still need to allow the additional characters that can sometimes be found within a phone number. This is just a case of adding the characters we want to allow into the square brackets. Here's the final regular expression:

```
/^[0-9+()-]/
```

We could build upon this and force a certain structure or maxlength, but I think this achieves exactly what we're looking for.

Using minlength, maxlength, min, and max

The other four directives AngularJS gives us for validation aren't nearly as exciting, but they can prove to be handy. For example, the `minlength` directive is perfect for enforcing password security and min/max proves to be handy in a shopping cart situation —only allowing a minimum of one and a maximum of whatever's in stock.

All four directives are used in the exact same way and accept a number as their value. Both `ng-minlength` and `ng-maxlength` look at the length of what's been input, whereas `ng-min` and `ng-max` look at the actual numerical value.

Here they are in action. As with all directives, you can use a combination to achieve the desired rules.

```
<input type="number" ng-min="1" ng-max="5">
<input type="password" ng-minlength="8" ng-maxlength="255">
```

Creating a custom validator

AngularJS does a great job of covering the majority of input types and use cases. However, there are times when you want a little more control. The real power comes with the ability to create our own custom validations. These custom validators are just directives with one special requirement. In order for them to work, there has to be an `ng-model` set on the element.

Since our application has no real need for a custom validator, let's look at how we can create a validator to ensure the input isn't in a preset list. This can be handy for ensuring a username is unique.

Let's call this directive `uniqueList`, and put it within our `contactsMgr.directives.js` file. We're going to restrict it to attribute only and utilize the `link` method. If you need a quick refresher, jump back to *Chapter 6*, *CRUD*, where we covered creating custom directives.

We can inject a controller as the fourth parameter of our `link` function. Angular knows which controller we're planning to use by looking at the directive's `require` property. We've set this to `ngModel`, so we have direct access to an API for the `ng-model` directive:

```
.directive('uniqueList', function(){
    return {
        restrict: 'A',
        require: 'ngModel',
        link: function(scope, elem, attrs, ctrl){

        }
    };
});
```

The AngularJS documentation has sections on all of the core controllers. The information on `ngModelController` can be found at `https://docs.angularjs.org/api/ng/type/ngModel.NgModelController`.

Within is a key method that we can take advantage of: `$setValidity`. This lets us define whether our model is valid or not. We're going to use this in conjunction with `scope.$watch` to check the validity whenever the model changes.

Let's first set that `scope.$watch` up within our directive's link method. We can fetch the name of our model from the `attrs` object:

```
.directive('uniqueList', function(){
    return {
        restrict: 'A',
        require: 'ngModel',
        link: function(scope, elem, attrs, ctrl){
            scope.$watch(attrs.ngModel, function(value){

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

As we already know, we can get access to the new value and only value from a watcher. For this example, we only need the new value. We'll be checking this against a list of usernames. Of course, this could be an HTTP request to the server, but for now we're going to hardcode an array or names:

```
.directive('uniqueList', function(){
    var usernames = [
        'bob',
        'john',
        'paul'
    ];

    return {
        restrict: 'A',
        require: 'ngModel',
        link: function(scope, elem, attrs, ctrl){
            scope.$watch(atts.ngModel, function(value){

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

The validity of the model is just a simple check to see if the value is in the array:

```
var valid = (usernames.indexOf(value) > -1) ? false : true;
```

The `$setValidity` method on the `ngModel` controller is very simple with just two parameters. The first is the error key and the second is a Boolean as to whether it's valid or not. Let's hook it up:

```
var valid = (usernames.indexOf(value) > -1) ? false : true;
ctrl.$setValidity('uniqueList', valid);
```

Now that everything is hooked up, let's take a look at our finished directive and how we can use it:

```
.directive('uniqueList', function(){
    var usernames = [
        'bob',
        'john',
        'paul'
    ];

    return {
        restrict: 'A',
        require: 'ngModel',
        link: function(scope, elem, attrs, ctrl){
            scope.$watch(atts.ngModel, function(value){
                var valid = (usernames.indexOf(value) > -1) ? false :
                  true;
                ctrl.$setValidity('uniqueList', valid);
            });
        }
    };
});
```

To use the directive, we just attach it to our input via an attribute:

```
<input type="text" ng-model="contact.name" unique-list>
```

Self-test questions

1. Name three directives that can be used for validation
2. How do we check the validity of our form?
3. How do we gain access to the `ngModel` controller when creating a custom validator?
4. How would we check our input against a regular expression?

Summary

We've taken a look at how validation works in AngularJS and how we can combine it with Bootstrap's styles to make our app more user friendly. Form validation was made easy with AngularJS's built-in directives and automatic validation of HTML5 inputs, such as `email` and `tel`.

We were able to check the validity of our form on submission and display appropriate error messages and warnings should it have failed. While the built-in validators are great for most use cases, we also looked at how we could build our own via a directive.

With our application complete, we'll take a look at a few community-built tools in the next chapter to make our lives easier while working with AngularJS.

12
Community Tools

Our contact manager app is now complete. We've gone from a blank page to a fully-fledged single-page CRUD application that connects to the server and validates perfectly. This chapter will highlight a couple of incredibly powerful and useful community tools that will make our lives easier when using AngularJS.

We'll setup Batarang and ng-annotate to work with our project. Batarang will give us a better look at our scope amongst other things, and ng-annotate will allow us to minify our JavaScript much more easily. This chapter will cover the following topics:

- Learning about Batarang and ng-annotate
- Installing Batarang and ng-annotate
- Inspecting the scope
- Monitoring our application's performance
- Utilizing ng-annotate with Grunt and gulp

Batarang

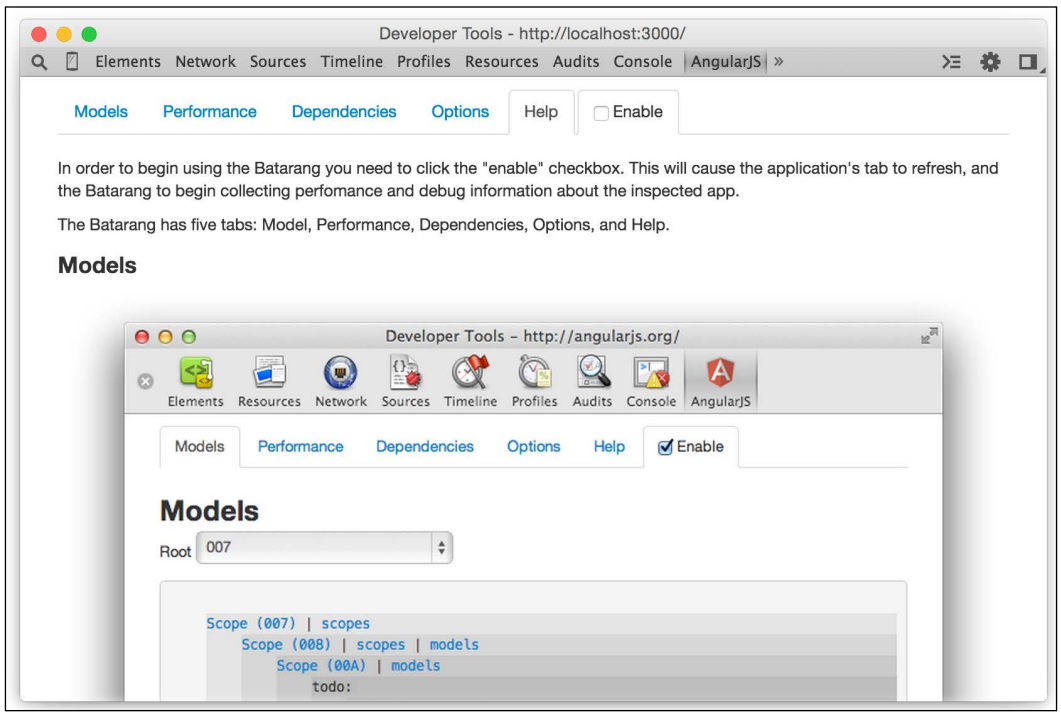
Batarang is a Chrome extension (sorry Firefox and Safari users, AngularJS is a Google project after all) that reveals an extra tab in our developer tools to enable us to profile and debug AngularJS apps.

Installing Batarang

Batarang is easy to install, as it's just a Chrome extension. Let's take a look at how we can get it hooked up:

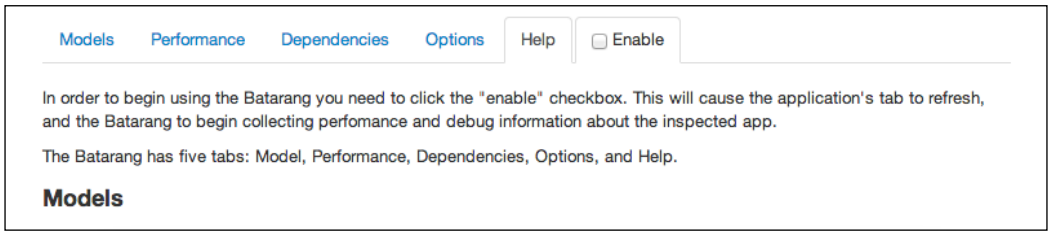
1. First, head over to `https://chrome.google.com/webstore/` and search for Batarang in the box on the top-left corner of the page. It should be the only result under the extensions section, which is shown in the following screenshot:

2. Click the button to install, and you'll now have a new tab in your web inspector, as shown in the following screenshot:



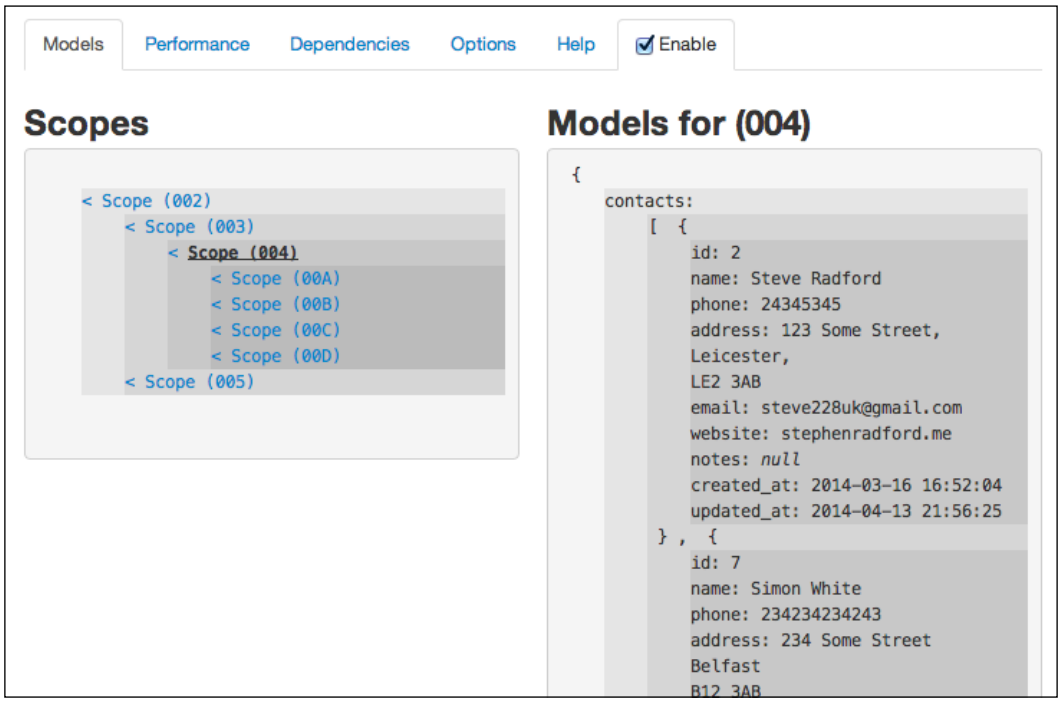
Inspecting the scope and properties

Perhaps the handiest feature Batarang provides is the ability to inspect the various scopes within our application. The extension adds a new tab to our web inspector; let's take a look inside:



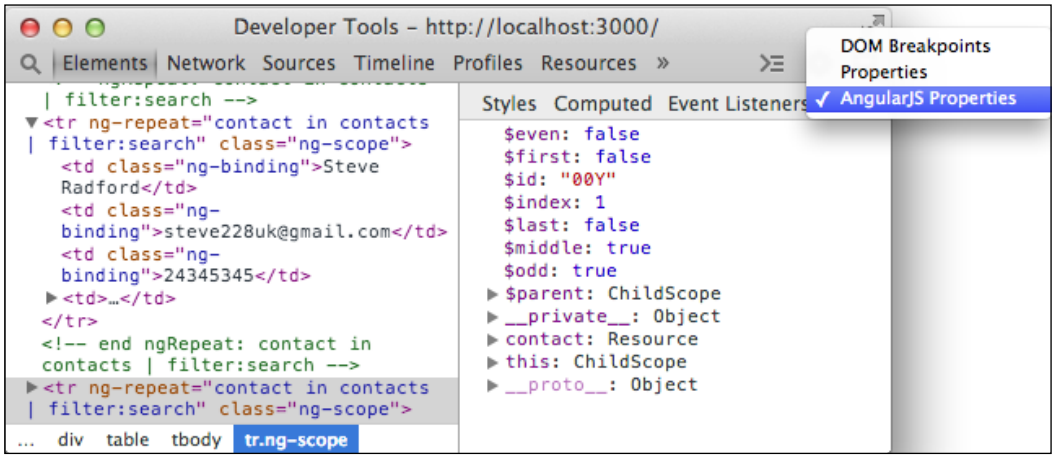
Batarang is split into five tabs: **Models**, **Performance**, **Dependencies**, **Options**, and **Help**. To use Batarang within our application, we need to check the **Enable** box. This will refresh the page and allow the extension to begin collecting the information it needs.

There are three ways we can inspect the scope and properties using Batarang. The most obvious way is to hit the **Models** tab where you'll be presented with a list of all your nested scopes.

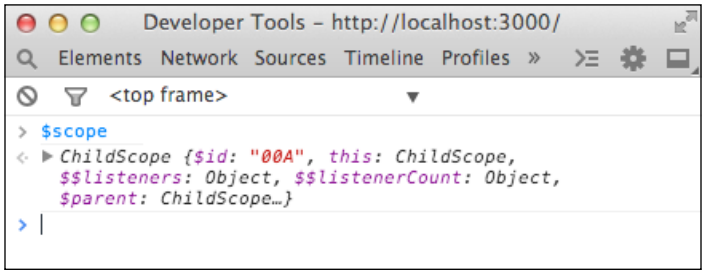


On the left of the **Models** tab is a list of all of the scopes on your page. Selecting these will show the models contained within this scope on the right. These update automatically whenever a value is changed on the page, making it extremely easy to see exactly what's going on with your models.

Batarang also adds an extra tab when inspecting our elements. This is called **AngularJS Properties** and shows everything Angular is associating with that element: things like detecting if it's even, the index of the item in an ng-repeat, or if it's a form element that's valid or invalid, as shown in the following screenshot:



The final thing Batarang gives us in terms of inspection is a handy little tool within our console. Once Batarang is enabled, you'll be able to type `$scope` in your console to see the scope for the last selected element, as shown in the following screenshot:

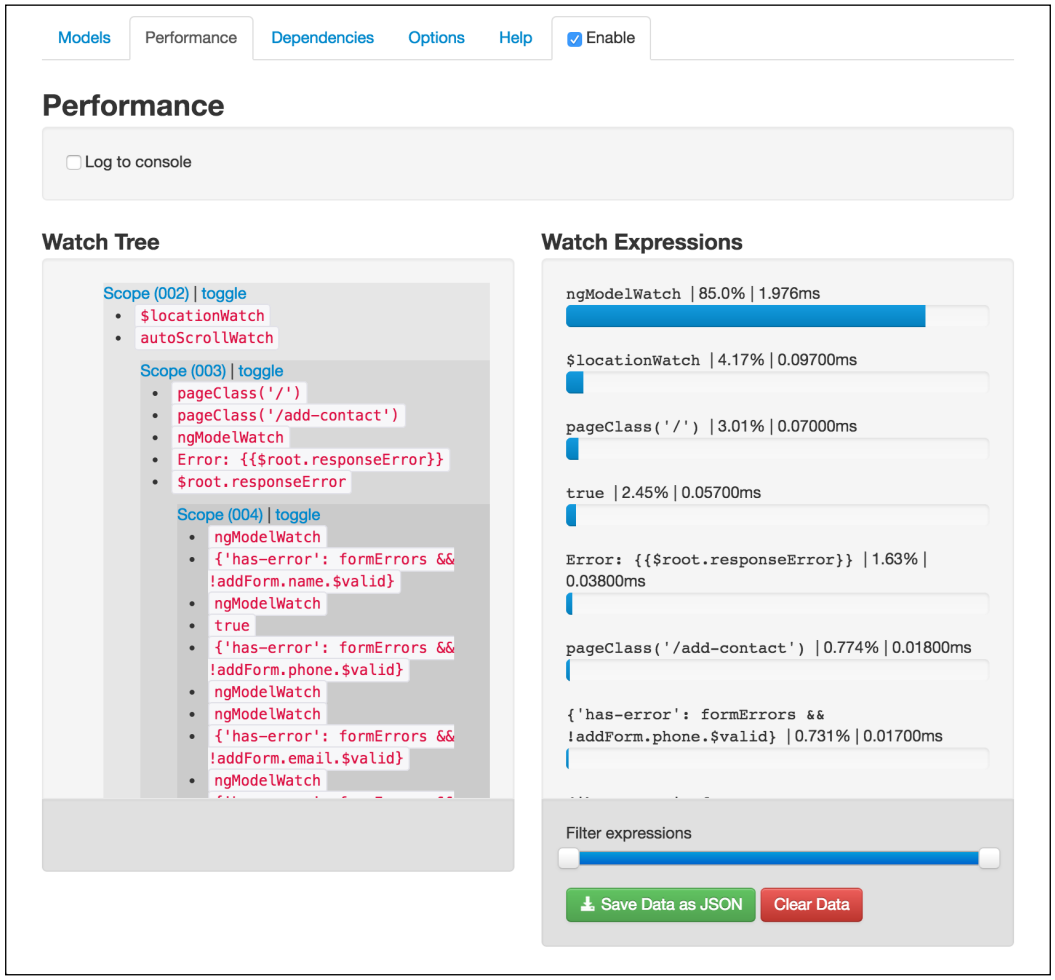


Monitoring performance

The **Performance** tab provides us with a handy list of everything that's being watched for changes by Angular, as well as a list of expressions with how long everything is taking to execute.

As with everything, it's worth keeping an eye on performance as your app grows. You might find you're being a bit too overzealous with the `$scope.$watch` functions or that one of your filters is taking a long time to execute.

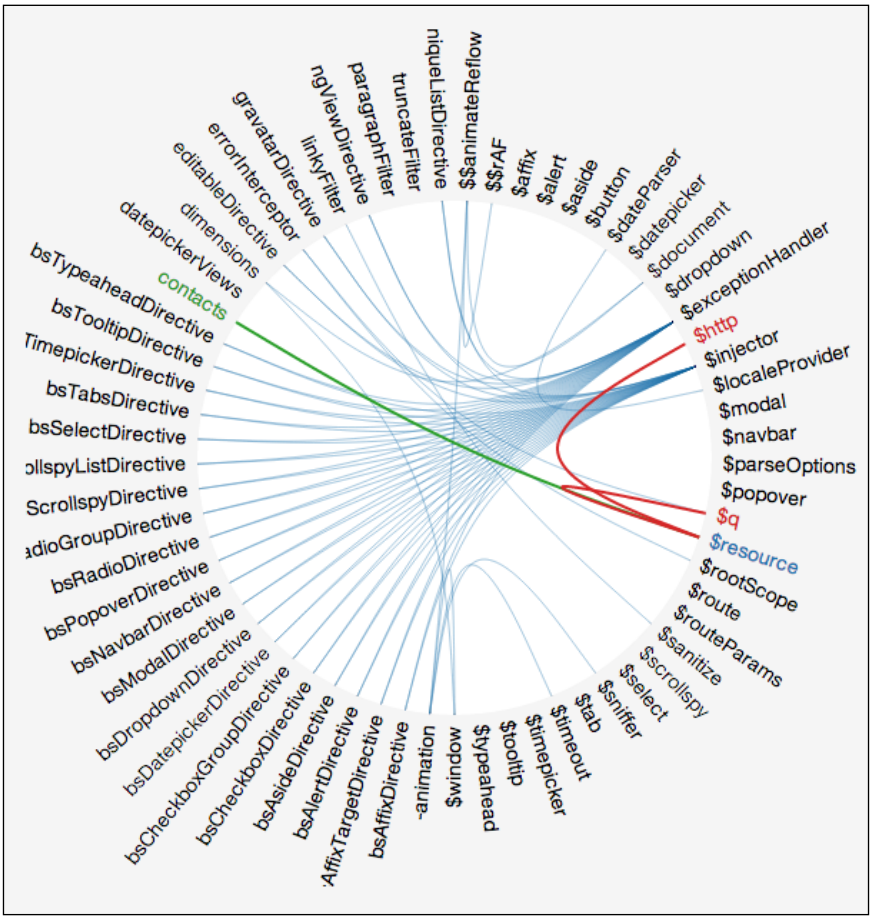
Here's a screenshot of the **Performance** tab from the **Add Contact** page of our application. On the left, you can see the **Watch Tree** section. This is a list of all the watchers on the page and displays which scope they're in. On the right, you can see a breakdown of which expressions are taking the longest.



Visualizing dependencies

The coolest tab in Batarang is without a doubt the **Dependencies** tab. This shows an interactive graph of everything your services rely over within your application. When you hover upon a service name, the names of services that rely on the hovered service are highlighted in green, and services depended upon by the service are highlighted in red.

In the following screenshot, we've hovered over **$resource**, which has highlighted **$http** and **$q** in red and contacts in green:



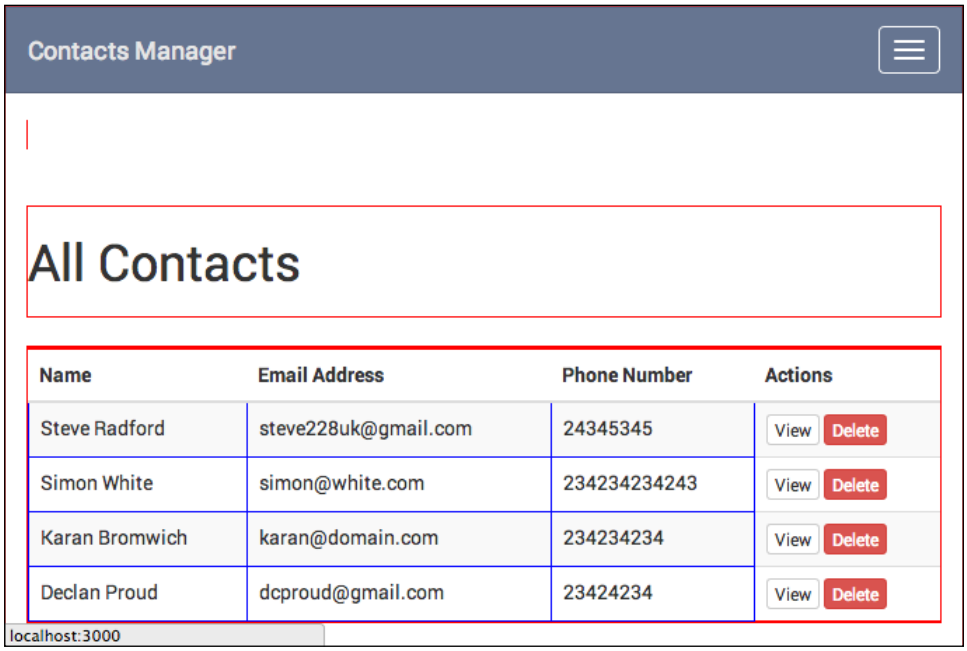
You'll probably have noticed that in the preceding screenshot there are a number of services that don't depend on anything and aren't being depended upon either. This can happen quite a lot, but it can also mean that these services aren't being utilized by your application, so they're worth keeping an eye on.

Batarang options

Tucked away under the **Options** tab are three handy checkboxes. These three boxes highlight applications, bindings, and scopes on our page by placing a colored border around them.

Applications have a green border; a binding is given a blue border; and scopes a red one. This can be powerful to see if a certain part of your app is in one scope or another, or if a binding is actually happening.

Here's what your app might look like with everything enabled:



ng-annotate

The ng-annotate is an open source project by Olov Lassus designed to remove one of the most frustrating side effects of using AngularJS. You'll remember that when minifying our code, we had to wrap our dependencies in an array so the names didn't get mangled. The project removes this requirement by looking at our code and wrapping it for us. In short, the tool is a pre-minifier and should be used to prepare our code for uglification.

Previously, we used ngMin by Brian Ford to accomplish the same thing. That project has now been deprecated in favor of ng-annotate.

Installing ng-annotate

The ng-annotate tool is an `npm` package and can be installed via our command line. There are also additional packages that work with our Grunt and gulp setups, but for now let's see how we can run the project manually:

1. Open up your terminal and run the following command to install the package globally on your computer:

   ```
   npm install -g ng-annotate
   ```

   Remember, if this returns a permissions error, you'll need to run it as admin. This can be done with the `sudo` command on a *nix based system or by running command prompt as an administrator on Windows.

   This will give you a new command to use within your terminal. If you run `ng-annotate`, you should be shown the help file for the tool. Using ng-annotate on a file is easy—we just use the `ng-annotate` command followed by our options and the name of the file we wish to process.

2. Change into your project's `js` directory and run the following:

   ```
   ng-annotate -r  controllers/app.js
   ```

   This should return the contents of the app controller but with one slight change—you will notice that all of the existing annotations around the dependencies have been removed. That's because we told ng-annotate to remove them with the `-r` option.

3. We can also get ng-annotate to add its own back in with `-a`. These two options can be used in tandem or on their own:

   ```
   ng-annotate -ra  controllers/app.js
   ```

4. This time, we get our controller with annotations returned. We can tell that ng-annotate has worked its magic, as the single quotations around our dependencies have been replaced with double quotes.

Okay, so `ng-annotate` works but it's pretty useless if we have to run it manually. How can we bring this into our task runners? Let's find out!

Using ng-annotate with Grunt

In order to use ng-annotate with Grunt, we need to install another `npm` package. This time though it's not global—we have to install it to our project. To do so perform the following steps:

1. First, we need to change into our project folder in the terminal:

   ```
   cd ~/path/to/contacts-manager
   ```

2. Now we can install the Grunt module, saving it to our project's dev dependencies so we can always install again later, or another developer working on the project can install everything required:

   ```
   npm install grunt-ng-annotate --save-dev
   ```

3. Before we make any changes to our Gruntfile, we mustn't forget to load the tasks from our newly installed package:

   ```
   grunt.loadNpmTasks('grunt-ng-annotate');
   ```

Configuring the task

All that's left to do now is configuring the `ngAnnotate` task within our Gruntfile and making a couple of tweaks to our watch task. Here's how we go about it:

1. Let's add it under our `less` task within the `grunt.initConfig` object.

   ```
   ngAnnotate: {

   }
   ```

2. Like most Grunt tasks, `ngAnnotate` accepts an options hash as well as multiple targets. Let's set up option first:

   ```
   ngAnnotate: {
       options: {
           remove: true,
           add: true,
           singleQuotes: true
       }
   }
   ```

   We've set three options here. We've opted to remove any pre-existing annotations, to add new ones with ng-annotate, and to use single quotations over double. Also available is the ability to define our own regular expression if we want to target a specific section.

3. Our target is going to look at all files within our `js` directory and save them with the `.annotated.js` extension. This way we can set up our watch task to run ng-annotate and then set uglify to look for files ending with `.annotated.js`.
4. The only files we really need to pay attention to are ours. Either everything within the vendor directory will have been pre-annotated, or it isn't Angular related. As such, we can point Grunt to look at everything except our vendor and build folders:

```
ngAnnotate: {
    options: {
        remove: true,
        add: true,
        singleQuotes: true
    },
    app: {
        src: [
            'assets/js/**/*.js',
            '!assets/js/vendor/*.js',
            '!assets/js/build/*.js'
        ]
    }
}
```

5. The first path within our `src` array includes all JS files within our project's `js` directory. As we figured out, we need to ignore the vendor and build folders. The two items that follow in our array do just this. The exclamation mark before the path tells Grunt we want to exclude all JS files within these folders.
6. Now that we've specified our source, we also need to tell ng-annotate what we want to do with our files once they've been processed. We already know that we want to rename the file to include the `.annotate.js` extension and then save them in the same place:

```
ngAnnotate: {
    options: {
        remove: true,
        add: true,
        singleQuotes: true
    },
    app: {
        src: [
            'assets/js/**/*.js',
            '!assets/js/vendor/*.js',
            '!assets/js/build/*.js'
        ],
        expand: true,
        ext: '.annotated.js',
```

```
            extDot: 'last'
        }
    }
```

7. In the preceding code, we've added three new properties to our target: `expand`, `ext`, and `extDot`. The `expand` property splits the path and allows us to change the extension. The `ext` property changes the extension, and the `extDot` property tells Grunt which dot in the filename to look at. In our case, it's the last one, but this covers us should we use multiple dots in our filenames.
8. Okay, now we're ready to run our task in the terminal. If we run it with the `--verbose` flag, we can see exactly what's happening:

```
grunt ngAnnotate --verbose
```

9. Everything looks like it's working perfectly, and while it is creating our new `.annotated.js` files, we're going to run into some trouble when we run our task for a second time. The following will be output when you run the preceding command again:

```
Writing assets/js/modules/contactsMgr.services.annotated.
  annotated.js
```

10. The task has looked at not only the `.js` files in our directories but also the `.annotated.js` files. That's because Grunt doesn't know the difference. It looks at the last period to determine the extension, so we just need to add one more exclusion to our `src` array to complete our task:

```
ngAnnotate: {
    options: {
        remove: true,
        add: true,
        singleQuotes: true
    },
    app: {
        src: [
            'assets/js/**/*.js',
            '!assets/js/**/*.annotated.js',
            '!assets/js/vendor/*.js',
            '!assets/js/build/*.js'
        ],
        expand: true,
        ext: '.annotated.js',
        extDot: 'last'
    }
}
```

11. Delete the additional `.annotated.annotated.js` files from your `js` directory and run the task one final time.

Hooking into our watch task

Now that our `ngAnnotate` task is working as expected, we can make a few tweaks to our `watch` and `uglify` tasks to run it automatically and minify our JavaScript, as follows:

1. The first thing we need to do is change the js target in our watch task to run `ngAnnotate` instead of `uglify`:

```
js: {
    files: [
        'assets/js/**/*.js'
    ],
    tasks: ['ngAnnotate']
},
```

2. We also need to add a couple of exclusions to our files array here. We don't need to watch our `build` directory, and we also need to tell Grunt to disregard any files ending with `.annotated.js`:

```
js: {
    files: [
        'assets/js/**/*.js',
        '!assets/js/build/*.js',
        '!assets/js/modules/**/*.annotated.js',
        '!assets/js/controllers/**/*.annotated.js'
    ],
    tasks: ['ngAnnotate']
},
```

3. Let's quickly test that everything is working. In your terminal, start up the `watch` task:

```
grunt watch
```

4. If you now save one of your JS files like your app controller, Grunt should detect there's been a change and fire up `ngAnnotate`:

```
Running "watch" task
Waiting...OK
>> File "assets/js/controllers/app.js" changed.

Running "ngAnnotate:app" (ngAnnotate) task
>> 8 files successfully generated.

Done, without errors.
```

5. Okay, that seems to be working well. Next, we can create a new target to look at our annotated files for changes and run our uglify task:

```
annotated: {
    files: [
        'assets/js/**/*.annotated.js',
    ],
    tasks: ['uglify']
},
```

6. This one is pretty easy and just looks for any changes in files with the `.annotated.js` extension. Lastly, we just need to make a couple of changes to the src array in uglify's build target:

```
src: [
    'assets/js/vendor/jquery.js',
    'assets/js/vendor/bootstrap.js',
    'assets/js/vendor/angular.js',
    'assets/js/vendor/angular-animate.js',
    'assets/js/vendor/angular-resource.js',
    'assets/js/vendor/angular-route.js',
    'assets/js/vendor/angular-sanitize.js',
    'assets/js/vendor/angular-strap.js',
    'assets/js/vendor/angular-strap.tpl.js',
    'assets/js/modules/*.annotated.js',
    'assets/js/controllers/*.annotated.js'

],
```

7. We've just made a couple of changes to the last two items in the array to look at the generated files rather than their un-annotated siblings. Okay, let's run the grunt watch task again and re-save one of our JS files to see what happens:

```
Running "watch" task
Waiting...OK
>> File "assets/js/controllers/app.js" changed.

Running "ngAnnotate:app" (ngAnnotate) task
>> 8 files successfully generated.

Done, without errors.
Running "uglify:build" (uglify) task
File "assets/js/build/ContactsMgr.js" created.

Done, without errors.
```

8. When we saved our `app.js` file, Grunt detected a change and ran the `ngAnnotate` task, generating eight files. Next, the `watch` task saw that there were changes to the annotated files that had just been generated and built our `ContactsMgr.js` file by running the `uglify` task. With all those changes, here's what our `Gruntfile.js` now looks like:

```
module.exports = function(grunt){

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    watch: {
      js: {
        files: [
        'assets/js/**/*.js',
        '!assets/js/build/*.js',
        '!assets/js/**/*.annotated.js'
        ],
        tasks: ['ngAnnotate']
      },
      annotated: {
        files: [
        'assets/js/**/*.annotated.js',
        ],
        tasks: ['uglify']
      },
        less: {
        files: [
        'assets/less/*.less'
        ],
        tasks: ['less:dev']
      },
        css: {
          files: [
          'assets/css/bootstrap.css'
          ],
        options: {
          livereload: true
        }
      }
    },
      uglify: {
        options: {
        banner: '/*! <%= pkg.name %> <%=
          grunt.template.today("yyyy-mm-dd") %> */\n'
```

```
    },
      build: {
      src: [
      'assets/js/vendor/jquery.js',
      'assets/js/vendor/bootstrap.js',
      'assets/js/vendor/angular.js',
      'assets/js/vendor/angular-animate.js',
      'assets/js/vendor/angular-resource.js',
      'assets/js/vendor/angular-route.js',
      'assets/js/vendor/angular-sanitize.js',
      'assets/js/vendor/angular-strap.js',
      'assets/js/vendor/angular-strap.tpl.js',
      'assets/js/modules/**/*.annotated.js',
      'assets/js/controllers/**/*.annotated.js'
      ],
      dest: 'assets/js/build/<%= pkg.name %>.js'
    }
  },
    less: {
      dev: {
      files: {
      'assets/css/bootstrap.css':
        'assets/less/bootstrap.less'
    }
  },
    production: {
    options: {
    cleancss: true
    },
    files: {
      'assets/css/bootstrap.css':
        'assets/less/bootstrap.less'
    }
  }
},
  ngAnnotate: {
    options: {
      remove: true,
      add: true,
      singleQuotes: true
      },
      app: {
        src: [
          'assets/js/**/*.js',
```

```
                '!assets/js/**/*.annotated.js',
                '!assets/js/vendor/*.js',
                '!assets/js/build/*.js'
                ],
                  expand: true,
                  ext: '.annotated.js',
                  extDot: 'last'
              }
          }
      });

      grunt.loadNpmTasks('grunt-contrib-uglify');
      grunt.loadNpmTasks('grunt-contrib-watch');
      grunt.loadNpmTasks('grunt-contrib-less');
      grunt.loadNpmTasks('grunt-ng-annotate');

      grunt.registerTask('default', ['ngAnnotate', 'uglify']);

};
```

Using ng-annotate with gulp

Just as with everything we've seen so far, there's also a gulp version of ng-annotate that will let us use this alternative task runner, if that's your preference. Let's see how we can set this up to work with our pre-existing gulpfile:

1. First, let's grab the package from `npm`:

   ```
   npm install gulp-ng-annotate --save-dev
   ```

2. Open up `gulpfile.js` and pull in the package we just installed:

   ```
   var ngAnnotate = require('gulp-ng-annotate');
   ```

3. This will give us a new function we can use within gulp's pipes. It's quite unbelievable how much quicker gulp is to set up to use ng-annotate in comparison to Grunt. All we need to do is add in a new pipe to our `uglify` task:

   ```
   gulp.task('uglify', function(){
       gulp.src(paths.js)
       .pipe(concat(pkg.name+'.js'))
       .pipe(ngAnnotate())
       .pipe(uglify())
       .pipe(gulp.dest('assets/js/build'));
   });
   ```

4. The pipe with the `ngAnnotate` function in can be dropped in just before our `uglify` pipe. Our `watch` task is already set up to run our `uglify` task whenever a JS file is changed, so that really is all we need to do to gulpfile!
5. We can now either run gulp `uglify` to manually run ng-annotate and minify our JavaScript, or use gulp `watch` to automatically detect changes. After those two small changes, here's what our `gulpfile.js` file looks like:

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var concat = require('gulp-concat');
var pkg = require('./package.json');
var less = require('gulp-less');
var livereload = require('gulp-livereload');
var ngAnnotate = require('gulp-ng-annotate');

var paths = {
    js: [
        'assets/js/vendor/jquery.js',
        'assets/js/vendor/bootstrap.js',
        'assets/js/vendor/angular.js',
        'assets/js/vendor/angular-animate.js',
        'assets/js/vendor/angular-resource.js',
        'assets/js/vendor/angular-route.js',
        'assets/js/vendor/angular-sanitize.js',
        'assets/js/vendor/angular-strap.js',
        'assets/js/vendor/angular-strap.tpl.js',
        'assets/js/modules/**/*.js',
        'assets/js/controllers/**/*.js'

    ],
    less: 'assets/less/**/*.less'
};

gulp.task('uglify', function(){
    gulp.src(paths.js)
    .pipe(concat(pkg.name+'.js'))
    .pipe(ngAnnotate())
    .pipe(uglify())
    .pipe(gulp.dest('assets/js/build'));
});

gulp.task('watch', function(){

    var server = livereload();
```

```
    gulp.watch(paths.js, ['uglify']);
    gulp.watch(paths.less, ['less']);
    gulp.watch('assets/css/bootstrap.css').on('change',
      function(file){
        server.changed(file.path);
    });

});

gulp.task('less', function(){
    gulp.src('assets/less/bootstrap.less')
    .pipe(less({
        filename: 'bootstrap.css'
    }))
    .pipe(gulp.dest('assets/css'));
});

gulp.task('default', ['uglify']);
```

Self-test questions

1. What is Batarang?
2. Name three tools Batarang gives you.
3. What are the two ways we can inspect the scope?
4. What did we use before ng-annotate?
5. What options does ng-annotate give us?
6. What problem does ng-annotate solve?

Summary

We've taken a look at two powerful, heavily recommended, and used community tools in this chapter. Batarang gives you the Swiss Army knife of debugging tools to help you build awesome web apps with Angular, and ng-annotate gets rid of that annoying quirk when minifying files.

Of course, both of these tools are optional and aren't necessary when using Angular, but both will help you along the way and you'll no doubt find yourself using them regularly. These are just two of the tools that have stemmed from an amazing AngularJS community. Explore and find out what else is out there to help you, and if you can't quite find what you're looking for, build it and give back to the community.

A

People and Projects to watch

Both AngularJS and Bootstrap have a massive following and two huge communities behind them. This means that there are plenty of projects surrounding both frameworks. Of course, behind every project are dedicated developers, so let's take a look at some of the people and projects to watch out for.

Bootstrap projects and people

Bootstrap is now in its third version, which means there are plenty of extensions to the core as well as handy tools to help you along the way.

The core team

Bootstrap started out as Twitter Bootstrap and was the brainchild of two engineers at the company: Mark Otto (`@mdo`) and Jacob Thornton (`@fat`). Both engineers have subsequently left the company but continue to contribute to the project.

There have been almost 600 contributors to Bootstrap at the time of writing, which shows just how powerful open source software can really be. Both of the original creators have now left Twitter, but Mark remains the most active contributor and maintainer of Bootstrap.

- **URL**: `http://www.getbootstrap.com`
- **Twitter**: `@twbootstrap`
- **People**: Mark Otto (`@mdo`), Jacob Thornton (`@fat`) et al.

Bootstrap Expo

Whenever I speak to people who know a little about Bootstrap but haven't yet had a play with it, I find that they seem to think Bootstrap is a fixed style. I think a lot of that bad reputation comes from the early projects that were built with it. There was a time when every jQuery plugin page you seemed to visit was built with unstyled Bootstrap.

As we've learnt though, Bootstrap is a whole lot more than that. It's a full-fledged frontend framework. To change this viewpoint, one of the Bootstrap creators started the Bootstrap Expo blog to showcase the most inspiring uses of Bootstrap around the Web.

- **URL**: `http://expo.getbootstrap.com`
- **People**: Mark Otto (`@mdo`)

BootSnipp

BootSnipp is an incredible resource for anyone working with Bootstrap. At its simplest, it's a collection of precoded components that you can copy and paste into your project. Things like navigations, carousels, and style-up modal windows can be found here.

But BootSnipp is a lot more than that. You can filter by Bootstrap version, take a look at other useful resources, and also use its handy form and button builders where you simply drag the elements you want and copy the HTML.

- **URL**: `http://www.bootsnipp.com`
- **Twitter**: `@BootSnipp`
- **People**: Maksim Surguy (`@msurguy`)

Code guide by @mdo

As your project increases in size, you'll find the need to follow some standards when it comes to HTML and CSS. Of course, when you're working on a project with other people, things can get even messier, and the need for some kind of style guide becomes even more apparent.

Mark Otto, one of the co-creators of Bootstrap, has put together a comprehensive guide of the standards he uses on projects. It's well worth taking a look at, but a lot of this is also just personal preference and what works well for you and your team. Use this as a starting point to establish your own sets of standards and rules.

- **URL**: `http://codeguide.co/`
- **People**: Mark Otto (`@mdo`)

Roots

Roots is an opensource, WordPress starter theme that comes packed with Bootstrap, Grunt, and the HTML5 Boilerplate to help you build awesome WordPress themes. While this may not be what we've been building throughout the course of this book, I want to open up to you the possibility that Bootstrap can be used for anything. It's an extremely versatile and solid platform to build your projects upon.

- **URL**: `http://roots.io/`
- **People**: Ben Word (`@retlehs`)

Shoelace

If you struggle getting your head around building or visualizing grids with Bootstrap, then it's probably worth taking a look at Shoelace. It's a handy little tool that will allow you to interactively build your applications grid and output all the markup you need in HTML, Jade, or EDN.

Grids can be saved and shared, and you can take a look at what your grid will look like on each device size. Additional classes can be added to rows and columns; however, it seems that you can only use the small size columns by default. Of course, you could always find and replace that afterwards, should you need to.

- **URL**: `http://www.shoelace.io`
- **People**: Erik Flowers (`@Erik_UX`) and Shaun Gilchrist

Bootstrap 3 snippets for Sublime Text

Sublime Text 2 and 3 are extremely popular text editors among developers, and it comes as no surprise that there's a plugin that includes a deluge of snippets for our favorite frontend framework.

The plugin allows you to quickly pull in any of Bootstrap's components and really lessens the learning curve. It can be installed via Package Control by searching for `Bootstrap 3 Snippets`, or you can, of course, just grab it from GitHub.

- **URL**: `https://github.com/JasonMortonNZ/bs3-sublime-plugin`
- **People**: Jason Morton (`@JasonMortonNZ`)

Font Awesome

This project started as a drop-in replacement for Bootstrap 2's image icons with some font icon equivalents. It's grown to become one of the biggest icon fonts around today and can be used with or without Bootstrap.

If you're looking to extend Bootstrap's already great range of icons, then give Font Awesome a try.

- **URL**: `http://fortawesome.github.io/Font-Awesome/`
- **People**: Dave Gandy (`@davegandy`)

Bootstrap Icons

Bootstrap comes with an awesome collection of icons out of the box, and it's even easily extended with Font Awesome. Bootstrap Icons was created to quickly search through the icons and grab the relevant class needed.

It's much quicker to find what you're looking for here than in the official documentation, as each icon has been tagged with multiple keywords. Simply type what you're looking for into the search box and the site will filter it out for you.

- **URL**: `http://bootstrapicons.com/`
- **People**: Brent Swisher (`@BrentSwisher`)

AngularJS projects and people

There are a number of AngularJS community projects that really can become part of your daily workflow. We've already touched on a couple throughout the book, but let's take a look at some more to keep an eye on.

The core team

As we know, AngularJS is a Google project, and as such, its core team is made up of employees at the big G. You may not know that the framework was initially created at a company called Brat Tech LLC by Miško Hevery and Adam Abrons to power a JSON storage service called `http://getangular.com/`. They later abandoned the service and open-sourced the framework we've come to know and love under the AngularJS name.

Miško continues to maintain the project as a Google employee along with several other engineers. Together they've expanded the framework, released countless additional modules, and created the popular Batarang Chrome extension.

- **URL**: `https://angularjs.org/`
- **People**: Miško Hevery (`@mhevery`), Adam Abrons (`@abrons`), Brian Ford (`@briantford`), Brad Green (`@bradlygreen`), Igor Minar (`@IgorMinar`), Votja Jína (`@vojtajina`) et al.

RestAngular

We've already touched upon two of the ways Angular allows you to connect to an API: `$http` and `ngResource`. There's also an extremely popular community project called RestAngular that takes a different approach.

The biggest difference with `ngResource` is that it uses promises, whereas `$resource` will automatically unwrap these for you. Depending on how your project works, this can be powerful, as it means you can resolve data on route load using `$routeProvider.resolve`.

If you're working with a RESTful API in your project and you're not entirely sold on `ngResource`, then this is definitely worth taking a look at.

- **URL**: `https://github.com/mgonto/restangular`
- **People**: Marin Gonto (`@mgonto`)

AngularStrap and AngularMotion

We've already seen and used AngularStrap in our project. It's a fantastic port of all of Bootstrap's core plugins over to native AngularJS directives. If you're using Bootstrap and AngularJS in tandem (as we have been) then it's a must-have module.

AngularMotion is designed to be used with AngularStrap, but it doesn't have to be. These are drop-in animations that work natively with `ngAnimate` and add something extra to your project. They can be used with `ng-show` and `ng-hide` as well as directives such as `ng-repeat` to animate adding or deleting items.

- **URL**: `http://mgcrea.github.io/angular-strap/` and `http://mgcrea.github.io/angular-motion`
- **People**: Olivier Louvignes (`@olouv`)

AngularUI

The AngularUI project is arguably the largest to sprout from the AngularJS community. It's split into several modules including UI-Utils, UI-Modules, and UI-Router.

The UI-Utils module is described as the Swiss Army Knife of tools, and it is. Small things like highlighting text, checking for key presses, and even fixing an element when it's scrolled past are available.

UI-Modules are nice little AngularJS modules with external dependencies on things like Google Maps or jQuery plugins. There are some powerful things in here, and I've used the Select2 module on numerous occasions.

Perhaps their most popular project is the UI-Router module. This provides true nested routing for Angular. It enables you to split your page into states, for example, you might have a state for your sidebar and another state for your main content. These can both have their own partial, and it enables the easier building of large pages and web apps.

There's even a module here for Bootstrap that is similar to AngularStrap, which is worth a look at.

- **URL**: `http://angular-ui.github.io/`
- **Twitter**: `@angularui`
- **People**: Nate Abele (`@nateabele`), Tasos Bekos (`@tbekos`), Andrew Joslin (`@andrewtjoslin`), Pawel Kozlowski (`@pkozlowski_os`), Dean Sofer (`@Unfolio`), Douglas Duteil (`@douglasduteil`) et al.

Mobile AngularUI

Unlike the AngularUI project, this one is actually for building a user interface. It's a simple mobile framework that uses both AngularJS and Bootstrap 3. Things feel fairly native, but there are parts like the side-nav that could be improved. It's still early days but this project has serious potential and is worth watching out for.

- **URL**: `http://mobileangularui.com/`
- **Twitter**: `@mobileangularui`
- **People**: mcasimir

Ionic

Ionic is incredible. It really is. It mashes together everything that's good about AngularJS and Cordova/Phonegap to enable you to build amazing hybrid apps with the web languages you already know.

Everything feels very native, and it's super easy to get started even if you've never built an app before. It uses AngularJS and the UI-Router extension alongside a heap of their own code. The best thing is that it's entirely open source and the whole thing can be contributed to by anyone on GitHub.

Oh, and they've even made the `ngCordova` module, which is full of directives that can be used to easily interface with an abundance of Cordova plugins.

- **URL**: `http://ionicframework.com/`
- **Twitter**: `@ionicframework`
- **People**: The Drifty Team (`http://drifty.com/`) – Andrew Joslin (`@ajoslin`) et al

AngularGM

Whilst AngularUI does include a directive to use Google Maps within Angular, I much prefer the simpler approach of AngularGM. The module enables easy creation of Google Maps within your project along with markers, InfoWindows, and polylines.

You can customize just about anything you want, from changing the map colors and settings to using a custom element for your InfoWindow or non-standard icon for your marker.

- **URL**: `https://github.com/dylanfprice/angular-gm`
- **People**: Dylan Price

Now it's your turn…

The sheer number of opensource projects out there for both Bootstrap and AngularJS is incredible. The great thing is that these can be contributed to by anyone—even you! If you find a bug, report it, or if you know how to fix it, submit a pull request and become a contributor.

Naturally, not every problem will already have a solution there for you to find. Now that you know how to use both frameworks, it's your turn to build something awesome and have fun.

B
Where to Go for Help

Even the most skilled developers get stuck from time to time, and there's no shame in asking for help. There are some specific places related to both Bootstrap and AngularJS that can help you on your way, should you need them.

The official documentation

The first place you should always head to if you have a certain issue or just need to jog your memory is the official documentation. Both Bootstrap and AngularJS have great documents. Previously, there were complaints that those of AngularJS were vague and lacking examples, but they've improved drastically in the last few years. For more details, refer to `http://www.angularjs.org` and `http://www.getbootstrap.com`.

GitHub issues

Angular and Bootstrap are both hosted on GitHub and both take advantage of the service's issue tracker. Should you come across a bug in either of the frameworks, this is where you should report it. Of course, if you know what the issue is and how to fix it, you can also submit a pull request and become a contributor to the project. For more details, refer to `http://www.github.com/angular/angular.js/issues` and `http://www.github.com/twbs/bootstrap/issues`.

Stack Overflow

You had to guess this one was coming, right? Stack Overflow is an awesome resource and is a great place to go if you have a specific question you want answered. Most of the time, you'll find someone else has had the same question and you can read through the answers. Otherwise, ask a new question and tag it with AngularJS or Twitter Bootstrap. For more details, refer to `http://www.stackoverflow.com`.

The AngularJS Google group

As you play more with AngularJS, I can guarantee that after some Google research, you'll find yourself at this group. It's the official group/forum for AngularJS and is very active. There are well over 11,000 topics on there, so it's worth giving it a search before you ask a new question. For more details, refer to `https://groups.google.com/forum/#!forum/angular`

Egghead.io

If you're looking for some video tutorials on AngularJS, then Egghead.io is probably the best resource around for them. Whilst there is a paid subscription service, there's also a large chunk of their library that is free to watch. If you want to learn a little more visually, refer to `https://egghead.io/tags/free`.

Twitter

This might seem like a pretty strange suggestion for a support resource, but there are some incredibly helpful people on Twitter. It may not be the best place to ask in-depth questions, but for small little tidbits it can be great.

It's obviously also a great place to meet lovers of both frameworks and participate in the respective communities. Both of the frameworks have their official Twitter accounts: `@angularjs` and `@twbootstrap`. Oh, and if you want to tweet to me, I'm `@steve228uk`.

Undoubtedly, as you learn more about AngularJS and Bootstrap, you'll need to refer to the documentation and ask for help less and less. One thing I've learnt is that it's great to pass on knowledge. You might be asking for help building that very specific directive, but that doesn't mean you can't help others too! When you have a free second, log on to something like Stack Overflow and try to answer a couple of questions—I bet you can answer more than you expect.

C
Self-test Answers

Chapter 1

1. Using the ng-app attribute
2. The double curly brace syntax: {{model}}
3. Model-View-Controller
4. Create a controller using a standard JS constructor and the ng-controller attribute
5. Jumbotron

Chapter 2

1. A Bootstrap navbar
2. 12 columns
3. Functions called from an attribute or custom element
4. ng-repeat

Chapter 3

1. Using the pipe symbol on a model: {{ modelName | filter }}
2. Using colons: {{ modelName | filter:arg1:arg2 }}
3. The filter named: filter
4. Using the $filter service by injecting the filter as its own service following the pattern: filternameFilter

5. An AngularJS module
6. Array, expression, and comparator

Chapter 4

1. ngRoute
2. The config method on our module
3. The $routeProvider service
4. Using the $routeProvider.when method
5. The $routeProvider.otherwise method
6. Using html5Mode

Chapter 5

1. As it's included on our root view
2. Several: table-bordered, table-striped, table-hover, and table-condensed
3. <button class="btn btn-primary btn-lg"><button>
4. A form group
5. Labels are aligned to the left of our elements
6. A help block
7. img-circle to create a circular image, img-rounded to create a rounded rectangle, and img-thumbnail to add a double border

Chapter 6

1. Custom service, $rootScope, application-wide controller
2. Value, Service, and Factory
3. The ngSanitize module
4. The controller method allows our directive to communicate with other directives, whereas link does not
5. The = means we can directly bind a model and @ means our directive will use the literal value of that attribute
6. Setting the restrict property to EM
7. Add some helper functions to our navbar
8. Using $index

Chapter 7

1. ngAnimate
2. AngularMotion
3. bs-
4. Click, hover, focus, and manually
5. show, hide, and toggle

Chapter 8

1. A promise
2. $http.get('http://localhost:8000').success(function(data) { $scope.contacts = data });
3. URL, Parameter Defaults, Actions
4. It acts as a placeholder
5. RESTAngular uses promises, and you don't have to write your placeholders when following REST
6. Real time

Chapter 9

1. Node
2. To tell NPM which packages we require
3. Uglify
4. We need to annotate them with arrays

Chapter 10

1. Variables, Mixins, Nested rules
2. Using the &:before or &:after pattern
3. Change the variable in variables.less
4. Style Bootstrap to look like Bootstrap 2

Chapter 11

1. Any of the following: required, ng-required, ng-pattern, ng-minlength, ng-maxlength, ng-min, ng-max
2. Checking the $valid or $invalid properties
3. Calling it in the require property
4. Using ng-pattern

Chapter 12

1. A Chrome extension to allow us to inspect AngularJS apps
2. Any of the following: Models, Performance, Dependencies, Inspector, highlighting applications, bindings, and scopes
3. From the Models tab through the web inspector by selecting AngularJS Properties
4. ngMin
5. Remove, add, and singleQuotes
6. The need to manually annotate dependencies

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Thank you for buying
Learning Web Development with Bootstrap and 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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

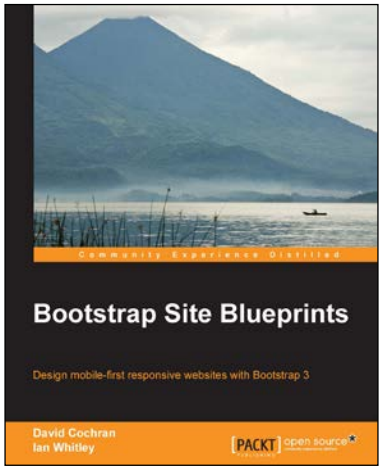
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced 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, then please 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.

[PACKT] open source
community experience distilled
PUBLISHING

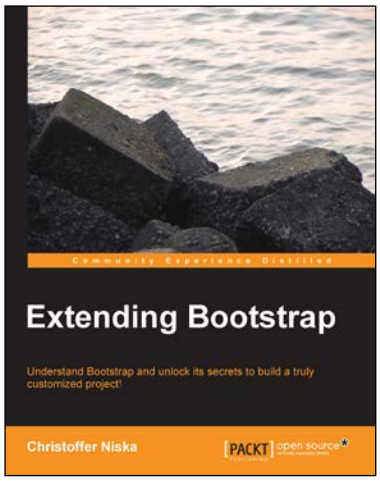


Bootstrap Site Blueprints

ISBN: 978-1-78216-452-4 Paperback: 304 pages

Design mobile-first responsive websites with Bootstrap 3

1. Learn the inner workings of Bootstrap 3 and create web applications with ease.
2. Quickly customize your designs working directly with Bootstrap's LESS files.
3. Leverage Bootstrap's excellent JavaScript plugins.

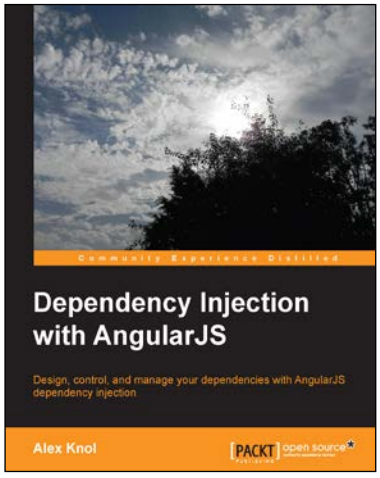


Extending Bootstrap

ISBN: 978-1-78216-841-6 Paperback: 88 pages

Understand Bootstrap and unlock its secrets to build a truly customized project!

1. Learn to use themes to improve your user experience.
2. Improve your workflow with LESS and Grunt.js.
3. Get to know the most useful third- party Bootstrap plugins.

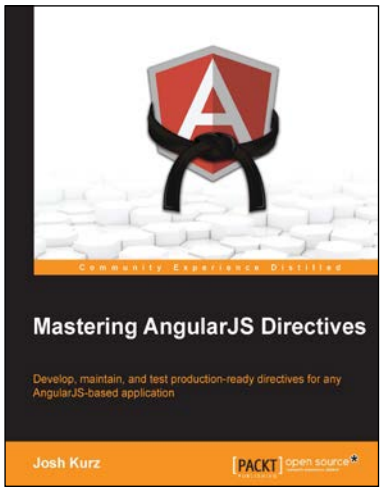


Dependency Injection with AngularJS

ISBN: 978-1-78216-656-6 Paperback: 78 pages

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

1. Understand the concept of dependency injection.
2. Isolate units of code during testing JavaScript using Jasmine.
3. Create reusable components in AngularJS.



Mastering AngularJS Directives

ISBN: 978-1-78398-158-8 Paperback: 210 pages

Develop, maintain, and test production-ready directives for any AngularJS-based application

1. Explore the options available for creating directives, by reviewing detailed explanations and real-world examples.
2. Dissect the life cycle of a directive and understand why they are the base of the AngularJS framework.
3. Discover how to create structured, maintainable, and testable directives through a step-by-step, hands-on approach to AngularJS.