AngularJS Starter

A concise guide to start building dynamic web applications with AngularJS, one of the Web's most innovative JavaScript frameworks

Dan Menard

[PACKT] PUBLISHING

Table of Contents

Instant AngularJS Starter

Instant AngularJS Starter

First published: February 2013

Production Reference: 1130213

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78216-676-4

www.packtpub.com

Credits

Author

Dan Menard

Reviewers

Stephane Bisson

Misko Hevery

Acquisition Editor

Rukhsana Khambatta

Commissioning Editor

Yogesh Dalvi

Technical Editor

Nitee Shetty

Project Coordinator

Sneha Modi

Proofreader

Aaron Nash

Production Coordinator

Prachali Bhiwandkar

Cover Work

Prachali Bhiwandkar

Cover Image

Conidon Miranda

About the author

Dan Menard is a web developer, originally from Canada and now living in California. He has many years of consulting work under his belt, which has taught him a great deal about ramping up on new technologies and the value of a good framework. He currently works at Netflix on the 10' UI team.

He has spoken at a number of conferences, most recently OSCON 2012. He also maintains a blog, and occasionally hangs out around Twitter and Google+. If you're in the bay area, you may be able to catch him at the Mountain View AngularJS meet-ups.

I'd like to thank my loving wife for her ongoing support of my work and work-related hobbies. I'd also like to thank Packt Publishing, for giving me the opportunity to write this book, and my reviewers, for all of their help and suggestions.

About the reviewers

Stephane Bisson is a developer at ThoughtWorks, a global IT consultancy. He is currently based in Chengdu, China. He has worked on several rich web applications for the medical, financial, and manufacturing industries.

Misko Hevery works as an Agile Coach at Google where he is responsible for coaching Googlers to maintain the high level of automated testing culture. This allows Google to do frequent releases of its web applications with consistent high quality. Previously he worked at Adobe, Sun Microsystems, Intel, and Xerox (to name a few), where he became an expert in building web applications in web-related technologies such as Java, JavaScript, Flex, and ActionScript. He is very involved in the Open Source community and an author of several open source projects such as Angular (http://angularjs.org) and JsTestDriver (http://code.google.com/p/js-test-driver).

www.packtpub.com

Support files, eBooks, discount offers and more

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

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

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

packtLib.packtpub.com

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

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web 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 nine entirely free books. Simply use your login credentials for immediate access.

Chapter 1. Instant AngularJS Starter

Welcome to *Instant AngularJS Starter*, your guide to building modern JavaScript applications with AngularJS. This book has been specially crafted to help you become productive with AngularJS as quickly and efficiently as possible. You will learn the basics of the framework, how to mold it into an MVC powerhouse, and everything you need to know about templates, data binding, modules, dependency injection, and directives.

This document contains the following sections:

So, what is AngularJS?: Find out what separates AngularJS from the other JavaScript frameworks out there, and why it just might be the best choice for your next project.

Hello World: Learn how to get up and running with AngularJS, so you can start building cutting-edge web applications today.

Quick Start – an MVC application in AngularJS: Brush up on the Model-View-Controller paradigm, and see how to build a fast, flexible MVC application using AngularJS.

Top 5 features you need to know about: Discover the strengths of AngularJS by exploring its most prominent features—templates, two-way data binding, modules, dependency injection, and directives.

People and places you should get to know: Get to know the top contributors, bloggers, and experts on AngularJS. Visit helpful links to tutorials, forums, articles, and Twitter feeds that will keep you up-to-date on all things about AngularJS.

So, what is AngularJS?

AngularJS is a JavaScript framework for building interactive, single-page applications for the modern Web.

If you are reading this book, I bet you already knew that. In fact, I bet you fit one of the following descriptions:

- You're an experienced web application developer, and you already have a good idea of what a JavaScript framework can do for you. You want to know what makes AngularJS special.
- You're relatively new to application development in JavaScript, and you have noticed that there are a number of JavaScript frameworks available but they all kind of look the same. You want to know what makes AngularJS special.

AngularJS is indeed very special, and we'll get to that in a moment. But before we can talk about what makes AngularJS unique, we have to look at the problem JavaScript frameworks in general are trying to solve: HTML being really bad at building interactive applications.

HTML was designed with static content in mind. Not only was it never meant to support rich applications, it really gets in the way when we try to take the Web in a more fluid direction. There's nothing dynamic about loading content once, on page load, for example.

We've come a long way with JavaScript—and even CSS to some extent—but HTML is still the fundamental base of the Web, and we're stuck with it no matter how ill suited it is for the beautiful, dynamic user interfaces we intend to build.

Enter the JavaScript framework, whose goal is to exploit the asynchronous nature of JavaScript while minimizing the ruthlessly static nature of HTML. There are two common approaches to this problem.

The first is to add a thick layer of JavaScript above the HTML. This is a double-edged sword; it's nice not to have to worry about how your actions in dynamic-JavaScript-land are going to map to the static world of HTML, but working so far from what is actually being rendered in the browser can easily turn into a performance nightmare if you don't fully understand how the framework interprets your actions.

The second approach is to leave view creation up to the developer, and provide a thin layer of JavaScript to tie into these views. Having less abstraction makes it more difficult for performance slowdowns to sneak into your application, but since the framework is assuming less responsibility for the HTML, the developer will need to create views manually.

AngularJS takes the second approach a step further. Rather than leaving the developer to fend for himself in the world of markup and styles, AngularJS integrates with HTML to extend the functionality of views and pull in dynamic bits of JavaScript as needed.

Everything that is special about AngularJS comes from the following approach:

- The view abstraction allows you to bind model objects directly in your HTML, letting the framework take care of how and when the DOM will be updated
- Every abstraction comes with a performance hit, but since the view abstraction is integrated tightly into HTML, it generally performs very well compared to other frameworks
- Extending HTML allows the framework to add features such as dependency injection, which allows you to write unit-testable views

The integrate-and-extend mantra in AngularJS has other benefits, as well. Since there's no giant abstraction to drag around, you can drop AngularJS into an existing web application

and immediately start using its features as much or as little as you like; even if you're already using another JavaScript framework.

Most of the features in AngularJS are centered around building and extending views, and the interaction between views and models. This affects how you will spend your time while working with AngularJS, so let's look at these two groups of features separately.

User interface controls are built declaratively, using HTML-based templates and a feature of AngularJS called directives. Directives are unique to AngularJS. We'll cover them in depth later in this book, but for now, think of them as a dynamic extension of HTML. Working with templates and directives allows you to reuse and further your skills with HTML and CSS, while still drawing dynamic behavior from JavaScript with help from the framework.

Model and view interaction is handled through annotations in your HTML. Like the user interface controls we just talked about, this is also a declarative process. Defining these connections ahead of runtime will allow you to take advantage of powerful AngularJS features such as data binding and dependency injection. Later in this book, we'll cover how these features work, and how they can help you build a robust and cohesive data layer.

AngularJS is an exciting framework to work with. It favors integration and extension over heavy abstraction, and encourages web application development best practices such as templating and declarative data modeling. It offers advanced features such as directives and dependency injection, which allow you to write fully testable applications—a huge benefit of AngularJS. These features will save you time and stress while building a more interactive web.

This book will walk you through how you can take advantage of the best parts of AngularJS. You'll learn how to use AngularJS to build a powerful, well-structured application, and all about the major enhancements and innovations AngularJS brings to the JavaScript framework table.

Whether you're a seasoned web application developer, or completely new to the craft, you'll pick up the skills you need to wield AngularJS with confidence and style.

Let's get started!

Hello World

Now that we have a clear understanding of what AngularJS is, and a vague idea of what makes it special, we can start using it to build great things! But all great things are built atop a solid foundation; so before we dive into a simple example, let's get your environment set up.

Recommended tools for AngularJS development

At a bare minimum, you only need a couple of things to build an AngularJS application:

- A text editor
- A web browser

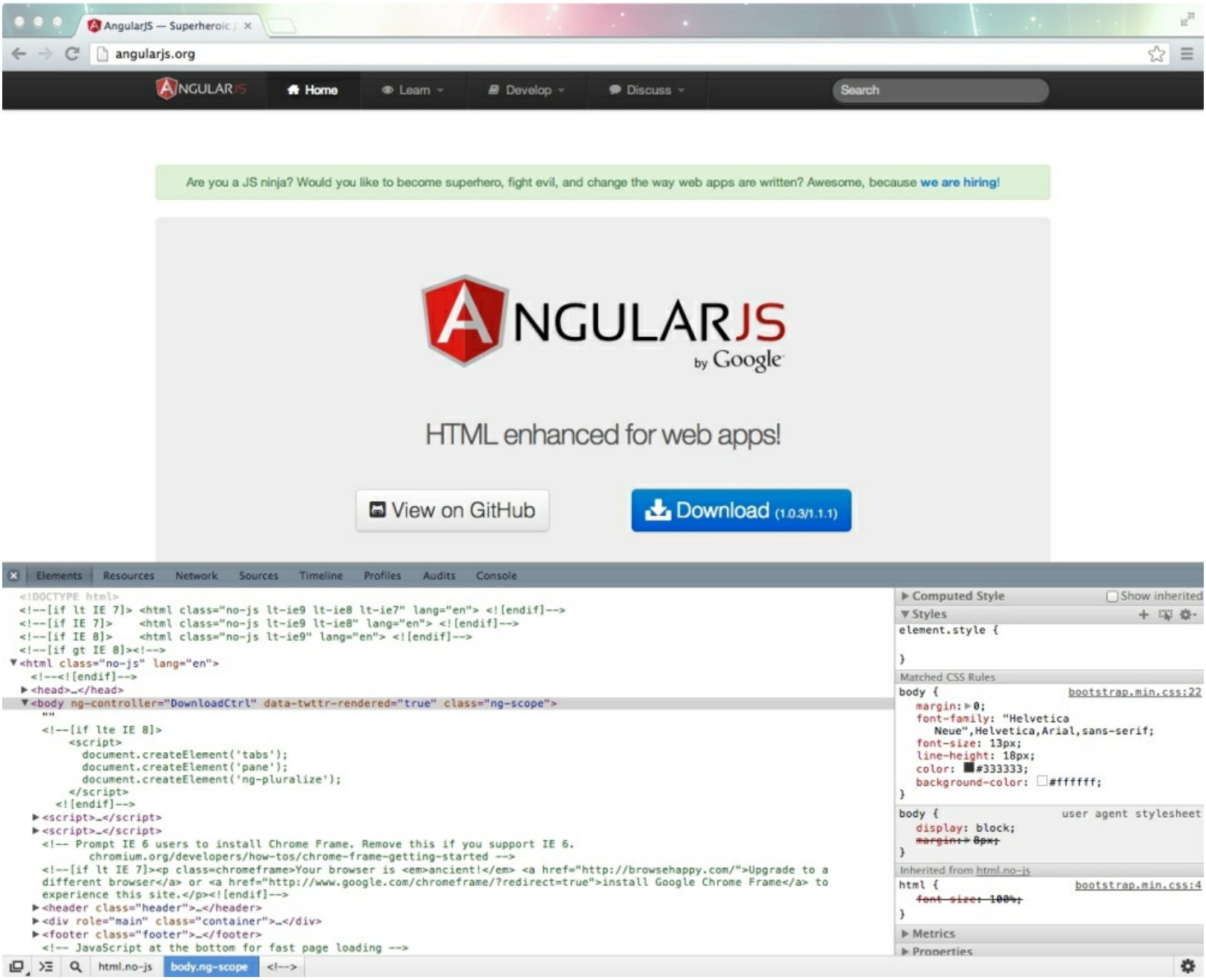
But, if we want to build great things, we're going to need great tools, not the bare minimum. AngularJS is a serious framework for serious development, and serious developers have serious tools.

Let's revisit that list:

- A text editor
- A good web browser
- A web server

A good web browser should have an integrated inspector and debugger, and should support modern HTML/JavaScript features. The latest version of all major browsers meets these criteria, so just make sure your favorite browser is up-to-date.

For example, I'm currently using the latest version of Google Chrome for OSX, and with its developer tools open, it looks like the following screenshot:



Once you have your editor and browser ready, you'll need a place to test your work. You can do this directly in your browser, but I'm going to strongly suggest you use a web server for a couple of reasons:

- Modern browsers don't like running files directly from your hard drive, and will often limit what your application can do when running directly from disk.
- When your application is finished and published for real people to use and love, it's going to be running on a web server. It's always a good idea to test your application in an environment that closely resembles what your users will see.

Setting up a local web server isn't especially difficult, and there are several AngularJS-friendly tools available to make this even easier for you.

One such tool is *Yeoman*, which is available at https://github.com/yeoman. Yeoman provides many useful features for the application developer, such as package management and project scaffolding. Of particular note is the Yeoman server command, which will start a

local web server that can host your AngularJS application.

Another option is *angular-sprout*, an AngularJS-specific bootstrapping tool. It supplies a similar set of features, and offers web server integration through *Node.js* (another handy tool for the aspiring web development maven). You can get these tools from https://github.com/thedigitalself/angular-sprout and http://nodejs.org respectively.

A very basic AngularJS application

With all our tools in place, we're ready to start coding! Let's just dive right into it. Pull up your text editor, and punch in a quick HTML skeleton as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to AngularJS</title>
  </head>
  <body>
    <h1>Hello, World.</h1>
  </body>
</html>
```

Tip

Downloading the example code

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

Remember to use a proper `DocType`. This isn't required by AngularJS, but it's important for helping web browsers render your content properly. Speaking of AngularJS, it's time to add the framework. We do this by adding it to the document head as shown in the following code snippet:

```
<head>
    <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.j
s'></script>
    <title>Welcome to AngularJS</title>
</head>
```

You'll notice that we're including AngularJS from the cloud, rather than downloading it and including it with a relative URL. This is perfectly fine, and the suggested way to access

AngularJS. The current version as of this writing is 1.0.3, but you should use the latest version of AngularJS, which is available at http://angularjs.org. We're also using the non-minified version; this is fine for development, but you should use the minified version in production.

If you run what we have so far in your web browser—pointing to your web server, of course!—you'll see a very basic HTML printout with no special AngularJS behavior. You can verify that AngularJS is loaded using the web inspector built into your browser, but as you can see, we're not doing anything with it yet.

Let's fix that. First we'll add a simple script to our head:

```
<head>
  <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.j
s'></script>
  <script>
    function Clock($scope) {
      $scope.currentTime = new Date();
    }
  </script>
  <title>Welcome to AngularJS</title>
</head>
```

Next, we need to tell AngularJS about our application. We do this by annotating our HTML tag:

```
<html ng-app>
```

Finally, we'll sprinkle a little AngularJS magic onto a paragraph tag:

```
  <body>
    <h1>Hello, World.</h1>
    <p ng-controller='Clock'>
      The current time is {{currentTime | date:'h:mm:ss a'}}.
    <p>
  </body>
```

Now if you run the application, you should see a message under the header telling you the current time in your local time zone. It may not look like much, but it's your first AngularJS application and it gives us some important insight into how AngularJS applications are built.

A few takeaways

Before we move on to something bigger, let's take a moment to reflect on what we have so far. For reference, here's the full application we just built:

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.j
s'></script>
    <script>
      function Clock($scope) {
        $scope.currentTime = new Date();
      }
    </script>
    <title>Welcome to AngularJS</title>
  </head>
  <body>
    <h1>Hello, World.</h1>
    <p ng-controller='Clock'>
      The current time is {{currentTime | date:'h:mm:ss a'}}.
    <p>
  </body>
</html>
```

Let's start with the `Clock` function. Believe it or not, this is what an AngularJS controller looks like—simple and unassuming. The only really interesting bit here is the `$scope` variable that the function accepts. Scope is a messy subject in JavaScript, but AngularJS tries to make your life a little easier by providing its own context, called `$scope`. We'll talk a lot more about `$scope` later, but for now, notice how we're using it; we're getting the current date from JavaScript's native date object, and storing it in the `currentTime` property so that we can use it in our application's view.

The `Clock` controller is bound to our user interface by the annotations we placed in our markup. The first annotation is in the HTML tag itself, way at the top of the file. By identifying a tag as `ng-app`, we're telling AngularJS that everything in our application is contained within that tag. Note that you can do this with any tag, not just the HTML tag. This is very valuable if you want to integrate AngularJS into an existing application, because it gives you full control over which elements are available to AngularJS. In our case, we could have put the `ng-app` annotation on the body or even the paragraph tag, and everything would still work because we don't use AngularJS in our markup until inside the paragraph tag.

The annotation on the paragraph tag, `ng-controller`, is what binds our controller (the `Clock` function) to our view (everything inside the paragraph tag). Like `ng-app`, we could move this annotation up to the body or HTML tags. It will work as long as it's defined alongside or on a child of `ng-app`, and anything alongside or below `ng-controller` will be able to use it.

The crux of our application is the contents of the paragraph tag. Specifically, the following bit of the AngularJS syntax:

```
{{currentTime | date:'h:mm:ss a'}}
```

The `currentTime` property is going to give us the value of the JavaScript date object we set up in the `Clock` function. That vertical bar next to it is our way of telling AngularJS that we want to format this data before displaying it. Specifically, we're using the built-in date formatter in AngularJS, and providing it with a string explaining how we want it formatted (we could have used our own formatting function, but there was no need to do that here).

Recall that AngularJS is all about integrating with and extending HTML to build a dynamic application. This is already visible in our simple example; we're using existing HTML and JavaScript entities, and enhancing them with annotations and the date filter from AngularJS.

Using these simple techniques, AngularJS is capable of creating fully standalone applications. Since AngularJS is an entirely client-side library, integrating with any server-side library is as easy as setting up an AJAX connection.

AngularJS has much more to offer us than what we've seen here. Let's build something bigger! In the next section, we'll explore how views, controllers, and models work together to create a scalable application architecture in AngularJS.

Quick start – an MVC application in AngularJS

Hello World examples are nice, because they introduce you to the framework's syntax and capabilities. But if you've ever tried to build a large product with a new JavaScript framework, you've probably run into the same problem I have—scale.

I don't know what you intend to build with AngularJS, but I'm fairly confident it won't fit in a single file like the example we just saw. You'll want to use multiple controllers, dynamic views, and models with real data. After all, interaction between models, views, and controllers is one of the things AngularJS does very well.

In this section, we're going to build an application following the **Model-View-Controller** (**MVC**) pattern. We'll combine multiple views, models, and controllers, and end up with a scalable architecture you can replicate for whatever you build next.

Before we dive into the code (and there's a lot of code in this section), let's do a quick review of the MVC pattern.

Step 1 – understanding the Model-View-Controller pattern

Model-View-Controller, or MVC, is a popular design pattern for building scalable web applications. The idea is to break your application into the following three parts:

1. Models, which contain data used in your application.
2. Views, which display data to the user and read user input.
3. Controllers, which format data for views and handle application state.

If you're not already familiar with this pattern, I urge you to look it up online. It's a powerful architecture for building scalable web applications. If you're in a rush, or you just need a quick refresher, here are the highlights.

MVC is about discipline. Models should only be concerned with storing data, and shouldn't know anything about controllers or views. Views only care about presenting data to the user and reading user input. Controllers tie models and views together. When a controller decides to load a view, it requests any necessary data from the model, and formats that data specifically for the view. When the view detects a user action, it feeds that information to the controller, which decides what to do next.

This has a number of benefits that you should research if you're new to MVC. For our

purposes, it really helps applications scale in a sensible way. That's what we'll focus on in this section.

Step 2 – building the Guidebook application

Let's build an application in AngularJS that follows the MVC pattern.

Why don't we make a guide to accompany this book? It could have a list of chapters, like an outline, and we could allow the user to add and remove notes for each chapter.

What would that look like in MVC? Let's break it down:

- We're going to need a view for the list of chapters. And since we're letting the user add notes, we'll need a view for user input as well.
- We have a few distinct actions in our application: view chapters/notes, add a note, and delete a note. Let's add a controller for each of those.
- Since we're storing a dynamic collection of notes, we'll need a model to manage them. It probably makes sense to put the chapter data in a model too, since that's also data.

We also need a bit of start-up logic to configure AngularJS for our application, and probably some CSS for our views. If we create a file hierarchy for all of this, we get the following:

```
Guidebook
  - Guidebook.html
  - Guidebook.js

  model
    - NoteModel.js
    - ChapterModel.js

  view
    - chapters.html
    - addNote.html
    - style.css

  controller
    - NotesController.js
    - ChapterController.js
```

I cheated a bit with our controllers; I'm putting the add note and delete note controllers in the same file, since they're kind of similar. We could actually put all of the JavaScript together in one giant file, if we wanted. The separation is only to make our lives easier; AngularJS doesn't care either way.

You may have noticed that we refer to our HTML files as views. This is not entirely correct; a view in AngularJS is actually a compiled HTML template, and what we're going to put in those HTML files are templates. Remember, AngularJS is about extending and enhancing existing web technologies—this is especially true for views.

Now that we know what we're building, let's have at it!

Step 3 – configuring AngularJS for the Guidebook application

We'll start with our AngularJS configuration files, `Guidebook.html` and `Guidebook.js`. This is where we'll define the structure of our application, and include our models and controllers.

First things first, let's pull up `Guidebook.html`, where we'll start hooking in our file structure:

```
<!DOCTYPE html>
<html ng-app='Guidebook'>
  <head>
    <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.j
s'></script>
    <script src='Guidebook.js'></script>
    <script src='controller/ChapterController.js'></script>
    <script src='controller/NotesControllers.js'></script>
    <script src='model/NoteModel.js'></script>
    <script src='model/ChapterModel.js'></script>
    <link rel='stylesheet' href='view/styles.css'>
    <title>AngularJS Starter Guidebook</title>
  </head>
  <body>
    <h1>Welcome to the AngularJS Starter Guidebook</h1>
    <div ng-view>
      <!-- Views are added here at runtime. -->
    </div>
  </body>
</html>
```

Some of this looks familiar. We have our DocType, we're including AngularJS, and we have a couple of annotations. We're also including all of our model and controller files, but none of our view files (views are mapped via JavaScript, which we'll get to in a minute).

Note that this time, we're defining a name for our application in the `ng-app` annotation. This isn't necessary, as we saw in our last example, but it's a good idea for real-world applications, because it helps us to namespace our models and controllers. (If we had two AngularJS applications on the same page, separate modules would help us remember which components belong to which application).

We also have a new annotation: `ng-view`. This tells AngularJS where to load our views. There can only be one instance of `ng-view`, and it should contain everything we want to load and change dynamically in the application. If something resides outside of `ng-view`, like the heading in our earlier example, it will be visible at all times.

Let's move on to `Guidebook.js`, which specifies our view mappings:

```
var guidebookConfig = function($routeProvider) {
  $routeProvider
    .when('/', {
      controller: 'ChaptersController',
      templateUrl: 'view/chapters.html'
    })
    .when('/chapter/:chapterId', {
      controller: 'ChaptersController',
      templateUrl: 'view/chapters.html'
    })
    .when('/addNote/:chapterId', {
      controller: 'AddNoteController',
      templateUrl: 'view/addNote.html'
    })
    .when('/deleteNote/:chapterId/:noteId', {
      controller: 'DeleteNoteController',
      templateUrl: 'view/addNote.html'
    })
  ;
};
var Guidebook = angular.module('Guidebook',
[]).config(guidebookConfig);
```

We'll start with the last line in the file. This is how you define an AngularJS namespace, called a `module` (we'll talk more about modules later). By setting up our application this way, we can keep our controllers and models in their own application namespace, which is generally a best practice, and especially important if this application is added to a page with other content.

As you can see, we're passing a list of routes to configure our application. Routes define which controller is used for a given URL, and which view template to show using that controller. Any parameters that the controller needs are passed as part of the URL. For example, if we access `http://your-webserver/guidebook.html#/chapter/2` then AngularJS will load `ChaptersController` with the `chapters.html` view, and pass along the value `2` as `chapterId`.

When we want to delete a note, say the fourth note from chapter one, we'll load a URL ending with `#/deleteNode/1/4`. This will pass `1` and `4` to `DeleteNoteController` as `chapterId` and `noteId`.

Routes contain more features than we've covered here. For example, the samples above are perfectly valid when accessed directly (this behavior is called deep linking). Routes also support back/forward navigation, meaning if the user hits a browser's back or forward button, AngularJS will return to the most recent internal URL instead of exiting the application altogether.

Declaring routes is a simple, powerful way to define how to move around within an AngularJS application. Even a very long list of routes provides a single source for your navigation information and controller view bindings. This is far more manageable than

scattering such logic implicitly across a number of files.

At this point, we have our application configured with a few routes, and we've told AngularJS which controller to load for each route, as well as which view to load for that controller. Let's now build some views!

Step 4 – creating views

As noted earlier, our view files contain annotated HTML templates. Our paragraph tag in the Hello World application was a template, and the templates we're going to create for our Guidebook application will be very similar, just a little larger.

Let's start with `chapters.html`, which defines the template for our chapter template:

```
<ul>
  <li ng-repeat='chapter in chapters | orderBy:"title"'>
    <h2>{{chapter.title}}</h2>
    <p>{{chapter.summary}}</p>
    <p>
      <a href='#/chapter/{{chapter.id}}'>
        {{chapter.notes.length}}
        <span ng-show='chapter.notes.length == 1'>note</span>
        <span ng-hide='chapter.notes.length == 1'> notes</span>
      </a>
      |
      <a href='#/addNote/{{chapter.id}}'>add note</a>
    </p>
    <ul class='notes' ng-show='chapter.id == >
      <li ng-repeat='note in chapter.notes | orderBy:"id"'>
        <div>#{{$index}}</div>
        <div><a href ng-click='onDelete(note.id)'>delete</a></div>
        <p>{{note.content}}</p>
      </li>
    <ul>
  </li>
</ul>
```

This is quite a bit more complicated than our Hello World view! I know there's a lot of AngularJS magic there, which may seem a little daunting. Let's just focus on the controller interaction for the time being, and we'll fill in the syntax gap later.

Let's start by thinking about what we're trying to do. We want to show all of the chapters of this book, and for the selected chapter, we also want to show any notes the user has created. Furthermore, each chapter needs an *add note* link, and each note needs a *delete me* link. Let's look at how our implementation works.

We begin with a list item with the `ng-repeat` annotation. Think of this as a `ForEach` loop in JavaScript. It's expecting a list of chapters in our `$scope`, and hopefully each chapter has a

title and a summary, because we're showing those on the next two lines.

After that we show a couple of HTML links (with a little AngularJS enhancement, as usual). Remember how our routes are set up: the first link will reload `ChaptersController` with `chapterId` of the current chapter (we're in a loop, so we'll make one of these links for each chapter). The second, about five lines later, will take us to `AddNoteController` with the `addNote` template, which we'll look at in a moment.

After that, we have another `ng-repeat` for the notes of the current chapter. We're using an `ng-show` annotation (like an `if` conditional) to only show the notes if the current `chapterId` in our outer loop matches `$scope.selectedChapterId`, and if we are showing notes, we'll print the ID and content of each one, and add a link that will call `$scope.onDelete()` when clicked.

Again, don't worry about mastering the syntax just yet. Instead, focus on how the view gets information from the controller. We expect all of the following:

- A list of chapters, each containing a title, summary, and a list of notes. All notes should have an ID and a content property.
- A property called `selectedChapterId`.
- A function called `onDelete()`.

We'll look at `ChaptersController` in a second, but first let's look at our other view, `addNote.html`:

```
<form name='addNote'>
  <label for='noteContent'>Enter a note about this chapter:</label>
  <textarea id='noteContent' ng-model='note.content'></textarea>
  <button ng-click='cancel()'>Cancel</button>
  <button ng-click='createNote()' ng-
disabled='!note.content'>Create Note</button>
</form>
```

This is a little easier to process. We have a form, with a label, a field, and two buttons. It looks like we're expecting a couple of functions from our controller: `cancel()` and `createNote()`. We also have an `ng-model` annotation on our text area.

`ng-model` is used to set up a data model shared by a view and its controller. This isn't a persisted model; it's just a convenient way for the view and controller to share data. This feature is called data binding, and it's a topic we'll cover in detail later in this book. For now, just think of `ng-model` as the view's way to set properties on `$scope` for the controller.

You're probably still a little shaky on views. That's ok! Concepts will start falling into place once we see the controller side of the equation. Off we go!

Step 5 – defining controllers

In our Hello World application, our controller was a simple function. We told AngularJS to bind that controller to our view with the `ng-controller` annotation. This time around, we've defined routes to bind controllers, so we need to define the controller a little differently. We can see this in `ChaptersController.js`:

```
Guidebook.controller('ChaptersController',
  function ($scope, $location, $routeParams, ChapterModel,
NoteModel) {
    var chapters = ChapterModel.getChapters();
    for (var i=0; i<chapters.length; i++) {
    chapters[i].notes =
NoteModel.getNotesForChapter(chapters[i].id);
    }
    $scope.chapters = chapters;
    $scope.selectedChapterId = $routeParams.chapterId;
    $scope.onDelete = function(noteId) {
    var confirmDelete = confirm('Are you sure you want to delete
this note?');
      if (confirmDelete) {
        $location.path('/deleteNote/' + $routeParams.chapterId +
'/' + noteId);
      }
    };
  }
);
```

As you can see, the controller is still largely a single function. We're simply adding it to our application namespace by calling `Guidebook.controller` (since Guidebook is the name we specified in our `ng-app` annotation).

Note the parameters our controller function accepts. There's the `$scope` variable, which we've seen before, but after that are all kinds of new things. What are these? They are explained as follows:

- `$location` is how AngularJS represents the current URL. We can use it to read or set the current location, allowing us to load new controllers/views.
- `$routeParams` are the parameters we pass via the URL. Remember how some of our route definitions had parameters? That's what we get in `$routeParams`.
- `ChapterModel` and `NoteModel` are our model objects. AngularJS will send these to our controller automatically, instantiated and ready to go.

That last point, about `ChapterModel` and `NoteModel`, is called dependency injection. We'll discuss it again later, but note for now that the order of the parameters doesn't matter.

The contents of the function are relatively straightforward. First, we get the chapter information from `ChapterModel` and combine that with any notes we retrieve from

`NoteModel`. This will get the chapter and note data we need for the view, and we're setting it on our `$scope` variable so that the view will be able to access it through `$scope.chapters`, as we saw in `chapters.html`.

Next, we set `selectedChapterId` to whatever `chapterId` we picked up in our URL. If there was no `chapterId` in the URL, this will be undefined, which is fine—we used `selectedChapterId` in the view to show notes if a chapter was selected. If there's no selection, we simply won't display any notes.

Finally, we're giving the view an `onDelete()` function to run if the user clicks on the *delete note* link we saw earlier. This function uses the `window.confirm()` method built into JavaScript (remember, AngularJS is an enhancement; we still have access to standard JavaScript features in our controller). If the user proceeds with the deletion, we use the `$location` parameter to change our URL and load `DeleteNoteController`.

We're moving quickly now! Let's finish looking at our controllers before we slow down and analyze the state of our application. Here's `NoteController.js`:

```
Guidebook.controller('AddNoteController',
  function ($scope, $location, $routeParams, NoteModel) {
    var chapterId = $routeParams.chapterId;
    $scope.cancel = function() {
      $location.path('/chapter/' + chapterId);
    }
    $scope.createNote = function() {
      NoteModel.addNote(chapterId, $scope.note.content);
      $location.path('/chapter/' + chapterId);
    }
  }
);

Guidebook.controller('DeleteNoteController',
  function ($scope, $location, $routeParams, NoteModel) {
    var chapterId = $routeParams.chapterId;
    NoteModel.deleteNote(chapterId, $routeParams.noteId);
    $location.path('/chapter/' + chapterId);
  }
);
```

We have two controller functions in this file, `AddNoteController` and `DeleteNoteController`. We're defining them the same way we defined our `ChaptersController`, and you can see that they take the same parameters, minus the `ChapterModel`, which they don't need.

The `DeleteNoteController` function is similar to `ChaptersController`. It grabs the `chapterId` and `noteId` from `$routeParams`, and uses those to send a delete call to our `NoteModel`. After that, we redirect back to the `ChaptersController` function.

The `AddNoteController` function is a little different. We're setting two functions here, for the

form we saw in `addNote.html`:

- `cancel()` will return to the chapter list, via the `ChaptersController` function
- `createNote()` will tell the `NoteModel` model object to add the new note, then redirect to the `ChaptersController` function

So far we've looked at both our views, and all three of our controllers. Let's take stock of what we have before moving on to the models.

For starters, there's a very loose coupling between controllers and views. Unlike our Hello World example, where we bound the controller directly in the view, we're using routes to give us a little more liberty. (The same view can be reused for multiple controllers, for example.)

We also saw two different ways of loading a controller. Ultimately, a new controller is only loaded when the URL changes, but we can do that from either the view or the controller. In `chapters.html`, for example, there are several HTML links that lead to new URLs. In our controllers, we changed the URL by modifying the `$location` parameter. These are both valid ways to load new controllers.

Lastly, there's `$scope`. Note that this is the only way for views and controllers to share information. The view expects the controller to set various properties on `$scope`, and it can also assign data to controllers through `$scope` using the `ng-model` annotation.

We're two thirds of the way through our MVC application, and the hard part is behind us. We've learned how to define views and controllers, and how they can interact with each other through routes and `$scope`. Let's add the final piece of the puzzle.

Step 6 – building models

Before we look at our model code, let's quickly flip back through our controllers and see what we need in our `ChapterModel` and `NoteModel`:

- In `ChaptersController`, we make a call to `ChapterModel .getChapters()` and `NoteModel.getNotesForChapter()`
- In `AddNoteController`, we make a call to `NoteModel.addNote()`
- In `DeleteNoteController`, we make a call to `NoteModel.deleteNote()`

It looks like our `ChapterModel` will be the simpler of the two. It only needs one function, and all that's going to do is return a list of chapters that never changes. Here's `ChapterModel.js`:

```
Guidebook.service('ChapterModel', function() {
  this.getChapters = function() {
    return [{
      id: 0,
      title: 'Chapter 1: So, What is AngularJS?',
      summary: 'Find out what separates AngularJS from...'
    }, {
      id: 1,
      title: 'Chapter 2: HelloWorld',
      summary: 'Learn how to get up and running with...'
    }, {
      id: 2,
      title: 'Chapter 3: QuickStart',
      summary: 'Brush up on the Model-View-Controller...'
    }, {
      id: 3,
      title: 'Chapter 4: Key AngularJS Features',
      summary: 'Discover the strengths of AngularJS...'
    }, {
      id: 4,
      title: 'Chapter 5: The AngularJS Community',
      summary: 'Get to know the top contributors...'
    }
  ]};
});
```

The data structure we're returning in the `getChapters()` function is very straightforward; it's a simple array of JavaScript objects. The function definition is also an easy one; as usual, we're using the application namespace, and we're defining the model as an AngularJS service. (We'll cover AngularJS services and their brethren in the next section, when we discuss dependency injection).

Our `NoteModel` will be considerably more interesting, because we need to be able to store and retrieve notes. Let's pull up `NoteModel.js`:

```
Guidebook.service('NoteModel', function() {
  this.getNotesForChapter = function(chapterId) {
    var chapter =
JSON.parse(window.localStorage.getItem(chapterId));
    if (!chapter) {
      return [];
    }
    return chapter.notes;
  };
  this.addNote = function(chapterId, noteContent) {
    var now = new Date();
    var note = {
      content: noteContent,
      id: now
    };
    var chapter =
JSON.parse(window.localStorage.getItem(chapterId));
    if (!chapter) {
      chapter = {
        id: chapterId,
        notes: []
      }
    }
    chapter.notes.push(note);
    window.localStorage.setItem(chapterId,
JSON.stringify(chapter));
  };
  this.deleteNote = function(chapterId, noteId) {
    var chapter =
JSON.parse(window.localStorage.getItem(chapterId));
    if (!chapter || !chapter.notes) {
      return;
    }
    for (var i=0; i<chapter.notes.length; i++) {
      if (chapter.notes[i].id === noteId) {
        chapter.notes.splice(i, 1);
        window.localStorage.setItem(chapterId,
JSON.stringify(chapter));
        return;
      }
    }
  };
});
```

If you're familiar with HTML5, you'll recognize right away that we're using local storage to store our notes. This is another example of why integrating with existing web technologies is such a powerful pattern: there are plenty of ways to store and transfer data on the Web already, and AngularJS lets us use whichever we please (even another JavaScript framework).

Like `ChapterModel`, we're defining `NoteModel` as a service. This time, however, we're defining three model functions. Let's look at each one individually.

`getNotesForChapter()` is up first. It takes `chapterId`, and it will use that `chapterId` as a

key in our local storage database. If it finds an entry for our chapter, we'll return the notes from that entry, and if it doesn't find anything, it returns an empty array.

`addNote()` is next. In addition to `chapterId`, it also takes some `noteContent`. We use `noteContent` to create a note object, using the current time as a unique ID. Like before, we use `chapterId` to pull up the current notes for a given chapter. If we don't have any yet, it's time to create a chapter entry, which just consists of `chapterId` and an empty list of notes. Finally, we add our newly created note, and save the updated chapter entry back to local storage.

`deleteNote()` is the last of our `NoteModel` functions. This time, we check for a chapter entry right away, and exit if we can't find one. (No sense deleting something that doesn't exist, right?) If there are notes for the given chapter, we cycle through them and delete the target note if we find it.

That's it for our models! They're quite a bit easier to digest than views and controllers, and this makes sense when you think about it. Views and controllers must concern themselves with data manipulation and user input, while models are entirely single-purpose, and only care about storing data. As we just saw, how data is stored is completely up to each model—whether it's a hardcoded list of data or a local storage interface, it doesn't matter to the controller.

By building our application following the Model-View-Controller pattern, we have an application architecture that will scale. If we wanted to add a new feature to our little Guidebook, we know exactly what steps to follow:

1. Create views based on what the user needs to see and do.
2. Define controllers to provide those views with the properties and functions that they require.
3. Build models to store data requested by the controllers.

Using the structure we have designed in this section, there's little room for bad design to slip into our application. We have a clean separation of purpose: our views and controllers only communicate through `$scope` and `ng-model`, and our models expose data retrieval functions to our controllers, who have no knowledge of how that data is stored.

With a solid foundation under our belt, it's time to get into the really exciting stuff. The next section dives deep into the inner workings of five key features that will transform you from an AngularJS developer to an AngularJS guru.

Let's roll!

Top 5 features you need to know about

Every framework has a set of features that differentiates it from the other frameworks out there. In this section, we're going to look at the features that make AngularJS special.

Knowing how to use these features will help you create seriously epic applications with AngularJS:

- Learning the inner workings of templates will help you build stronger views
- Mastering modules will help you break up your application into sensible components
- Understanding data binding will help you strengthen the relationship between your views and the data they interact with
- Digging into dependency injection will help you organize your controllers and the components they depend on
- Seeing directives in action will help you build powerful, reusable view components

Let's begin with templates.

Templates

We already know about templates. We used two in our Guidebook application, one for our chapter view, and one for our add note view. But how do those views work?

We know we can use `$scope`, but we also saw `$index` in our chapter view. Are there other properties like this?

What about annotations? We've seen a few useful ones already, but what else is out there that we can use?

Let's start by discussing the difference between properties such as `$scope`, and annotations such as `ng-repeat`.

There are two ways to invoke AngularJS from a view. As we saw earlier in our chapter view, we can use double curly braces to print something directly to the user:

```
<ul>
  <li ng-repeat='chapter in chapters | orderBy:"title"'>
    <h2>{{chapter.title}}</h2>
    <p>{{chapter.summary}}</p>
    <p>
      <a href='#/chapter/{{chapter.id}}'>
        {{chapter.notes.length}}
        <span ng-show='chapter.notes.length == 1'>note</span>
        <span ng-show='chapter.notes.length != 1'>notes</span>
      </a>
      |
      <a href='#/addNote/{{chapter.id}}'>add note</a>
    </p>
    <ul class='notes' ng-show='chapter.id == selectedChapterId'>
      <li ng-repeat='note in chapter.notes | orderBy:"id"'>
        <div>#{{$index}}</div>
        <div><a ng-click='onDelete(note.id)'>delete</a></div>
        <p>{{note.content}}</p>
      </li>
    <ul>
  </li>
</ul>
```

Any time we use double curly braces, we're asking AngularJS to render some specific bit of content, and print it out on the page. For example, `{{chapter.title}}` will print the title of the chapter object that we hopefully have in scope, and `{{$index}}` will print our current position in the encompassing `ng-repeat`.

You may be wondering why we need to specify a dollar sign (`$`) when we render content such as `{{$scope}}` and `{{$index}}`, but not when we render content such as `{{chapter.title}}`. The reason for this has to do with how content is made visible to our view.

Annotations such as `ng-repeat` and `ng-show` get full access to anything in `$scope`. So when we write `ng-repeat='chapter in chapters'`, AngularJS will iterate over `$scope.chapters`, and create a local (to the view) object called `chapter`.

Content that is local to the view is called without a dollar sign. This is why we say `{{chapter.title}}` (correct) instead of `{{$chapter.title}}` (incorrect). When we add a dollar sign, we're specifying an AngularJS keyword such as `$scope` or `$index`. These are not properties we have defined; they're properties AngularJS maintains internally.

So, if you're rendering some content using an AngularJS keyword, use a dollar sign. If you're rendering content that is already defined in your view, do not use a dollar sign.

Speaking of dollar sign properties, we've only seen two AngularJS keywords so far: `$scope` and `$index`. What else does AngularJS make available to us?

These are divided into two groups: properties all views get by default, and properties made

available by annotations.

`$scope` is an example of a property all views get by default. This is uncommon in AngularJS; there are only a few properties exposed in this manner.

In addition to `$scope`, allow me to introduce `$id` and `$window`. `$id` will contain a unique ID for the current scope. Since annotations such as `ng-repeat` generate a new child scope for each iteration, we can use this to add a unique handle to each iteration. For example, if we wanted to give all of our chapter headings a unique ID, we could make the following change to our heading:

```
<h2 id='{{$id}}'>{{chapter.title}}</h2>
```

`$window` will provide a reference to JavaScript's native window object. It's recommended to always use AngularJS' `$window` instead of JavaScript's window. This is important for testing, as `$window` can be properly stubbed in a test environment.

The other way AngularJS keywords are exposed to views is through annotations. For example, the `ng-repeat` annotation exposes the `$index` property, which holds the value of the current position in the collection we're iterating over.

`ng-repeat` also provides Boolean flags for specific positions in the list:

- `$first` will be true for the first iteration only
- `$last` will be true for the final iteration only
- `$middle` is true for all iterations except the first and last

In AngularJS, `ng-repeat` is the only annotation that exposes additional keywords. This is still an important concept, as we'll see how to create our own keywords in this manner later in this section when we talk about directives.

Views are only half the story. Let's take a quick look at controllers, and how they influence templates.

You may recall a few dollar sign properties in our `ChaptersController`:

```
Guidebook.controller('ChaptersController',
  function ($scope, $location, $routeParams, ChapterModel,
NoteModel) {
    var chapters = ChapterModel.getChapters();
    for (var i=0; i<chapters.length; i++) {
      chapters[i].notes =
NoteModel.getNotesForChapter(chapters[i].id);
    }
    $scope.chapters = chapters;
    $scope.selectedChapterId = $routeParams.chapterId;
    $scope.onDelete = function(noteId) {
      var confirmDelete = confirm('Are you sure you want to delete
this note?');
      if (confirmDelete) {
        $location.path('/deleteNote/' + $routeParams.chapterId +
'/' + noteId);
      }
    };
  }
);
```

We know what `$scope`, `$location`, and `$routeParams` are for from our controllers discussion in the previous section. What else do controllers get access to?

Turns out, quite a bit. There are actually a couple dozen properties that controllers can access, and most of them are far more obscure than `$scope`, `$location`, and `$routeParams`. Many are only necessary in very, very special cases, but there's one we haven't see yet that is worth introducing.

We didn't address localization in our Guidebook application. This is a must for most real-world applications. Generally, you will want to do this before you get to your view, but if there are some last-minute adjustments that are locale-dependent—like loading different header images for different locales—AngularJS can help you out with the `$locale` property.

`$locale` will return the current locale of the user's browser. This could be read in a controller to load a different string or image, or set if we want to allow the user the option of changing locales.

One gotcha related to localization, while we're on the subject: If you want to allow multiple locales, you should include the corresponding locale library from AngularJS: http://code.angularjs.org/1.0.0rc9/i18n-1.0.0rc9. This will add proper defaults for things such as date and time formatting.

Let's switch gears and look a little more closely at annotations.

We've seen a handful of these already. In our Hello World example, we used `ng-controller` to bind our controller to our view. Our Guidebook application introduced us to `ng-app` and `ng-view`, which we used to specify where in the DOM our AngularJS content and views would live. Our chapters view taught us a number of new and very useful

properties:

- `ng-repeat` iterates over a collection, creating one copy of all the markup within it for each iteration
- `ng-show` shows an element based on a Boolean condition
- `ng-click` calls a function when clicked

And finally, our `NoteController` showed us how to send data to a controller using `ng-model`.

As we discussed a moment ago, annotations often create properties local to the view. For example, our `ng-repeat` in our chapters view created a `chapter` object that the view could use to load properties from that specific chapter.

We also saw that `ng-repeat` creates a few local AngularJS objects we can use: `$index`, `$first`, `$last`, and `$middle`.

Something we haven't seen yet is the concept of local annotations—annotations that only work within a specific parent annotation. For example, there are the `ng-class-even` and `ng-class-odd` annotations. These are used to give different class attributes to an element within an `ng-repeat` annotation. If we wanted to alternate the background color of each note in our chapters view, we could do that with `ng-class-even` and `ng-class-odd`, as shown in the following code snippet:

```
<ul class='notes' ng-show='chapter.id == selectedChapterId'>
  <li ng-repeat='note in chapter.notes | orderBy:"id"' ng-class-
even='"even"' ng-class-odd='"odd"'>
    <div>#{{$index}}</div>
    <div><a ng-click='onDelete(note.id)'>delete</a></div>
    <p>{{note.content}}</p>
  </li>
<ul>
```

Now the even-numbered notes will get the class `even`, and the odd-numbered notes will get the class `odd`. If we added some CSS to give different background colors to `ul.notes li.even` and `ul.notes li.odd`, our notes would have alternating background colors.

There is one other set of annotations that can only be used inside a specific parent:

`ng-switch-when` and `ng-switch-default` can be used inside of `ng-switch`. This allows you to build a switch statement into your template. We could have done this instead of using two `ng-show` annotations in our chapters view:

```
<a href='#/chapter/{{chapter.id}}'>
  {{chapter.notes.length}}
  <ng-switch on='chapter.notes.length'>
    <span ng-switch-when='1'>note</span>
    <span ng-switch-default>notes</span>
  </ng-switch>
</a>
```

The `ng-switch` syntax is much cleaner than using `ng-show` and `ng-hide` for a large list of items. I kind of prefer `ng-show` and `ng-hide` over `nng-switch` for binary situations like our note/notes problem, but there is actually an even better solution in our case. Behold, `ng-pluralize` is given in the following code snippet:

```
<a href='#/chapter/{{chapter.id}}'>
  {{chapter.notes.length}}
  <ng-pluralize count='chapter.notes.length'
    when='{"1": "note", "other": "notes"}'>
  </ng-pluralize>
</a>
```

`ng-pluralize` is specifically designed to handle issues with pluralization, like our single note versus many or zero notes. In addition to a cleaner syntax, `ng-pluralize` also has semantic value; when someone else looks at your code and sees `ng-pluralize`, it's obvious what that code is doing.

There are a few other useful annotations we didn't use in our Guidebook application. Let's now look at some that I find very useful when working with AngularJS.

We used the `ng-click` event to handle user interaction in a couple of places. AngularJS also provides annotations for mouse events, such as `ng-mouse-enter` and `ng-mouse-move`. As a rule of thumb, always look for an AngularJS equivalent before using a standard JavaScript event handler. Like `$window`, this is better for testability, and it will also allow you to specify a handling function in your controller, like we did with `ng-click` and the `onDelete()` function.

AngularJS also provides a lot of wrappers for forms and input elements. The `ng-form` annotation is identical to an HTML form, except that it can be nested (HTML forms cannot). Input elements can be extended with properties such as `ng-required`, and `ng-pattern`. These are coming to standard HTML input elements in HTML5, but the AngularJS versions work in a number of older browsers. If you're aiming to support browsers that are more than a few years old, you should always use the `ng-` equivalents of any HTML5 input attributes.

Along the same lines, AngularJS provides functionality that is coming to other HTML elements in HTML5, but may not work in legacy browsers. Some examples are `ng-readonly`, `ng-selected`, and `ng-disabled`. Like the input element annotations, these work

just like their HTML5 counterparts, with the benefit of working in older browsers.

Another interesting annotation is `ng-cloak`. When a web page is rendered, the browser will render the full content in one pass. When building complex dynamic applications, we tend to do a lot of complex dynamic things in JavaScript, and sometimes that will spill over and happen after the initial page load. If we're still adjusting our views after page load, this can sometimes generate a flicker effect where content kind of blinks onto the screen, or shuffles position slightly. Now, in an idea world, you would refactor your view to better organize your visual dependencies, but sometimes we're pressed for time and need a quick fix, or we're stuck supporting a wish-it-would-die version of IE, and it seems to flicker no matter what we do.

That's where `ng-cloak` comes in. If you annotate an element with `ng-cloak`, AngularJS will hide it initially, and show it as soon as the page has been rendered.

Note

This is one of the things I really like about AngularJS. Every framework will cover common use cases, such as `ng-repeat` and `ng-show` equivalents, but AngularJS also covers the really obscure cases with properties such as `ng-cloak`.

One last subject before we move on: Remember way back in our very first Hello World application, when we printed out the current time?

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.j
s'></script>
    <script>
      function Clock($scope) {
        $scope.currentTime = new Date();
      }
    </script>
    <title>Welcome to AngularJS</title>
  </head>
  <body>
    <h1>Hello, World.</h1>
    <p ng-controller='Clock'>
      The current time is {{currentTime | date:'h:mm:ss a'}}.
    <p>
  </body>
</html>
```

We used the date filter to render the date in a readable format. We've also seen the `orderby` filter, which we used in our `ng-repeat` to sort chapters alphabetically by title. We

could also filter `ng-repeat` results with the `filter` filter, which takes a Boolean condition to decide whether each result should be shown or hidden. Here are a few other useful filters provided by AngularJS:

- The `currency` filter will render currency according to the current locale
- The `uppercase`, `lowercase`, and `number` filters help format text and numbers
- Text can be automatically JSON-formatted using the `json` filter
- The `linky` filter will automatically add HTML links to any links within a section of text

I've tried to give a full overview of the most useful view properties and annotations, but there are still quite a few I haven't covered here. For a full list, I encourage you to consult the official AngularJS API documentation at http://docs.angularjs.com/api.

Now that we have a solid understanding of what we can do with templates in AngularJS, we're ready to look at an important AngularJS feature we've been neglecting so far.

Let's talk data binding!

Two-way data binding

For all the fancy stuff you can build with a framework like AngularJS, there are just some basic user interface patterns that seem to exist in every application.

For example, how often do you find your use cases boiling down to these three simple steps?

1. You give the user an input control.
2. The user interacts with the control.
3. You update some other part of the screen to reflect the input.

We can illustrate this quite simply with a very basic color picker. Based on what we've seen so far, we'd probably implement something like the following:

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.m
in.js'></script>
    <script>
      function ColorController($scope) {
        $scope.setColor = function() {
          var colours = document.getElementsByTagName('input');
          document.getElementById('label').style.color = 'rgb('
            + colours[0].value + ','
            + colours[1].value + ','
            + colours[2].value + ')';
        }
      }
    </script>
    <title>AngularJS Color Picker</title>
  </head>
  <body ng-controller='ColorController'>
    <h1 id='label' style='color: rgb(128,128,128);'>Drag to change
colour:</h1>
    <input type='range' min='0' max='255' step='1'ng-model='r' ng-
change='setColor();'>
    <input type='range' min='0' max='255' step='1'ng-model='g' ng-
change='setColor();'>
    <input type='range' min='0' max='255' step='1'ng-model='b' ng-
change='setColor();'>
  </body>
</html>
```

This will work, and it kind of uses some AngularJS features to make things a bit easier, but there's a lot to dislike about this code. I mean, we defined `ng-model` and didn't even use it. (But we did need it; `ng-change` will crash if you don't have an `ng-model` available to the element.)

And what is that controller even doing? Sure it has a bit of logic, but we could just as easily inline that function on each input element. That function isn't actually controlling anything.

Finally, do we really need to use `getElementById`? Isn't the whole point of AngularJS to enhance existing technologies, and help us avoid doing so much manual DOM-labor?

Well, it turns out this is a perfect situation to use data binding.

Data binding is the idea that some user input can be tied directly to view output. There's no point going through a controller if we're just going to present the output the same way every time. That should be the view's job, anyway; the controller shouldn't be telling the view how to render itself.

Data binding is common in a lot of JavaScript frameworks, but it's particularly elegant in AngularJS. Here's how our color picker looks if we build it with data binding:

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.m
in.js'></script>
    <title>AngularJS Color Picker</title>
  </head>
  <body ng-init='r=128; b=128; g=128;'>
    <h1 style='color: rgb({{r}},{{g}},{{b}});'>Drag to change
colour:</h1>
    <input type='range' min='0' max='255' step='1' ng-model='r'>
    <input type='range' min='0' max='255' step='1' ng-model='g'>
    <input type='range' min='0' max='255' step='1' ng-model='b'>
  </body>
</html>
```

That's more like it! Here we're using the `ng-init` annotation to pre-set the `r`, `g`, and `b` values in our `ng-model`. By marking `ng-model` on each of our input elements, those elements are directly bound to the `r`, `g`, and `b` values. When the input fields are updated, AngularJS automatically keeps the text color up-to-date with the latest values.

But it's called two-way data binding, isn't it? We've only shown that the binding works in one direction—from `ng-model` to the raw values. What if we change the raw values? Does that update the `ng-model` instances, too?

Let's add a little extra code to show that the binding works in the other direction as well:

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.m
in.js'></script>
    <title>AngularJS Color Picker</title>
  </head>
  <body ng-init='r=128; b=128; g=128;'>
    <h1 ng-click='r=128; b=128; g=128;'style='color: rgb({{r}},
{{g}},{{b}});'>Drag to change colour, click here to reset.</h1>
    <input type='range' min='0' max='255' step='1' ng-model='r'>
    <input type='range' min='0' max='255' step='1' ng-model='g'>
    <input type='range' min='0' max='255' step='1' ng-model='b'>
  </body>
</html>
```

Now if we click the text on the page, the color will change back to its original gray, and the input ranges will reset.

Before we move on to something else, let's talk a little more about `ng-model`.

When you first hear the term `ng-model`, it sounds like it's going to be a model object kind of

like the controller object that matches `ng-controller`. But that's not really what it is; there is no such object.

After a few examples, you start to think it's kind of like `ng-view`. After all, `ng-view` isn't something we typically extend in JavaScript; we just use it to enhance HTML to build our views. But that's not really what `ng-model` is for either, is it? We're not enhancing some pre-existing model.

`ng-model` is really responsible for view data. It's fantastic at managing data within a view. We've used it for data binding here, but we also used it back in our Guidebook's add note view to send form data to our `AddNoteController`.

When you see `ng-model`, resist the urge to think "That's a model where I can store my data". Instead, think "That's a place to put data that I want to move around my view".

We've now seen an example of how to use two-way data binding to build a dynamic view with very little code. We've learned a bit more about `ng-model`, and clarified what it is and what it isn't responsible for.

For our next trick, let's take a deep dive into the depths of AngularJS. Let's get much more comfortable with modules.

Modules

We caught a glimpse of modules at the start of our Guidebook application. Remember our configuration file? It is given as follows:

```
var guidebookConfig = function($routeProvider) {
  $routeProvider
    .when('/', {
      controller: 'ChaptersController',
      templateUrl: 'view/chapters.html'
    })
    .when('/chapter/:chapterId', {
      controller: 'ChaptersController',
      templateUrl: 'view/chapters.html'
    })
    .when('/addNote/:chapterId', {
      controller: 'AddNoteController',
      templateUrl: 'view/addNote.html'
    })
    .when('/deleteNote/:chapterId/:noteId', {
      controller: 'DeleteNoteController',
      templateUrl: 'view/addNote.html'
    })
  ;
};
var Guidebook = angular.module('Guidebook',
[]).config(guidebookConfig);
```

That last line is where we defined our Guidebook module. As we discussed earlier, this creates a namespace for our application. What we haven't discussed yet is how it also gives us a mechanism for creating AngularJS components.

For example, look at how we defined the add and delete note controllers:

```
Guidebook.controller('AddNoteController',
  function ($scope, $location, $routeParams, NoteModel) {
    var chapterId = $routeParams.chapterId;
    $scope.cancel = function() {
      $location.path('/chapter/' + chapterId);
    }
    $scope.createNote = function() {
      NoteModel.addNote(chapterId, $scope.note.content);
      $location.path('/chapter/' + chapterId);
    }
  }
);

Guidebook.controller('DeleteNoteController',
  function ($scope, $location, $routeParams, NoteModel) {
    var chapterId = $routeParams.chapterId;
    NoteModel.deleteNote(chapterId, $routeParams.noteId);
    $location.path('/chapter/' + chapterId);
  }
);
```

Since Guidebook is a module, we're really making a call to `module.controller()` to define our controller. We also called `Guidebook.service()`, which is really just `module.service()`, to define our models.

One advantage of defining controllers, models, and other components as part of a module is that it groups them together under a common name. This way, if we have multiple applications on the same page, or content from another framework, it's easy to remember which components belong to which application.

The second advantage of defining components as part of a module is that AngularJS will do some heavy lifting for us.

Take our `NoteModel`, for instance. We defined it as a service by calling `module.service()`. AngularJS provides some extra functionality to services within a module. One example of that extra functionality is dependency injection. This is why in our `DeleteNoteController`, we can include our note model simply by asking for it in the controller's function:

```
Guidebook.controller('DeleteNoteController',
  function ($scope, $location, $routeParams, NoteModel) {
    var chapterId = $routeParams.chapterId;
    NoteModel.deleteNote(chapterId, $routeParams.noteId);
    $location.path('/chapter/' + chapterId);
  }
);
```

This only works because we registered both `DeleteNoteController` and `NoteModel` as parts of the same module. (Dependency injection is really neat, by the way; we'll talk more about it later in this section.)

Let's talk a little more about `NoteModel`. When we define controllers on our module, we call `module.controller()`. For our models, we called `module.service()`. Why isn't there a `module.model()`?

Well, it turns out models are a good bit more complicated than controllers. In our Guidebook application, our models were relatively simple, but what if we were mapping our models to some external API, or a full-fledged relational database?

Because there are so many different types of models with vastly different levels of complexity, AngularJS provides three ways to define a model: as a service, as a factory, and as a provider.

We've already seen how we define a service in our `NoteModel`; it is as follows:

```
Guidebook.service('NoteModel', function() {
  this.getNotesForChapter = function(chapterId) {
    ...
  };
  this.addNote = function(chapterId, noteContent) {
    ...
  };
  this.deleteNote = function(chapterId, noteId) {
    ...
  };
});
```

It turns out this is the simplest type of model. We're simply defining an object with a number of functions that our controllers can call. This is all we needed in our case, but what if we were doing something a little more complicated?

Let's pretend that instead of using HTML5 local storage to store and retrieve our note data, we're getting it from a web server somewhere. We'll need to add some logic to set up and manage the connection with that server.

The service definition can help us a little here with the setup logic; we're returning a function, so we could easily add some initialization logic.

When we get into managing the connection, though, things get a little messy. We'll probably want a way to refresh the connection, in case it drops. Where would that go? With our service definition, our only option is to add it as a function like we have for `getNotesForChapter()`. This is really hacky; we're now exposing how the model is implemented to the controller, and worse, we would be allowing the controller to refresh the connection.

In this case, we would be better off using a factory. Here's what our `NoteModel` would look like if we had defined it as a factory:

```
Guidebook.factory('NoteModel', function() {
  return {
    getNotesForChapter: function(chapterId) {
      ...
    },
    addNote: function(chapterId, noteContent) {
      ...
    },
    deleteNote: function(chapterId, noteId) {
      ...
    }
  };
});
```

Now we can add more complex initialization logic to our model. We could define a few functions for managing the connection privately in our factory initialization, and give our data

methods references to them. The following is what that might look like:

```
Guidebook.factory('NoteModel', function() {
  var refreshConnection = function() {
    ...
  };
  return {
    getNotesForChapter: function(chapterId) {
      ...
      refreshConnection();
      ...
    },
    addNote: function(chapterId, noteContent) {
      ...
      refreshConnection();
      ...
    },
    deleteNote: function(chapterId, noteId) {
      ...
      refreshConnection();
      ...
    }
  };
});
```

Isn't that so much cleaner? We've once again kept our model's internal workings completely separate from our controller.

Let's get even crazier. What if we backed our models with a complex database server with multiple endpoints? How could we configure which endpoint to use?

With a service or a factory, we could initialize this in the model. But what if we have a whole bunch of models? Do we really want to hardcode that logic into each one individually?

At this point, the model shouldn't be making this decision. We want to choose our endpoints in a configuration file somewhere.

Reading from a configuration file in either a service or a factory will be awkward. Models should focus solely on storing and retrieving data, not messing around with configuration updates.

Provider to the rescue! If we define our `NoteModel` as a provider, we can do all kinds of configuration from some neatly abstracted configuration file.

Here's how our `NoteModel` looks if we convert it into a provider:

```
Guidebook.provider('NoteModel', function() {
  this.endpoint = 'defaultEndpoint';
  this.setEndpoint = function(newEndpoint) {
    this.endpoint = newEndpoint;
  };
  this.$get = function() {
    var endpoint = this.endpoint;
    var refreshConnection = function() {
      // reconnect to endpoint
    };
    return {
      getNotesForChapter: function(chapterId) {
        ...
        refreshConnection();
        ...
      },
      addNote: function(chapterId, noteContent) {
        ...
        refreshConnection();
        ...
      },
      deleteNote: function(chapterId, noteId) {
        ...
        refreshConnection();
        ...
      }
    };
  };
});
```

Now if we add a configuration file to our application, we can configure the endpoint from outside the model:

```
Guidebook.config(function(NoteModelProvider) {
  NoteModelProvider.setEndpoint('anEndpoint');
});
```

Providers give us a very powerful architecture. If we have several models, and we have several endpoints, we can configure all of them in one single configuration file. This is ideal, as our models remain single-purpose and our controllers still have no knowledge of the internals of the models they depend on.

Modules provide an easy, flexible way to create components in AngularJS. We've seen three different ways to define a model, and we've discussed the benefits of using a module to create an application namespace.

We'll revisit modules one final time when we talk about directives, at the end of this section. For now, let's move on to another important AngularJS feature—dependency injection.

Dependency injection

Dependency injection is an exciting feature, and one not typically found in JavaScript frameworks. We've used dependency injection already, to load model instances into our controllers.

Recall our `DeleteNoteController`, and how it takes our `NoteModel` as a parameter:

```
Guidebook.controller('DeleteNoteController',
  function ($scope, $location, $routeParams, NoteModel) {
    var chapterId = $routeParams.chapterId;
    NoteModel.deleteNote(chapterId, $routeParams.noteId);
    $location.path('/chapter/' + chapterId);
  }
);
```

We never call the function that instantiates `DeleteNoteController`, so we never explicitly pass in the `NoteModel`. How does it get there?

Typically, when one component in an application depends on another component, we need to explicitly map that dependency. In standard JavaScript, if you have an object that depends on another object, it's your responsibility to make sure a proper reference gets there. Nobody's going to do that for you, and you're going to have to do some grunt work to set that up in your code.

What we're doing here with AngularJS is different. We've defined `NoteModel` and `DeleteNoteController`, but we're not explicitly telling them about each other in our code. Instead, AngularJS will look at our `DeleteNoteController` at runtime, see that it requires something called `NoteModel`, and look for something called `NoteModel` in our Guidebook module. When it finds our `NoteModel`, it creates an instance and sends it to `DeleteNoteController`.

This is called dependency injection, because AngularJS is injecting a dependency (`NoteModel`) into our `DeleteNoteController`. In fact, `$scope`, `$location`, and `$routeParams` are also dependencies, and are also being injected.

This is convenient for us, because we don't need to spend as much time and focus sorting out how our controllers are going to get references to our models. It's also very important for unit testing, because it allows test frameworks to inject mock objects into our code at runtime, in place of our real ones.

The downside to dependency injection is that at first, it kind of looks like magic. This can make a framework less approachable for novice developers, and can cause frustration when it doesn't "just work".

To wield dependency injection with confidence, it's important to fully understand what's happening behind the curtain.

Since AngularJS is open source, we can check it out for ourselves. Here is a snippet of the AngularJS file we're using, available at http://ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.js:

```
function bootstrap(element, modules) {
  element = jqLite(element);
  modules = modules || [];
  modules.unshift(['$provide', function($provide) {
    $provide.value('$rootElement', element);
  }]);
  modules.unshift('ng');
  var injector = createInjector(modules);
  injector.invoke(
    ['$rootScope', '$rootElement', '$compile', '$injector',
      function(scope, element, compile, injector) {
        scope.$apply(function() {
          element.data('$injector', injector);
          compile(element)(scope);
        });
      }
    ]
  );
  return injector;
}
```

The `bootstrap()` method is called at startup to initialize important parts of AngularJS. This is where the injector is created—the heart of dependency injection in AngularJS.

There are two interesting method calls here relating to dependency injection: `createInjector()` and `injector.invoke()`. We'll get to `createInjector()` in a moment, but first let's talk about `injector.invoke()`.

Think back to our `DeleteNoteController`. We've defined it as a function called `DeleteNoteController()`. The `injector.invoke()` method is what eventually calls that function.

As a convenience, `injector.invoke()` can also accept a list of object references, and a function to run for each of them. That's what we see happening in `bootstrap()`, and it's notable because there are some interesting objects being invoked, which are as follows:

- `$rootScope` is the root scope object. (When we use `$scope`, we're referring to a child of `$rootScope`.)
- `$rootElement` is the root element in our markup. Usually, this is wherever you've placed your `ng-app` annotation.
- `$compile` is a service that matches annotations in our markup to objects in AngularJS. (These objects are called directives—more on those soon.)
- `$injector` is a reference to the injector itself. Yes, the injector can inject itself to components that depend on it.

Now, what about our `DeleteNoteController`? How is it invoked?

Note the line where `createInjector()` is being called. That modules object will contain our Guidebook module. (If you want to see where that comes from, check out the `angularInit()` function. It's another delicious slice of AngularJS startup pie, but a little too large to reprint here, and not quite relevant to our current topic.)

If we take a peak inside `createInjector()`, we'll see the following nifty line:

```
forEach(loadModules(modulesToLoad), function(fn) {
instanceInjector.invoke(fn || noop); });
```

`modulesToLoad` is the modules object passed in through `createInjector()`, containing our Guidebook module. On this line, AngularJS is going to make a call to `loadModules()`, and is expecting to get back an array. This is run through `forEach()`, thus calling `invoke()` on each item in the array.

Stepping into `loadModules()`, we can see that this is where our module is injected, which will let the injector know about components such as our `NoteModel`. The `loadModules()` function returns an array of modules, which will contain our Guidebook module, ready to be invoked by the waiting `forEach()` block.

The act of calling `invoke()` will run all our Guidebook module's component functions, such as `DeleteNoteController()`, and because the module was just injected, proper references to components such as `NoteModel` will already be in place, ready to go.

To summarize, the flow for dependency injection looks something as follows:

1. We write a component that AngularJS will be able to invoke (such as `DeleteNoteController`), with dependencies that AngularJS will be able to inject (such as `NoteModel`).
2. At startup AngularJS creates an injector and passes in our module (Guidebook), containing all our components (`DeleteNoteController`, `NoteModel`, and so on).
3. The injector injects our dependencies (such as `NoteModel`), so that they'll be ready when the components that need them (such as `DeleteNoteController`) are invoked.
4. Our components are invoked, causing their dependencies to be injected, and their internal functions to be run.

We just walked through steps 2, 3, and 4. But step 1 raises an interesting question: What exactly can AngularJS invoke? Another way to phrase this is: How can we annotate our components to tell AngularJS that they can be invoked?

In the following code block we recall how we defined our `DeleteNoteController`:

```
Guidebook.controller('DeleteNoteController',
  function ($scope, $location, $routeParams, NoteModel) {
    var chapterId = $routeParams.chapterId;
    NoteModel.deleteNote(chapterId, $routeParams.noteId);
    $location.path('/chapter/' + chapterId);
  }
);
```

Note how we're passing a string version of our function name, `'DeleteNoteController'`, to `module.controller()` along with our `DeleteNoteController` function. AngularJS can use this name directly when it injects and invokes our components, because we've explicitly told the framework how to reference each component.

There are actually two other ways to annotate components in AngularJS so that they are properly handled during dependency injection. One is to manually register our function's dependencies with the injector, and is given as follows:

```
var deleteNoteController = function ($scope, $location,
$routeParams, NoteModel) {
    var chapterId = $routeParams.chapterId;
    NoteModel.deleteNote(chapterId, $routeParams.noteId);
    $location.path('/chapter/' + chapterId);
  }
);
deleteNoteController.$inject = ['$scope', '$location',
'$routeParams', 'NoteModel'];
```

This is just as valid as what we've been doing so far, but it's slightly more work and a little less maintainable; if we ever change the dependencies in our function definition, we have to remember to update them in the manual `$inject` as well.

The second way to annotate components for dependency injection is to do what we did in our Hello World application:

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.m
in.js'></script>
    <script>
      function Clock($scope) {
        $scope.currentTime = new Date();
      }
    </script>
    <title>Welcome to AngularJS</title>
  </head>
  <body>
    <h1>Hello, World.</h1>
    <p ng-controller='Clock'>
      The current time is {{currentTime | date:'h:mm:ss a'}}.
    <p>
  </body>
</html>
```

Here, we're not specifying anything that will help the injector or invoker map the `$scope` dependency to the `Clock()` function. AngularJS is actually smart enough to figure this out on its own, because it will parse the parameter list, see `$scope`, and look for something called `$scope` to inject. But there is one major disadvantage to this approach:

What will happen if we minify this file?

Suddenly the `$scope` parameter is no longer called `$scope`. It's probably been shortened to the letter A or something similar. AngularJS will parse the parameter list, see A, look for something called A to inject, and find nothing.

This will cause our app to crash, which is obviously not ideal. The other two styles of defining components and their dependencies will survive minification.

This concludes our deep dive into dependency injection. Hopefully, seeing how `inject()` and `invoke()` work under the hood has helped to demystify the magic behind dependency injection in AngularJS.

We're running out of features to explore! Join me in our final investigation, where we'll look at the most powerful AngularJS feature of all—directives.

Directives

Do you remember what makes AngularJS special?

As we covered in the introduction, AngularJS is special because of how it solves the HTML/JavaScript interaction problem. Instead of layering abstractions on top of

abstractions, AngularJS integrates and extends HTML and JavaScript to add dynamic functionality.

Directives are a feature unique to AngularJS, and they really exemplify that integrate and extend mantra.

The following code snippet helps recall our Guidebook's chapters view:

```
<ul>
  <li ng-repeat='chapter in chapters | orderBy:"title"'>
    <h2>{{chapter.title}}</h2>
    <p>{{chapter.summary}}</p>
    <p>
      <a href='#/chapter/{{chapter.id}}'>
        {{chapter.notes.length}}
        <span ng-show='chapter.notes.length == 1'>note</span>
        <span ng-show='chapter.notes.length != 1'>notes</span>
      </a>
      |
      <a href='#/addNote/{{chapter.id}}'>add note</a>
    </p>
    <ul class='notes' ng-show='chapter.id == selectedChapterId'>
      <li ng-repeat='note in chapter.notes | orderBy:"id"'>
        <div>#{{$index}}</div>
        <div><a ng-click='onDelete(note.id)'>delete</a></div>
        <p>{{note.content}}</p>
      </li>
    <ul>
  </li>
</ul>
```

This is a good template. It's concise, and you just need a quick glance to see what's going on. But good templates don't usually stay that way.

Think back to the last big project you worked on. Maybe it was for a client, or maybe it was at work or school, or even just something ambitious you wanted to build for yourself.

I bet it started out pretty well, didn't it? You probably had nice, clean code that was well written, had comments, and made perfect sense. A child could read it.

Now what was it like when you finished it? Did that beautiful clarity from the early days carry all the way through to the end? Or did a bunch of features sneak in and bloat up your codebase?

View code is especially vulnerable to these sorts of problems. Even with a smart framework that helps you build good templates, things can quickly get out of hand.

In our Guidebook application's case, the chapter view is really the main view of the application. What if our boss or our client (or that pesky voice inside our head that loves to

tinker with things) wants to add some more content here?

That notes section is just begging to be expanded. We could include little share buttons for social networks, or the ability to nest comments, or vote them up or down. Can you imagine what that would do to our nice little template?

Not so concise anymore, I bet. Our `chapters.html` file will grow from a handful of kilobytes to several hundred, we'll have `div` blocks nested 20 levels deep, and we'll waste all kinds of time scrolling up and down the entire file looking for that one line we need to change.

Nobody wants that. What we want to do is something like the following:

```
<ul>
  <li ng-repeat='chapter in chapters | orderBy:"title"'>
    <h2>{{chapter.title}}</h2>
    <p>{{chapter.summary}}</p>
    <p>
      <a href='#/chapter/{{chapter.id}}'>
        {{chapter.notes.length}}
        <span ng-show='chapter.notes.length == 1'>note</span>
        <span ng-show='chapter.notes.length != 1'>notes</span>
      </a>
      |
      <a href='#/addNote/{{chapter.id}}'>add note</a>
    </p>
    <gb-note-list ng-show='chapter.id == selectedChapterId' />
  </li>
</ul>
```

The chapter view doesn't really care how the notes are rendered; it just wants the notes to be there. If the notes want to have a bunch of extra whiz-bang features, that's fine; our chapter view simply wants to render a list of notes per chapter.

As it turns out, we're not the only ones that think this is a neat idea. One of the things we can do with directives is create reusable, HTML-style elements. Here's the tiny amount of JavaScript we need to make that template work, in a file I'll call `NoteList.js`:

```
Guidebook.directive('gbNoteList', function() {
  return {
    restrict: 'E',
    templateUrl: 'view/directives/noteList.html'
  };
});
```

Fairly straightforward, isn't it? We're defining a directive on our Guidebook module, and telling it where to find its content: a new folder called directives inside our views directory. That's also where I put `NoteList.js`—I like to keep my directives' JavaScript and markup files together, but that's personal preference. AngularJS doesn't care where files go, so long as they're referenced properly.

Don't worry about the restrict keyword for now; we'll cover that in a few minutes. First, let's take a look at `noteList.html`:

```
<ul>
  <li ng-repeat='note in chapter.notes | orderBy:"id"'>
    <div>#{{$index}}</div>
    <div><a ng-click='onDelete(note.id)'>delete</a></div>
    <p>{{note.content}}</p>
  </li>
</ul>
```

Now that looks familiar! It's the same content we had originally. This is because, by default, directives inherit scope; the note directive's markup can access everything the chapter view can access. It turns out this is actually a bad thing, and we'll talk about why in just a moment. Let's wrap up a couple of loose ends first.

In order for our directive to be included in the application, we need to load the script in our head, in `guidebook.html`:

```
<head>
    <script
src='http://ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.j
s'></script>
    <script src='Guidebook.js'></script>
    <script src='controller/ChapterController.js'></script>
    <script src='controller/NotesControllers.js'></script>
    <script src='model/NoteModel.js'></script>
    <script src='model/ChapterModel.js'></script>
    <script src='view/directives/NoteList.js'></script>
    <link rel='stylesheet' href='view/styles.css'>
    <title>AngularJS Starter Guidebook</title>
</head>
```

Finally, a quick note on naming conventions: I like to preface any directives I make with a couple of letters, just to make it extra clear to anyone reading my code that those are my directives and not something AngularJS provides. In this case, I chose `gb-` as a shorthand for Guidebook.

It's also worth highlighting that AngularJS does some funny business with the names of directives properties, like the directive's name. Specifically, it converts any directive properties written in camel case, `likeThisOne`, to snake case, `like-this-one`. We can see in our case that it converted `gbNoteList` to `gb-note-list`. Keep this in mind when writing your views. If we'd written `<gbNoteList>` in our chapters view markup, our directive wouldn't have been loaded.

Now, let's talk about code reuse.

One of the really huge advantages of directives is that they're easy to configure for reuse.

For example, let's think about what would happen if we wanted to make a new view in our Guidebook application that just showed a giant list of all the notes for every chapter. Before we were using directives, this would be a really big problem for us; we would have to duplicate the view code related to notes in that new view, and maintain both copies.

With directives, we're much better off. All we need to do is use `ng-repeat` to show a `gb-note-list` element for each chapter. We could write that in a matter of minutes, and every time we need to update the note list to add a new social network sharing button, we only need to change one file—the note list directive markup.

Now, before we can go reusing this lovely note list element we created, we have to revisit scope. I mentioned just a moment ago that inheriting scope is a bad thing. There are a few reasons for this.

First, the note list directive can mess with all our chapter view's data. This is completely unnecessary, and dangerous. It would be easy for a naïve developer to add some code here that relies on or modifies chapter data. This could cause a lot of unnecessary bugs; directives are supposed to be self-contained, and nobody is going to think to check the note list directive when something weird happens to our chapter data.

Second, we're opening the door to name clashing. What if the note directive and the chapters view each had a variable with the same name? One would clobber the other, which would be very dangerous.

We want our directives to have their own scope. AngularJS can do this for us, but we need to provide it with a little more information. Here's how we would have to modify our directive definition:

```
Guidebook.directive('gbNoteList', function() {
  return {
    restrict: 'E',
    templateUrl: 'view/directives/noteList.html',
    scope: {
      show: '=show',
      notes: '=notes',
      orderValue: '@orderBy',
      onDelete: '=deleteNoteHandler'
    }
  };
});
```

By defining the `scope` property, we're explicitly listing what we want in our directive's scope, and how we expect to get it. For example, `orderValue: '=@orderBy'` means that we are adding the `orderValue` property to our scope, and views that use our directive can pass that value to us using the `order-by` annotation.

The characters beginning our expected value are important: An equals sign (`=`) means pass

the value as-is, which is useful for arrays and functions. The *at* symbol (`@`) will render the given information as a string. There's also the ampersand (`&`), which will render the content as an expression.

In our example, we could have actually written the following:

```
scope: {
  show: '=',
  notes: '=,
  orderValue: '@orderBy',
  onDelete: '=deleteNoteHandler'
}
```

This is a shortcut notation provided by AngularJS for cases where the property name is the same on both sides of the directive's scope. (For example, we are expecting a `notes` value, and use it to set a `notes` property locally; we can use the shortcut notation because we're using the term `notes` in both cases.)

With our new directive definition in place, we have to make a few updates to both the calling code (the chapter view) and our directive markup. Here's how the calling code changes:

```
<gb-note-list notes='chapter.notes'
  show='chapter.id == selectedChapterId'
  order-by='id'
  delete-note-handler='onDelete' />
```

Note that the `notes`, `show`, `order-by`, and `delete-note-handler` properties are the same ones we defined in our directive definition's `scope` property.

Our directive's markup must also change to reflect the new scope changes:

```
<ul class='notes' ng-show='show'>
  <li ng-repeat='note in notes | orderBy:orderValue'>
    <div>#{{$index}}</div>
    <div><a ng-click='onDelete(note.id)'>delete</a></div>
    <p>{{note.content}}</p>
  </li>
<ul>
```

Now we have a reusable directive we could drop into any number of views. It will render correctly as long as the calling code provides the `notes`, `show`, `orderValue`, and `onDelete` properties.

Let's talk about that `restrict` property we haven't covered yet. It's used to restrict how our directive should be rendered. In our case, we're using `E`, which stands for `Element`. This means that our directive must be instantiated in markup, as an HTML-style element.

There are three other ways to specify how a directive should be rendered:

`C` is for `Class`. This allows us to render a directive via an element's class attribute. For our directive, that would have looked as follows:

```
<div class='gb-note-list' notes='chapter.notes'
  show='chapter.id == selectedChapterId'
  order-by='id'
  delete-note-handler='onDelete'>
</div>
```

Using the class attribute is useful if you are programmatically generating DOM, and you would like some of that DOM to use directives. It's also a handy way to target CSS at your directives.

`M` is for `Comment`. This allows us to render a directive via an HTML comment, like so:

```
<!-- directive:gb-note-list -->
```

The other methods of invoking a directive all use custom properties that will cause HTML validators to freak out. If validation is important to you, you can use `Comment` to write semantically valid AngularJS markup while still using directives, sort of. In our case, this actually won't work, because HTML comments don't support attributes, and we need to pass in a function, among other things.

`A` is for `Attribute`. This allows us to render a directive via an attribute, like so:

```
<div gb-note-list notes='chapter.notes'
  orderBy='id'
  show='chapter.id == selectedChapterId'
  deleteNoteHandler='onDelete'>
</div>
```

You may notice that this looks a little familiar. That `gb-note-list` annotation looks a whole lot like the `ng-` annotations we've been looking at all along, doesn't it?

Well, that's because anytime we've used an annotation throughout this entire book, such as `ng-repeat` or `ng-show`, we've actually used a directive. So you already know quite a bit about directives; we've just been referring to them as annotations.

The proper term is directive. I didn't mean to mislead you; it was simply easier to refer to them as annotations because annotation is a familiar term, and you need to know a lot about AngularJS before you can truly understand what a directive is and why it's so important.

Let's sum up everything we know about annotations/directives:

- Directives are used to add dynamic functionality to an HTML template
- AngularJS provides a number of useful directives out of the box, such as `ng-repeat`, `ng-model`, and `ng-click`
- You can build your own directives, and instantiate them as elements, classes, comments, or attributes
- You should use directives to write cleaner, more reusable view templates

Together we've explored a lot more than just directives! We started by talking casually about AngularJS and why it's special. We walked through some Hello World code samples, and then created a full-blown MVC application— with an architecture designed to scale in the real world.

After that, we looked in-depth at the most important aspects of AngularJS. We examined view templates, and got a feel for the kinds of directives included in AngularJS. We saw how two-way data binding can be used to build a concise, but highly interactive interface. We did a deep dive into modules, to understand how to build components for advanced applications. We explored dependency injection, and took a brief tour of how AngularJS works internally. And we saw how to build reusable components using directives—the crux of a great AngularJS application.

I hope you've found this guide useful. Our time together has almost come to a close, but I have a few remaining pages to help you continue your journey with AngularJS.

The final section lists additional resources to further your skills with AngularJS. There are links to official pages and documentation, and notes about meet-ups and where to go for help.

People and places you should get to know

You began your journey by learning about how AngularJS is different. You were introduced to the syntax, and got a feel for AngularJS in our Hello World application. You fought through our Guidebook application, chiseling a reusable Model-View-Controller architecture in the process. You took on one important feature after another, finally coming to conquer templates, data binding, modules, dependency injection, and directives.

With your newfound skills and your vast range of knowledge, you're ready to take on your next challenge. You're an AngularJS ninja. You've finished training, and are ready for battle.

Before you go, I'd like to leave you with a few resources you can use to hone your skills even further. Think of them as a dojo, where you can meet and train with AngularJS masters, continuing to evolve the deadly art of web craftsmanship.

Official AngularJS websites

Here are all the official AngularJS entities. These are the places to go if you need to get the latest version of AngularJS, or hear the latest news about what's next for your new favorite JavaScript framework. Refer to the following list:

- http://angularjs.org: The official AngularJS website is your source of truth for all things AngularJS.
- http://blog.angularjs.org: The official AngularJS blog is a great place to hear the latest news about upcoming releases, meet-ups, and tutorials.
- http://docs.angularjs.org/api: I'd suggest bookmarking this one. AngularJS has excellent documentation, and many of the entries contain links to tutorials or other related developer resources.
- http://code.angularjs.org: Every now and then you may need to look at some code or documentation from an earlier version of AngularJS.
- https://github.com/angular/angular.js: The AngularJS GitHub page is where you can download the AngularJS source, should you ever need it, and the home of the AngularJS bug tracker, where you're encouraged to log issues with the framework.

Where to go for help

Everybody struggles sometimes, even with easy-to-use frameworks such as AngularJS. Fortunately, you're not in this alone. There are a whole bunch of people out there who are just waiting to help you through whatever challenges you may face. Here's where you can find them:

- http://groups.google.com/group/angular: The AngularJS Google group is a forum-

like interface for developers to talk about and help each other with AngularJS-related issues.
- http://stackoverflow.com/questions/tagged/angularjs: As usual, Stack Overflow is a great place to go if you have specific questions about some bug you can't seem to fix.
- http://webchat.freenode.net/?channels=angularjs&uio=d4: For a slightly more meta experience, check out the #angularjs channel on freenode. This is a place where developers like to congregate to discuss the AngularJS framework.

Social networks

These days it's trendy for every new framework or service to have a social media presence. Well AngularJS isn't just in it for the trends; the YouTube, Twitter, and Google+ accounts for AngularJS pump out a constant stream of useful information.

Pick your poison(s):

- http://www.youtube.com/user/angularjs: AngularJS has its own YouTube channel, which I highly recommend checking out. There are recordings of talks, tutorials, interviews, and all kinds of other helpful resources.
- https://plus.google.com/+AngularJS/posts: Circle AngularJS on Google+ to get access to case studies, blog posts, and other online community content promoted by AngularJS, including Q&A hangouts.
- https://twitter.com/angularjs: Follow the AngularJS Twitter feed to keep up with current events. There are often tweets about conferences, meet-ups, and other gatherings of people passionate about AngularJS.
- http://twitter.com/packtopensource: While we're on the topic of Twitter, did you know Packt has a Twitter account? If you're interested in open source technologies, you should check them out.

AngularJS meet-ups

Sometimes we all just need to get away from our screens. The AngularJS community tries to help you with this by organizing regular meet-ups around the world!

If this sounds like something you'd like to attend sometime, here's where you can find information related to recurring meet-ups:

- http://angularjs.meetup.com

If none of those are anywhere near you, why not consider creating one yourself? The beauty of the Web is how easy it is to discover people around you with the same interests.