Quick answers to common problems

AngularJS Web Application Development Cookbook

Over 90 hands-on recipes to architect performant applications and implement best practices in AngularJS

Matt Frisbie

[PACKT] open source*
PUBLISHING community experience distilled

AngularJS Web Application Development Cookbook

Over 90 hands-on recipes to architect performant applications and implement best practices in AngularJS

Matt Frisbie

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

AngularJS Web Application Development Cookbook

First published: December 2014

Production reference: 1191214

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

ISBN 978-1-78328-335-4

`www.packtpub.com`

Cover image by Suyog Gharat (`yogiee@me.com`)

Credits

Author
Matt Frisbie

Reviewers
Pawel Czekaj
Patrick Gillespie
Aakash Patel
Adam Štipák

Commissioning Editor
Akram Hussain

Acquisition Editor
Sam Wood

Content Development Editor
Govindan K

Technical Editors
Taabish Khan
Parag Topre

Copy Editors
Deepa Nambiar
Neha Vyas

Project Coordinator
Shipra Chawhan

Proofreaders
Simran Bhogal
Maria Gould
Ameesha Green
Paul Hindle

Indexer
Mariammal Chettiyar

Graphics
Abhinash Sahu

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Matt Frisbie is currently a full stack developer at DoorDash (YC S13), where he joined as the first engineer. He led their adoption of AngularJS, and he also focuses on the infrastructural, predictive, and data projects within the company.

Matt has a degree in Computer Engineering from the University of Illinois at Urbana-Champaign. He is the author of the video series *Learning AngularJS*, available through O'Reilly Media. Previously, he worked as an engineer at several educational technology start-ups.

About the Reviewers

Pawel Czekaj has a Bachelor's degree in Computer Science. He is a web developer with strong backend (PHP, MySQL, and Unix systems) and frontend (AngularJS, Backbone.js, jQuery, and PhoneGap) experience. He loves JavaScript and AngularJS. Previously, he has worked as a senior full stack web developer. Currently, he is working as a frontend developer for Cognifide and as a web developer for SMS Air Inc. In his free time, he likes to develop mobile games. You can contact him at `http://yadue.eu`.

Patrick Gillespie is a senior software engineer at PROTEUS Technologies. He has been working in the field of web development for over 15 years and has both a Master's and Bachelor's degree in Computer Science. In his spare time, he enjoys working on web projects for his personal site (`http://patorjk.com`), spending time with his family, and listening to music.

Aakash Patel is the cofounder and CTO of Flytenow, a ride sharing platform for small planes. He has industry experience of client-side development using AngularJS, and he is a student at Carnegie Mellon University (CMU).

Adam Štipák is currently a full stack developer. He has more than 8 years of professional experience with web development. He specializes in AMP technologies (where A stands for Apache, M for MySQL, and P for PHP). He also likes other technologies such as JavaScript, AngularJS, and Grunt. He is also interested in functional programming in Scala. He likes open source software in general.

Writing about a subject as tumultuous as JavaScript frameworks is a bit like bull riding.

To Jordan, my family, and my friends—you helped me hang on.

Table of Contents

Preface

"Make it work. Make it right. Make it fast."

Back when the world was young, Kent Beck forged this prophetic sentiment. Even today, in the ultra-modern realm of performant single-page application JavaScript frameworks, his idea still holds sway. This nine-word expression describes the general progression through which a pragmatic developer creates high-quality software.

In the process of discovering how to optimally wield a technology, a developer will execute this progression many times, and each time will be a learning experience regarding some new understanding of the technology.

This cookbook is intended to act as a companion guide through this process. The recipes in this book will intimately examine every major aspect of the framework in order to maximize your comprehension. Every time you open this book, you should gain an expanded understanding of the brilliance of the AngularJS framework.

What this book covers

Chapter 1, *Maximizing AngularJS Directives*, dissects the various components of directives and demonstrates how to wield them in your applications. Directives are the bread and butter of AngularJS, and the tools presented in this chapter will maximize your ability to take advantage of their extensibility.

Chapter 2, *Expanding Your Toolkit with Filters and Service Types*, covers two major tools for code abstraction in your application. Filters are an important pipeline between the model and its appearance in the view, and are essential tools for managing data presentation. Services act as broadly applicable houses for dependency-injectable modules and resource access.

Chapter 3, *AngularJS Animations*, offers a collection of recipes that demonstrate various ways to effectively incorporate animations into your application. Additionally, it will dive deep down into the internals of animations in order to give you a complete perspective on how everything really works under the hood.

Chapter 4, *Sculpting and Organizing Your Application*, gives you strategies for controlling the application initialization, organizing your files and modules, and managing your template delivery.

Chapter 5, *Working with the Scope and Model*, breaks open the various components involving ngModel and provides details of the ways in which they can integrate into your application flow.

Chapter 6, *Testing in AngularJS*, gives you all the pieces you need to jump into writing test-driven applications. It demonstrates how to configure a fully operational testing environment, how to organize your test files and modules, and everything involved in creating a suite of unit and E2E tests.

Chapter 7, *Screaming Fast AngularJS*, is a response to anyone who has ever complained about AngularJS being slow. The recipes in this chapter give you all the tools you need to tune all aspects of your application's performance and take it from a steam engine to a bullet train.

Chapter 8, *Promises*, breaks apart the asynchronous program flow construct, exposes its internals, then builds it all the way back up to discuss strategies for your application's integration. This chapter also demonstrates how promises can and should integrate into your application's routing and resource access utilities.

Chapter 9, *What's New in AngularJS 1.3*, goes through how your application can integrate the slew of new features and changes that were introduced in the AngularJS 1.3 and the later AngularJS 1.2.x releases.

Chapter 10, *AngularJS Hacks*, is a collection of clever and interesting strategies that you can use to stretch the boundaries of AngularJS's organization and performance.

What you need for this book

Almost every example in this book has been added to JSFiddle, with the links provided in the text. This allows you to merely visit a URL in order to test and modify the code with no setup of any kind, on any major browser and on any major operating system. If you want to replicate an example outside of JSFiddle, all the external content (AngularJS, AngularJS modules, third-party libraries and modules) is served from `https://code.angularjs.org/` and `https://cdnjs.com/`.

Chapter 6, Testing in AngularJS, involves setting up a testing framework, which should be able to be accomplished on any major Unix-based operating system (OS X and, Linux). The test suite is built on top of Grunt, Karma, Selenium, and Protractor; all of these and their dependencies can be installed through npm.

Who this book is for

There are already plenty of introductory resources to guide a green developer into the thick of AngularJS. This cookbook is for developers with at least basic knowledge of JavaScript and AngularJS, and who are looking to expand their perspective on the framework.

The goal of this text is to have you walk away from reading about an AngularJS concept armed with a solid understanding of how it works, insight into the best ways to wield it in real-world applications, and annotated code examples to get you started.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "By cleverly using directives and the `$compile` service, this exact directive functionality is possible."

A block of code is set as follows:

```
(index.html)

<!-- specify root element of application -->
<div ng-app="myApp">
  <!-- register 'my-template.html' with $templateCache -->
  <script type="text/ng-template" id="my-template.html">
    <div ng-repeat="num in [1,2,3,4,5]">{{ num }}</div>
  </script>

  <!-- your custom element -->
  <my-directive></my-directive>
</div>
```

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

```
(app.js)

.directive('iso', function () {
  return {
    scope: {}
  };
});
```

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

```
npm install protractor grunt-protractor-runner --save-dev
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The following directive will display **NW**, **NE**, **SW**, or **SE** depending on where the cursor is relative to it."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1
Maximizing AngularJS Directives

In this chapter, we will cover the following recipes:

- Building a simple element directive
- Working through the directive spectrum
- Manipulating the DOM
- Linking directives
- Interfacing with a directive using isolate scope
- Interaction between nested directives
- Optional nested directive controllers
- Directive scope inheritance
- Directive templating
- Isolate scope
- Directive transclusion
- Recursive directives

Introduction

In this chapter, you will learn how to shape AngularJS directives in order to perform meaningful work in your applications. Directives are perhaps the most flexible and powerful tool available to you in this framework and utilizing them effectively is integral to architecting clean and scalable applications. By the same token, it is very easy to fall prey to directive antipatterns, and in this chapter, you will learn how to use the features of directives appropriately.

Building a simple element directive

One of the most common use cases of directives is to create custom HTML elements that are able to encapsulate their own template and behavior. Directive complexity increases very quickly, so ensuring your understanding of its foundation is essential. This recipe will demonstrate some of the most basic features of directives.

How to do it…

Creating directives in AngularJS is accomplished with a directive definition object. This object, which is returned from the definition function, contains various properties that serve to shape how a directive will act in your application.

You can build a simple custom element directive easily with the following code:

```
(app.js)

// application module definition
angular.module('myApp', [])
.directive('myDirective', function() {
  // return the directive definition object
  return {
    // only match this directive to element tags
    restrict: 'E',
    // insert the template matching 'my-template.html'
    templateUrl: 'my-template.html'
  };
});
```

As you might have guessed, it's bad practice to define your directive template with the `template` property unless it is very small, so this example will skip right to what you will be using in production: `templateUrl` and `$templateCache`. For this recipe, you'll use a relatively simple template, which can be added to `$templateCache` using `ng-template`. An example application will appear as follows:

```
(index.html)

<!-- specify root element of application -->
<div ng-app="myApp">
  <!-- register 'my-template.html' with $templateCache -->
  <script type="text/ng-template" id="my-template.html">
    <div ng-repeat="num in [1,2,3,4,5]">{{ num }}</div>
  </script>

  <!-- your custom element -->
  <my-directive></my-directive>
</div>
```

When AngularJS encounters an instance of a custom directive in the `index.html` template, it will *compile* the directive into HTML that makes sense to the browser, which will look as follows:

```
<div>1</div>
<div>2</div>
<div>3</div>
<div>4</div>
<div>5</div>
```

JSFiddle: `http://jsfiddle.net/msfrisbie/uwpdptLn/`

How it works...

The `restrict: 'E'` statement indicates that your directive will appear as an element. It simply instructs AngularJS to search for an element in the DOM that has the `my-directive` tag.

Especially in the context of directives, you should always think of AngularJS as an HTML compiler. AngularJS traverses the DOM tree of the page to look for directives (among many other things) that it needs to perform an action for. Here, AngularJS looks at the `<my-directive>` element, locates the relevant template in `$templateCache`, and inserts it into the page for the browser to handle. The provided template will be compiled in the same way, so the use of `ng-repeat` and other AngularJS directives is fair game, as demonstrated here.

There's more...

A directive in this fashion, though useful, isn't really what directives are for. It provides a nice jumping-off point and gives you a feel of how it can be used. However, the purpose that your custom directive is serving can be better implemented with the built-in `ng-include` directive, which inserts a template into the designated part of HTML. This is not to say that directives shouldn't ever be used this way, but it's always good practice to not reinvent the wheel. Directives can do much more than template insertion (which you will soon see), and it's best to leave the simple tasks to the tools that AngularJS already provides to you.

Working through the directive spectrum

Directives can be incorporated into HTML in several different ways. Depending on how this incorporation is done, the way the directive will interact with the DOM will change.

How to do it...

All directives are able to define a `link` function, which defines how that particular directive instance will interact with the part of the DOM it is attached to. The `link` functions have three parameters by default: the directive scope (which you will learn more about later), the relevant DOM element, and the element's attributes as key-value pairs.

A directive can exist in a template in four different ways: as an HTML pseudo-element, as an HTML element attribute, as a class, and as a comment.

The element directive

The element directive takes the form of an HTML tag. As with any HTML tag, it can wrap content, have attributes, and live inside other HTML elements.

The directive can be used in a template in the following fashion:

```
(index.html)

<div ng-app="myApp">
  <element-directive some-attr="myvalue">
    <!-- directive's HTML contents -->
  </element-directive>
</div>
```

This will result in the directive template replacing the wrapped contents of the `<element-directive>` tag with the template. This element directive can be defined as follows:

```
(app.js)

angular.module('myApp', [])
.directive('elementDirective', function ($log) {
  return {
    restrict: 'E',
    template: '<p>Ze template!</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html());
      // <p>Ze template!</p>
      $log.log(attrs.someAttr);
      // myvalue
    }
  };
});
```

[JSFiddle: `http://jsfiddle.net/msfrisbie/sajhgjat/`]

Note that for both the tag string and the attribute string, AngularJS will match the CamelCase for `elementDirective` and `someAttr` to their hyphenated `element-directive` and `some-attr` counterparts in the markup.

If you want to replace the `directive` tag entirely with the content instead, the directive will be defined as follows:

```
(index.html)

angular.module('myApp', [])
.directive('elementDirective', function ($log) {
  return {
    restrict: 'E',
    replace: true,
    template: '<p>Ze template!</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html());
      // Ze template!
      $log.log(attrs.someAttr);
      // myvalue
    }
  };
});
```

[JSFiddle: `http://jsfiddle.net/msfrisbie/oLhrm194/`]

This approach will operate in an identical fashion, but the directive's inner HTML will not be wrapped with `<element-directive>` tags in the compiled HTML. Also, note that the logged template is missing its `<p></p>` tags that have become the root directive element as they are the top-level tags inside the template.

The attribute directive

Attribute directives are the most commonly used form of directives, and for good reason. They have the following advantages:

- They can be added to existing HTML as standalone attributes, which is especially convenient if the directive's purpose doesn't require you to break up an existing template into fragments

- It is possible to add an unlimited amount of attribute directives to an HTML element, which is obviously not possible with an element directive
- Attribute directives attached to the same HTML element are able to communicate with each other (refer to the *Interaction between nested directives* recipe)

This directive can be used in a template in the following fashion:

```
(index.html)

<div ng-app="myApp">
  <div attribute-directive="aval"
     some-attr="myvalue">
  </div>
</div>
```

A nonstandard element's attributes need the `data-` prefix to be compliant with the HTML5 specification. That being said, pretty much every modern browser will have no problem if you leave it out.

The attribute directive can be defined as follows:

```
(app.js)

angular.module('myApp', [])
.directive('attributeDirective', function ($log) {
  return {
    // restrict defaults to A
    restrict: 'A',
    template: '<p>An attribute directive</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html());
      // <p>An attribute directive</p>
      $log.log(attrs.attributeDirective);
      // aval
      $log.log(attrs.someAttr);
      // myvalue
    }
  };
});
```

JSFiddle: `http://jsfiddle.net/msfrisbie/y2tsgxjt/`

Other than its form in the HTML template, the attribute directive functions in pretty much the same way as an element directive. It assumes its attribute values from the container element's attributes, including the attribute directive and other directives (whether or not they are assigned a value).

The class directive

Class directives are not altogether that different from attribute directives. They provide the ability to have multiple directive assignments, unrestricted local attribute value access, and local directive communication.

This directive can be used in a template in the following fashion:

```
(index.html)

<div ng-app="myApp">
  <div class="class-directive: cval; normal-class"
       some-attr="myvalue">
  </div>
</div>
```

This attribute directive can be defined as follows:

```
(app.js)

angular.module('myApp', [])
.directive('classDirective', function ($log) {
  return {
    restrict: 'C',
    template: '<p>A class directive</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html());
      // <p>A class directive</p>
      $log.log(el.hasClass('normal-class'));
      // true
      $log.log(attrs.classDirective);
      // cval
      $log.log(attrs.someAttr);
      // myvalue
    }
  };
});
```

JSFiddle: `http://jsfiddle.net/msfrisbie/rt1f4qxx/`

It's possible to reuse class directives and assign CSS styling to them, as AngularJS leaves them alone when compiling the directive. Additionally, a value can be directly applied to the directive class name attribute by passing it in the CSS string.

The comment directive

Comment directives are the runt of the group. You will very infrequently find their use necessary, but it's useful to know that they are available in your application.

This directive can be used in a template in the following fashion:

```
(index.html)

<div ng-app="myApp">
  <!-- directive: comment-directive val1 val2 val3 -->
</div>
```

The comment directive can be defined as follows:

```
(app.js)

angular.module('myApp', [])
.directive('commentDirective', function ($log) {
  return {
    restrict: 'M',
    // without replace: true, the template cannot
    // be inserted into the DOM
    replace: true,
    template: '<p>A comment directive</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html())
      // <p>A comment directive</p>
      $log.log(attrs.commentDirective)
      // 'val1 val2 val3'
    }
  };
});
```

JSFiddle: `http://jsfiddle.net/msfrisbie/thfvx275/`

Formerly, the primary use of comment directives was to handle scenarios where the DOM API made it difficult to create directives with multiple siblings. Since the release of AngularJS 1.2 and the inclusion of `ng-repeat-start` and `ng-repeat-end`, comment directives are considered an inferior solution to this problem, and therefore, they have largely been relegated to obscurity. Nevertheless, they can still be employed effectively.

How it works...

AngularJS actively compiles the template, searching for matches to defined directives. It's possible to chain directive forms together within the same definition. The `mydir` directive with `restrict: 'EACM'` can appear as follows:

```
<mydir></mydir>

<div mydir></div>

<div class="mydir"></dir>

<!-- directive: mydir -->
```

There's more...

The `$log.log()` statements in this recipe should have given you some insight into the extraordinary use that directives can have in your application.

See also

- The *Interaction between nested directives* recipe demonstrates how to allow directives attached to the same element to communicate with each other

Manipulating the DOM

In the previous recipe, you built a directive that didn't care what it was attached to, what it was in, or what was around it. Directives exist for you to program the DOM, and the equivalent of the last recipe is to instantiate a variable. In this recipe, you will actually implement some logic.

How to do it...

The far more common use case of directives is to create them as an HTML element attribute (this is the default behavior for `restrict`). As you can imagine, this allows us to decorate existing material in the DOM, as follows:

```
(app.js)

angular.module('myApp', [])
.directive('counter', function () {
  return {
    restrict: 'A',
    link: function (scope, el, attrs) {
      // read element attribute if it exists
      var incr = parseInt(attrs.incr || 1)
        , val = 0;
      // define callback for vanilla DOM click event
      el.bind('click', function () {
        el.html(val += incr);
      });
    }
  };
});
```

This directive can then be used on a `<button>` element as follows:

```
(index.html)

<div ng-app="myApp">
  <button counter></button>
  <button counter incr="5"></button>
</div>
```

JSFiddle: `http://jsfiddle.net/msfrisbie/knk5znke/`

How it works...

AngularJS includes a subset of jQuery (dubbed jqLite) that lets you use a core toolset to modify the DOM. Here, your directive is attached to a singular element that the directive sees in its linking function as the element parameter. You are able to define your DOM modification logic here, which includes initial element modification and the setup of events.

In this recipe, you are consuming a static attribute value `incr` inside the `link` function as well as invoking several jqLite methods on the element. The element parameter provided to you is already packaged as a jqLite object, so you are free to inspect and modify it at your will. In this example, you are manually increasing the integer value of a counter, the result of which is inserted as text inside the button.

There's more...

Here, it's important to note that you will never need to modify the DOM in your controller, whether it is a directive controller or a general application controller. Because AngularJS and JavaScript are very flexible languages, it's possible to contort them to perform DOM manipulation. However, managing the DOM transformation out of place causes an undesirable dependency between the controller and the DOM (they should be totally decoupled) as well as makes testing more difficult. Thus, a well-formed AngularJS application will never modify the DOM in controllers. Directives are tailor-made to layer and group DOM modification tasks, and you should have no trouble using them as such.

Additionally, it's worth mentioning that the `attrs` object is read-only, and you cannot set attributes through this channel. It's still possible to modify attributes using the element attribute, but state variables for elements can be much more elegantly implemented, which will be discussed in a later recipe.

See also

- In this recipe, you saw the `link` function used for the first time in a fairly rudimentary fashion. The next recipe, *Linking directives*, goes into further detail.
- The *Isolate scope* recipe goes over the writable DOM element attributes that can be used as state variables.

Linking directives

For a large subset of the directives you will eventually build, the bulk of the heavy lifting will be done inside the directive's `link` function. This function is returned from the preceding compile function, and as seen in the previous recipe, it has the ability to manipulate the DOM in and around it.

How to do it...

The following directive will display **NW**, **NE**, **SW**, or **SE** depending on where the cursor is relative to it:

```
angular.module('myApp', [])
.directive('vectorText', function ($document) {
  return {
    template: '<span>{{ heading }}</span>',
    link: function (scope, el, attrs) {

      // initialize the css
      el.css({
        'float': 'left',
        'padding': attrs.buffer+"px"
      });

      // initialize the scope variable
      scope.heading = '';

      // set event listener and handler
      $document.on('mousemove', function (event) {
        // mousemove event does not start $digest,
        // scope.$apply does this manually
        scope.$apply(function () {
          if (event.pageY < 300) {
            scope.heading = 'N';
          } else {
            scope.heading = 'S';
          }
          if (event.pageX < 300) {
            scope.heading += 'W';
          } else {
            scope.heading += 'E';
          }
        });
      });
    }
  };
});
```

This directive will appear in the template as follows:

```
(index.html)

<div ng-app="myApp">
  <div buffer="300"
       vector-text>
  </div>
</div>
```

JSFiddle: `http://jsfiddle.net/msfrisbie/a0ywomq1/`

How it works...

This directive has a lot more to wrap your head around. You can see that it has `$document` injected into it, as you need to define event listeners relevant to this directive all across `$document`. Here, a very simple template is defined, which would preferably be in its own file, but for the sake of simplicity, it is merely incorporated as a string.

This directive first initializes the element with some basic CSS in order to have the relevant anchor point somewhere you can move the cursor around fully. This value is taken from an element attribute in the same fashion it was used in the previous recipe.

Here, our directive is listening to a `$document mousemove` event, with a handler inside wrapped in the `scope.$apply()` wrapper. If you remove this `scope.$apply()` wrapper and test the directive, you will notice that while the handler code does execute, the DOM does not get updated. This is because the event that the application is listening for *does not occur* in the AngularJS context—it is merely a browser DOM event, which AngularJS does not listen for. In order to inform AngularJS that models might have been altered, you must utilize the `scope.$apply()` wrapper to trigger the update of the DOM.

With all of this, your cursor movement should constantly be invoking the event handler, and you should see a real-time description of your cursor's relative cardinal locality.

There's more...

In this directive, we have used the `scope` parameter for the first time. You might be wondering, "Which scope am I using? I haven't declared any specific scope anywhere else in the application." Recall that a directive will inherit a scope unless otherwise specified, and this recipe is no different. If you were to inject `$rootScope` to the directive and log to the `$rootScope.heading` console inside the event handler, you would see that this directive is writing to the `heading` attribute of the `$rootScope` of the entire application!

See also

- The *Isolate scope* recipe goes into further details on directive scope management

Interfacing with a directive using isolate scope

Scopes and their inheritance is something you will frequently be dealing with in AngularJS applications. This is especially true in the context of directives, as they are subject to the scopes they are inserted into and, therefore, require careful management in order to prevent unexpected functionalities. Fortunately, AngularJS directives afford several robust tools that help manage visibility of and interaction with the surrounding scopes.

If a directive is not instructed to provide a new scope for itself, it will inherit the parent scope. In the case that this is not desirable behavior, you will need to create an isolate scope for that directive, and inside that isolate scope, you can define a whitelist of parent scope elements that the directive will need.

Getting ready

For this recipe, assume your directive exists inside the following setup:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <div iso></div>
  </div>
</div>

(app.js)

angular.module('myApp', [])
.controller('MainCtrl', function ($log, $scope) {
  $scope.outerval = 'mydata';
  $scope.func = function () {
    $log.log('invoked!');
  };
})
.directive('iso', function () {
  return {};
});
```