Community Experience Distilled

Mastering AngularJS Directives

Develop, maintain, and test production-ready directives for any AngularJS-based application

Josh Kurz

[PACKT] PUBLISHING open source* community experience distilled

Mastering AngularJS Directives

Develop, maintain, and test production-ready directives for any AngularJS-based application

Josh Kurz

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Mastering AngularJS Directives

First Published: June 2014

Production Reference: 1090614

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

ISBN 978-1-78398-158-8

`www.packtpub.com`

Cover Image by Josh Kurz (`jkurz25@gmail.com`)

Credits

Author
Josh Kurz

Reviewers
Pete Bacon Darwin
Lee Howard
Darius Riggins
Iwan van Staveren
Ruoyu Sun

Commissioning Editor
Kunal Parikh

Acquisition Editor
Subho Gupta

Content Development Editor
Neil Alexander

Technical Editors
Pragnesh Bilimoria
Indrajit A. Das
Shashank Desai

Copy Editors
Dipti Kapadia
Insiya Morbiwala
Aditya Nair

Project Coordinator
Kartik Vedam

Proofreaders
Simran Bhogal
Ameesha Green
Maria Gould
Paul Hindle
Linda Morris

Indexer
Priya Subramani

Graphics
Abhinash Sahu

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Josh Kurz is a client-side technician who constantly pushes the realms of frontend technologies by mixing new-age theories and proven Computer Science concepts. He has successfully shown that AngularJS can be used to create some of the fastest, most usable data visualization applications while working at Turner. He also has a true passion for open source code and believes it is one of the reasons for his success. Currently, outside of work, he is practicing to become a black belt in Jiu Jitsu.

I would like to dedicate this book to the people who helped make this book a reality. Many of these people are part of the AngularJS community and push the bounds of what is conceivable every day. The technical editors of this book all deserve a round of applause, as they have done such a wonderful job. My co-workers at Turner also help raise the bar every day, showing me what it takes to be a professional. I would also like to thank Invidia Studios for the amazing artwork. Last but not least, my wonderful fiancé deserves the most appreciation, as she is so patient and caring and has helped in more ways than I can even begin to express.

About the Reviewers

Pete Bacon Darwin is a freelance programmer who is currently working with the AngularJS team at Google. He has a degree in Math from Cambridge University. He worked for a bunch of consulting companies before giving it all up to look after his kids and do coding in the background.

When he isn't coding or parenting, Pete teaches Aikido and wishes he could find time to do more climbing and mountaineering.

Pete co-authored *Mastering Web Application Development with AngularJS, Packt Publishing*.

Darius Riggins is a veteran full-stack developer who focuses on solving challenging problems with creative solutions.

Ruoyu Sun is a designer and developer living in Hong Kong. He is passionate about programming and has contributed to several open source projects. He founded several tech start-ups using a variety of technology before going into the industry. He is the author of the book *Designing for XOOPS, O'Reilly Media*.

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

Table of Contents

Preface

AngularJS offers a new outlook on web development that is changing more and more opinions everyday. The reason people are agreeing with AngularJS's direction is because of its orthogonal views on encapsulation and separation. Separation of logic into structurally defined realms is AngularJS's specialty, and this truth allows more focus to be placed on domain specific logic.

Directives offer the biggest form of encapsulation inside AngularJS applications. This is true because of its focus on separating the view from the model. For years, developers have combined different types of client-side logic that has no business being coupled together. The decoupling of the view and model has begun to take full effect in modern web applications, and AngularJS directives have been built with this mindset at their core.

Many say that directives are the most difficult piece of AngularJS to learn. People say this because directives take a new approach to JavaScript conventions, which has not been done before. There are not many libraries that focus on declarative approaches to handle the relationship between HTML and JavaScript. These new concepts seem difficult at first glance, but once their logic is understood, things fall into place rather quickly.

Many use cases can be solved with a simple directive or a set of directives that work in unison with each other. We will go over how to create these simple directives and how to train your mind to immediately consider them as solutions. The stages that this book goes through build on top of a singular idea. This idea is how to properly take some data model, and effectively render it and all of its changes in the view.

Once directives are understood for what they are, many great use cases can be accomplished with them. Some of the most important directives are actually built into the core itself, with the same tools available to any application. This book shows us how to use the different options made available by AngularJS to create a wide range of directives that serve many different purposes.

The differences between the directives created in this book are pretty broad. There are stopwatches, stoplights, media players, and stock charts, which can not only work together but also work individually just as well. The only central theme to each is that they are considered black belt directives and, as such, are tested and implemented just like any other production-ready software.

What this book covers

Chapter 1, *The Tools of the Trade*, introduces us to what a directive is, how it is created, and the different options that can be used to create them. Its main purpose is to introduce directives from a high-level perspective so that anyone can digest their meaning. To do this, the chapter is broken down into different parts that consist of basic examples showcasing the use of the different options.

Chapter 2, *Building a Stopwatch Directive*, introduces us to the first directive that we build in this book. The stopwatch building process goes through iterations that shed light on different design aspects. Throughout the design process, the directive is tested thoroughly to prove that its logic is correct and that any change made to it does not introduce bugs.

Each decision that went into the architecture of the directive is discussed and explained by showing the change, and then going into details about that change. The overall goal of the chapter is to create a useful directive that can be used in many different applications and to get ideas stirring about your own custom directives.

Chapter 3, *Harnessing External JavaScript Libraries with Directives*, discusses how many applications rely on third-party libraries to accomplish advanced DOM manipulation. These libraries can be integrated smoothly with any AngularJS application and can still abide by the concepts made by the majority in the community. The purpose of this chapter is to showcase best practices when integrating third-party libraries into AngularJS applications.

Chapter 4, *Compiling the Advantages*, shows you how being able to utilize AngularJS's compile cycle at will is useful in many different instances. There are few use cases that require the use of the $compile service, and these are discussed in detail. This chapter also showcases how useful it can be to generate DOM attached to AngularJS's scope in conjunction with third-party libraries and dynamic templates.

Chapter 5, *Communication between Directives*, shows that directives are very useful in normal circumstances. They are even more useful when they work in unison with each other to accomplish similar tasks. There are many ways to get directives to work together. Some ways include basic scope inheritance, and others include sharing portions of execution context.

This chapter takes a deep dive into the many possible ways to get directives to work with each other. No matter what their relationship, there is always a way to get two directives to work in collaboration. The examples here also help showcase how to write integration tests in order to prove the logic being used to integrate works as desired.

Chapter 6, *Working with Live Data*, shows that data is what makes applications important. If it were not for the data, then there would be no reason to push the Web forward. This chapter showcases the philosophies behind developing directives and their approach to working with live data.

Since the data is coming from a live source, we have kept scale in mind throughout the design of all of these examples. The scale considerations bring a large focus on different ways to write directives that deal with large amounts of data.

Chapter 7, *Optimization and Code Quality*, shows the importance of making sure that an application is fast and how the ability to stay agile is detrimental to its lifespan. AngularJS gives us many opportunities to write clean, fast code that does amazing things. However, with all great things comes great responsibility.

AngularJS can be used in inefficient ways that can drastically slow down a web page. This chapter showcases some things to watch out for when writing directives. Since directives are the main reason to create massive amounts of bindings, we go over how to keep the total number of bindings to a minimum. This chapter also goes over the benefits that AngularJS brings in terms of quality code and what this means for organized HTML views.

Chapter 8, *Directives and Animations*, shows why directives play an important role in how animations are integrated. This is because AngularJS animations have been built in a way to create another encapsulation layer that plays directly by working alongside directives. This chapter shows us how to use the animation service in core directives and how to create custom directives that use animations.

Chapter 9, *Conclusion*, wraps up the book with an overall summary of closing statements. There are references made to all the sections of the book, and each is given an overview. The overall goal is to finalize the ideas and concepts that have been portrayed.

What you need for this book

This book has been written to work with the 1.2.x branch. The directives that have been created will work with future versions of AngularJS, but there may be slight changes that need to occur in these future versions.

The examples that have been built are being showcased at `http://angulardirectives.joshkurz.net/` and in the GitHub repo, `https://github.com/joshkurz/Black-Belt-AngularJS-Directives`. The instructions to install the project can be located in the README file on the GitHub repo and here.

The following are the normal requirements for today's project standards:

- npm install -g grunt-cli http-server
- npm install
- bower install
- grunt
- *grunt protractor

All of the necessary modules are installed via npm. Grunt is used in this project to create the build file, which is tested against and used in `angulardirectives.joshkurz.net`. Protractor is used for all E2E tests in this book. To run Protractor tests, a web server needs be hosting the files on localhost. It is recommended to use the npm module http-server. Once you have these packages installed and all of the tests pass, you can play around with whatever you want inside the repo. Please send a PR if you would like to contribute to the project as well.

Who this book is for

If you are a developer who has previous JavaScript experience and some AngularJS experience, this is the book for you. New AngularJS users will be able to follow the concepts of this book, but there may be some references to AngularJS-specific material that is not fully defined in the book.

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, directive names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"The `ngInclude` directive creates an opportunity to write clean organized views."

A block of code is set as follows:

```
$routeProvider.when('/mediaelement', {
  templateUrl:'directives/demo/mediaelement/
   mediaelementView.tpl.html',
  reloadOnSearch: false,
  controller:'mediaelementCtrl'
 });
```

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

```
$routeProvider.when('/mediaelement', {
 templateUrl:'directives/demo/mediaelement/
  mediaelementView.tpl.html',
 reloadOnSearch: false,
 controller:'mediaelementCtrl'
});
```

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

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you. You can also download the code from GitHub at any time by going to `https://github.com/joshkurz/Black-Belt-AngularJS-Directives`.

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
The Tools of the Trade

The leading edge of web development moves quicker than most mortal humans can keep up with. There are so many different techniques to learn and harness that it can sometimes seem overwhelming. Thankfully, there is a wonderful JavaScript framework called AngularJS that helps us mere mortals become something greater.

AngularJS offers many different facets of technology that can be used to accomplish different tasks efficiently and effectively. There is no specific implementation that AngularJS is built from that is more powerful than a directive. A directive may be defined as an official or authoritative instruction. This is a modern nontechnical term for a directive. In AngularJS, directives still follow this definition; however, a more technical description could be a set of instructions, the ultimate goal of which is to read or write HTML.

Directives can be used to solve many different use cases related to **Document Object Model** (**DOM**). Directives allow developers to create new HTML elements that can do almost anything inside AngularJS. Teaching the browser new functionality to provide DOM manipulation, creation, and event detection is a new idea that is just becoming popular.

AngularJS takes a different approach to how JavaScript and HTML5 work in unison with each other. There is no longer a need to constantly traverse the DOM tree for every bit of functionality that needs to be created. Directives allow for a declarative approach, which separates DOM manipulation logic and business logic. This separation means that our new applications are more readable, testable, and perform better.

Introduction to directives

When first learning about AngularJS, directives can create a magical illusion that hides the logic associated with the view's actions. This hidden logic is by design and is the reason AngularJS is so popular. The gory details of every directive are not important to some developers, but these details are the lifeblood of custom directives.

A directive is essentially just a JavaScript factory function that is defined inside of a given AngularJS module. The function returns an object that holds a set of instructions for the AngularJS HTML compiler. This object can either be a function that is run once the element is linked to the scope (link function) or a JSON representation of more advanced instructions that ultimately should also contain a link function. Returning a JSON instruction representation is referred to as returning a definition object and is the community-preferred method of writing directives.

Definition objects can have a finite number of options, made available by the AngularJS public directive API. Let's break down a simple definition object as follows:

```
var definitionObject = {
   restrict: 'EA',
   link: function(scope, element, attrs){
     element.text('Hello Directive World');
   }
};
```

Downloading the example code

You can download the example code files for all Packt books that 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. The code can also be found at `https://github.com/joshkurz/Black-Belt-AngularJS-Directives/tree/master/chapters`.

This definition object has two properties that instruct the AngularJS compiler to do specific tasks. These instructions state that the directive can only be created on an element or attribute, and once it's found, to set its text to `'Hello World'`. The specific syntax needed to add this directive to the AngularJS compiler is as follows:

```
app.directive('bbHelloWorld', function(){
  return definitionObject;
});
```

> The directive is named `bbHelloWorld` because we are namespacing all of our directives with `bb`. This is a community-preferred method of directive writing because creating reusable code that does not interfere with other applications is the goal.

Now that our directive has been created, we can use it in any HTML template that is part of our `ng` application. To call this directive, we write something like the following lines of code:

```
<bb-hello-world></bb-hello-world>
```

The output will be as follows:

```
Hello Directive World
```

This was an example of the most basic type of directive. There is nothing wrong with creating simple directives that accomplish basic tasks, but AngularJS allows for all levels of directives to be written. The complete Definition Object API should be understood to create more advanced directives.

Directive Definition Object API

The compiler is given instructions from the directive's definition object. These instructions can be integers, strings, booleans, or JavaScript objects. Their purpose is to give the developer many different options of control when initializing and dynamically manipulating DOM according to the given model.

Priority

The `priority` integer is used when multiple directives are set on the same element. AngularJS collects all of the known directives on an element by any of the defined `restrict` properties and runs each directive's compile, prelink, and postlink functions in a given order. The order is specified by `priority`. Lower priority compile and prelink functions are run last; however, the postlink function is the opposite. The default value is zero. Negative values are allowed in case directives need to be compiled after default directives.

Terminal

The `terminal` field is a Boolean whose default value is `false`. The terminal value of a directive applies to lower priority directives defined on the same element and all of their child directives. Setting a `terminal` field to `true` states that the applicable directives will not compile during the initial directive collection. The initial collection can be run in AngularJS during the initial bootstrap or a manual `$compile` function call.

An HTML example looks like the following:

```
<div directive-one directive-two directive-three>
  <directive-four></directive-four>
</div>
```

The following code snippet shows how the directive definition of a terminal directive looks:

```
app.directive('directiveTwo', function() {
    return {
        priority: 10,
        terminal: true,
        link: function(scope,element,attrs) {
            //link logic
        }
    }
});
```

If `directiveOne` and `directiveThree` have a priority lower than 10, then they will not run on this given `div` element. The `directiveFour` directive will not run because it is a child of the terminal directive. Just because these elements have not been run during the initial collection phase does not mean that they cannot be run at a later time, depending upon other directive definition objects that could have been set.

The `terminal` option is used in the core library by a few different directives. The most notable are `ngRepeat`, `ngIf`, and `ngInclude`.

Scope

A `scope` is a JavaScript object that is created by AngularJS during the initial bootstrap. This initial scope is referred to as the `$rootScope` directive and can be created in two different ways. The `ngApp` directive or `angular.bootstrap` can be called on an element. Either way, the result is the same.

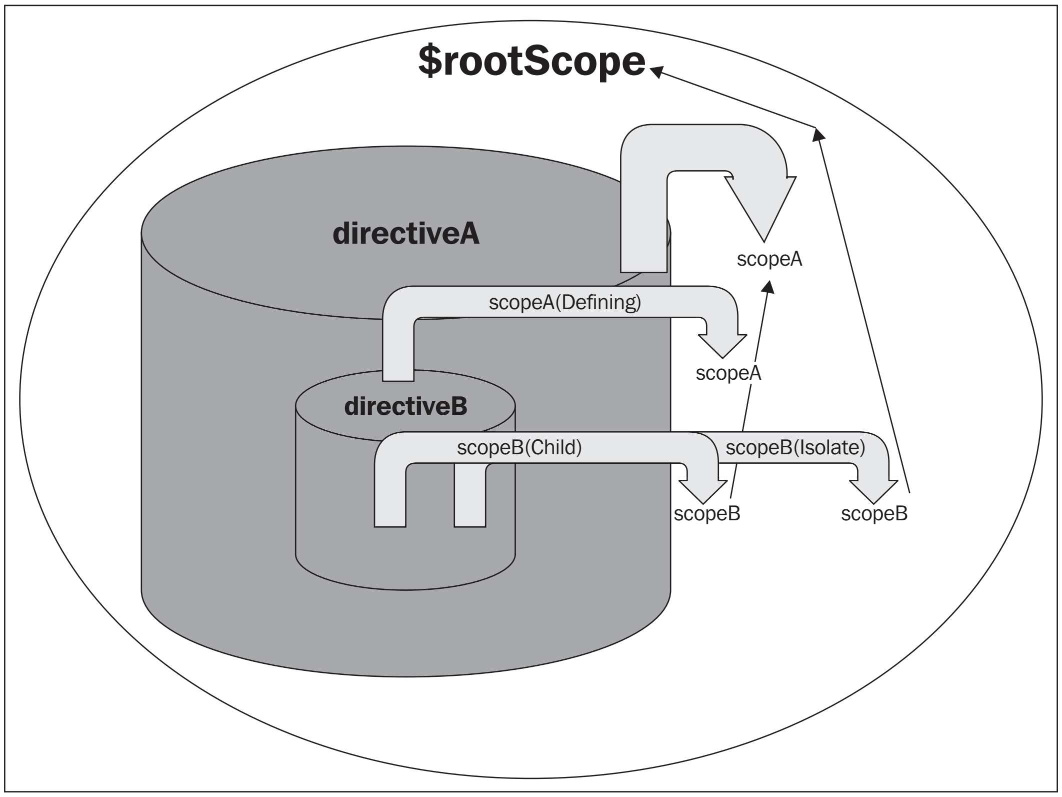
Once the application has been bootstrapped and a `$rootScope` directive has been created, subsequent child directives are in charge of creating new types of scopes. Together, these new scopes create a hierarchy of communication channels. There are specific rules for how each channel is allowed to communicate with each other.

The following two different types of scopes denote the communication rules that are observed by the scope hierarchy:

- The child scope
- The isolate scope

These two scope objects have different types of communication abilities. Child scopes are created using normal JavaScript prototypal inheritance, which means they inherit their parents' attributes but have their own context. Isolate scopes create a separate context that only has a root scope as a commonality between itself and its defining scope.

The scope is what drives a directive's ability to keep its view in sync with its associated data models. The scope can hold any number of data models and can either watch their values for changes or just read from them. There are three different options available when creating new scopes via a directive definition object: defining scopes, child scopes, and isolate scopes. Let's go over an example of a simple app and its scope hierarchy in the following diagram:



There are two directives present in the demo application represented by the preceding diagram. Each cylinder is a directive that either creates its own scope or uses its defining scope. For example, the **directiveA** cylinder creates **scopeA** and **directiveB** creates either **scopeB** or uses its defining scope, that is, **scopeA**. Let's say that **directiveA** creates a child scope called **scopeA**. Depending upon the scope definition of **directiveB**, three options are available for the scope it creates. The **directiveB** cylinder could just use its defining scope. A new child scope could be made, which would create prototypal inheritance from **scopeA**. Lastly, an isolate scope could be created, which would prevent **directiveA** from accidentally reading or writing to **scopeA**.

There is a specific syntax that is used to create the following three possible options for the scope of a directive:

- **False/default**: This is to use the defining/containing scope as the directives own scope
- **True**: This is to create a new scope, which prototypically inherits from the defining/containing scope
- **Object hash**: This is to create an isolate scope (the hash defines the details of the scope)

Two directives that do not create new scopes are `ngShow` and `ngHide`. These directives perform tasks declared on the element's attributes depending upon the defined scope. Since the directive is not actually ever going to change the model itself, it is safe for it to exist on the defining scope. Directives that alter the model in some way should either be given a child scope or an isolate scope.

Child scopes are useful when creating directives that live around other directives of their kind. This is because child scopes are not accessible to sibling elements; so by writing to a child scope, the directive is ensuring that none of its siblings will have any of their data overwritten. Child scopes create a prototypal inheritance model. There are some nuances to dealing with child scopes that are covered in detail in *Chapter 5*, *Communication between Directives*.

Isolate scopes are probably the most widely used scope option in directives in the open source community. This is because of the different options that are available with it. These options allow for a very unique and special functionality. There are three options available when defining an isolate scope variable. These options are the values of the isolate scopes definitions. Isolate scopes allow for special types of data binding.

> Isolate scopes are defined by three available values that perform different types of data binding.

The following code snippets shows scope variables:

```
   // String representation of a defining scope's variables
Javascript: Scope: {'name': '@'}
HTML: <div bb-stop-watch name="{{localName}}"></div>

// An expression executed on the defining scope
Javascript: Scope: {'name' : '&'}
HTML: <div bb-stop-watch name="newName = localName + ' ha ha'"></div>

   // Two Way Data Binding
   JavaScript: Scope: {'name': '='}
HTML: <div bb-stop-watch name="localName"></div>
```

All of these defined local scope variables are called `scope.name`, which are passed by a directive attribute called `name` as well. The attribute could be called something other than `name`, with a syntax that specifies the attributes names. The following code snippet is the same example, just with different attribute's names that define the value:

```
   // String representation of a defining scope's variables
JavaScript: Scope: {'name': '@theName'}
HTML: <div bb-stop-watch the-name="{{localName}}"></div>

// An expression executed on the defining scope
JavaScript: Scope: {'name' : '&theName'}
HTML: <div bb-stop-watch the-name=
  "newName = localName + ' ha ha'"></div>

   // Two Way Data Binding
   JavaScript: Scope: {'name': '=theName'}
HTML: <div bb-stop-watch the-name="localName"></div>
```

Each of these variables performs specific tasks as denoted by the comments above the scope definitions in the preceding code snippet. The first represents a string that is interpolated from the defining scope. The second represents an expression that returns some variable to be set as the local scope attribute definition. The expression used in the preceding example is simple, but these can be defined on any scope and can be as complex as needed.

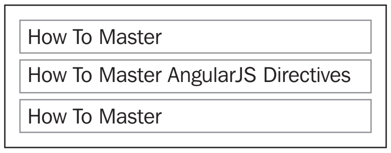
> Expressions that are set inside of curly brackets in an AngularJS application will be evaluated on every digest. This means that the developer should be careful to not set expensive expressions that could cause the digest cycle to take longer than 50 ms.

The third variable is used to create two-way data binding between the parent and its isolate scopes. This means that a watch is set on the variable automatically, and once one of the variables changes, all of the variables that reference each other will change as well. This is one of the most common AngularJS tactics and also one of the most spectacular.

The following is another example of three different input directives that utilize each one of the methods:

```
  */ HTML Templates */
<div bb-string term="{{term}}"></div>
<div bb-expression term="theTerm = term + ' AngularJS Directives'"></
div>
   <div bb-two-way term="term"></div>


*/ Demo Javascript Module */
angular.module('demoApp', [])
.controller('demoCtrl', function($scope){
    $scope.term = 'How To Master';
})
.directive('bbString', function(){
    return {
        scope: { term: '@'},
        template: '<input ng-model="term">'
    }
})
.directive('bbExpression', function(){
    return {
        scope: { term: '&'},
        template: '<input ng-model="term">',
        link: function(scope, element, attrs){
            scope.term = scope.term();
        }
    }
})
.directive('bbTwoWay', function(){
    return {
        scope: { term: '='},
        template: '<input ng-model="term">'
    }
});
```

Take a look at the following screenshot that shows the differences between the different directives:


How To Master

How To Master AngularJS Directives

How To Master

Typing into the first or second directives will not alter the parent `$scope.term` variable, but typing into the third input will alter the first directive's model and its parent directives. A live example is available at `http://jsfiddle.net/joshkurz/x22y2/`.

There is a fourth type of scope that can be used inside of a directive. This scope is only created when the directive is utilizing transclusion. The scope created is a child of the definition scope, and when inserted into the directive element, it becomes a sibling of the directive scope, whatever it may be.

Controller

The controller definition option creates mostly all the controllers created in an AngularJS application. This is not a very common insight, but it is true. Even the AngulaJS routing functions that associate a controller with a specific URL, wrap their associated template with `ngController` and feed the data used in the route definition of this directive.

A controller is a constructor that creates a new context (that is… `this`) that can define its own variables and functions every time its own constructor function is called. This constructor function can be called in various ways. Some common ways are to have a controller associated with a route, using an `ngController` directive directly, or create a custom directive that properly uses the named `controller` option to initialize a new instance of a controller.

The `controller` option is a string or inline function. If the value is a string, then it maps to a controller constructor function set on a module that the directive is tied too. Controllers and directives work together to keep the view aligned with the model.

The main purpose of having directives utilize their own private controllers is for interdirective communication. This is most commonly a requirement with reusable directives that work independently of each other, but depend on shared resources that instruct state changes. A directive can require any amount of controllers necessary to perform its tasks, whatever they may be.

A qualifier for a `controller` option is a set of directives that will always have a parent-child relationship and have the need to communicate with each other. The parent directive should have the main controller defined on its definition object. The child directive should require this controller.

Let's build a simple `bbStopLight` directive that is broken down by two individual directives that share a parent-child relationship. The parent `bbStopLightContainer` directive is used to contain all of the child `bbStopLight` directives. The `bbStopLightContainer` directive is in control of which `bbStopLight` will be active. This information needs to be communicated to all the associated directives.

For the following example, we only show the parent `bbStopLightContainer` directive; the child `bbStopLight` directive will be discussed at a later time:

```
// The container for the stopLight's
angular.module('TrafficLight')
 .directive('bbStopLightContainer', [ function() {
      return {
        controller: 'bbStopLightCtrl',
        scope: {options: '='}
       };
      }])
.controller('bbStopLightCtrl', function($scope,$interval){



          // setting options to the function's context
          this.options = $scope.options;

          this.setNextState = function(){
          state = $scope.options.state;
          // setting next state logic
        };
```

```
            $interval(this.setNextState,this.options.interval);
        });
```

If the value of the controller value defined on the definition object is an @ sign, then the controller name will be the directive's name, for example, `stop-light="stopLightCtrl"`. This is how `ngController` works.

Directive controllers are meant to create a medium for communication. This medium is used by the directive controllers to pass objects between related directives. Some nonintuitive behaviors of the directive's controllers are that the only objects accessible in the actual directives via the require field are defined on the function's context. This means that setting functions and attributes on the controller's scope object will not allow for its objects to be shared in directives that require it. The controller's scope is specific to the controller itself and is not shared between directives.

The `bbStopLightContainer` directive can always refer to what its current state is. Now that we have a parent container directive that will control the information for all our stoplights, it's time to create a `bbStopLight` directive and show how it works in collaboration with this new controller.

Require

The `require` field is a string or an array of strings. Each string represents a directive that provides a controller. The `require` field is essentially a key that helps AngularJS map the controller we want to pass into the link function. This is done during the pre and post stages of the link functions, which means the controllers are available in both as the fourth parameter. The controller that is passed into a directive during the link function is a representation of either a controller on a parent directive or the current directive. This is why directives that use `require` must have some type of relationship with the directive that they require a controller from.

The following four different options are available when initializing the `require` field on a definition object:

- `require: 'directiveName'` – This option searches for a directive on the current element; if not found, it throws an error
- `require: '?directiveName'` – If a directive is not present on an element, this option passes a null value to the link function
- `require: '^directiveName'` – This option searches for a directive on an element's parent directives, and if not found, throws an error
- `require: '^?directiveName'` – This option searches the parent directives for the directive and passes a null value to the link function if not found

Once the controller is passed to the directive, it can be read from, or written to, and this is true for any other directive that requires that controller. This gives direct access to the directives that require it and to other directives that also require the same instance of this controller. Remember that the shared controller is an instance of the controller that is created on the directive that calls the controller constructor. This opens up many options for child directives that perform specific tasks, depending on the state of the parent directive and its controller.

Let's allow the `bbStoplight` directive to communicate with its parent controller (`bbStopLightContainer`) by adding the `require` field to its definition object with the help of the following code snippet:

```
    // now bbStopLight requires the bbStopLightContainers Controller
angular.module('TrafficLight')
  .directive('bbStopLight', function(svgService) {
    return {
        require: '^bbStopLightContainer',
        scope: {},
        link: function(scope,element,attrs,stopLightCtrl) {
            // the logic that determines what to do with the
            // linked stop light element
        }
    };
});
```

Now, we can access the `bbStopLightCtrl` directive's objects that are publicly exposed, which means we can set our own options for each specific stoplight and turn it on or off. To do this, we could observe the values for a change on the `bbStopLightCtrl` directive and update the `bbStopLight` directive accordingly.

ControllerAs

The `controllerAs` field is a string that represents an alternative way to reference the directive's controller from the template. Once the `controllerAs` field has been set to a name, the function context (this) turns into the string representation of the `controllerAs` value.

It is the same as using `$scope.controllerAsValue =` this and being able to reference it inside of the HTML or template.

Restrict

The `Restrict` field is a character value that represents how the directive is defined in the DOM. AngularJS has a built-in compile function that is run on initialization and at the developer's choice. The `$compile` service is its public access point. This compile function collects directives and parses the DOM tree recursively, looking for directives by matching each element's `nodeType` to the list of directives attached to the defined app's modules. There are four different types of representations a directive can have, and all of them are denoted by the `Restrict` option listed as follows:

- `Restrict: 'A'` – This option represents the directive that is an attribute of the element (default), which implies that Angular is looking for `<div my-directive></div >`
- `Restrict: 'E'` – This option represents the directive that is an element, which implies that Angular is looking for `<my-directive></my-directive>`
- `Restrict: 'C'` – This option represents the directive that is a class definition, which implies that Angular is looking for `<div class="my-directive"></div>`
- `Restrict: 'M'` – This option represents the directive that is a comment, which implies that Angular is looking for `<!-- directive: my-directive attrs-->`

Any of the options of the restrictions mentioned in the preceding list can be combined together to create a directive that can have multiple options. Cross-browser compatibility is the biggest reason to go with the A restriction most of the time. IE 8 and 9 require special shivs to work with element directives, and there are some issues that IE has with reading comment directives as well. Class directives work across browsers, and when used properly, can be useful. The comment directive can have special use cases, such as when compiling a directive and creating a compiled comment directive, which are not seen by the user.

Template

The `template` option is a string or a function that returns a string. The string represents the HTML that the directives inject into the DOM once fully compiled and linked. Templates are very useful and help keep HTML source files clean and readable. Directives can be created inside of other directive templates; they allow for nested dynamic directive creation possibilities.

Templates have access to the directive's scope during the linking phase of a directive. This allows for any scope variable or function to be added to the template and utilized in the same fashion as any other HTML markup inside of an AngularJS application. The only stipulation is that the developer has to make sure that the objects being used in the template are actually available in the directive's scope.

Being able to determine what will be active on the directive's scope during runtime depends upon how the scope is defined in the definition object and where it lives in the DOM tree.

When the `template` value is a function, the two parameters available during runtime are `tElem` and `tAttrs`, which are the element and attributes that the directive has access to during the compile phase. However, the values of the attributes are pre-interpolated, so they must be hardcoded values in HTML so that significant value can be derived from them. This allows the developer to request dynamic templates with the `$http` service depending on the attributes set on the element and the values that are taken from the app's current state.

Inside the `template` function, any variable that has been Dependency Injected into the directive's functional context will be available and can be utilized to determine conditions.

The following is an example of an `Animated Menu` directive that utilizes the `template` function just to showcase its syntax:

```
angular.module('Menus', [])
  .directive('bbAnimatedMenu', [function(){
    return{
      restrict: 'EA',
      replace: true,
      template: function(tElem, tAttrs){
        return '<div class="animated-menu animated-menu-
          vertical animated-menu-left">' +
          '{{hello}}'  +
    '</div>';
        },
        link: function(scope, elem, attrs){
          scope.showMenu = function() {};
        }
     };
}])

.controller('menuCtrl', function($scope){
    $scope.hello = 'Hello';
    $scope.hello2 = ' World';
})

// This directive is Called in HTML with this syntax
<div bb-animated-menu test="{{hello}}"><div>
```

Inside the `template` function, the `tElem` and `tAttrs` objects are exactly equal to what they are in the static DOM. This is because the `template` function is run before the compile phase even happens, and there has been no interpolation on the `hello` attribute. The final result in the menu will be a `div` parameter, with the correlating CSS classes and some text that reads `Hello World`. An example of this function is available at `http://jsfiddle.net/joshkurz/qJfa4/3/`.

It is apparent that this function could be cleaned up to some extent, and that is exactly what the `templateUrl` field is for.

TemplateUrl

The `TemplateUrl` option is a string or a function that returns a string, which maps to a template located in `templateCache` or needs to be requested via HTTP. This is the field that should be utilized instead of the template when writing directives in production. Utilizing the `templateUrl` field allows directives to be much more readable, and the same goes for the templates themselves.

Once the template is fetched, the exact same rules apply as the `template` field. The `templateUrl` function utilizes the same `tElem` and `tAttrs` objects as the template option. The benefit to using `templateUrl` is the added readability.

Using `templateUrl` as a function is very important and allows for directives to be able to use dynamic templates based off of attribute values. This is a very declarative approach to programming and writing directives. The benefits are grand and allow for a flexible directive that can be used in many different ways.

Replace

The `replace` field is a Boolean that has a default value of `false`. If `replace` is set to `true`, then the element will replace the defining element in the DOM during the compile phase. This is useful when the defining directive is present just for syntactical purposes and the template holds all of the real DOM that the view should hold. If `replace` is `false`, then the template element will just be appended as a child of the defining element. `false` is the default value of `replace`.

Transclude

The `Transclude` option is a Boolean or a string that has a default value of false. If `transclude` is set to `true`, then AngularJS will copy the element's child elements from the DOM before compiling the template to store a link function for later use. If `transclude` is set to 'element', then AngularJS will compile the entire element and store its link function for later use.

There are a few other directive options that work well with transclusion. Earlier, we alluded to a terminal being useful when transclusion is used by directives. This is because any directive that uses the `terminal` option would not allow its children or other directives with a lower priority to be compiled. The `transclude` function forces them to be compiled, but their link functions are used at a specified time rather than upon directive initialization. `Transclude` also works with the `replace` option in a similar manner. If the defining directive uses `replace`, then its original contents will be lost without transclusion.

The transclude process takes place during the directive's compile phase. When compiling a node, if a transclusion option has been set, then either that node's specified elements or its child elements will be passed into another compile function call. This compile function returns a separate link function that is passed into the directives link function as the fifth parameter. The link function is prebound to a new transclusion scope, which is a child scope of the defining scope. Even though the `transclude` function is prebound to the correct scope, it can be overwritten when the `transclude` function is called. A normal use case of passing a different scope is to create a new child scope from the directive's scope and pass that new scope in.

Let's add some transclusion to the `animated-menu` directive and see how we can access its values, as shown in the following code snippet:

```
app.directive('bbAnimatedMenu', function(){
  return{
    restrict: 'EA',
    replace: true,
    transclude: true,
    templateUrl: 'animatedMenu.tpl.html',
    link: function(scope, elem, attrs, nullCtrl, transcludeFn){
      //setting a variable that represents the cloned element,
      //which is where the original contents of this element were
      //before the compile function ran and generated the new
      //templated element.
      var clonedElement = transcludeFn(function(clone){
        return clone;
      });
      elem.append(clonedElement);
      scope.showMenu = function() {};
    }
  };
});
```

Now that we are transcluding the menu, it can semantically create content that was relevant to the defining scope. This allows for the fulfillment of requirements that call for generating the DOM based upon `$parent` level scope objects that are not accessible inside of the directive through suggested means. This also keeps original bindings intact through prototypal inheritance. Transcluding this way is also great for readability, and it keeps the templates limited to content related to specific directives. A live demo is available at `http://jsfiddle.net/joshkurz/qJfa4/4/`.

There is also a directive called `ngTransclude` that allows the developer to insert the transcluded DOM into a specified place in the template of a directive. The `ngTransclude` directive is useful when a simple clone of the original contents is needed inside a new templated element.

A simple example of `ng-transclude` being used on a directive is shown in the following code snippet:

```
angular.module('DemoExamples').directive
  ('demoTransclusion', function(){
    return{
        restrict: 'EA',
              transclude: true,
        template: '<div class="container">' +
                  '<div ng-transclude><div>' +
                  '</div>',
        // simplified for readability
    };
}]);
```

Now, the compiled and linked DOM will be inserted into the div element where the `ng-transclude` attribute is set. This is great for clean, easy cloning when there are no additional changes that need to be made to the original contents.

Compile

The `Compile` option is a function that returns either an object or a single post-link function. If the returned item is an object, then it can contain two fields, which are either pre or post. The purpose of the compile function is to offer optimization techniques that can be run before the directive's DOM is linked to a scope. The `compile` function does not have access to any scope. This means that any attribute that is read during the `compile` phase will be the pre-interpolated value.

The life cycle of the `compile` function depends on its definition, but the order in which it is called depends on the priority set by the definition object. If a directive has a higher priority, then its `compile` function will be run before those that have a lower compile value and that are set on the same element.

This directive option is not to be confused with the `$compile` service that AngularJS offers in its API as a Dependency Injected variable. The difference is that this `compile` function does not traverse the given element to register any associated directives. The `compile` option's main purpose it to return a variable link function that defines how the directive should be linked to the DOM and scope.

The compile function is the basis of all core and custom directives. Normally, when creating directives, there is no need to utilize this field because its returned value is exactly the same as the link function would be.

An example compile function for the animated menu is shown in the following code snippet:

```
compile: function(tElem, tAttrs){

          return function(scope, elem, attrs, nullCtrl, transcludeFn){
            var clonedElement = transcludeFn(function(clone){
              return clone;
            });
            elem.append(clonedElement);
            scope.showMenu = function() {};
          }

}
```

The `tElem` and `tAttrs` objects are template values that are preinterpolated. Only template alterations should be made inside the context of the compile functions. DOM manipulation, event registration, and compiling should be done in the returned link function.

The `compile` function returns the exact same link function that the animated menu used in the previous `transclude` example. The difference is that now the link function has access to the templated attributes and elements. This opens up many options for different use cases, such as directives with variable link functions, because the template or the application state could determine which link function should be returned.

Another example of the compile function utilizing the pre and post objects is shown in the following code snippet:

```
compile: function(tElem, tAttrs){

    return {
        pre: function(scope, elem, attrs, controller, transcludeFn){
          scope.showMenu = function() {
            elem.toggleClass('animated-menu-push-toright' );
          };
        },
        post: function(scope,elem,attrs,controller,transcludeFn){
         var clonedElement = transcludeFn(function(clone){
            return clone;
          });
          elem.append(clonedElement);

          scope.showMenu();
        }
    }
}
```

The pre and post object fields, used in this version of the animated menu's `compile` function, both return link functions. The difference is that the pre function will always be run before any post link function that is defined on the `animatedMenu` directive. It is also not safe to perform DOM manipulation in the pre link function because the post link function will fail to locate specific elements.

Another important factor about the `compile` function is how it performs some of its optimization techniques. Any directive that uses `transclude`, such as `ngRepeat`, will compile directives only once. The `link` function will be run many times, which proves that placing logic in the `compile` phase, that does not need scope interaction, is faster.

Link

The `link` definition option can either be a function or an object. If the link definition is a function, then it is considered a postlink function. If the `link` definition is an object, then it can contain pre or post object keys that map to individual link functions that are run in a synchronous order. The post link function is the only place where DOM manipulation, which depends on scope variables, should occur in an AngularJS application. This is so all of the elements can be precompiled and their scope linked to DOM data. Doing so will ensure that all elements are available at the correct time and that the specific directive logic has been performed as intended.

The link function is synonymous with the compile function's returned object. Most of the examples that we have used in this chapter so far contain `link` functions. This is because most directives only contain link function definitions. The link function is the last function to be called from a directive's definition. If the element contains multiple directives, then the link function is considered a composite link function that holds the data set by all of the other directives on the given element and its child elements.

Directives are collected recursively in AngularJS. Their link functions are run in the opposite order that they are compiled in. AngularJS traverses the DOM, starting at the root, and compiles every directive it finds. Then, as AngularJS comes back up the tree, each post link function is run. The parent directive will only run its post link function once all of the child directives have already done so. If a directive was compiled first, then its DOM and scope will be linked last.

Almost all of a directive's logic will be placed inside of the link function. This is because it is the safest place to perform DOM manipulation as we can ensure that all of the directive's child elements are compiled and linked. It is common to find directives with fat link functions. Although this is not a wonderful technique, it is used in the community often.

> The **`bbStopLight`** directive creates an SVG element based on the width and height of the defining element and watches the parent container for state changes.

Let's take a look at a detailed version of the `bbStopLight` link function in the following code snippet:

```
link: function(scope,element,attrs,stopLightCtrl) {

            var context = element[0].getContext('2d');

            scope.options =  angular.extend({
              attrsState: attrs.state,
              height: element[0].height,
              width: element[0].width
            },stopLightCtrl.options);

            function getStopLightState(){
              return stopLightCtrl.options.state;
            }

            svgService.setUpStopLight(context,scope.options);
            scope.$watch(getStopLightState, function(newV,oldV){
              if(newV !== oldV){
                svgService.changeColor(context,scope.options.
attrsState,newV);
              }
            });
}
```

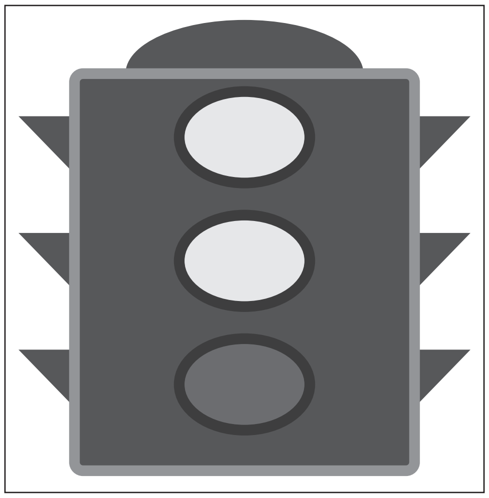
The `link` function for the `bbStopLight` directive consists of a couple of different functions. First, a context variable is set as a canvas element. This will be used as a parameter in the coming functions, so we can perform all of the SVG-related tasks we need to. Note that the specific isolated scope has an object value set to it, which are the options for the parent container and some extended information that is specific to an instance of the `bbStopLight` directive.

Once we have the context and the options, we can initialize the SVG circle that represents the `bbStopLight` directive. The `svgService` has been injected to the context of this directive so that we can call its specific SVG functions. This is so we can keep the cyclomatic complexity down inside of the link function.

Keeping the cyclomatic complexity low in `link` functions is especially important and easily achievable by utilizing services and factories. This also opens up more room to test individual functions that may not have been testable if left as private functions in the `link` function's context.

Now that we have our `bbStopLight` directive set, we need to set its color. The color should be `#ccc` or the color of the current state of the container only if it matches the isolated `bbStopLight`. This is done by setting up a watch on the scope and calling `getStopLightState` on every digest. During the first digest, this function will run because the comparison of `newV` and `oldV` will be true because `oldV` will equal undefined on the first digest. Once `svgService.changeColor` runs, our `bbStopLight` directive will be the color that it is supposed to be. Then, every time the interval fires in the controller of `bbStopLigthContainer`, this watch will be fired and the `svgService.changeColor` method will be run, subsequently changing the color of `bbStopLight` to its correct state.

The following output shows the change in the color of the traffic light:



The `link` functions are where the bulk of the logic for a specific directive gets located. The reason for this is that in the post link function, all of the compiled information about a directive and its child elements is known.

Wrapping up definition objects

The definition object is a set of instructions used when AngularJS compiles and links the DOM against a specified scope. The whole purpose of a definition object is to separate out the logic behind building DOM structures that live in the AngularJS digest cycle. Once completely linked, these DOM structures offer a versatile and dynamic solution to manipulating the view.

Each individual definition field has a distinct purpose and subsequently should be used if the requirements for the given directive call for it. This is not a commonly used set of definition object fields. The use of each field depends widely on individual implementation. Because of this, it is good to know when and where to use each definition field and how to use the fields in correlation with each other.

Summary

We have gone over every definition object option available in detail and the method to utilize these on directives for different purposes. We have successfully created a traffic light that consists of two directives, a directive controller, and a service. We also created an animated menu that utilizes different aspects of the `compile` definition field to create its `link` function that is needed to accomplish its tasks.

Another focus of this chapter, was on how and when to use individual definition fields according to specific requirements for individual directives. In every case, the definition fields needed can vary, but there are some very similar concepts that can be reused through almost every directive.

DOM manipulation, which depends on scope variables or child elements, should only be done inside the post-linking function returned by the compile definition. This is to ensure that all of the directive's elements and child elements are compiled, evaluated, and interpolated against the given directive's scope and transcluded scope.

The templates used in a directive can be very complex and should always be put into `templateUrl` for the best readability possible. The readability is advantageous for both the directive and the template. Also, one of the major advantages of using a `templateUrl` option is that we can create a declarative directive that uses dynamic templates.

Transclusion should be performed when the contents of a DOM element need to be cloned and attached to the newly linked element. It is possible to transclude just the directive's child elements or the entire directive itself. Transclusion works great with other options such as `terminal` or `replace`.

The `scope` field is very important and should be used correctly. There are three different types of scopes that a directive can have. These scopes have specific purposes and their own set of benefits. The most robust scope is the isolate scope option that allows for the directive to use two-way data.

These directive definition object options offer different ways of accomplishing simple to very advanced tasks. The key to controlling how and when to use these options correctly lays in knowing them in detail and practicing their utilization.

In the next chapter, we will build our first directive. This directive will utilize many of the options we have described in this chapter. We will also go over some different types of testing techniques.

2
Building a Stopwatch Directive

The stopwatch is a full-fledged example of how to build an AngularJS directive from start to finish. This chapter will cover the correct way to break down requirements and consolidate them into actionable items. We will explain how the directive API offers different ways to accomplish similar tasks and how to best choose what options accomplish tasks in the most efficient and readable manner.

The purpose of this chapter is to showcase simple and efficient techniques when writing directives. We do this by writing a stopwatch and showing how to accomplish its set of requirements. The stopwatch's definition object could be set up in different ways. We will first see how to create the stopwatch in its most simple form and then add optimization techniques that make the stopwatch more advanced.

Breaking down the stopwatch

Many implementations of stopwatches are available all over the Web, which use various different techniques and styles. This book's stopwatch is meant to be simpler and easier to implement inside of a modern web app. The directive API offers the options needed to create a sleek and straightforward piece of code that completes all the given requirements. Each of the stopwatch's requirements can be accomplished in many ways inside of an AngularJS context.

We will go over each requirement needed to build a stopwatch. When we break down each requirement, we will start to see what definition object options could be used to perform the task at hand. The major differences between the implementations that will be discussed are how to best set up a directive's link function and when and how to use some more advanced directive API options.

Stopwatch requirements

The stopwatch has the following set of requirements that need to be fulfilled and be able to work with multiple stopwatches on the same page:

- The ability to track its own time
- The ability to start and stop on command
- The ability to reset to zero on command
- The ability to log each lap

The requirements can be accomplished in different ways. It would be common to write this directive with a link function that is full of logic that accomplishes all of these requirements. Writing directives whose link functions contain all of the implementation logic is standard and accomplishes the basic requirements; however, writing directives in this manner does not facilitate the most readable, concise code.

A simple pseudo code example of this is as follows:

```
{
  scope: true,
  link: function(scope, element, attrs){
     scope.startStop = function(){//stop or start the timer}
     scope.reset = function(){//reset the timer}
     scope.logTime = function(){//log time}
  }
}
```

This example shows a simple link function and a `scope` object. The link function sets the required functions onto a child scope. We know that it is a child scope because the `scope` option evaluates to `true`. This means that the directive will be able to communicate with its defining scope through prototypical inheritance. This implementation of the directive API options will ultimately accomplish the requirements, but it could get out of hand quickly as we add more and more logic to the stopwatch.

Prototypical inheritance is a JavaScript design pattern that allows for objects to inherit from the prototypes of parent objects. This is a basic JavaScript model that is used heavily in AngularJS `$scope` objects, which are prototype objects that either prototypically inherit from their parent in the DOM or from `$rootScope`.

Deciding whether or not to use a child scope or an isolate scope is not always intuitive and requires a breakdown of what the directive is doing. Usually, isolate scopes are required when templates are needed to accomplish a directive's use cases. Many times, a directive only needs a child scope. In the case of the stopwatch, all we are doing is tracking the amount of time from a start and stop point. This is a simple use case that can be accomplished with just a child scope that allows for declarative markup inside of the stopwatch's child DOM elements.

Child scopes are meant for directives that expose public functions to its DOM elements. These public scope functions can be used in combination with other directives that can all work together on the same child scope. Child scopes allow the developer a medium to express normal JavaScript scopes into HTML, which is a revolutionary concept in itself.

An isolate scope is meant for a directive that has a specific API that should be exposed onto its scope. These directives are used when a reusable element is being created that either contains a template or the element's transcluded DOM. So, the overall rule is that if a directive needs to consolidate and force a specific API to its surroundings, it should use an isolate scope. A directive can benefit in many ways from using an isolate scope, but they should be held back as the last resort for a directive to accomplish its given requirements.

The negative aspect of using the previous format to write a directive is that the link function will be very large and contain logic that is not specific to the life cycle of the directive. To combat against the link function getting out of hand, it is very common to write services or factories that contain business logic. This business logic can then be injected into the directive's closure, giving the link function access to its exposed context. Creating directives with custom helper services is highly recommended to keep redundancy low and readability high.

The stopwatch could also be written with a controller that contains its implementation logic. This would allow for the separation of concerns, which would lead to more readable code. Writing the stopwatch with a controller also allows for a simple, reusable API that can be used in other directives that share some parent-child relationship. An isolate scope would be required if the directive was using a controller. This is because the controller is instantiated before the child scope is linked to the DOM. An Isolate Scope is created in a different order to create the isolation from the current prototypical pattern and allow for all linking phases to use the correct isolate scope object.

Even though we are not 100 percent certain as to how we would like to implement the specific details of the stopwatch, we can still write some simple tests that prove its creation logic.

The basics of testing

The **Test Driven Development** (**TDD**) process is a very maintainable development standard and is what AngularJS is built on. No quality AngularJS application or directive should go untested, and the extent of the tests should always be ever expanding. We will now talk about some of the tests that the directive uses to prove its logic. These tests will help explain how the directive will be created in the DOM before any code is written.

When writing tests for directives, the AngularJS lifecycle is much more apparent, and the "AngularJS Magic" that people refer to starts to reveal itself. This helps when trying to understand what a directive's lifecycle is and how it works at the basic level.

Testing directives is especially important. Directives are one of the hardest concepts to grasp in AngularJS, and DOM manipulation techniques are notoriously buggy when implemented incorrectly. Since AngularJS comes baked in with its own tests, there is no excuse not to test. As more tests are written, it becomes more apparent how the internals of AngularJS work.

Jasmine is the testing framework that is used in conjunction with Karma. Karma is the built-in test runner for AngularJS applications. A brief overview of how to set up Karma can be found at `https://github.com/karma-runner/karma-jasmine`

The unit tests should prove the basic logic behind the stopwatch's functionality and check if the stopwatch can be initialized correctly. The unit tests should not only prove the logic behind the code, but they should also help achieve a more advanced level of understanding as to what exactly it will take to most efficiently create the directive.

There are different ways to break down the describe blocks that are used to test directives. Each describe block should hold logic that performs like functions so that the readability of the tests remains high. In this chapter, we will break them down into the following three categories:

- Tests that prove the initial creation of the directive
- Factory/Service interactions
- Filter tests

The specific tests that will be shown are all part of the overall describe block of the stopwatch; it utilizes two modules and sets some global variables that are injected into the directive. These three service instances that are being injected into each of the tests are needed to call functions inside of the tests themselves, as they have their own specific uses while testing.

The following code snippet shows the predefined context of the stopwatch's clauses:

```
beforeEach(module('AngularBlackBelt.Stopwatch'));

beforeEach(inject(function (_$rootScope_, _$compile_,_$interval_) {
    scope = _$rootScope_.$new();
    $compile = _$compile_;
    $interval = _$interval_;
    scope.options = {
        interval: 100,
        log: []
    };
}));
```

The first `beforeEach` clause sets the context of the tests. Specifically, `beforeEach` calls the `angular.module` function, which either searches for a predefined module or sets a new module instance. In this specific case, it finds the predefined `AngularBlackBelt.Stopwatch` module and loads it into the execution context.

The second `beforeEach` clause utilizes the AngularJS injector function to find services that will be used to perform the tests. We wrap the Angular factories with underscores, so we can assign them to local variables. This is one way to add dependencies in tests. These services each have a specific purpose and are essential to the test's passing; however, they are not essential to the logic that the tests are meant to prove.

Creation tests

The creation tests of the stopwatch are to prove that the directive either failed to be initialized or was created as expected. Once the creation occurs, the intended scope objects should also be linked to the DOM's scope and contain their proper objects. These tests are very basic, but they serve as the base of our directive's tests and will allow us to add more in the future easily.

The following test will help us ensure that the correct options get passed into the directive. This test is supposed to always throw an exception because we are not passing any options into the directive. The directive needs at least an empty object, so it has a reference in memory to allow for AngularJS data binding to take effect in the DOM.

```
it('should throw an error if there are no options set on the element',
  function() {
    expect(function(){
      var stopwatch = $compile('<div bb-stopwatch></div>')(scope);
      scope.$apply();
      }).toThrow('Must pass an options object
        for the stopwatch to work correctly.');
});
```

The following test shows that if all of the directives' initialization requirements are met, then errors will not be thrown. We compile the directive and add it to the digest cycle so that we can mimic real interactions with DOM or `$scope`. This is being done with the `$compile service` method.

After the stopwatch is compiled, we call the `$apply` method. The `$apply` method runs the first digest cycles, which allows us to run all the watcher functions and initialize the directive as it would be in production. Refer to the following code:

```
it('Should not throw an error with an options object', function()
  {
    expect(function(){
      scope.optionsObject = {};
      var stopwatch = $compile('<div bb-stopwatch
        options="optionsObject"></div>')(scope);
      scope.$apply();
    }).not.toThrow();
});
```

The preceding test proves that all we need is an empty object for the directive to initialize successfully. It is important to understand why the `$apply` function is being used in the test. Directives are initialized before the first digest cycle starts. In production, a digest cycle can cause a directive to show an error if that directive is dependent on the digest cycle for initialization. So, because of this, we add the `$apply` function to closely mimic what would ensue in a live application.

In a specific case when creating a stopwatch, we will need a type of interval that calls an `updateTime` function. An interval is needed so that we can constantly update the `elapsedTime` function. The interval frequency should either be set from an object or as a default value. In this case, the default value is `100`. This test showcases that if no interval field is passed into the directive, the default interval will be set as `100`. Refer to the following code:

```
it('Should set the default interval value to 100 milliseconds',
  function() {
    var stopwatch = $compile('<div bb-stopwatch
      options="newObject"></div>')(scope);
    scope.$apply();
    expect(stopwatch.scope().options.interval).toBe(100);
});
```

Different functions are available to the `angular.element` object. Here, we call the `scope` function because we want to test the stopwatch's scope to make sure that it sets the correct values. If we used an isolate scope, we use `isolateScope` as the function to test with instead.

The `$compile` function returns a link function that can be run with a scope parameter to interpolate its template values. This link function returns an angular element. The angular element is almost equivalent to a `$(element)` object, and if jQuery is included in the window, then it is.

The following code snippet proves that if we send a custom interval into the stopwatch, it will override the default value:

```
it('Should contain all relative functions', function() {
  var stopwatch = $compile('<div bb-stopwatch
    options="options"></div>')(scope);
    scope.$apply();
    expect(stopwatch.scope().stopTimer).not.toBe(undefined);
    expect(stopwatch.scope().startTimer).not.toBe(undefined);
    expect(stopwatch.scope().resetTimer).not.toBe(undefined);
});
```

The previous creation tests prove that the directive does not throw any unexpected errors when being created and sets the appropriate default values on the scope once initialized. Now, we can move on and write the directive to showcase how to write the basics of the stopwatch directive and evolve it into a more advanced one.

Writing the stopwatch

Most directives use a link function to achieve their functionality in collaboration with AngularJS scope objects. This is because the link function is usually all that is required to accomplish most requirements that a directive has. The reason the link function (post link) is so useful is because it is always run when the directive's DOM and child scope have been attached to their associated AngularJS scope. This allows us to ensure that all the data properties are present and all the DOM elements have been compiled into an AngularJS context.

In this case, we are using a basic child scope declaration for the directive. This allows for the directive to achieve its functionality inside of its own private scope, which inherits from its defining scope. This inheritance is what allows for data binding between the parent and child scopes, and it is the simplest way to achieve the directive. Refer to the following code:

```
app.directive('bbStopwatch', ['StopwatchFactory',
  function(StopwatchFactory){
    return {
      restrict: 'EA',
      scope: true,
      link: function(scope, elem, attrs){

        if (!attrs.options){
          throw new Error('Must Pass an options object from the
            Controller For the Stopwatch to Work Correctly.');
          }

          var stopwatchService = new
            StopwatchFactory(scope[attrs.options]);

          scope.startTimer = stopwatchService.startTimer;
          scope.stopTimer = stopwatchService.stopTimer;
          scope.resetTimer = stopwatchService.resetTimer;

        }
    };
}])
```

The directive is namespaced with a bb, Black-Belt, because it is the best practice to namespace directives. This is so other projects can easily plug and play the directives.

The business logic is placed inside of the `stopwatchService` function, which is being created by the `StopwatchFactory` function. This is fed to the options object as a parameter and creates a new service object that will be used to manipulate the `elapsedTime` object. This is a very clean and organized way to write directives. The `stopwatchFactory` function can now be tested separately, allowing for the separation of the DOM logic that we were expecting to see using AngularJS.

The `bbStopwatch` function uses a child scope. As we prove in our tests, the directive will throw an error if the `options` attribute is not defined. This is because the stopwatch is dependent on the knowledge about exactly where it will be reading and writing from. By feeding the directive a `hash` object, we allow for a more organized input/output channel for the stopwatch that fixes any prototypical inheritance issue that the child scope would have caused.

When writing to a `hash` object from inside of a child scope, the `hash` object's properties will still reference its variables from its defining scope.

There are many benefits to just using a child scope in a directive. Some of these include simplicity, a prototypical inheritance model, and expected template behavior. The child scope created by the `bbStopwatch` function is accessible via any DOM that is defined as a child of the `bbStopwatch` function. This means that we can now place a stopwatch directive on any element and attach the link function's `$scope` methods to it.

In JavaScript, the `$scope` options will look like the following code:

```
$scope.stopwatchOptions = {interval: 100};
```

In HTML, the `$scope` methods will look like the following code:

```
<div bb-stopwatch options="stopwatchOptions">
 {{stopwatchOptions.elapsedTime}}
  <button ng-click="startTimer()">Start</button>
  <button ng-click="stopTimer()">Stop</button>
  <button ng-click="resetTimer()">Reset</button>
</div>
```

This allows for very dynamic creations of DOM, which can be reused throughout an application with many different implementation patterns. Another benefit of using child scopes is that we keep the definition context of the scope variables that we are passing into the directive.

If we were using an isolate scope in this instance, we would define the `stopwatchOptions.elapsedTime` function inside of the curly brackets as the `options.elapsedTime` function because the isolate scope created would not prototypically inherit from the parent scope and would set its own value based on its definition object. This is not 100 percent apparent and could cause readability issues.

This is the simplest form of the stopwatch directive. A working version of the stopwatch directive can be found at `http://jsfiddle.net/joshkurz/5LCXU`.

Stopwatch's business logic

The factory function used for the stopwatch is where all the main logic is placed. Since the logic is reusable and can be used in other places around our application, we created a factory. This factory returns a singleton object that can be created many times, which allows the stopwatch directive to accomplish its most important requirements. The majority of the work is done in this factory; however, it is organized in a clean and readable manner.

There is only one dependency that the factory relies on to be successful in its attempts to provide the stopwatch's functionality. This dependency is the `$interval` service, which fires a closure function at a set time in a repeated form. The `$interval` service is a wrapper for the native JavaScript `window.setInterval` method, but it calls `$scope.$apply` on every execution, and the `$interval` service offers a simple way to cancel its current interval.

The full analysis for this can be found after the source code. The following code snippet is the full `StopwatchFactory` function:

```
stopwatch.factory('StopwatchFactory', ['$interval',
  function($interval){

    return function(options){

      var startTime = 0,
      currentTime = null,
      offset = 0,
      interval = null,
      self = this;

      if(!options.interval){
        options.interval = 100;
        }

      options.elapsedTime = new Date(0);
```

```
        self.running = false;

        function pushToLog(lap){
          if(options.log !== undefined){
            options.log.push(lap);
            }
          }

        self.updateTime = function(){
          currentTime = new Date().getTime();
          var timeElapsed = offset + (currentTime - startTime);
          options.elapsedTime.setTime(timeElapsed);
          };

        self.startTimer = function(){
          if(self.running === false){
            startTime = new Date().getTime();
                  interval =
              $interval(self.updateTime,options.interval);
              self.running = true;
              }
            };

            self.stopTimer = function(){
              if( self.running === false) {
            return;
            }
            self.updateTime();
            offset = offset + currentTime - startTime;
            pushToLog(currentTime - startTime);
            $interval.cancel(interval);
            self.running = false;
            };

            self.resetTimer = function(){
          startTime = new Date().getTime();
          options.elapsedTime.setTime(0);
          timeElapsed = offset = 0;
        };

        return self;

        };

  }]);
```

That's a lot of code, but it's the biggest function definition the stopwatch contains. One of the most important takeaways from this snippet is that the factory's contexts is referenced as the `self` object, and the most important function is set to it. This is so we can offer a public API. We return the `self` object, which means that anything that is set to it will be available as a public function. This is how JavaScript replicates a private/public functional context model. This is very useful in all JavaScript applications and especially when writing directives. The cleanliness and readability of the code is much higher in this case than if we were to add this entire context into a directive's link function.

We also use the `self` object to assist us with referencing issues. This is so we can still reference `updateTime` inside of the interval callback function. When the interval fires, the context of the function is nonexistent, and so is the `this` object, because it is only available inside of the factory's context.

The rest of the logic is to allow the stopwatch to track its `elapsedTime` while fulfilling all of its initial requirements.

Business logic tests

The main purpose of having a factory contain the logic for the stopwatch is to keep track of the `elapsedTime` function in its isolated instance that is clean and testable. Adding a factory to the directive also cuts down cyclomatic complexity, which helps the overall readability and maintainability of the code. The factory allows us to create functions that have consolidated logic that can be publicly referenced in a JavaScript execution context, allowing for testing possibilities. Business logic is very important for testing in isolation as it does not depend on integrations of any type to complete its overall task.

The following code snippet is the `beforeEach` function that sets the service object to be used in the factory's unit tests:

```
beforeEach(inject(function
  (_$rootScope_,_$interval_,StopwatchFactory) {
    scope = _$rootScope_.$new();
    $interval = _$interval_;
    scope.options = {
      interval: 100,
      log: []
      };
    stopwatchService = new StopwatchFactory(scope.options);
}));
```

The `StopwatchFactory` function uses the `Date` object to set its elapsed time. We do not want to stub the `Date` object for the tests. This is because we want the factory's functions to act exactly how they would in production. It is for this reason that we are testing how many times the `updateTime` function is being called, because we trust that the JavaScript `Date` object always sets the right time.

The test creates a new instance of the `StopwatchFactory` function, which will be interacted with in the same manner that a directive would be interacted with. This will prove that the logic is correct. The next factory test shows that once the interval is flushed for 1000 milliseconds, the controller's `updateTime` function is called 10 times since the default interval time is 100 milliseconds. To accomplish this, we set a spy object on the `updateTime` function.

Now, we need to prove that when we start the timer and clock, the timeout is flushed for x milliseconds and the `updateTime` function is called x/interval times. This test uses the `$interval.flush` function to mock how much time has passed. This allows us to force the interval in the factory function to fire, which then calls the `updateTime` function. Refer to the following code:

```
it('Should call updateTime when the timer is started and should
  call it every 100 milliseconds', function() {
    spyOn(stopwatchService, 'updateTime');
    jasmine.Clock.useMock();
    stopwatchService.startTimer();
    $interval.flush(1000);
    expect(stopwatchService.updateTime.callCount).toBe(10);
    $interval.flush(1000);
    expect(stopwatchService.updateTime.callCount).toBe(20);
    $interval.flush(1000);
    expect(stopwatchService.updateTime.callCount).toBe(30);
});
```

The following test makes sure that the `stopwatchService` function is not calling the `updateTime` function at every interval if the `stopTimer` method has been called:

```
it('Should not call updateTime if the timer is stopped',
  function() {
    spyOn(stopwatchService , 'updateTime');
    stopwatchService.startTimer();
    $interval.flush(1000);
    expect(stopwatchService.updateTime.callCount).toBe(10);
    //calls update time one more time whenever we stop the timer
      so the elapsedTime has the most up to date time.
```

```
        stopwatchService.stopTimer();
        $interval.flush(1000);
        expect(stopwatchService.updateTime.callCount).toBe(11);
    });
```

The following test proves that the `stopwatchService` function is logging the time at the appropriate instances:

```
    it('Should append to the options log object', function() {
      spyOn(stopwatchService , 'updateTime');
      stopwatchService.startTimer();
      $interval.flush(1000);
      //calls update time one more time whenever we stop the timer so
        the elapsedTime has the most up to date time.
      stopwatchService.stopTimer();
      expect(scope.options.log.length).toBe(1);
    });
```

When these tests have passed, it will prove that the factory is updating the `elapsedTime` object as expected and the interval can be stopped and started on command. It also proves that the factory works with a log array and appends messages to it correctly.

Now, we need to make sure that we are clearing the stopwatch's interval correctly and not letting it hang around at all. When working with intervals, one has to be very careful to make sure that no intervals are left behind.

Let's consider a production example. If the stopwatch is compiled on a scope, which is set underneath a view and the view switches, that scope broadcasts a `$destroy` event. This `$destory` event cleans up the scope objects, but it does not destroy the stopwatch interval automatically.

This is a very big problem, but it can be solved rather easily. The next test proves that once the `killTimer` method is called, the `updateTimer` function will never be called again. The following code snippet shows that we can programmatically stop the timer and disallow any more intervals to fire:

```
    it('Should not call updateTime if the interval has been
      destroyed', function() {
        spyOn(stopwatchService , 'updateTime');
        stopwatchService.startTimer();
        $interval.flush(1000);
        expect(stopwatchService.updateTime.callCount).toBe(10);
        stopwatchService.cancelTimer();
        $interval.flush(1000);
        expect(stopwatchService.updateTime.callCount).toBe(10);
    });
```

Note how the `callCount` function is still 10, even after the `interval.flush()` function. This proves that the interval was destroyed as expected, and no more subsequent calls to `updateTime` were made. All we have to do now is add a `subscribe` function on the `$destory` event that calls this factory function, and we can ensure that our intervals will always be cleaned up. This test would fail with the current `bbStopwatch` directive.

Optimizing the stopwatch

Now that we have created the directive in a super simple form, let's optimize it a little more and also add some functionality. The original `bbStopwatch` function accomplishes all of the original requirements. Some great optimization techniques that could be achieved on the stopwatch would be to use a `compile` function instead of a `link` function or clean up intervals when scopes are destroyed.

The `compile` function is used in different situations, but mostly, it is an optimization tool that allows for pre-interpolated DOM to conditionally be checked or manipulated. In the case of the stopwatch, we use it to short circuit the directive if the initial option's attribute requirements are not met. Any type of element or attribute conditions that must be met for the directive to achieve successful creation, which is not reliant upon scope objects or that element's children, should be implemented during the compile phase.

To accomplish this, all we have to do is take the attributes being checked from the link function and place the logic in the `compile` function. It is important to know that because we do this, the attributes check will only be run once per directive, and the directive will not have to run its expensive link function if the conditions are not met. This is a minor optimization, but it is the correct implementation. Refer to the following code:

```
compile: function(tElem, tAttrs){

  if (!tAttrs.options){
    throw new Error('Must Pass an options object from the
      Controller For the Stopwatch to Work Correctly.');
    }


    return function(scope, elem, attrs, controller, transclude){
      // same exact link function as we had before.
      };
}
```

Now, the stopwatch will call its compile function first and make sure that the requirements are met. If the requirements are indeed met, the link function will be returned and instantiated with its associated scope in the same manner as before.

Since the stopwatch uses the `$interval` service that comes with Angular Version 1.2.x, its pretty easy to set, clear, and allow an interval to run without having to worry about calling the `scope.$apply()` function ourselves. The only caveat is that at the moment the `$interval` service of AnguarJS does not automatically clear the interval when the `$interval` object's specific context is destroyed.

The following text can be found at `http://javascript.info/tutorial/memory-leaks#setinterval-settimeout`:

> "*For* `setInterval`*, the completeness occurs on* `clearInterval`*. That may lead to memory leaks when the function actually does nothing, but the interval is not cleared.*"

It is for this reason that we add an `observer` method and wait for the `scope's $destroy` method to execute. By adding the destroy event observer, we make sure that the interval is cleared according to plan. The following code snippet ensures that we do not have any memory leaks associated with the stopwatch's interval functions:

```
scope.$on('$destroy', function(node){
  stopwatchService.cancelTimer();
});
```

Many directives could benefit from using this approach. There are many variables that get left behind when scopes are destroyed. These variables are not able to become garbage collected, which means that a memory leak has occurred. Getting used to always clearing out intervals, timeouts, and other, nonreferenced local variables when the scope is destroyed is the best practice.

The stopwatch's interval function is made available by a tiny addition to the `stopwatchFactory` function. Refer to the following code:

```
self.cancelTimer = function(){
    $interval.cancel(interval);
    }}
```

Once this has code has been added the factory, the stopwatch has the ability to cancel its own interval at any time.

Stopwatch's filter

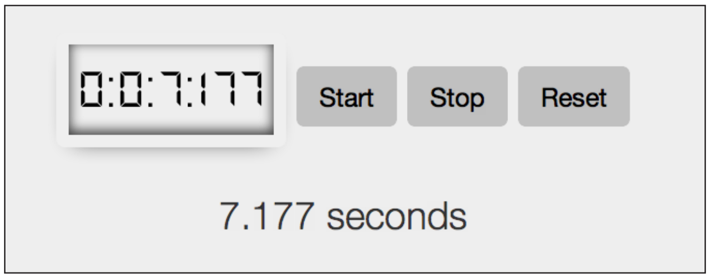
The stopwatch relies on a filter that converts the `elapsedTime Date` object into a readable format. This filter is used inside of whichever DOM element is representing the time. This shows even more separation of the presentation logic that the core stopwatch functionality does not rely on. The following code snippet shows how the logic would be used in HTML:

```
<div class="stopwatch numbers">
  {{options.elapsedTime | stopwatchTime}}
</div>
```

The code is very simple and not exactly relevant to how the stopwatch works internally, so we will just show a test to prove that it works. There are four tests on the Black Belt repo, but they all serve the same purpose, which is to make sure that the filter returns the correct stopwatch time format depending on the date object that it is being passed. In the following code snippet, we test that the filter returns a string that represents that 1 hour has passed:

```
it('Should have 1 hour elapsed', function() {
  stopwatchTimeFilter = $filter('stopwatchTime');
  var newDate = new Date(1000 * 60 * 60);
  expect(stopwatchTimeFilter(newDate)).toBe('1:0:0:0');
});
```

The stopwatch is complete. We have successfully created our first, working AngularJS directive. This directive has many use cases and can be used in the production of applications all over the Web. You can find the live example at `http://jsfiddle.net/joshkurz/5LCXU/6/`. The following screenshot is an example of how the stopwatch functions; it is shown on the Black Belt site (`http://angulardirectives.joshkurz.net/dist/#/stopwatch`):



Summary

The stopwatch is now fully functional and ready for production. There are many things it can be used for. Some more advanced use cases could be called for the stopwatch to work in collaboration with other directives that depend on the `elapsedTime` object, or having some resource function post the logs to a server. All of these are possible and easily achievable now that the basics are implemented.

The main cause for creating the stopwatch was not because there were no stopwatches on the Web, but to showcase the different options available when creating a directive and how these options help decouple many aspects of AngularJS development.

The stopwatch directive utilizes three main options made public by the directive API. The `Scope`, `Compile`, and `Link` fields give the directive the ability to achieve many of its requirements in a simple and organized format.

AngularJS makes it very easy to decouple many different aspects of the stopwatch's logic. No selectors are needed, which can constrain a piece of code whose purpose is to manipulate DOM according to some model object. In this specific case, we get rid of the need for DOM selectors by allowing AngularJS scopes to work with declarative DOM. Now, there is no need to even write code to update this directive's HTML. This is monumental, and a wonderful example of why AngularJS is at the leading edge of the JavaScript MVC/MVVM/MV* framework technology.

The main concern we came across when building the stopwatch was to make sure that it did not leak memory by allowing its interval to live forever. This concern was taken care of by the `$on` observer method, which is available to all scope objects inside of an AngularJS application. Since all of the scopes emit a `$destory` event that traverses all the child scopes, cleaning up is easy. If there were any other objects that would be considered dangling references left inside of the destroyed scope, the stopwatch would set them equal to null here as well.

In later chapters, we will be implementing the directive to an application, allowing it to communicate and work in collaboration with other directives and controllers.

3 Harnessing External JavaScript Libraries with Directives

JavaScript is a powerful language that gives the developer many different opportunities to build rich, dynamic web pages and applications. It can take years to master the language and know all of the different possibilities available. This is when third-party libraries come into play. Time is a necessity in every application's life cycle. AngularJS works beautifully with these JavaScript libraries when used correctly.

There are a plethora of different JavaScript libraries that are built with jQuery. Most of these libraries focus on manipulating the DOM for various reasons. The open source world offers many of these wonderful jQuery plugins free of charge. Since there are so many different types of jQuery plugins available that offer much of the basic requirements needed for a modern web application, there is much value in knowing how to incorporate any third-party library into an AngularJS application.

The AngularJS community is large and vibrant. Many of these jQuery plugins have already been created by the community and are easily available. AngularUI is the most popular online organization, which is known for creating well-built and well-tested directives. Some of their directives utilize third-party libraries, and some are written with pure AngularJS scripts.

This chapter builds two directives that utilize third-party libraries. One uses a third-party gauge library and one is the `fullCalendar` directive from AngularUI. The purpose of all the directives built in this chapter is to showcase different techniques to integrate these third-party libraries into the AngularJS digest cycle.

The main differences in the community techniques for creating directives comes by way of how the developer incorporates `ngModel` to tie a dataset to the AngularJS life cycle. The `ngModel` directive is a core directive that is used by many elements to declaratively set the `watch` functions in HTML markup. Refer to the following code:

```
<input ng-model="theString">
{{theString}}
```

Now, whatever we type into the input will automatically be updated inside of the curly brackets. This is one of the most spectacular features AngularJS offers, and it is all because of the `ngModel` directive and its isolated scope.

> Working with third-party libraries calls for the need of an isolate scope because we want as many options for data binding available to us as possible.

It should be made clear that if a third-party library's purpose is to manipulate and dynamically create the DOM, it should be converted into a directive. There are third-party libraries that do not need to be wrapped as directives, and their methods can be used with factories and services. Socket.IO is a great example of a library that could be used without the need to create a directive to utilize its features. Once libraries such as Socket.IO have been wrapped correctly in an injectable factory or service, they can be used with other directives to accomplish the DOM-related tasks.

Incorporating third-party libraries

jQuery is by far the most popular JavaScript library available. It offers many functions whose purpose is to control the DOM efficiently and effectively. AngularJS teaches us that these methods should only be used inside of directives. Now, with AngularJS, there is much less need to create directives with IDs and crawl the DOM using selectors. The element is always available, either precompiled or postcompiled, inside of a directive.

It should be noted that AngularJS offers a jQLite object that is a much smaller version of jQuery, but it does offer many of the same powerful functions. The need to include jQuery comes when incorporating other third-party libraries that depend on it into an application.

> When building directives from scratch, it is usually easier to write pure AngularJS implementations. These usually come out simpler and more effective than directives that are integrated with untested third-party libraries.

The following are the three essential requirements behind wrapping any jQuery-based library with AngularJS:

- The data that the directive uses needs to be watched for changes
- The directive's DOM should be updated when changes occur
- The events emitted by the third-party library need to call `$apply` and resync the model if they change that data in any way

The AngularJS digest cycle is the basis for the model being kept in sync with the view at all times. The only way jQuery-based libraries can be used in the proper AngularJS manner is if their events and data are hooked into this digest cycle.

There are many different techniques used in the community to accomplish this requirement. The main purpose they all accomplish is to call the third-party libraries' main function on the element provided by the directive's link function.

> Let's not forget that the only reason we are writing a directive that uses a third-party library is because it does some very advanced DOM manipulation.

The link function is the only place where third-party libraries should ever be called on an element. When a specific directive's link function is invoked, the directive and its child directives get compiled by AngularJS and their templates interpolated with the directive's given scope. This ensures that all of the necessary DOM and scope elements are in place and can be used accordingly.

All third-party libraries are not the same, and some need to be used in different ways to achieve the required result. They all do have one thing in common; that is, they need to call their libraries' main function on their DOM element.

The first step to writing any directive that uses a third-party library is pretty much the same as writing any other AngularJS directive (writing tests).

Testing directives that use third-party libraries

Application-wide unit tests and end-to-end tests have been around for a long time. JavaScript testing has just recently started to build momentum in the community. That said, older third-party libraries do not usually have quality testing suites, which means that they were not built for testing. So, subsequently they will probably have a lot of private functions and variables that are not accessible to scopes outside of their closures. This is not always an issue, but in some cases it can be troublesome when trying to test third-party directives in a TTD manner.

Testing libraries such as Jasmine and Mocha allow for different types of function spying, which are a large part of the solution to testing third-party libraries. The spy function can expose the methods called by the object being watched, the method call count, and the parameters used. These spy functions can prove that the directive has called the appropriate library function. From there, we will trust that the third-party library is going to do what it documents that it does.

The following code snippet is a sample test that creates a spy on a sample third-party library function and tests to make sure that once the directive is created, the third-party library function is called:

```
it('should make sure the thirdPartyFunction is called, function ()
  {
  spyOn($.fn, 'thirdPartyFunction');
  expect($.fn.thirdPartyFunction.callCount).toBe(0);
  $compile('<div sample-directive></div>')(scope);
  expect($.fn.thirdPartyFunction.callCount).toBe(1);
});
```

Not all third-party libraries add their main function to the jQuery `$` object, as shown in the following code:

```
App.directive('sampleDirective', function(){
  return {
    scope:{localVar: 'ngModel'},
    link: function(scope, element, attrs){
      //calling the third party function on the element
      element.thirdPartyFunction();
      }
    };
});
```

This is as basic of an example as it gets. The `thirdPartyFunction` is being called on the Angular element, which is the same as using a jQuery selector to perform the DOM-related task that it was included for.

Not all JavaScript libraries hook themselves as a prototype function to jQuery. Some use different techniques to expose the libraries' functionalities to the application node. If this is the case, they are accessible for spying by some other means rather than directly on the function itself. In some instances, a constructor is attached to the window object as a new function. The constructor's prototype methods can be spied on and checked for the proper calls that should have been made. This is the case in the following example.

In other instances, the function is not a constructor but a closure exposed on the window, much like the `$` object or the `angular` object. These types of libraries are easily spyable in the same manner as well, as shown in the previous example.

Wrapping the gauge.js file

Many websites have requirements, such as showcasing data in different formats, so that the user can consolidate and consume the data as fast and effectively as possible. Gauges help provide some of this functionality. There are many different gauge libraries available for use in the open source market. For this example, the decision was to go with gauge.js (`http://bernii.github.io/gauge.js`).

Gauge.js is a simple, canvas-based library that has an easy API for rendering gauges in a wide variety of formats. When coupled with AngularJS, its power becomes focused and intensified. For this implementation, we need to do the following:

- Render the gauge on a canvas element
- Attach the value and the options to the `$scope` function
- Make sure the gauge updates itself when its current value or options change

Once these requirements are met, the gauge will be completed, and it can be incorporated into any AngularJS application.

Testing the gauge directive

To test the `gauge` directive, the requirements need to be proven. The following tests prove that the directive can only be applied to canvas elements, and that when the scope changes, the directive calls the correct function:

```
describe('Creating A gauge-js Directive', function () {

  it('should throw an error if not set on a canvas
    element',function(){
    expect(function(){
      var gauge = $compile('<div gauge-js></div>')(scope);
      }).toThrow('guage-js can only be set on a canvas element.
        DIV will not work.');
    });

  it('should not throw an error when creating on a canvas
    element', function() {
    expect(function(){
      var gauge = $compile('<canvas gauge-js current-value="value"
        options="configOptions"></canvas>')(scope);
      }).not.toThrow();
    });
```

This following test is a little different. Since we know that we are going to be relying on a `watch` function to update our directive when scope objects change, we need to call the `$scope.$apply` function to simulate this. The `$apply` function will fire the directive's `watch` function, which in turn will call the gauge constructor with the appropriate parameters.

```
  it('should set the Gauge when the directive is compiled and linked
    to the DOM', function() {
    spyOn(Gauge.prototype, 'setOptions').andCallThrough();
    var gauge = $compile('<canvas gauge-js current-value="value"
      options="configOptions"></canvas>')(scope);
    scope.$apply();
    expect(Gauge.prototype.setOptions)
      toHaveBeenCalledWith(scope.configOptions);
    });

  });
```

The first two tests prove that the element was created, and that if it was created on any other type of object than a canvas, that it will throw the proper error. The last test sets a spy on the gauge's prototype function, `setOptions`, and makes sure that it is called with the correct parameters during the initialization of the gauge.

Writing the gauge directive

The `gauge` directive is simple, but when bound together with AngularJS, it is a very powerful tool. The main demo that gauge.js uses can be found at `http://bernii.github.io/gauge.js/`. This example showcases how the gauge can be utilized when its inputs automatically update the gauge (view). The example shown here includes over 140 lines of code. The full directive and example in AngularJS is only 80 lines of code, and this includes the HTML for the view, as shown in the following code:

```
angular.module('GaugeJs', []).directive('gaugeJs', function(){
  return {
    restrict: 'A',
    scope: {
      options:'=',
      currentValue: '=ngModule'
      },
    compile: function(tElem, tAttrs){

  if ( tElem[0].tagName !== 'CANVAS' ) {
    throw new Error('guage-js can only be set on a canvas element.
      ' + tElem[0].tagName + ' will not work.');
    }

  return function(scope, element, attrs){

    var gauge;

    function setGauge(options){
      gauge = new Gauge(element[0]).setOptions(scope.options);
      gauge.maxValue = scope.options.maxValue; // set max gauge
        value
      gauge.set(scope.currentValue);
      }

    scope.$watch('options', function(newV, oldV){
      setGauge(scope.options);
      },true);

      scope.$watch('currentValue', function(newV,oldV){
        if(scope.currentValue > scope.options.maxValue){
          gauge.set(scope.options.maxValue);
          } else {
            gauge.set(scope.currentValue);
            }
          });
        };
      }
    };
});
```

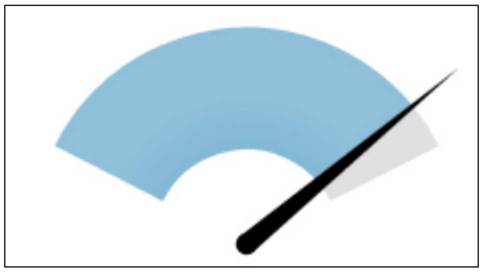
The `gauge` constructor has to be fed a raw canvas element. This is because of the inner workings of the gauge.js library.

Some libraries vary and take the wrapped jQLite or jQuery objects. Some only work with the raw element.

Once the gauge constructor is initialized and rendered, the only requirements left to achieve are the automatic updates.

The two watch functions take care of all the data binding updates that the gauge needs. The directive performs two different types of watches here. The first watch function is a deep watch. This will traverse the JSON options and check for differences. If there are differences, the watch function will be fired and the gauge will be reset with the new options. The second watch function just observes the scope.currentValue variable. The currentValue function is set with ngModule so that this directive follows normal AngularJS directive implementation. The currentValue variable does not need to be watched with a deep watcher because we are not working with a JSON object at this point.

This is all it takes to render the gauge directive. The full demo has been reconstructed and can be found at http://jsfiddle.net/joshkurz/8W2Z5/. Now, we have a fully functional gauge that can be altered and updated from any controller methods, as shown in the following screenshot:



Writing scope interaction tests

The following set of tests prove that the scope variables are attached inside of the isolate scope, and whenever they are changed, the gauge updates itself automatically. These tests are more closely related to the integration of the AngularJS scope variables and how their changes affect the gauge directive, as shown in the following code:

```
describe('Testing the Scope of a Gauge Directive', function () {

  var gauge,
  gaugeScope;
```

```
     beforeEach(function(){
       gauge = $compile('<canvas gauge-js ng-module="value"
         options="configOptions"></canvas>')(scope);
       gaugeScope = gauge.isolateScope();
       });

     it('make sure the isolateScope has the correct values attached',
       function() {
       expect(gaugeScope.currentValue).toBe(25);
       expect(gaugeScope.options).toBe(scope.configOptions);
       });

     it('when the currentValue changes the gauge is updated
       correctly', function() {
       var oldDataUrl = gauge[0].toDataURL();
       scope.value = 100;
       scope.$apply();
       expect(gauge[0].toDataURL()).not.toBe(oldDataUrl);
       });

     it('when the options change the gauge updates itself',
       function() {
       var oldDataUrl = gauge[0].toDataURL();
       scope.configOptions = 1000;
       scope.$apply();
       expect(gauge[0].toDataURL()).not.toBe(oldDataUrl);
       });

   });
```

The first test in this describe statement makes sure that the isolate scope of the `gauge` directive is set with the correct values. This is done by creating a `gaugeScope` object before each of the tests form the `isolateScope` method provided by the compiled directive.

The `isolateScope` method is only available to directives that actually have an isolate scope.

The second test proves that the gauge automatically updates itself when the parent value changes. Since the gauge library does not offer any get methods for the current gauge, we have to create a workaround by testing the `dataUrl` function.

The third test is similar to the second; however, instead of the value, the options will be changed. When any one of the options is changed, the gauge should update itself as well. This proves all of the initial requirements for the `gauge` directive. The `gaugeJs` directive showcases a simple way to wrap a third-party library into a usable AngularJS directive. The main purpose of the directive is to take care of all updates automatically, according to the changes in the model. This takes care of many of the lines of Boilerplate JavaScript, which can be replaced with the required business logic.

Wrapping the fullCalendar.js file

The Arshaw `fullCalendar` jQuery plugin is by far the most popular calendar plugin available in the open source world. The options that come as stock are mostly all a developer needs to create a rich, enterprise-level application that depends on some type of calendar functionality. The calendar can do amazing things, and when AngularJS is added to the picture, these amazing things become very easy to implement.

There are a few requirements that the calendar plugin has to be considered complete in all aspects; they are as follows:

- The calendar's events and options should be watched for changes
- The entire `fullCalendar` API should be available from a controller
- Multiple calendars must work together in the same view flawlessly
- The calendar should work with all documented types of events
- The calendar should update itself whenever any of its events change

Introduction to the calendar directive

Every library that needs to be incorporated into an AngularJS application is going to be implemented slightly differently. Many simple directives do not need any type of advanced functionality to check for changes and call their update functions for themselves.

Writing the `fullCalendar` directive is a little trickier than most. This is because of the functionality required to watch all of the events and call the correct functionality at the appropriate time. Most of the code of the `calendar` directive is used to watch the events and keep track of their state.

The calendar's event objects are very large and reference themselves. Due to this fact, they cannot be watched directly because the `calendar` directive creates digest errors to do so. Workarounds have to be made throughout the life cycle of the `calendar` directive to keep the model and view in sync.

Many advanced-level directives evolve as they progress. The `calendar` directive has been through a lot of changes since the beginning of its creation. The first `calendar` directive just watched the length of the events array and would update whenever this length changed. This was a very simple implementation, and the pitfalls of only watching the length of the array are obvious. The second approach was to use an array of event sources and watch its own length and the length of all of the objects inside it. This opened the calendar's API up almost fully and allowed for much more advanced calendars to be created.

> Watching large objects can cause AngularJS to bog down and throw errors. This is why the decision was made to watch just the length of the array.

The third and current form of the directive uses a very advanced tracking technique (written by Gleb Mazovetskiy) to watch the events in the event sources array. This tracking technique is built by taking a fingerprint of each event in the `calendar` directive. The fingerprint hash object uses a string representation of each event as its keys and a custom representation of each event as the keys' value. This fingerprint object is stored in memory inside of the directive's closure and watched by every digest for changes. This simply means that if the fingerprint object changes in any way, the `calendar` directive will call the corresponding `fullCalendar` method. The method that is called will be specific to the change that occurred to the fingerprint object.

The following code shows a test that proves that the `calendar` directive calls its correct method depending upon the change in an event's title. The `calendar` directive is in charge of knowing which function to call depending upon what changes are made to the events that the `calendar` directive is using to render its DOM elements.

```
it('should make sure that if we just change the title of the event
that it updates itself', function () {
            var calendarCalls = $.fn.fullCalendar.mostRecentCall.
args[0].eventSources;
            scope.events[0].title = 'change title';
            scope.$apply();
            calendarCalls = $.fn.fullCalendar.mostRecentCall.args[0];
            expect(calendarCalls).toEqual('updateEvent');
});
```

Testing the fullCalendar directive

There are three types of tests for the `fullCalendar` directive. The first set of tests builds the calendar in the DOM and spy on the `fullCalendar` jQuery function exposed by the library. The second set of tests works directly with the calendar's controller and makes sure that all of the change tracking is going according to plan. The third set of tests works with the different types of configurations, which are set before the calendar is compiled.

This is a subset of the full test suite that the calendar has. These tests were chosen to best showcase how to test third-party libraries. Only the first and second sets are detailed in this book.

The `fullCalendar` method is a prototype method that is exposed on the jQuery `$` object. This allows for Jasmine to spy on this method and check to make sure that it is being called with the correct options depending upon the changes we make to the model.

The tests use an array of events; they are declared in a `beforeEach` clause at the beginning of the spec file. The `eventSources` array is an array of event sources.

The following code snippet is an on-event source. This is what builds the calendar and is the source of what drives the calendars. The tests use these events to prove that their functionality is working as expected.

```
scope.events = [
  {title: 'All Day Event',start: new Date(y, m, 1),url:
    'http://www.angularjs.org'},
  {title: 'Long Event',start: new Date(y, m, d - 5),end: new
    Date(y, m, d - 2)},
  {id: 999,title: 'Repeating Event',start: new Date(y, m, d - 3,
    16, 0),allDay: false},
  {id: 999,title: 'Repeating Event',start: new Date(y, m, d + 4,
    16, 0),allDay: true}];
$scope.eventSources = [$scope.events];
```

Testing the calendar's initialization and MVC functionality

Some would say that the following test does not need an `apply` function to work properly, but it is required since the `calendar` directive relies on the `watch` function for its initialization:

```
beforeEach(function(){
  spyOn($.fn, 'fullCalendar');
  $compile('<div ui-calendar="uiConfig.calendar"
    ng-model="eventSources"></div>')(scope);
  scope.$apply();
});
```

The following test is checking to prove that we called the initial `fullCalendar` method with the correct objects. The objects are based on the events that we have in our `$scope.events` array.

```
    it('should set up the calendar with the correct options and events',
    function () {
      expect($.fn.fullCalendar.mostRecentCall.args[0].eventSources[0]
        length).toBe(4);
      expect($.fn.fullCalendar.mostRecentCall.args[0]
        eventSources[0][0]title).toBe('All Day Event');
      expect($.fn.fullCalendar.mostRecentCall.args[0]
        eventSources[0][0].url).toBe('http://www.angularjs.org');

    });
```

Test to make sure that when an event is added to the calendar, everything is updated with the new event. This event's purpose is to prove that whenever the view model changes, the directive automatically calls the correct function accordingly. The `$apply` function is called to fire a digest on the calendar's scope, which will fire its `watch` function, as shown in the following code:

```
    it('should call its renderEvent method', function () {
        expect($.fn.fullCalendar.mostRecentCall.args[0].eventSources[0].
        length)
          .toEqual(4);
        expect($.fn.fullCalendar.callCount).toEqual(1);
        scope.addChild(scope.events);
        scope.$apply();
        expect($.fn.fullCalendar.callCount).toEqual(2);
        expect($.fn.fullCalendar.mostRecentCall.args[0])
        toEqual('renderEvent');
        scope.addChild(scope.events);
        scope.$apply();
        expect($.fn.fullCalendar.callCount).toEqual(3);
        expect($.fn.fullCalendar.mostRecentCall.args[0])
        toEqual('renderEvent');
    });
```

The next test proves that the calendar works with different types of events. It works with an object called `calEventsExt`, which is an events object in the extended form. This is shown as follows:

```
scope.calEventsExt = {
  color: '#f00',
  textColor: 'yellow',
  events: [ //eventObjects ]
};
it('should make sure the calendar can work with extended event
  sources', function () {
  scope.eventSources.push(scope.calEventsExt);
  scope.$apply();
  var fullCalendarParam = $.fn.fullCalendar
    mostRecentCall.args[0];
  expect(fullCalendarParam).toEqual('rerenderEvents');
});
```

This section was about proving simple interactions with the `calendar` directive that are rendered in the DOM. We mimicked interactions that take place in production environments, as well as made sure that the initialization goes according to plan in different circumstances.

Writing the fullCalendar directive

Writing the `fullCalendar` directive is similar to writing other directives that utilize third-party libraries. There is more code involved with the `calendar` directive, but the truth remains that all third-party directives rely on calling the third-party initialization function on the element provided by the directive's compile and link functions.

The `calendar` directive is no different in these circumstances. The calendar calls its `fullCalendar` initialization method inside of a `watch` function to check for changes to its `options` object. This is a recommended technique as it provides two methods of functionality. The directive is initialized upon the first digest, and whenever any of the options change, the directive automatically recreates itself with the new options.

The `calendar` directive follows normal directive creation practices, but it also uses advanced techniques to attach functions, which it relies upon to update itself correctly. These methods will be discussed in depth. The full source can be found in the AngularUI repository at `https://github.com/angular-ui/ui-calendar`.

```
angular.module('ui.calendar', [])
.constant('uiCalendarConfig', {})
.controller('uiCalendarCtrl', function(//changeWatcher logic){})
.directive('uiCalendar',
  ['uiCalendarConfig',function(uiCalendarConfig) {


  return {
    restrict: 'A',
    scope: {eventSources:'=ngModel'},
    controller: 'uiCalendarCtrl',
    link: function(scope, elm, attrs, controller) {

      //These are the local variables. The eventSourceWatcher and
        eventsWatcher are created by calling a controller
        constructor function. These objects used to set custom
        eventWatcher functions inside of the calendar's closure.
      var sources = scope.eventSources,
      sourcesChanged = false,
      eventSourcesWatcher = controller.changeWatcher(sources,
        controller.sourcesFingerprint),
      eventsWatcher = controller.changeWatcher(controller
        allEvents, controller.eventsFingerprint),
      options = null;
      //The getOptions functions resets the local calendarSettings
        options objects which is used to update the calendar upon
        initialization and any time these options change.
      function getOptions(){
        var calendarSettings = attrs.uiCalendar ?
          scope.$parent.$eval(attrs.uiCalendar) : {},
          fullCalendarConfig;

          fullCalendarConfig = controller
            getFullCalendarConfig(calendarSettings,
            uiCalendarConfig);

          options = { eventSources: sources };
          angular.extend(options, fullCalendarConfig);

          var options2 = {};
          for(var o in options){
            if(o !== 'eventSources'){
            options2[o] = options[o];
            }
          }
        return JSON.stringify(options2);
        }

      scope.destroy = function(){
```

```
  if(attrs.calendar) {
    scope.calendar = scope.$parent[attrs.calendar] =
      elm.html('');
    } else {
    scope.calendar = elm.html('');
    }
  };

  scope.init = function(){
    scope.calendar.fullCalendar(options);
    };

  eventSourcesWatcher.onAdded = function(source) {
scope.calendar.fullCalendar('addEventSource', source);
sourcesChanged = true;
};

eventSourcesWatcher.onRemoved = function(source) {
  scope.calendar.fullCalendar('removeEventSource', source);
  sourcesChanged = true;
  };

eventsWatcher.onAdded = function(event) {
  scope.calendar.fullCalendar('renderEvent', event);
  };

  eventsWatcher.onRemoved = function(event) {
    scope.calendar.fullCalendar('removeEvents', function(e)
      { return e === event; });
    };

  eventsWatcher.onChanged = function(event) {
    scope.calendar.fullCalendar('updateEvent', event);
    };

  eventSourcesWatcher.subscribe(scope);
  eventsWatcher.subscribe(scope, function(newTokens,
    oldTokens) {
    if (sourcesChanged === true) {
      sourcesChanged = false;
      // prevent incremental updates in this case
      return false;
    }
  });
```

```
            scope.$watch(getOptions, function(newO,oldO){
              scope.destroy();
              scope.init();
            });
          }
        };
    }]);
```

The main points that should be expressed about the `calendar` directive are its organization techniques, its subscription process, how it wraps functions in `apply` method, and how it makes the `calendar` object public on its defining scope.

The `calendar` directive uses the controller to help organize its functions in a manner that reduces complexity. This is mainly to maintain high readability and allow for reusability. Now that the calendar has its own controller, any other directive can share its state and help alter the events as well, which opens up many windows for extending an application's functionality.

The `calendar` controller returns a subscription object that is used to set the `onAdded`, `onChanged`, and `onRemoved` functions. This subscription process allows the `calendar` controller to only contain functionality that is specific to its creation and teardown that renders it in the DOM.

Subscription-based functions allow for the calendar to remain very testable and easily extendable by developers who do not know about the change watcher functionality.

One of the requirements that the `calendar` controller has is that it needs to be able to call any `fullCalendar` method from a controller method. To accomplish this, the `calendar` controller allows for the user to define a calendar attribute to the directive with some string. This string will then be used as the key when setting the `calendar` object, returned by `fullCalendar`, to the `$parent` scope. Since the directive utilizes an isolate scope, we can assure that it has a parent scope whenever it is being initialized. This allows for the developer to choose what the name of the calendar should be in the markup.

The `calendar` controller makes it possible to add functions to its option set by means of passing in the `$scope` functions as values. These functions are called by the `calendar` controller outside of the AngularJS digest cycle, and in turn, each function should call the `$apply` method.

The `calendar` controller knows which functions to call the `apply` method on. It is set inside of the controller whenever the `calendar` controller gets its options. Each option is traversed over, and if it is a function, it is wrapped in a timeout function. The timeout function is the safest way to call an `$apply` method outside of AngularJS. This is because it is an asynchronous method that ensures that a digest will be fired on the JavaScript execution thread, as shown in the following code:

```
wrapFunctionWithScopeApply = function(functionToWrap){
  var wrapper;

  if (functionToWrap){
    wrapper = function(){
      // This happens outside of angular context so we need to
        wrap it in a timeout which has an implied apply.
      // In this way the function will be safely executed on the
        next digest.

      var args = arguments;
      $timeout(function(){
        functionToWrap.apply(this, args);
        });
      };
    }

  return wrapper;
};
```

Now, the developer can write methods for the `fullCalendar` directive, and they do not need to worry about whether they should call the `apply` method or not. When writing directives, it is recommended that the directive take care of calling `apply` method if it is ever needed, as shown in the following code:

```
/* alert on Resize */
$scope.alertOnResize = function(event, dayDelta, minuteDelta,
  revertFunc, jsEvent, ui, view ){
    $scope.alertMessage = ('Event Resized to make dayDelta ' +
      minuteDelta);
};
```

The `fullCalendar` directive has been created and tested properly. We have gone over all of the details behind its internals except for the controller itself. The controller code is more advanced and not specifically relevant to third-party directives in general. It can be viewed in the AngularUI GitHub repository.

All of the initial requirements have been met for `calendar`, so now it can be installed on any production AngularJS application fluently and effectively. The live example can be found at `http://embed.plnkr.co/UGaI1FytAbIbqsfJPTzU/preview`.

Summary

There are many different ways to incorporate third-party libraries into AngularJS applications. These methods vary depending on the type of library that is being incorporated into the application. If the third-party library's main purpose is to manipulate and control the view layer, it should be made into a directive. This allows for AngularJS to control the timing of the libraries' initialization calls.

It is very common to use an isolate scope when working with third-party directives. This allows for directives to easily use the common `ngModel` attribute and ensures that they do not interfere with other directives. Isolate scopes also give directives extra data binding features for free.

Every third-party library directive that is written should have the proper test suites that make sure that its proper functions are being called at the given time and that it is being created in the DOM accordingly. Writing tests for third-party directives can be different from writing native AngularJS directive tests. This usually calls for spy methods; they allow the tests to ensure that the third-party library is calling the correct functions at the correct time.

Since many useful directives have already been written and are being maintained on GitHub, it's fairly easy to find any type of directive and just start using it right away. The real bonus points are in knowing how to create these without the help of the community. This is a very large confidence booster because anything is possible for the application now, and it does not have to depend on others in the open source community.

4
Compiling the Advantages

The directives made so far in this book have been fairly trivial. While they utilized many different available directive definition object properties, they have not really been focused on dynamic DOM manipulation. The purpose of this chapter is to showcase how a directive can dynamically create and destroy new HTML that is compiled and linked to the AngularJS DOM tree.

The `$compile` service offered by AngularJS is used in a few places throughout the AngularJS core. The initial bootstrap, which AngularJS uses to jumpstart an application, is just one big `compile`. From there on, any directive that utilizes `compile` will be adding more content to the already compiled DOM.

> Using the `$compile` service is the equivalent of calling `angular.bootstrap` on a given element.

The compiler searches for interesting factors about the semantic markup, such as curly brackets or restricted directives. Once the interesting objects are found, their templates are gathered and their `compile` functions are run. The service returns a composite `link` function that serves as the glue between the DOM and the model.

> The `$compile` service is needed in advanced situations. Some good examples of directives that compile DOM are `ngInclude` and `ngView`.

The `$compile` function can be used anywhere inside of an application but should only be used inside of directives. Some developers say that `$compile` can be used inside of controllers without affecting an application. This is true in many cases, but it is always safe to do any type of DOM manipulation inside of the `link` function. This is so that AngularJS has the opportunity to consolidate all of the application's data and we are guaranteed that all elements and attributes will be set.

A controller's `factory` function is always initialized before the directive's `link` function, which means that it is not safe to do DOM manipulation. Filters should also never use the `$compile` function. A filter is initialized on every digest, which means that an element that is compiled in the `$digest` cycle would be compiled on every single digest. This would really go against all Angular Zen philosophies.

The `compile` provider is the one that does the bulk of the work for the `$compile` service. This is the closure that takes care of all DOM traversal inside an AngularJS application. Collecting directives and applying their definition objects is the `compile` provider's main concern. This is a very detailed and advanced process, which includes many different steps. The main high-level steps are as follows:

- Collecting the directives
- Applying each conditional definition object
- Recursively running each directive's `compile` function in the order of priority

The `compile` definition function's main purpose is to optimize the creation of the directive and to return the `link` function. The optimization comes into play by only compiling an element once, no matter how many times its `link` function is run.

The `link` function that is returned is a composite pre link and post link function. What does this mean? There are two stages in the linking phase. The first linking phase is performed after the `scope` object has been created for the element, but not its children. The second linking phase is the post link function. During the post link function's execution, all children DOM elements and their scopes are guaranteed to be existent. Because of this guarantee, most of the logic is placed inside of a directive's `link` function.

The composite `link` function is an array of `link` functions built from a single element's directives. The composite `link` array will run each function synchronously, one after the other. This order is specified by the priority of the directive and is the opposite for post link functions. Once this process is done, the element will be attached to the DOM and will be fully interpolated.

Common use cases for compiling the DOM

There are a given set of common requirements for using the `$compile` service in an AngularJS application. The following requirements have been collected by researching many different directives that use the `$compile` service and from the interjections made about how AngularJS provides the ability to transclude DOM:

- Directives that use transclusion
- Recursive directives
- Template-based directives
- DOM manipulation testing

This list breaks down the different possibilities to manually compile DOM. The following sections will go over transclusion, recursion, and template-based directives. DOM manipulation testing is used throughout the entire book, so we will not cover this in detail.

> The research was taken from the AngularJS AnugularUI repositories, various Stack Overflow posts, and the AngularJS irc channel.

Using transclusion in a directive

The following quote can be found in the documentation of transclusion (visit `http://en.wikipedia.org/wiki/Transclusion`):

> *"In computer science, transclusion is the inclusion of a document or part of a document into another document by reference."*

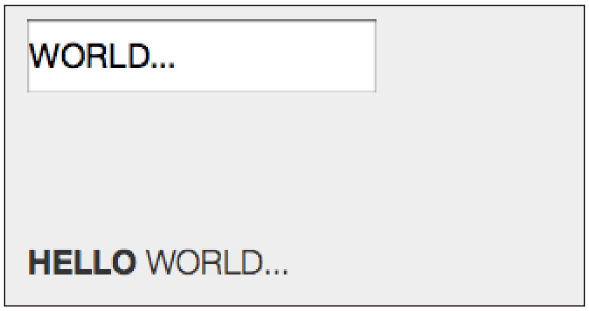
AngularJS uses transclusion to separate DOM from its original container and place it in a new container. The separated DOM is stored in memory and wrapped in a `closure` function, which returns a new linking function. The returned linking function allows access to the prebound template. Optionally, the binding can be overridden by any scope passed into the function as the first parameter.

What does this mean? Any semantic markup can be yanked from the DOM and placed anywhere inside of the directive's DOM. Take for example the following simple markup:

```
    <super-component>
      <span> {{world}} </span>
    </super-component>
Here is the definition object for the superComponent.
.directive('superComponent', function () {
    return {
      restrict:'E',
      template: '<div><strong>{{hello}}</strong>
        <span ng-transclude></span></div>',
      transclude: true,
      scope: true,
      link :function(scope, element, attrs){
        scope.hello = ' HELLO!';
        }

      };
});
```

This directive uses `ng-transclude` to apply its transclusion. The `ng-transclude` function tells the application where to put all of the transcluded DOM elements. This is the simplest way to utilize transclusion in a directive.

The final DOM rendering depends upon what `$scope.world` equals in its defining `$scope`. Let's say that we have some input with an `ng-model="world"` attribute and its value equals `WORLD`.... The example can be found at `http://jsfiddle.net/joshkurz/JNndE/`. The final DOM rendering would look as follows:



Unveiling transclusion

At a high level, transclusion is a simple idea, and AngularJS offers simple functionality to allow this process to take place anywhere in the DOM. The lower-level details of transclusion are a bit more complicated and require an in-depth understanding of the `compile` function.

Transclusion implicitly calls the `compile` function, which means that anytime a directive transcludes HTML, it will compile it as well. Transclusion uses the exact same `compile` function as the injectable `compile` service. This is how transclusion offers you the ability to create sibling relationships inside whatever scope it is defined in. This means that any transcluded element's binding will always stay intact, unless further specified by the user.

The element that is transcluded by a directive can vary depending upon the instruction given by the directive's definition object (that is, `element` or `true`). The need for either type of transclusion depends upon the use case.

[The ngRepeat function uses element transclusion to copy its contents and compile them once.]

Transclusion is also a wonderful optimization technique. The link function of a compiled element can run more than once in the life cycle of an AngularJS application. In contrast, the compile function only runs once. This means that transclusion is the perfect candidate for a directive whose purpose is to duplicate DOM over and over again.

This example uses ngRepeat to create the superComponent directive.

The JavaScript code is as follows:

```
.controller('SuperCtrl', ['$scope', function ($scope) {
      $scope.numbers = [1,2,3,4,5,6,7,8,9,10];
      }]);
```

The HTML code is as follows:

```
<div ng-controller="SuperCtrl">
  <div ng-repeat="number in numbers">
    <super-component>
          <span> {{number}}</span>
    </super-component>
  </div>
</div>
```

What will this block of HTML do? How many times will superComponent be compiled and linked in the DOM? These questions are essential to understanding the nature of transclusion.

Let's cut straight to the chase. The superComponent directive is only compiled once in this example. Its link function will run ten times. This is because of how ng-repeat treats the superComponent directive. The entire element is stored and compiled in memory during the application's bootstrap. This means that anytime a number is added to the number's array, ngRepeat will just call its transcluded link function to place the superComponent directive after the previous element in the DOM. In this case, the ten superComponent directives will be appended to div that ngRepeat is instantiated on.

The ngRepeat function uses the transclude function that is passed into the link function to insert the newly interpolated DOM into the element. This is an advanced way to accomplish the DOM insertion. The main reason to use this process is to overwrite the scope of the element being transcluded. In the ngRepeat function's case, it is a must-have functionality because every iteration of ngRepeat requires its own fresh child scope. Let's super charge our superComponent by using this advanced technique to accomplish the transclusion desired.

Here is the souped-up version of the `superComponent` directive. Let's assume that we are still calling this directive inside of `ngRepeat` that runs off an array of ten numbers, and each child scope created by `ngRepeat` has an associated number value. Refer to the following code:

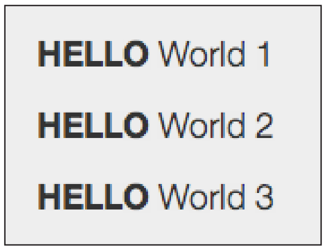
```
.directive('superComponent', function () {
    return {
      restrict:'E',
      template: '<div><strong>{{hello}}</strong></div>',
      transclude: true,
      replace: true,
      scope: true,
      compile: function(tElem,tAttrs){
        return function(scope, element, attrs, ctrl, transclude){
            scope.hello = 'HELLO ';
            scope.number = "World " + scope.number;
            transclude(scope, function(clone){
                element.append(clone);
                });
            }
          }
        };
})
```

The main point of interest here is the `transclude` object passed into the `link` function as the fifth parameter. This is the `link` function that has been compiled and prebound to its original scope. Now, we want to add something extra to the `child` scope's declaration of the number variable. So, in the link function, we prepend the number with the string `World`, and then pass the directive's scope into the `transclude` function. The `transclusion` function interpolates the precompiled element based on the scope that is passed in, and then calls the attach function, which is its second parameter.

The first scope parameter is optional and does not have to be used. The `transclusion` function will use its original scope if its first parameter is a function.

The clone is then appended to the element. The directive also sets the replace property on the definition object because we want the transcluded span element to be placed directly next to the template. If the replace property is not set, then the `transclude` element will be appended next to the `superComponent` DIV in the DOM.

The example can be found at `http://jsfiddle.net/joshkurz/JNndE/2/`. The final result is 10 DIV elements with `Hello World + {{number}}` as their preinterpolated text, shown as follows:



Transclusion can also be accomplished in a custom fashion without using the `transclude` property on a definition object. This can be useful in many different advanced use cases.

Creating recursive directives

A `recursive` directive calls itself however many times it is needed based upon some `data` object. The object can be organized in any way, but most commonly it is based on a parent-child model. This means that there are nested objects within nested objects that will make up the overall structure of the finally rendered DOM.

The `recursive` directives are needed for various use cases. The most classic cases for a `recursive` directive are a drop-down menu or a nested `comment` directive. Any directive that builds a similar template based on a `data` model and has a need to call itself is a perfect candidate for a `recursive` directive. AngularJS makes it very simple to allow directives that call themselves. However, the following are a few caveats that we must go over before we start describing how to make recursive directives:

- Always compile the contents of the element so that we do not get into an infinite loop
- The directive can use transclusion, but the transcluded nodes need to use `ngIf` to determine whether the element should be rendered in the DOM

In computer science, the `recursive` functions are known to be expensive and dangerous if used incorrectly. They are also known to be wonderfully useful, and the only solution to some very difficult problems when used correctly. This is no different when working with `recursive` directives.

Each individual `recursive` directive needs to compile its children and so on and so forth. This will render an entirely new DOM structure with very little code. Let's go over an example `tree` directive, and explain how to accomplish this functionality in a couple of different ways.

The custom recursive tree directive

A `tree` directive is very useful when creating nested navigation that changes depending on different variables. The DOM nodes that make up the navigation are created based off of a data model that represents the structure of the tree. The structure has a root element which has `children`, which can have `children`. We are going to show a couple of different ways to achieve this recursive behavior in AngularJS.

All of the examples used in this section are going to be working with this data model. Refer to the following code:

```
{ name : "Super Grandpa",

  children: [{
      name : "Super Man",

      children: [{
          name : "Super Boy",

          children: []
          },
        {
          name : "Super Girl",

          children: []
          }]
      }]
  }
```

Using transclusion and a templateUrl with the treeNode directive

The `treeNode` directive takes any DOM element that is specified in the semantic markup and recursively assigns new nodes to the DOM based off of the original transcluded element. This directive accomplishes this by using `ng-transclude` combined with a template fetched from the template cache.

A template cache is used in all of the directives in this book that use the `templateUrl` function, instead of making an HTTP request for the template. This is for performance and testing reasons.

Testing the treeNode directive

The `treeNode` directive's requirements are to render transcluded DOM as many times as necessary based off of the data model given. Refer to the following code:

```
describe('Creating The Tree Node With A Template Directive',
  function () {
    var treeNode;

    // We are compiling the element before every test, so that we
      have it in memory and we can make sure that it is
      initialized without errors every time.
    beforeEach(function(){
        treeNode = $compile('<div tree-node-template
          family="node">' +
          '<a href="" ng-click="family.show = !family.show">{{
            family.name }}</a>' +
          '</div>')(scope);
        scope.$apply();
        expect(treeNode).not.toBe(undefined);
        });

    //This test proves that the treeNode directive creates 4 a
      tags inside of the element.
    it('should Expect the tree node to have the correct amount of
      a tags', function() {
      expect(treeNode.find('a').length).toBe(4);
    });

    //This test proves that the text of the treeNode directive's a
      tags is correct.
    it('should Expect the tree node to have the correct text for
      each child a', function() {
      var treeANodes = treeNode.find('a');
      expect($(treeANodes[0]).text()).toBe('Super Grandpa');
      expect($(treeANodes[1]).text()).toBe('Super Man');
      expect($(treeANodes[2]).text()).toBe('Super Boy');
      expect($(treeANodes[3]).text()).toBe('Super Girl');
      });

    //This test proves that the directive is working with dynamic
```

```
         scope changes accordingly.
       it('should Expect the tree node to update itself when the
         nodes change their visibility', function() {
         expect(treeNode.find('a').length).toBe(4);
         scope.node.children[0].show  = false;
         scope.$apply();
         expect(treeNode.find('a').length).toBe(2);
         scope.node.show  = false;
         scope.$apply();
         expect(treeNode.find('a').length).toBe(1);
         });
     });
```

All of these tests in collaboration with each other prove that the `treeNode` directive creates the DOM elements as expected. We do not need to test any further since the directive uses transclusion and has the ability to use any type of DOM element. In order to prove that it is recursively assigning DOM, the initial tests should suffice.

The treeNodeTemplate directive

The `treeNodeTemplate` directive stores its contents in memory during the compile phase. This allows the directive to be created and recreated without the need to compile itself again. This copies the same functionality that transclude uses. We cannot utilize the pure `transclude` function because AngularJS cannot see the parent transcluded directive during the compile phase and will throw errors. This is very important when working with the `recursive` directives because by saving the template element in the memory, we are being as efficient as the normal `transclude` process and we are able to accomplish the given requirements. Refer to the following code:

```
angular.module('treeNodeTemplateModule',
  ['directives/treeNodes/treeNodeTemplate.tpl.html'])
.directive("treeNodeTemplate", function($compile) {
    return {
      restrict: "EA",
      transclude: true,
      scope: {family: '='},
      templateUrl: 'directives/treeNodes/treeNodeTemplate
        tpl.html',
      compile: function(tElement, tAttr) {
        var contents = tElement.contents().remove();
        var compiledContents;
        return function(scope, element, attrs, ctrl, transclude) {
            //setting the value to true so the tree will always
              start open. This could be a configuration of some
```

```
                sort.
              scope.family.show = true;

              if(!compiledContents){
                compiledContents = $compile(contents, transclude);
                }

              compiledContents(scope, function(clone) {
                element.append(clone);
                });
            };
          }
        };
    });
```

The `transclude` function is passed into the `compile` function so that every recursively compiled element has a reference to the original transcluded element. This is the key to allow the recursion to work as expected because now the directive and every subsequent `child` directive have a reference to their defining scope's context. The example can be seen here `http://plnkr.co/edit/E9qJmb?p=preview`.

The treeNode directive using only transclusion

The next directive that we will create is going to do the exact same thing as the prior directive, except that it will not use a template at all. This allows you to have an even more flexible semantic markup, which gives the directive more use and makes it more valuable to an application. It is also more efficient because it only has to compile itself once by leaning on `ngRepeat` and its template DOM. This optimization shows that using pure transcluded content to accomplish recursion can be slightly better than using a template.

The tests that we have written do almost the same thing as the `treeNodeTemplate` directive. The only difference is that the nodes are not initially open. This means that the order in which the tests check DOM elements is the opposite. This makes it apparent that the recursion is working as intended on both directives without duplicating the test logic.

Testing the treeNode directive

The following is the `treeNode` template that is being compiled to make this test generate its given elements:

```
treeNode = $compile('<div tree-node-no-template>' +
            '<ul id="testList" class="list-group">' +
              '<li class="list-group-item">' +
                '<a href=""><span class="btn" ng-show="node.children
                  && !node.show" ng-click="node.show=!node
                  show">[+]</span>' +
                '<span class="btn" ng-show="node.children &&
                  node.show" ng-click="node.show=!node
                  show">[-]</span>' +
                '{{node.name}}</a>' +
              '</li>' +
              '<li ng-if="$parent.node.show" class="list-group-item"
                ng-repeat="node in node
                children" ng-transclude></li>' +
            '</ul>' +
          '</div>')(scope);
```

The next test proves the following two most important requirements:

- The `treeNode` function is rendering the DOM correctly
- The `treeNode` function is updating itself on any change in the model

Refer to the following code:

```
it('should Expect the tree node to update itself when the nodes
  change their visibility', function() {
    expect(treeNode.find('a').length).toBe(1);
    scope.node.show = true;
    scope.$apply();
    expect(treeNode.find('a').length).toBe(2);
    expect($(treeNode.find('a')[1]).text()).toBe('[+][-]Super
      Man');
      scope.node.children[0].show = true;
      scope.$apply();
      expect(treeNode.find('a').length).toBe(4);
      expect($(treeNode.find('a')[2]).text()).toBe('[+][-]Super
        Boy');
      expect($(treeNode.find('a')[3]).text()).toBe('[+][-]Super
        Girl');
});
```

This test uses an `a` tag as its selector. We are using a jQuery method to find the child `a` elements inside of the compiled `treeNode` function. Each `treeNode` tag is expected to contain two spans and the node name as the text. These spans are written in the semantic markup that would otherwise be written in the source code of an application. These elements could be anything, but the `a` tag was chosen to wrap all of the DOM elements that are being recursed because it is easy to track and test with jQuery.

It is ok to have special conditions in tests as long as they do not alter the environment in which the directive will operate. This is another way of saying that you shouldn't alter a directive's source just so a test will pass.

The treenodeNoTemplate directive

The `treeNodeNoTemplate` directive showcases a different way to provide recursive directive behavior. This directive is more flexible and dynamic than its prior counterpart only because it does not use a template. Using just transclusion allows you to perform more dynamic use cases rather than creating multiple templates based upon each use case. The example can be found at `http://plnkr.co/edit/QD8KfOhwy2xOoxb0NN6g?p=preview`.

Refer to the following code:

```
angular.module('treeNodeNoTemplateModule', [])
.directive('treeNodeNoTemplate', ['$compile', function($compile) {
  return {
    restrict: 'EA',
    scope: true,
    compile: function(element, attr) {
      var $template = element.clone().contents();
      element.html(''); // clear contents

      var linkFn = $compile($template, function(scope,
        cloneAttachFn)   {
        return linkFn(scope, cloneAttachFn);
        });

      return function($scope, $element, $attr) {

        linkFn($scope, function(contents) {
          $element.append(contents);
          });
      };

    }
  };
}]);
```

The `treeNodeNoTemplate` directive does almost exactly the same thing as its sibling `treeNodeTemplate` directive that uses a template. The main difference is that this directive is much more optimized than its templated sibling. This is because the `compile` and `link` functions are only called once during the entire life cycle of the directive. No matter how the model changes, the `recursive` directives do not have to make subsequent `link` function calls. This is because the directive relies on `ngInclude` and `ngRepeat` to do all of its dirty work. All the directive has to do is create itself in the DOM and link its scope to its template. The main gotcha here, that is not apparent, is that `transclude` is actually not being called on the definition object and no `transclusion` function is being passed into the `link` function. Instead, the directive creates its own custom `transclusion` function in the compile phase, which in this case is being called the `linkFn` function. This `linkFn` function takes the place of the `transclude` function found in the prior directive.

What is the significance of using a custom `transclusion` function?

Because of the recursive nature of this directive, the `transclusion` option cannot be used. If a `transclude` property is added to the directive and it is used instead of the custom `linkFn` function created, then AngularJS will throw an error.

These two directives accomplish some wonderful recursive techniques with very little effort from the developer's standpoint. However, note that this same `treeNode` directive can be accomplished with just a combination of `ngInclude` and `ngRepeat`. The example can be found at `http://angulardirectives.joshkurz.net/dist/#/treeNodes`.

Compiling templates and their many values

Using a dynamic template is not the same as using the `template` or `templateUrl` function available in the directive definition object. This is because both the functions do does not have access to the scope and are actually a hardcoded value in the source or a variable returned by a function. The `ngInclude` directive take this a step further and allows custom template compilation based upon a `scope` variable. Today, `ngInclude` interpolates its source value and updates its own DOM contents whenever the source changes. This is common practice among directives that deal with compiling templates, and we are going to focus on this dynamic nature of AngularJS. This means that our directives will have access to the scope when they determine which template to compile. There are many advantages to using this method, but there are some disadvantages as well.

It is not always in the directive's best interest to interpolate the URL it is going to use as a template.

The disadvantages of interpolating a `templateUrl` directive or template in the `link` function are related to performance and security. The `link` function is fairly expensive, and if not used carefully, it can run many times and cause memory leaks. This also leads to performance issues that can cause lag and other related issues. The security risks involve directives that could potentially allow HTML injection attacks. HTML injection attacks can be minimized by using the `$sce` service of AngularJS. The `$sce` service is a security service that makes sure that interpolated strings pass variable high-level injection attacks.

[Injection attacks and other client-side security issues are always best thwarted by securing the server to its maximum capacity.]

The advantages vastly outweigh the disadvantages, and if the requirements call for the use of dynamically defined templates, then they must be implemented. Let's look at the directive that utilizes dynamic templates to render its DOM.

Introduction to the media player directive

There are many different types of video players in the market today, which come in many different shapes and sizes. We are going to create one video player that will work with multiple libraries—*One Media Player to Rule Them All*. The directive has a strong focus on templates rather than on exposing a public API. This will allow the `video player` directive to remain as flexible as possible.

The templates for the video player can vary greatly and can range from simple to very complex. The point of creating the directive is to allow any video template to use any video player library or a pure HTML5 video player template that uses no third-party library to control its functionality.

Requirements for the media player directive

The requirements for the `media player` directive open the doors for many options regarding video players. These requirements are very general and force the directive to allow the use of multiple libraries or multiple templates. This dynamic-template functionality is only possible because of the `$compile` service and its ability to generate live interpolated DOM. The following are the requirements for the `media player` directive:

- Needs to work with multiple third-party libraries
- Should be able to be a pure HTML5 video player
- Should update itself if its template changes

These requirements call for a generic directive that can work with almost any type of media-element library. This approach will allow the `mediaPlayer` directive to create any number of video templates that can be used by any number of third-party libraries to create the expected media DOM elements. The ability to conditionally use any media library based upon a `scope` variable creates many possible use cases.

Testing the media player directive

To test this directive, we need to make sure that it can be created for all of its possible scenarios. The different test cases created are a combination of different templates and media types. These tests just make sure that there are no errors thrown during the compiling of the directive and make sure the correct third-party library being called.

[There are more tests for the `bbMediaPlayer` directive in the Black Belt repo. These were chosen to prove that the directive does in fact work with multiple libraries and dynamic templates.]

The first test uses the third-party media library, which is called `mediaelement.js`. This test spies on the `mediaelementplayer` function so that we can make sure that it is being called on the media element. The only odd piece about this test is the use of the `$timeout service` function. Refer to the following code:

```
it('create a mediaelement object when media-type and template url
  are interpolated strings and call the mediaelement method with
  the correct options', function() {
    spyOn($.fn, 'mediaelementplayer');
    scope.mediaType = "mediaelementplayer";
    scope.template = "directives/mediaelement/mediaelement.tpl.html";
    scope.$apply();
    var mediaPlayer = $compile('<div bb-media-player
      media-type="{{mediaType}}" video-config="goodVideoObj"
      template-url="{{template}}"></div>')(scope);
      scope.$apply();
      $timeout.flush();
      expect($.fn.mediaelementplayer.callCount).toBe(1);
      expect($.fn.mediaelementplayer.mostRecentCall
        args[0]).toBe(scope.goodvideoObj);
});
```

This test proves that `bbMediaPlayer` works with the `flowplayer` method, which is another third-party library. This is the same test as the previous one, except that it uses a different template and media type. Refer to the following code:

```
it('create a flowplayer object when media-type and template url
  are interpolated string and call the flowplayer method with the
  correct options', function() {
    spyOn($.fn, 'flowplayer');
    scope.mediaType = "flowplayer";
    scope.template = "directives/mediaPlayer/flowplayer.tpl.html";
      scope.$apply();
      var mediaPlayer = $compile('<div bb-media-player
        media-type="{{mediaType}}" video-config="goodVideoObj"
        template-url="{{template}}"></div>')(scope);
      scope.$apply();
      $timeout.flush();
      expect($.fn.flowplayer.callCount).toBe(1);
      expect($.fn.flowplayer.mostRecentCall.args[0])
        toBe(scope.goodvideoObj);
});
```

The following test proves that `mediaPlayer` works with a pure HTML5 template as well:

```
it('create a pure HTML5 media element', function() {
    expect(function(){
    var mediaPlayer = $compile('<div bb-media-player
    video-config="goodVideoObj" template-url="directives/
    mediaPlayer/pureHtml5Player.tpl.html"></div>')(scope);
      scope.$apply();
      $timeout.flush();
}).not.toThrow() });
```

Writing the media player directive

The `bbMediaPlayer` directive is a powerful tool that allows multiple types of media content to be distributed to its users. There are many benefits of using a directive that can accomplish a wide range of use cases. To accomplish this wide range of use cases, the `bbMediaPlayer` directive uses the `$compile` service to dynamically create DOM and interpolate scope variables with the template DOM.

The `bbMediaPlayer` directive is installed and initialized in the DOM as follows:

```
<div media-player media-type="{{mediaType}}"
  video-config="activeVideo" template-
  url="{{currentFlowplayer}}"></div>
```

This directive points to a template that is compiled in the `link` function, allowing the interpolated string to be read and linked to the defining `isolated` scope. The `activeVideo` object will be available on the `isolate` scope object inside of the template for any other media type. The `mediaType` attribute states which media library should be called on the element.

One of the main advantages of working with templates that compile DOM is that once they are written and fully tested, they can be utilized in an infinite amount of ways. Developers of many different skill sets can write templates, which means that an application's development and progress does not rely on experienced developers. Directives of this nature allow rapid development to ensue.

The following is an example of a simple template:

```
<div class="pureHTML5Player">
  <video autoplay>
    <source type="video/mp4" ng-src="{{trustSrc
      (videoConfig.playlist[0], '.mp4')}}">
    <source type="video/ogg" ng-src="{{trustSrc
      (videoConfig.playlist[0], '.ogv')}}">
  </video>
</div>
```

This template is as simple as it gets for video elements. The HTML5 valid browsers will pick up on the video tag and render the video as expected. This video will not have any controls associated with it, but these can be written or a third-party library can be used to accommodate the missing controls.

The directive itself is quite simple and utilizes basic logic to accomplish its requirements. Refer to the following code:

```
angular.module('AngularBlackBelt.mediaPlayer',
  ['directives/mediaPlayer/flowplayer.tpl.html'
  ,'directives/mediaPlayer/flowplayerSlideshow.tpl.html',
  'directives/mediaPlayer/pureHtml5Player.tpl.html'])
.directive('bbMediaPlayer', ['$sce', '$compile', '$templateCache',
  '$timeout', function($sce, $compile, $templateCache, $timeout) {
      return {
        restrict: 'A',
        scope: {
          videoConfig: '='
          },
        compile: function(tElem,tAttrs){

          if (!tAttrs.templateUrl){
            throw new Error('Must Give media-player a templateUrl
              to look for.');
            }

          return function(scope, element, attrs) {

            if (typeof scope.videoConfig !== 'object'){
              throw new Error('videoConfig must be an object');
              }

            var newElement,
            mediaPlayer;

            function getConfigurations(){
              scope.videoConfig.templateUrl = attrs.templateUrl;
              return scope.videoConfig;
              }

            scope.trustSrc = function(filePath,ext) {
              return $sce.trustAsResourceUrl(filePath + ext);
              };

            function init(){
              newElement = $compile($templateCache.get
                (attrs.templateUrl).trim())(scope);
              element.html('').append(newElement);
              $timeout(function(){
                if(attrs.mediaType && attrs.mediaType !== ''){
                  mediaPlayer = newElement[attrs.mediaType]
                    (scope.videoConfig.options);
                  }
                });
            }

          scope.$watch(getConfigurations, function(newV,oldV) {
            init();
            },true);

        scope.$on('$destroy', function(node){
            if(mediaPlayer.remove){
              mediaPlayer.remove();
              }
            mediaPlayer = null;
            element.html('');
            });
```

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

Breaking down the media player directive

The `mediaPlayer` directive is broken down into three separate important pieces, which together make the full functionality available. The three pieces consist of the attributes that are watched, which are `templateUrl` and `mediaType`. The attributes that are watched allow the directive to recompile itself whenever they change. Using the `$templateCache` service allows the directive to be faster and almost mimic perfect synchronization. Not using HTTP to request for HTML templates also provides a sense of security. The last security feature provided by the `mediaPlayer` directive is the use of the core `$sce` service.

The `$sce` service is a security service that provides different types of security features for HTML injection and **Cross-Origin Resource Sharing** (**CORS**).

The `trustAsResourceUrl` method allows the application to request video files that are hosted on different domains. The video files are the only files being requested outside of the server because all of the templates are stored in the template cache during the build time. This plays a large role in the security of the directive, which should be a priority when dealing with directives that compile templates from a dynamic source.

The templates are added into the module definition array as the first parameter of the `angular.module` function. This puts the templates into the template cache. These are the only templates available to the directive, unless more templates are programmatically put into the template cache. This is a security feature that the `mediaPlayer` directive utilizes to help stop HTML injection. This is the second reason why the `$templateCache` service is important to the `bbMediaPlayer` directive, the first reason being performance.

Utilizing advanced templates

Template-based directives offer advanced functionality because the templates can be as complex as the developer wants. They can even have other AngularJS directives inside of them. These directives will be compiled by the `mediaPlayer` directive and will be called inside of the template.

This fact allows the `mediaPlayer` directive to be very powerful while keeping it simple. Let's go over some different examples of advanced templates that utilize AngularJS directives inside of them to accomplish the functionality that gives the `mediaPlayer` directive a master's touch.

The mediaelement templates

The `mediaelement` template is a third-party library that utilizes HTML templates, which contain video tags arranged in certain ways, and converts them into a media player with many different types of features. The library sets a function named `mediaelementplayer` onto the `$.fn` object, which is what is used as the directive media type. The Black Belt demo uses the following two different types of the `mediaelement` templates:

- Basic `mediaelement` templates that work with a statically hosted video file from `archive.org`
- A YouTube `mediaelement` template that allows `mediaelement` to stream videos directly from YouTube

Refer to the following code for the `mediaelement` template:

```
<video width="100%" height="100%" preload="none"
  ng-attr-poster="{{videoConfig.thumbnail}}"
  ng-src="{{trustSrc(videoConfig.filePath, '.mp4')}}">
  <source ng-src="{{trustSrc(videoConfig.filePath, '.mp4')}}"
    type="video/mp4">
  <source ng-src="{{trustSrc(videoConfig.filePath, '.webm')}}"
    type="video/webm">
  <source ng-src="{{trustSrc(videoConfig.filePath, '.ogv')}}"
    type="video/ogg">
  <object width="100%" height="100%" type="application/
    x-shockwave-flash" data="{{trustSrc(videoConfig.filePath,
    '.swf')}}">
    <param name="movie" value="{{trustSrc(videoConfig.filePath,
      '.swf')}}">
    <param name="flashvars" value="controls=true&amp;
      file={{trustSrc(videoConfig.filePath, '.mp4')}}">
  </object>
</video>
```

This template uses another core directive called `ngAttrPoster`. This directive sets the `poster` attribute whenever it is interpolated. The reason that it is needed is because the browser sets the `poster` attribute with the interpolated values before AngularJS gets a chance to interpolate its value. This is common with AngularJS directives that work with default attributes.

> The ngAttr directive is a core directive that works with any attribute placed after the Attr directive. This is a very useful and flexible directive.

The template uses ngSrc to set its source tag because of the same issue that the poster attribute has with the interpolation values of AngularJS. In this specific case, ngSrc calls the trustSrc method provided by the mediaPlayer directive. The rest of the template uses pure HTML5 syntax to set the video element as needed.

The YouTube template that is used is transformed by mediaelement into an embed element. The template is simple to write and is a very powerful tool when used correctly. The Black Belt application uses a demo that utilizes AngularUI's typeahead by hitting YouTube's API and sets the video of this template based off of the chosen result. The end result is useful and attractive to any level of a user. The demo can be found at http://angulardirectives.joshkurz.net/dist/#/mediaelement. Refer to the following code:

```
<video width="100%" height="100%" preload="none">
    <source type="video/youtube"    src="{{trustSrc(videoConfig.
filePath, '')}}" />
</video>
```

The flowplayer templates

Flowplayer is another third-party media library. The benefits of using it include default HTML5 support and more options to create real-life video players. The directive is initialized a little differently than the previous implementation. This is because here we use interpolated values to set the media-type attribute. This could have been used in the previous mediaelement instantiation of the media player. Refer to the following screenshot:

Re-Imagining the Browser with AngularJS
RE-IMAGINING THE BROWSER WITH AngularJS
Miško Hevery
Creator of AngularJS, Google
January 31, 2013
San Francisco, CA

Refer to the following HTML code for the media player:

```
<div media-player media-type="{{mediaType}}" video-
config="activeVideo" template-url="{{currentFlowplayer}}"></div>
```

This HTML gives the media player the ability to dynamically change everything about itself. At the whim of any change in the `scope` variable, these values could dramatically alter the look, feel, and functionality of the `mediaPlayer` directive. The ability to utilize a single directive in many different possible scenarios is a huge plus and is attainable by using the `$compile` service with a combination of directives that follow this template style.

The following two templates created for the Black Belt application accomplish two separate use cases:

- Utilize the `ng-repeat` directive to provide the playlist for the media player
- Create a mock of a flowplayer demo, which uses a pre-roll before allowing the user to watch videos

The template that utilizes `ng-repeat` to provide the playlist functionality is further proof that the sky is the limit for this directive and other directives of this nature. Refer to the following code:

```
<div id="dots" class="player">
  <video autoplay>
    <source type="video/mp4" ng-src="{{trustSrc
      (videoConfig.playlist[0], '.mp4')}}">
    <source type="video/ogg" ng-src="{{trustSrc
      (videoConfig.playlist[0], '.ogv')}}">
  </video>

  <div class="fp-playlist">
    <a ng-repeat="video in videoConfig.playlist"
      href="{{video}}.mp4" id="dot{{$index}}"></a>
  </div>
</div>
```

This template will be rendered as a video player that has little dots at the bottom-left of the player, which stand for each available video in the playlist. The playlist can be altered live during runtime, which results in the `mediaPlayer` directive updating itself and re-initializing the template. The final result is a dynamic video player that utilizes a playlist that is fed and controlled by AngularJS. The demo can be found at `http://angulardirectives.joshkurz.net/dist/#/flowplayer`.

The second template example for the `mediaPlayer` directive is to showcase more of the endless possibilities available with the `mediaPlayer` directive and the simplicity of the semantics and implementation when dealing with AngularJS templates to accomplish complex tasks. Refer to the following code:

```
<div class="flowplayer">
  <video>
    <source type="video/mp4" src="http://stream.flowplayer.org/
      download/640x240.mp4">
    <source type="video/webm" src="http://stream.flowplayer.org/
      download/640x240.webm">
    <source type="video/ogg" src="http://stream.flowplayer.org/
      download/640x240.ogv">
  </video>

  <div class="fp-playlist">
    <a class="is-advert" href="http://stream.flowplayer.org/
      download/640x240.mp4"></a>
    <a ng-repeat="video in videoConfig.playlist"
      href="{{video}}.mp4">Video {{$index + 1}}</a>
  </div>

  <div class="preroll-cover">pre-roll</div>
</div>
```

This template renders a pre-roll ad right when the first play is hit. This would be a great opportunity for companies to make money off of ad plays or to self promote a product. The functionality is already built into the flowplayer, and this template just unleashes its power into the AngularJS environment.

The `mediaPlayer` directive is completed, and its functionality is full of wonderful features available to any application that is in need of utilizing media as a medium to connect and interact with users. The `$compile` function has given the `mediaPlayer` directive the ability to utilize any user-generated template. This feature, alongside the ability to work with different third-party libraries for video functionality, is what makes this directive so unique and useful.

Summary

The `$compile` service made available by AngularJS's provider system is very powerful. Directives are always going to be the safest place for its dynamic use, which is true because of the order of operations performed by AngularJS's compile cycle. AngularJS uses the `$compile` service internally on many different occasions. The most important being the initial bootstrap phase, which is what brings AngularJS to life.

Any time the `$compile` service is used in an application is just another mini bootstrapping that is spawned by some programmatic function. This means that to bring some piece of HTML into an AngularJS application, it must be compiled and linked to the DOM.

Since the `$compile` service collects and conditionally applies definition objects to all of the directives in an associated piece of DOM, when does the `$scope` function come into play?

The `$compile` service returns a `link` function whose sole purpose is to attach a given `scope` object to a piece of DOM element and interpolate all of its bindings and events accordingly. The reason the compile and linking phase are broken into two different pieces is for optimization purposes and overall logic separation.

In this chapter, we covered how the compile cycle works in detail and how to incorporate this dynamic process into your own applications correctly. Most of the time, there is no need for custom compilation in an application, but there are a few instances that do require the use of AngularJS's `compile` functionality.

The different use cases for calling the `compile` function in a directive are as follows:

- Directives that use transclusion
- `Recursive` directives
- Dynamic template-based directives

These use cases make up the majority of the directives that utilize dynamic DOM creation and breakdown. Transclusion is not apparently calling the `compile` function, but don't let the genius of the AngularJS framework confuse you. Transclusion is nothing more than a mechanism to rip a piece of a DOM out of the tree during the `compile` phase and storing its transpiled `link` function in the memory to be called at a later time.

The `recursive` directives are built on the idea of transclusion and use its optimized ideologies to create complex DOM elements with very minimal code. These DOM elements are semantically much more simple than their static counterparts which use many lines to accomplish what the `recursive` directives can do in much less. To show case transclusion and recursion, we created a `treeNode` directive that accomplishes the same functionality in two different ways.

Dynamic template-based directives are the bases for two major AngularJS core directives. The `ngInclude` and `ngView` directives are the two major directives that allow applications to control HTML in a simplified and organized format. To showcase the power of template-based directives, we created a `mediaPlayer` directive. The `mediaPlayer` directive can compile any template that has been placed in AngularJS's `$templateCache`. This is so that the directive can work in an almost perfectly synchronous manner, speeding up the entire compile and linking processes.

The benefits of using the `compile` function in an application are enormous. The `compile` service is a sword yielded by only the strongest and most apt AngularJS developers and once its process is mastered, the rest of the way becomes simpler.

5
Communication between Directives

Directives are meant to control DOM and be in charge of their own view layer. Many directives need to be able to work in collaboration with each other to achieve certain requirements. There are many ways to write directives that allow this type of behavior. We will be going over many possible types of directive communication in this chapter.

Collectively, these are the methods that are used to communicate with each other. Each has advantages and disadvantages, which will be discussed and explained. The main topics that will be covered in this chapter are:

- Utilizing scope objects to share data between related directives
- Using services to share data and function context
- Broadcasting, emitting, and listening for specific events
- Using `require` and directive controllers to communicate and share functionality

Testing integrated directives

The act of communication infers that multiple parties are going to be working in unison with each other. This fact suggests that the unit testing processes that have been taken so far in this book will not suffice. When testing multiple directives together, it is common to use integration testing.

[Integration tests can be provided by the tools used in all of the tests previously written in this book.]

Integration tests

The use of multiple modules together to prove specific use cases is considered an integration test. Testing multiple directives working in collaboration with each other is expected when proving communication between directives is working properly. To accomplish this, multiple directives need to be integrated and compiled together. The integration in this book is referring to the creation of an element that has multiple directives appended to each other inside of a containing element. This type of integration is how all of the integration tests are carried out.

Let's go over a basic example of an integration test.

This test describes two modules that could interact with each other. Let's assume that the JavaScript for the `module1` directive will update the `module2` directive's test attribute when clicked. The main focus here is to see how the integration tests are written so that there is no confusion about how and why different directives are being compiled in one DOM element. Refer to the following code:

```
describe('Basic Communication with Directives', function () {

  beforeEach(module('BasicCommunicationExamples'))

    var directives,
        bbDirective1,
        bbDirective2;

    beforeEach(function(){
      var integration = angular.element('<div>' +
                            '<div bb-directive1></div>' +
                            '<div bb-directive2></div>' +
                          '</div>');
      directives = $compile(integration)(scope);
      scope.$apply();
      bbDirective1 = $(directives.find('.directive')[0]);
      bbDirective2 = $(directives.find('.directive')[1]);
    });

         it('should start with a message that says Not Clicked',
function(){
      expect(bbDirective1.text()).toBe('Not Clicked');
      expect(bbDirective2.text()).toBe('Not Clicked');
    });

    it('should alter each others text when clicked', function(){
      bbDirective1.click();
      expect(bbDirective2.text()).toBe('bbDirective1 Clicked');
      bbDirective2.click();
      expect(bbDirective1.text()).toBe('bbDirective2 Clicked');
    });
  });
```

These tests prove that `bbDirective1` and `bbDirective2` are interacting with each other exactly as they are supposed to. There are two key takeaways to pay attention to when looking at this describe block:

- Both directives are compiled together in one integrated DIV
- Each directive is taken out of the DIV and interacted with specifically

All of the integration tests in this chapter will compile DOM elements in this combined fashion. The directives are compiled together in one integrated DIV to prove their collaboration with each other works in a manner that would be used in a production environment. This allows the best quality tests to be written and executed.

Using scope objects for communication

Communication between directives can be by means of many different interactions. The most common and easy to achieve is to use the common `$scope` object. This is usually the first form of communication developers use when writing directives that communicate with each other.

When writing directives that communicate and use model objects, it is recommended to follow certain development ideologies related to code organization and declaration. The following are the two major related AngularJS ideologies that should always be implemented when writing directives, to create the most extendable and readable code base:

- Declaratively define what variables a directive watches as an attribute
- Avoid forcing directives to write to their defining scope

Using the scope for more advanced communication than simple examples can become troublesome once other directives and controller methods start editing data. The reason for the trouble is because it can be hard to debug what directives are writing to what variables. This is when child and isolate scopes come into play. Child and isolate scopes allow directives to use their own private scope object that can be read by only specific scopes that live in the correct hierarchy. By using a child or isolate scope, the directive can now safely write to its own scope with less chance of polluting its defining scope.

Using child scopes

Child scopes offer the ability to mimic basic JavaScript inheritance in the DOM tree. This is useful when creating directives whose data always communicates in a child that has lower value than a parent or a parent that has lower value than a child path. Child scope directives offer the following two types of functionality:

- Read from the parent and only write to its own private scope
- Read and write to both parent and private scope

The ability to achieve each individually depends on the instance of the data type that is being read from or written to. The reason for this is the prototypical nature of JavaScript.

JavaScript defines its variables with a prototypical inheritance model. This means that the global scope is the parent and all functions create their own private child prototypical scope instance. This instance will read from its own scope for variable definitions and then move upward until it finds a scope with the defined variable. Non-intuitive prototypical issues can be caused by writing to variables that are defined on parent scopes inside of child scopes. This can be seen in AngularJS as well and is a common issue among new developers.

Let's look at a basic example and break down what is happening with an AngularJS example.

The following code snippet shows how a child scope is implemented using HTML:

```
<div ng-app="scopeDemonstration">
  <ul>
    <li>Parent: <input ng-model="hey"></li>
    <li>Child 1: <div bb-hello-child></div></li>
    <li>Child 2: <div bb-hello-child></div></li>
    <li>Child 3: <div bb-hello-child></div></li>
  </ul>
</div>
```

The following code snippet shows how a child scope is implemented using JavaScript:

```
angular.module('scopeDemonstration', [])
.directive('bbHelloChild',
function () {
  return {
    restrict: 'A',
    scope: true,
    template: '<input ng-model="hey">'
    };
    }
)
```

The `ngModel` directive does not create a child scope. It creates a controller that is used by the core `input` directive to update its defining scope every time the text input changes. Each `bbHelloChild` directive uses a text input in collaboration with `ngModel` as well.

Let's go over a series of possibilities that could happen in this example and explain the output. The live example can be found at `http://jsfiddle.net/joshkurz/4q68V/` with the following possibilities:

- Entering a value for the `Parent` input: All of the children will be updated because they do not have a string literal named `scope.hey` located in them.
- Entering values first for the `Parent` input and then for `Child 1`: Only the input model for `Child 1` is updated. JavaScript will find `scope.hey` in the child scope and will not search any higher in the scope hierarchy.
- Entering values first for the `Child 1` input and then for `Parent`: Only `Child 2` and `Child 3` will be updated with the new parent model.

Possibility 2 is true because once `Child 1` has been typed into, it will receive a value written to its scope named `hey`. The `Child 2` and `Child 3` inputs do not have this `hey` value, so they continue to read up the prototypical scope hierarchy until they see that the `Parent` does. The `Parent` value is what is read and displayed into the input.

Some people consider this to be unexpected behavior, but it is basic JavaScript 101. Writing the data model in a flat structure like this allows child scopes to only read from the parent scope, which in some instances is what the directive should do.

To allow the child scope to write to the parent, the data model needs to be an object rather than just a string literal. This can be achieved in different ways. The most common way is to use DOT notation to access the variables and set the `ngModel` value equal to an object with the key value `hey` attached to it.

An example of bypassing the nuances of scope inheritance can be found at `http://jsfiddle.net/joshkurz/4q68V/1/`.

The following code shows the implementation using HTML:

```
Parent Scope HTML Declaration: <input ng-model="hey.hello">
```

This same technique can be used in the `bbHelloChild` directive.

The following code shows the implementation using JavaScript:

```
template: '<input ng-model="hey.hello">'
```

Let's go over the same inputs and see what outputs they render now that we are using an object to store our string values. Refer to the following possibilities:

- Enter a value for the `Parent` input: All of the children will be updated because they do not have an object named `scope.hey` located on their context
- Enter values first for the `Parent` input and then for `Child 1`: All of the inputs will be updated because the child `scope.hey` object is a reference to the parent `scope.hey` object
- Enter values first for the `Child 1` input and then for `Parent`: `Child 2` and `Child 3` will be updated with the new parent model, but `Child 1` was typed into first and it created its own `scope.hey` object, which cannot be overwritten by `Parent`

Great directives can be written once these concepts are understood. Let's create some advanced examples using child scopes. These examples will communicate with our previously created `bbStopwatch` directive.

Creating a wasFast directive

The `bbStopwatch` directive creates an array of log times. These log times can be viewed in any desired manner. The `wasFast` directive creates a child scope that prototypically inherits from its parent. The `wasFast` directive reads from the stopwatch's log array and determines whether each value was fast or not based off of some arbitrary conditions. The final result is an element that renders an informative message based on the speed and adds an associated CSS class to pretty up the message.

The following are the requirements for the `wasFast` directive:

- Needs to be able to work with any scope name as the actual log value
- Should render text based off of the log value
- Should append an associated time class to the element

Unit testing

The `describe` block for the `wasFast` directive proves all of its basic requirements. For the sake of simplicity, we are only showing the first unit test in the `describe` block. The others prove that the average and slow conditions work as expected. The test examples can be found at `https://github.com/joshkurz/Black-Belt-AngularJS-Directives/commit/40cdb849fa9f7f5e053b77a077da2515ae0fab04#diff-1cda6f2d12e35eac4676b5ce60648baeR102`. Refer to the following code:

```
describe('The wasFast directive', function () {
  var wasFast;
  function compileWasFast(){
    wasFast = $compile('<div was-fast time="testLog"></div>')(scope);
    scope.$apply();
  }

  it('should append the correct super fast text and the fast class to
the
    directive', function() {
        scope.testLog = 100;
        compileWasFast();
        expect(wasFast.text()).toBe('0.1 seconds (Super Dog Speed)');
        expect(wasFast.children().eq(0).hasClass('fast')).toBe(true);
  });
});
```

This test is basic and proves that the `wasFast` directive is setting the correct class to the element based on the log value it is reading.

Integration tests

The combination of isolate scopes and child scopes are used to fulfill the final integrated communication requirements in this next example. The `wasFast` directive has a child scope that reads the stopwatch's log value. The log value is being iterated over in an `ngRepeat` directive.

Once the `wasFast` directive is initialized in the DOM, it will create its correct values; this was shown in the preceding unit test. Refer to the following code:

```
describe('Integration between the stopwatch and the wasFast
directive',
  function () {

    var integration,
        logs,
        stopwatch;
    beforeEach(function(){
      var preCompileElement = angular.element('<div>' +
                               '<div class="stopwatch"
options="stopwatch"
                                 bb-stopwatch override="true">' +
                                 '<button ng-click=
                                   "startTimer()">start</button>'+
                                 '<button ng-click=
                                   "stopTimer()">stop</button>'+
                              '</div>' +
                               '<div class="logs" ng-repeat="log in
                                 stopwatch.log">'+
                                 ' <div class="wasFast" was-fast
                                   time="log"></div>' +
                                 ' <div class="fastRunner" fast-runner
                                   time="log" pics="testPics"></div>'
+
                               '</div>' +
                             '</div>');

      integration = $compile(preCompileElement)(scope);
      scope.$apply();
      stopwatch = integration.find('.stopwatch');
    });

    it('should append a super fast child to the directive', function()
{
        logs = integration.find('.logs');
        expect(logs.children().length).toBe(0);
        expect(scope.stopwatch.log.length).toBe(0);
        $(stopwatch.children()[0]).click();
        $(stopwatch.children()[1]).click();
        logs = integration.find('.logs');
        expect(scope.stopwatch.log.length).toBe(1);
        expect(logs.children().eq(0).text().split('(')[1]).toBe('Super
Dog
          Speed)');
    });
});
```

The following are some important things to take away from this test:

- The `bbStopwatch` and `wasFast` directives are being created together in the same element. This can be confusing to look at because of the stringified template. This is the fastest and most efficient way to compile large chunks of DOM in an integration test.
- We are using jQuery to find and interact with the directives inside of the integration element.
- We are splitting the text value of the `wasFast` directive because the start and stop clicks were synchronous, and there is no way to tell how much time actually went by.

Implementing the wasFast directive

The `wasFast` directive is a simple directive that has most of its logic implemented in the `link` function. The directive inherits its data from its defining scope. The `wasFast` directive is expecting a time attribute to be set on the directive, which is read from the child scope. The following final result of the `wasFast` directive is a paragraph element that has an associated class and text that represents the speed and the time:

- **Super Dog Speed**: 0.803 seconds
- **Human Speed**: 2.436 seconds
- **Super Slow Speed**: 20.623 seconds

The following example is using `ngRepeat` to iterate over the logs in the `bbStopwatch.log`. The time attribute is important because it is telling the directive the name of the scope variable it should be looking for:

```
<div class="log" ng-repeat="log in stopwatch.log">
      <div was-fast time="log"></div>
</div>
```

This is the actual directive. As you can see, there aren't many special features to it. Its purpose is to allow for a clean HTML markup. This is accomplished by appending a paragraph element to the `wasFast` directive with its associated information, as shown in the following code:

```
app.directive('wasFast', function () {
  return {
    restrict:'EA',
    scope: true,
    template: '<p class="wideLoad" ng-
class="speedClass">{{logText}}<p>',
    link: function(scope, element, attrs){

      scope.logText = scope[attrs.time]/1000 + ' seconds';

      if(scope[attrs.time] < 1000){
        scope.speedClass = 'fast';
        scope.logText += ' (Super Dog Speed)';
      } else if(scope[attrs.time] < 5000){
        scope.speedClass = 'average';
        scope.logText += ' (Human Speed)';
      } else {
        scope.speedClass = 'slow';}
        scope.logText += ' (Super Slow Speed)';
      }
    };
});
```

The directive gets the scope time and divides it by `1000`. This converts the milliseconds into seconds and makes the time more readable. The `logText` function is a string that gets set at the beginning of the linking phase. Then conditions are checked and the values are appended to this string to represent what the time frame means. During these conditional statements, the class is set as well. The `logText` function is bound to the paragraph elements text via the curly brackets and the class is bound via `ngClass`.

The `wasFast` directive communicates with its parent scope to determine what time should be displayed in its DOM. This is a simple directive that provides a very effective and efficient way to display a conditional representation of how fast the logged time variable was. Using child scopes to communicate across directives is a very clean and natural expression of how directives can work with each other.

Creating a fastRunner directive

We want to go one step further and create another directive that animates a GIF across the screen to represent the time. Let's assume that the time frame is associated with a runner and we want to show GIFs of different types of runners moving across the screen. To do this, we create another directive that is called `fastRunner`. This directive will be a marquee element that has some image and speed associated with it.

The requirements for the `fastRunner` directive are as follows:

- Render a marquee element with some dynamic speed
- Render an image element with some dynamic source

Since the speed data is already being calculated inside of the `wasFast` directive, we can use `wasFast` to create `fastRunner`. This will allow the scopes to communicate the speed and image data that the `fastRunner` directive needs to implement itself.

To accomplish this, we will edit the `wasFast` directive to compile and append the `fastRunner` directive. Only the `wasFast` link function will be edited to accomplish this integration.

The first change is to add the following variables to the `link` function:

```
var runner = ''",
runnerSpeed = 0;
```

The next change is to assign appropriate values to `runner` and `runnerSpeed` inside each of the `wasFast` condition, as follows:

```
if(scope[attrs.time] < 1000){
  speedClass = 'fast';
  runnerSpeed = 100;
  runner = '/dist/images/runningDog.gif';
  scope.logText += ' (Super Dog Speed)}
```

[Each condition has specific values that will be set depending upon the speed that was calculated.]

The last change is to `$compile` and append the `fastRunner` directive to the `wasFast` element as follows:

```
var fastRunnerElem angular.element('<div fast-runner runner="' +
  runner + '" speed="' + runnerSpeed + '"></div>')
var runnerNode = $compile(fastRunnerElement)(scope);
element.append(runnerNode);
```

[Now `wasFast` is fully equipped and ready to integrate with the `fastRunner` directive.]

Integration testing

The integration test that proves a collaboration between `wasFast` and `fastRunner` will be an adjustment of the integration test written for `wasFast` and `bbStopwatch`. The only difference will be that we will now be checking for marquee elements and image source attributes.

So the only change will be the addition of two assertions as follows:

```
expect(integration.find('marquee').attr('scrollamount'))
  toBe('100');
expect(integration.find('img').attr('src'))
  toBe('/images/runningDog.gif');
```

These assertions prove that the integration was a success. The `wasFast` directive is creating and communicating the correct scope values to the `fastRunner` directive.

Implementing the fastRunner directive

Let's write the actual `fastRunner` directive. The `fastRunner` directive sets a watcher on its scope that looks for changes in its attributes. This watcher updates the scope variables whenever the directive's attributes change. This is theoretically manually setting up a binding on the attribute's value; the only difference is now there is only one binding rather than two. Refer to the following code:

```
.directive('fastRunner', function () {
    return {
      restrict:'EA',
      template: '<marquee behavior="scroll"
        scrollamount="{{speed}}" direction="right"><img
        ng-src="{{runner}}"/></marquee>',
      link: function(scope, element, attrs){

        function getTheAttrs(){
          return attrs.runner + attrs.speed;
          }

          scope.$watch(getTheAttrs, function(){
              scope.runner = attrs.runner;
              scope.speed = parseInt(attrs.speed,10);
              });
          }
        };
});
```

The `fastRunner` directive creates a marquee element that scrolls a GIF across the screen that represents the speed of a runner. The optimization techniques used here bring the number of watchers from `2` to `1`.

The following is the output of the implementation:

0.561 seconds (Super Dog Speed)

An alternative that could have been used would have required the `wasFast` directive to use an isolate scope whose value was two way data bound.

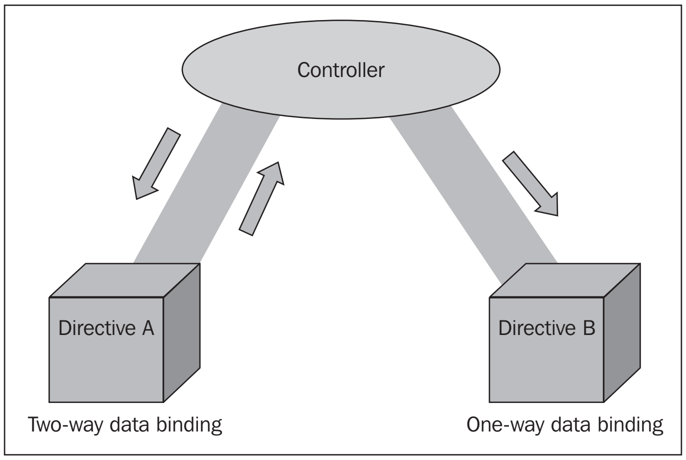
How to use isolate scopes

Isolate scopes are very important and offer many advantages to single page applications. The advantages provide special types of communication patterns that side-step JavaScript's prototypical inheritance patterns. Isolate scopes offer the following two types of communication techniques:

- One-way data binding
- Two-way data binding

Communication between directives using isolate scopes starts with the parent scope. The parent scope will hold the data model that a directive requires for successful communication with another directive that uses that same model.

Think of isolated scopes as the pipe from these scopes that go directly to the directive. There are different types of pipes available that can be used by each directive. Refer to the following diagram:



The different pipes offer multiple forms of communication. The different forms are available via the scope definition object key value. While both one and two-way data binding allow directives to communicate with controllers in different ways. Only two-way data binding allows directives to communicate with each other on a sibling basis rather than using a parent child model.

A directive that uses two-way data binding can write to a directive that only uses one-way data binding. This is because once two-way data binding is instantiated, the isolate scope gets access to edit the data, which is the same as defining the scope's data. When the isolate scope updates a model value and there are directives that rely on that data model, those directives are also updated. No matter what environment the two-way data bound directive lives in, its data is always available to both parties. If one party changes the data, the other party will automatically notice and update itself as needed. This is because the edits are actually happening to the same data model, which follows AngularZen philosophy, which states that the model is the single source of truth.

Let's rewrite the `wasFast` and `fastRunner` directives to use isolate scopes. This will allow both of the directives to be initialized in the DOM as siblings, rather than forcing `wasFast` to create the `fastRunner` directive.

The tests that we wrote previously do not have to be changed. All of the final integration should work in the exact same way. The difference is how the view is initialized. Now we are going to create the `fastRunner` directive in the DOM template.

The ng-repeat function that renders the wasFast directive will have another child. The extra child will be the fastRunner directive.

The following code is before implementing the isolate scope:

```
<div class="log" ng-repeat="log in stopwatch.log">
  <div was-fast time="log"></div>
</div>
```

The following code is after implementing the isolate scope:

```
<div class="log" ng-repeat="log in stopwatch.log">
  <div was-fast time="log"></div>
  <div class="fastRunner" fast-runner time="log"
    pics="pics"></div>
</div>
```

The number of view layer possibilities increase by initializing each directive with two-way data binding. If the log values are changed by any source at any moment, all of the directives' (stopwatch, wasFast, and fastRunner) log values will be updated automatically. We write each directive to auto-update its own view by setting a watcher on time, which will call a function to update its view accordingly.

The new scope and link option value for wasFast has been refactored to only include logic that is specific to its own view. This alleviates any crossover and confusion when working with these directives.

The following code shows the refactored wasFast scope and link options:

```
scope: {time : '='},
link: function(scope, element, attrs){

  function changeMessage(){
    scope.logText = scope.time/1000 + ' seconds';
    if(scope.time < 1000){
      scope.speedClass = 'fast';
      scope.logText += ' (Super Dog Speed)';
      } else if(scope.time < 5000){
      scope.speedClass = 'average';
      scope.logText += ' (Human Speed)';
      } else {
      scope.speedClass = 'slow';
      scope.logText += ' (Super Slow Speed)';
      }
    }
  scope.$watch('time',changeMethod);
}
```

Now, every time the time value changes, the view for this isolate scope will be updated with its associated values. The same goes for the new `fastRunner` directive.

The following code shows the refactored `fastRunner` scope and link options:

```
scope: {time: '=', pics: '='},
template: '<marquee behavior="scroll"
  scrollamount="{{runnerSpeed}}" direction="right"><img ng-
  src="{{pics[runnerSpeed]}}"/></marquee>',
link: function(scope, element, attrs){

  function changeSpeed(){
    if(scope.time < 1000){
      scope.runnerSpeed = 100;
      } else if(scope.time < 5000){
      scope.runnerSpeed = 10;
      } else {
      scope.runnerSpeed = 1;
      }
    }

  scope.$watch('time', function(newV,oldV){
    changeSpeed();
  });
}
```

The biggest change to this directive is how the images are being passed into the scope. This allows business logic to be placed in the controller where it belongs. The only item that is being changed in the directive on each time change is the `runnerSpeed` variable, which is now the key to the pictures array. The isolate scope makes this easy to achieve without having to evaluate parent scopes and creating logic that depends on scope relationships.

Another great benefit of using the `=` sign to represent the isolated scope is that now we can pass in the actual log value and not try to read it off of a child scope. This means that we can now watch the last value of an array. This results in a much more declarative view. Refer to the following code:

```
<div was-fast time="stopwatch.log[stopwatch.log
  length-1]"></div>
```

The directive will update itself every time a new log value is appended to the stopwatch's log. There are a plethora of obvious advantages to being able to assign values to directives in this declarative manner.

Relying on the $rootScope function

The following quote can be found in the documentation of AngularJS (visit `http://docs.angularjs.org/api/ng.$rootScope`):

> "*Every application has a single root scope. All other scopes are descendant scopes of the root scope.*"

The `$rootScope` function plays a very important role in AngularJS. It is the only scope object all view layers have access to at all times for free. Whatever is set on the `$rootScope` function can be accessed in any child or isolated scope, unless prototypical inheritance gets in the way. The high availability of this service is the reason for its misuse in applications. There are times when using `$rootScope` is good, but then there are times when custom services should be created.

Some good use cases for using `$rootScope` are in services, where there is no access to any other scope. Another great use case is for propagating messages up or down the scope tree so that all facets of an application receive a given notification.

Broadcasting to other directives

Directives are always linked to a `scope` variable. This `scope` variable has multiple public methods available. Three of these methods are `$broadcast`, `$emit`, and `$on`. The `$broadcast` and `$emit` functions are meant to dispatch messages across an application. Either message dispatcher function can be used, but depending on the parent-child relationship of the scopes actively listening for messages, there is a correct choice for which method to use.

When broadcasting an event, the message flows in a parent-to-child fashion from the defining scope. This means that if a message is broadcasted on the `$rootScope` function, then all scopes will be able to interact with that event. The interaction will only occur if a given scope has set up an `$on` listener function for that specific event. The `$emit` function is the inverse of `$broadcast`. Its job is to dispatch messages in an upward crawl through the scope tree.

When a `$broadcast` or `$emit` function is fired, two parameters are passed into the `$on` function that is listening to the event. The first parameter is the `event` object, which has specific arguments that can search for unique values and functions that can stop the propagation of the notification to further scopes. The specific arguments can be found at `http://docs.angularjs.org/api/ng.$rootScope.Scope#methods_$on`.

The second parameter passed into the `listener` function is the custom argument that was passed into the `$broadcast` or `$emit` function. This could be any object that is specific to the application and the notification being undertaken.

Directives can `$broadcast` messages on the `$rootScope` function to ensure that wherever the listener is, it will get the message. There are optimization best practices to only call a `$broadcast` or `$emit` function depending on the parent-child relationship. These optimizations speed up the overall time it takes for the message to get to its final destination. Using the `$rootScope` function to invoke communication throughout an entire application is ok and not considered bad practice if the parent-child relationship is not always apparent.

Communicating with media players

Let's take an example based off of a Stack Overflow post, which can be found at `http://stackoverflow.com/questions/18780402/angularjs-communication-between-directives`.

The overall goal is to create a set of directives that communicate with each other in a way that only allows one player to be on at any given time. When a directive is turned on, all of the other directives need to be turned off.

Integration testing for the bbBroadcastingPlayer directive

Let's go ahead and write an integration test that proves these directives do work in unison with each other and communicate the correct data with each other. Refer to the following code:

```
describe('Broadcasting events between the players', function () {
    var controllerPlayer;
    beforeEach(function(){
        var integration = angular.element('<div>' +
          '<div class="player" bb-broadcast-player></div>' +
          '<div class="player" bb-broadcast-player></div>' +
          '<div class="player" bb-broadcast-player></div>' +
          '<div class="player" bb-broadcast-player></div>' +
          '</div>');

        controllerPlayer = $compile(integration)(scope);
        scope.$apply();
        });

    it('Should start out with its text reading no and then once
      clicked change it to yes.', function(){
      var bbPlayer = $(controllerPlayer.find('.player')[0]);
      expect(bbPlayer.text().trim()).toBe('is playing: no');
      bbPlayer.find('.btn').click();
      expect(bbPlayer.text().trim()).toBe('is playing: yes');
    });

it('Should only ever have one yes at a given time', function(){
    var players = controllerPlayer.find('.player');
    var bbPlayer1 = $(players[0]);
    var bbPlayer2 = $(players[1]);
    var bbPlayer3 = $(players[2]);
    var bbPlayer4 = $(players[3]);
    expect(bbPlayer1.text().trim()).toBe('is playing: no');
    expect(bbPlayer2.text().trim()).toBe('is playing: no');
    expect(bbPlayer3.text().trim()).toBe('is playing: no');
    expect(bbPlayer4.text().trim()).toBe('is playing: no');
    bbPlayer1.find('.btn').click();
    expect(bbPlayer1.text().trim()).toBe('is playing: yes');
    expect(bbPlayer2.text().trim()).toBe('is playing: no');
    expect(bbPlayer3.text().trim()).toBe('is playing: no');
    expect(bbPlayer4.text().trim()).toBe('is playing: no');
    bbPlayer3.find('.btn').click();
    expect(bbPlayer1.text().trim()).toBe('is playing: no');
    expect(bbPlayer2.text().trim()).toBe('is playing: no');
    expect(bbPlayer3.text().trim()).toBe('is playing: yes');
    expect(bbPlayer4.text().trim()).toBe('is playing: no');
  });

});
```

This test is compiling multiple directives into one integration element. This integration element is being interacted with to prove that all directives inside of the integration are communicating and working correctly. Specifically, we are first testing to make sure all of the element's text reads `no`. The next test checks if a directive is clicked, its text should be `yes` but all of the other directives' text should be `no`. This is a pretty verbose test to prove such a simple concept, but the proof is in the pudding as they say.

Implementing the bbBroadcastPlayer directive

The `bbBroadcastPlayer` directive uses a child scope because it is writing to the scope and we want to make sure that its defining scope is not polluted by any of its values. The most important concept that goes into this implementation is the fact that it is using `$rootScope` to broadcast the event. The `$rootScope` function is used because there is no parent-child relationship between the `player` directives. They all create their own child scope, so emitting or broadcasting an event from their own scope would be pointless. Refer to the following code:

```
angular.module('broadcastingDirectives', [])
.directive('bbBroadcastPlayer', ['$rootScope',
    function ($rootScope) {
      return {
        restrict: 'A',
        replace: false,
        scope: true,
        templateUrl: 'directives/communicationExamples/
          playerTemplate.tpl.html',
        link: function (scope, iElement, iAttrs, controller) {

          scope.player = {isPlaying : 'no'} ;

          scope.play = function() {
            $rootScope.$broadcast('turnOff');
            scope.player.isPlaying = 'yes' ;
            };

          scope.$on('turnOff', function(event){
        scope.player.isPlaying = 'no';
        });
    }
  };
}
]);
```

When the `listener` function is fired, the `isPlaying` value of the player is set to `no`. Since the `$broadcast` and `$emit` functions are blocking functions and are run synchronously, we can safely turn off all values and then set the clicked scope `player.isPlaying` value to `yes`.

[Using a directive controller can accomplish this exact functionality.]

Collaborating with controllers

Normal controllers that are used in AngularJS applications hold business logic and drive data model manipulation. There are slight differences between application controllers and directive controllers. Directive controllers are more like `service` objects. They are singletons and can be shared between directives of the same element or children of the instantiated controller.

To request a controller, a directive needs to have a specific definition object parameter, called `require` set. The `require` value defines which directive or directives should inject their controller instances into the `link` function requesting them. To require another directive's controller, the requiring directive must be declared on the same element or be a child of the directive instantiating the controller.

A controller's context is made available to a directive's `link` function once it has been injected. The controller's context is whatever is set onto this value of the controller's closure. This gives controllers the ability to offer a public API to directives, which in turn makes it possible to reuse business logic across associated directives. The controller's context also offers communication possibilities between all directives that share its instantiation.

Requiring the basics

The benefits of requiring a controller versus injecting a service is to embed the actual functional context with the `link` function, which means a directive's `scope` can have access to another directive's execution context. To create a directive that shares its controller's context means that it has something worth sharing. An example could be two directives that will always live together like a carousel and its slides, or a directive that will share its data and functionality with other directives, such as `ngModel`.

The main use cases for the `require` option are basic, but cover a wide variety of possibilities. There are two basic use cases for `require`, as follows:

- The directive needs to share data with another directive
- Another directive's controller has useful functionality that is needed to accomplish a set of tasks

These are very general and cover 99 percent of the total use cases for `require`. Some specific directives that utilize controllers in the core are `ngModel`, `ngForm`, and `ngSwith`. These directives allow other directives the ability to have access to their controller's public context. This opens up many doors in terms of usability and functionality.

Communication and reusable APIs are essential to the health of an application. They increase the ability to keep code duplication down and allow the logical interaction with shared data across all layers of an application.

The biggest gotcha when working with controllers and requiring them in other directives is to make sure that the parent-child relationship is correct and known. A directive cannot require a directive's controller if it has no relationship with it in the DOM. Only a directive that is defined on the same element or is a child of the required directive can actually use the requested controller. There is a specific syntax to define how `require` values should look for the directive's `definition` object. This syntax can be read about in more detail in *Chapter 1, The Tools of the Trade*, and can also be found at `http://docs.angularjs.org/api/ng.$compile` in the section about `require`.

Using controllers for the bbPlayer directive

The `bbBroadcastingPlayer` directive created in the section prior to this was notifying other directives when a specific event happened to ensure that they were all in sync with each other. This same functionality can be accomplished by means of using a directive controller to keep track of the players. Using a controller gives a more efficient medium to accomplish this type of communication and is more extendable for future use cases that could come along.

Integration testing

The test that is used is exactly the same as the test for the `broadcasting` directive. The only difference is the integration element that is used. Now we are calling the `bbPlayer` directive, and it has a container directive called `bbPlayerContainer`. Refer to the following code:

```
beforeEach(function(){
    var integration = angular.element('<div bb-player-container>'
      +
    '<div class="player" bb-player></div>' +
    '<div class="player" bb-player></div>' +
    '<div class="player" bb-player></div>' +
    '<div class="player" bb-player></div>' +
    '</div>');

  controllerPlayer = $compile(integration)(scope);
  scope.$apply();
})
```

The same integration test should pass in the exact same way as it did when we were broadcasting the click event throughout the entire application's scope tree. The test was about proving that clicking on a directive will set that player's text to `yes` and all others to `no`. There should never be two players with `yes` as their text values at one time.

Implementing the bbPlayer and bbPlayerContainer directives

The `player` container is needed because this is where the controller is going to be created. The directive that is in charge of initializing the controller is always either the parent of or at least set on the same directive that is doing the requiring. Refer to the following code:

```
angular.module('controllerPlayers',
  []).directive('bbPlayerContainer', [
    function () {
      return {
        restrict: 'A',
        controller: function(){

          var players = [];
          this.addPlayer = function(player){
            players.push(player);
            };
          this.turnOffPlayers = function(){
            for(var i = 0;i < players.length;i++){
              players[i].isPlaying = 'no';
              }
            };
          }
        };
      }
]);
```

The `bbPlayerContainer` directive does not have a `link` function. This is because its not doing any DOM manipulation. Its sole purpose it to keep track of its child players and create a public API that each player can use. This public API will offer a player the ability to turn off all players and add themselves to the array of players. Refer to the following code:

```
angular.module('controllerPlayers', [])
.directive('bbPlayer', [
    function () {
        return {
          restrict: 'A',
          replace: false,
          require: '^bbPlayerContainer',
          scope: true,
          templateUrl: 'directives/communicationExamples/
            playerTemplate.tpl.html',
          link: function (scope, element, attrs, controller) {

            scope.player = {isPlaying : 'no'};

            scope.play = function() {
              controller.turnOffPlayers();
              scope.player.isPlaying = 'yes' ;
              };

            controller.addPlayer(scope.player);
            }
          };
        }
  ]);
```

The newly refactored `bbPlayer` directive is using the require definition parameter to locate the `bbPlayerContainer` controller. The ^ sign in front of the directive being required tells AngularJS to look at this directive's parents for a directive called `bbPlayerContainer`. If it is found, it will add it as the fourth parameter. There are two other options used to require data as well. A live example can be found at `http://jsfiddle.net/joshkurz/B8x3Q/3/`.

Creating a fastClicker directive

The `bbStopLight` directive has already been created, but it does not really do anything special or eye catching by itself. The purpose of `bbStopLight` is to tell an object when it is ok to perform a task. Let's create a directive that requires the `bbStopLight` controller to know when to perform its given instructions.

The directive that will be created is going to be called `fastClicker`. Its purpose will be to become active and clickable once the `stopLight` directive turns green. This is a simple directive that will showcase how to use a controller of another directive and reuse their public APIs to accomplish new requirements.

The `bbStopLight` directive transcludes its DOM to accomplish its final form. The `fastClicker` directive will be a child of `bbStopLight` and live inside of its transcluded DOM. Usually, directives that use transclusion have the ability to offer their controllers for other directives to require. This is only for a more flexible development standard and can be bypassed by using a directive that offers the ability to use dynamic templates.

Integration testing

This integration test will be very easy to achieve since this directive is so simple and only has two requirements.

The following are the requirements for `fastClicker`:

- Disable the button when the `stopLight` is not `green`
- Enable the button when the `stopLight` is `green`

This integration test can be coded as follows:

```
describe('Creating A fastClicker directive inside a stopLight
directive',
  function () {

  var stopLight,
      fastClicker;

  beforeEach(function(){
    var integration = angular.element('<div bb-stop-light-container
      options="options">' +
                               '<canvas bb-stop-light></canvas>' +
                               '<canvas bb-stop-light></canvas>' +
                               '<canvas bb-stop-light></canvas>' +
                               '<fast-clicker options="stopwatch"
                                 bb-stopwatch></fast-clicker>' +
                             '</div>');
    stopLight = $compile(integration)(scope);
    scope.$apply();
    fastClicker = stopLight.find('fast-clicker');
    ctrl = $controller('bbStopLightCtrl', {$scope:  scope, $interval:
      $interval});
    scope.$apply();
  });

  it('should activate its own button', function() {
      var fastClickerChild = fastClicker.children()[0];
```

```
        expect(fastClickerChild.hasAttribute("disabled")).toBe(true);
        expect(ctrl.options.state).toBe('red');
        ctrl.setNextState();
        ctrl.setNextState();
        scope.$apply();
        expect(ctrl.options.state).toBe('green');
        expect(fastClickerChild.hasAttribute("disabled")).toBe(false);
    });

    it('should allow for clicking the fast clicker and log the time',
      function() {
        expect(scope.stopwatch.log.length).toBe(0);
        ctrl.setNextState();
        ctrl.setNextState();
        scope.$apply();
        $(fastClicker.children()[0]).click();
        expect(scope.stopwatch.log.length).toBe(1);
    });
  });
```

The integration element that is being created contains the `fastClicker` directive. This should render an object that is disabled when the stoplight is not `green`. This test proves that this use case has occurred successfully. To do this, the most import aspect of the test is how the controller is created in the `beforeEach` block. This controller is created so that we can programmatically call its public functions to trip the `fastClicker` directive into falling into its enabled state.

Writing the fastClicker directive

The `fastClicker` directive has a basic template, which uses `ngDisabled` to accomplish its requirement. This is a simplification that AngularJS provides developers with when creating directives that add and take away attributes such as `disabled` and certain classes. Refer to the following code:

```
<button class="btn" ng-click="stopRaceTimer()"
  ng-disabled="canClick() == false">Race</buton>
```

Stock AngularJS directives should be used as much as possible rather than creating custom directives.

The following code is the JavaScript for `fastClicker`:

```
app.directive('fastClicker', function () {
  return {
    restrict:'EA',
    template: 'directives/communicationExamples/fastClicker.tpl.html',
    require: '^bbStopLightContainer',
    link: function(scope, element, attrs, ctrl){
      scope.canClick = function(){
        if(ctrl.options.state === 'green'){
          return true;
        } else {
          return false;
        }
      };
    }
  };
});
```

What the `fastClicker` directive is doing is requiring the `bbStopLightContainer` controller, which holds all of the public state information. This directive is not stopping the stoplight when it becomes `green`, so the end result would be a button that is flipping from enabled to disabled on some interval when the stoplight is ON. An example can be found at `http://jsfiddle.net/joshkurz/SupQ2/`.

Wiring up the stopwatch

The `fastClicker` directive is nice, but it doesn't really do anything cool. Let's integrate it with the `bbStopwatch` directive so it can start and stop a timer, which will subsequently update the stopwatch's time log array. Remember that by updating the stopwatch's time array, we will be able to create the `wasFast` and `fastRunner` directives as well. This will combine every example in this chapter together, but will be fairly simple to achieve now that the leg work is out of the way.

To add the `stopwatch` controller to the `fastClicker` directive, all we have to do is update the `require` option in the `fastClicker` directive and write the correct HTML markup.

The correct markup will include the addition of a `stopwatch` attribute to the `fastClicker` directive and setting its `options` attribute, which is two way data bound.

Now the `fastClicker` directive will look as follows:

```
<fast-clicker options="stopwatch" bb-stopwatch></fast-clicker>
```

To gain access to the `bb-stopwatch` controller from inside of the `fastClicker` directive, we will require the `bb-stopwatch` controller. Now, `fastClicker` is requiring multiple controllers, which means the syntax in the directive needs to be updated. Refer to the following code:

```
require: ['?stopwatch', '^stopLightContainer']
```

Since the `stopwatch` directive is being set onto the same element, we could not add any special selector (`?` or `^`), but we still want our previous test to pass, so we add the question mark in front of the stopwatch, giving the `fastClicker` directive the option to be set on the `stopwatch` directive or not.

The final changes to the `fastClicker` directive are shown as follows:

```
angular.module('AngularBlackBelt.fastClicker', ['AngularBlackBelt.
StopWatch'])
.directive('fastClicker', function () {
    return {
        restrict:'EA',
        templateUrl: 'directives/communicationExamples/fastClicker.
tpl.html',
        require: ['?bbStopwatch', '^bbStopLightContainer'],
        link: function(scope, element, attrs, ctrl){

           var raceTime = new Date();
           scope.canClick = function(){
             if(ctrl[1].options.state === 'green'){
                ctrl[1].killInterval();
                ctrl[0] && ctrl[0].stopwatchService.startTimer();
                return true;
             } else {
                return false;
             }
           };

           scope.stopRaceTimer = function(){
             ctrl[0] && ctrl[0].stopwatchService.stopTimer();
             ctrl[1].setNextState();
           };
        }
    };
})
```

The most obvious change is that now we are requesting the controller functions from an array of controllers rather than just one controller object. We are also short-circuiting the stopwatch's controller from calling itself if it is not present. This allows the tests to remain passing and does not introduce any breaking changes into our application. The final result can be found at `http://angulardirectives.joshkurz.net/dist/#/stoplight`.

Summary

Communicating across directives is an essential technique that most applications require to fulfill advanced requirements. There are many different ways to carry out interdirective communication, and each way has its own purpose and place.

Using scope objects to share data across directives is common in many applications and often doesn't even require a directive to achieve. The first directive written in this chapter showcased the differences between using child scopes and isolated scopes to communicate data.

Broadcasting and emitting data helps let other directives know what events and activities are happening across the application. The difference between a broadcaster and an emitter is that a broadcaster notifies up the scope tree and the emitter notifies downward. If there is no parent-child relationship between the directives then the `$rootScope` function can be used to make sure that all the directives listening for the event will be notified correctly.

Directive controllers can be shared and injected into other directives via the `require` options made available on the directive's definition object. This option tells the compiler that the directive would like access to a specified directive's controller. Using another directive controller gives the requesting directive access to its entire public context. Using directive controllers is a clean and efficient way to achieve inter-directive communication.

Communication between directives is a very important aspect directives. Many requirements are based on integrations between two or more different components inside of an application. These components must be able to share and manipulate data in efficient and effective ways.

6
Working with Live Data

Big Data is a new field that is growing every day. HTML5 and JavaScript applications are being used to showcase these large volumes of data in many new interesting ways. Some of the latest client implementations are being accomplished with libraries such as AngularJS. This is because of its ability to efficiently handle and organize data in many forms.

Making business-level decisions off of real-time data is a revolutionary concept. Humans have only been able to fathom metrics based off of large-scale systems, in real time, for the last decade at most. During this time, the technology to collect large amounts of data has grown tremendously, but the high-level applications that use this data are only just catching up.

Anyone can collect large amounts of data with today's complex distributed systems. Displaying this data in different formats that allow for any level of user to digest and understand its meaning is currently the main portion of what the leading-edge technology is trying to accomplish. There are so many different formats that raw data can be displayed in. The trick is to figure out the most efficient ways to showcase patterns and trends, which allow for more accurate business-level decisions to be made.

We live in a fast paced world where everyone wants something done in real time. Load times must be in milliseconds, new features are requested daily, and deadlines get shorter and shorter. The Web gives companies the ability to generate revenue off a completely new market and AngularJS is on the leading edge. This new market creates many new requirements for HTML5 applications. JavaScript applications are becoming commonplace in major companies. These companies are using JavaScript to showcase many different types of data from inward to outward facing products.

Working with live data sets in client-side applications is a common practice and is the real world standard. Most of the applications today use some type of live data to accomplish some given set of tasks. These tasks rely on this data to render views that the user can visualize and interact with. There are many advantages of working with the Web for data visualization, and we are going to showcase how these tie into an AngularJS application.

AngularJS offers different methods to accomplish a view that is in charge of elegantly displaying large amounts of data in very flexible and snappy formats. Some of these different methods feed directives' data that has been requested and resolved, while others allow the directive to maintain control of the requests. We will go over these different techniques of how to efficiently get live data into the view layer by creating different real-world examples. We will also go over how to properly test directives that rely on live data to achieve their view successfully.

Techniques that drive directives

Most standard data requirements for a modern application involve an entire view that depends on a set of data. This data should be dependent on the current state of the application. The state can be determined in different ways. A common tactic is to build URLs that replicate a snapshot of the application's state. This can be done with a combination of URL paths and parameters.

URL paths and parameters are what you will commonly see change when you visit a website and start clicking around. An AngularJS application is made up of different route configurations that use the URL to determine which action to take. Each configuration will have an associated controller, template, and other forms of options. These configurations work in unison to get data into the application in the most efficient ways.

AngularUI also offers its own routing system. This UI-Router is a simple system built on complex concepts, which allows nested views to be controlled by different state options. This concept yields the same result as ngRoute, which is to get data into the controller; however, UI-Router does it in a more eloquent way, which creates more options. AngularJS 2.0 will contain a hybrid router that utilizes the best of each.

Once the controller gets the data, it feeds the retrieved data to the template views. The template is what holds the directives that are created to perform the view layer functionality. The controller feeds directives' data, which forces the directives to rely on the controllers to be in charge of the said data. This data can either be fed immediately after the route configurations are executed or the application can wait for the data to be resolved.

AngularJS offers you the ability to make sure that data requests have been successfully accomplished before any controller logic is executed. The method is called resolving data, and it is utilized by adding the `resolve` functions to the route configurations. This allows you to write the business logic in the controller in a synchronous manner, without having to write callbacks, which can be counter-intuitive.

The XHR extensions of AngularJS are built using `promise` objects. These `promise` objects are basically a way to ensure that data has been successfully retrieved or to verify whether an error has occurred. Since JavaScript embraces callbacks at the core, there are many points of failure with respect to timing issues of when data is ready to be worked with. This is where libraries such as the Q library come into play. The `promise` object allows the execution thread to resemble a more synchronous flow, which reduces complexity and increases readability.

The $q library

The `$q` factory is a lite instantiation of the formally accepted Q library (`https://github.com/kriskowal/q`). This lite package contains only the functions that are needed to defer JavaScript callbacks asynchronously, based on the specifications provided by the Q library. The benefits of using this object are immense, when working with live data.

Basically, the `$q` library allows a JavaScript application to mimic synchronous behavior when dealing with asynchronous data requests or methods that are not thread blocked by nature. This means that we can now successfully write our application's logic in a way that follows a synchronous flow.

ES6 (**ECMAScript6**) incorporates promises at its core. This will eventually alleviate the need, for many functions inside the `$q` library or the entire library itself, in AngularJS 2.0.

The core AngularJS service that is related to CRUD operations is called `$http`. This service uses the `$q` library internally to allow the powers of promises to be used anywhere a data request is made. Here is an example of a service that uses the `$q` object in order to create an easy way to resolve data in a controller. Refer to the following code:

```
this.getPhones = function() {
  var request = $http.get('phones.json'),
  promise;

  promise = request.then(function(response) {
    return response.data;
    },function(errorResponse){
      return errorResponse;
      });

  return promise;
}
```

Here, we can see that the `phoneService` function uses the `$http` service, which can request for all the phones. The `phoneService` function creates a new request object, that calls a `then` function that returns a `promise` object. This `promise` object is returned synchronously. Once the data is ready, the `then` function is called and the correct data response is returned.

This service is best showcased correctly when used in conjunction with a `resolve` function that feeds data into a controller. The `resolve` function will accept the `promise` object being returned and will only allow the controller to be executed once all of the phones have been resolved or rejected.

The rest of the code that is needed for this example is the application's configuration code. The `config` process is executed on the initialization of the application. This is where the `resolve` function is supposed to be implemented. Refer to the following code:

```
var app = angular.module('angularjs-promise-example',
  ['ngRoute']);

app.config(function($routeProvider){
  $routeProvider.when('/', {
    controller: 'PhoneListCtrl',
    templateUrl: 'phoneList.tpl.html',
    resolve: {
      phones: function(phoneService){
        return phoneService.getPhones();
        }
      }
    }).otherwise({ redirectTo: '/' });
})

app.controller('PhoneListCtrl', function($scope, phones) {

  $scope.phones = phones;

});
```

A live example of this basic application can be found at `http://plnkr.co/edit/f4ZDCyOcud5WSEe9L0GO?p=preview`.

Directives take over once the controller executes its initial context. This is where the `$compile` function goes through all of its stages and links directives to the controller's template. The controller will still be in charge of driving the data that is sitting inside the template view. This is why it is important for directives to know what to do when their data changes.

How should data be watched for changes?

Most directives are on a need-to-know basis about the details of how they receive the data that is in charge of their view. This is a separation of logic that reduces cyclomatic complexity in an application. The controllers should be in charge of requesting data and passing this data to directives, through their associated `$scope` object.

Directives should be in charge of creating DOM based on what data they receive and when the data changes. There are an infinite number of possibilities that a directive can try to achieve once it receives its data. Our goal is to showcase how to watch live data for changes and how to make sure that this works at scale so that our directives have the opportunity to fulfill their specific tasks.

There are three built-in ways to watch data in AngularJS. Directives use the following methods to carry out specific tasks based on the different conditions set in the source of the program:

- Watching an object's identity for changes
- Recursively watching all of the object's properties for changes
- Watching just the top level of an object's properties for changes

Each of these methods has its own specific purpose. The first method can be used if the variable that is being watched is a primitive type. The second type of method is used for deep comparisons between objects. The third type is used to do a shallow watch on an array of any type or just on a normal object.

Let's look at an example that shows the last two watcher types. This example is going to use jsPerf to showcase our logic. We are leaving the first watcher out because it only watches primitive types and we will be watching many objects for different levels of equality.

This example sets the `$scope` variable in the app's `run` function because we want to make sure that the jsPerf test resets each data set upon initialization. Refer to the following code:

```
app.run(function($rootScope) {
 $rootScope.data = [
  {'bob': true}, {'frank': false}, {'jerry': 'hey'}, {'bargle':
    false},
  {'bob': true}, {'bob': true}, {'frank': false}, {'jerry':
    'hey'},{'bargle': false},{'bob': true},{'bob':    true},
    {'frank': false}];
});
```

This `run` function sets up our `data` object that we will watch for changes. This will be constant throughout every test we run and will reset back to this form at the beginning of each test.

Doing a deep watch on $rootScope.data

This `watch` function will do a deep watch on the `data` object. The `true` flag is the key to setting off a deep watch. The purpose of a deep comparison is to go through every object property and compare it for changes on every digest. This is an expensive function and should be used only when necessary. Refer to the following code:

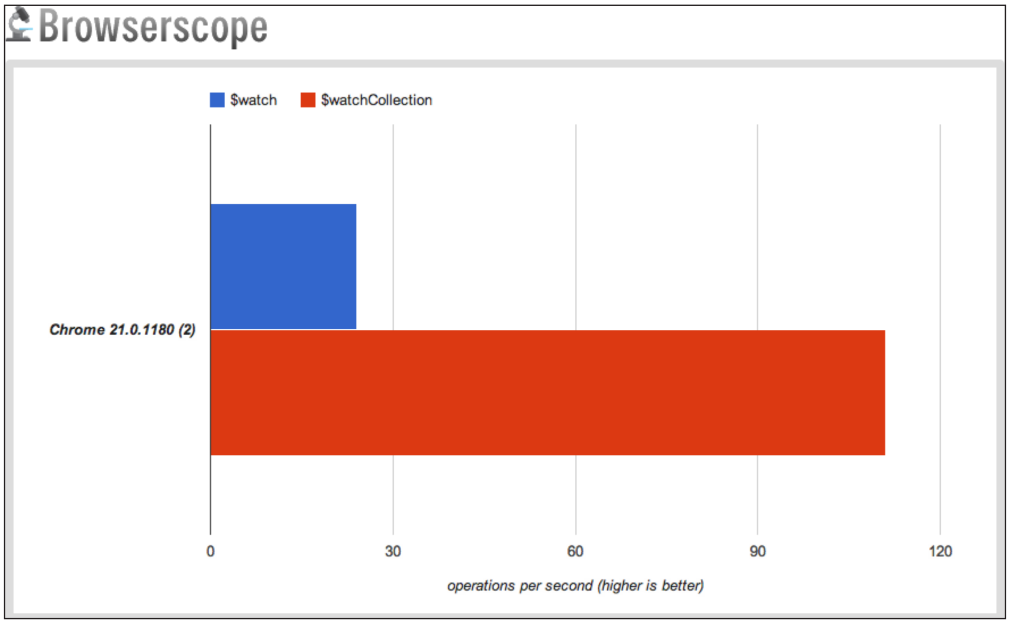
```
app.service('Watch', function($rootScope) {
  return {
    run: function() {
      $rootScope.$watch('data', function(newVal, oldVal) {
    },true);
    //the digest is here because of the jsPerf test. We are using
      this run function to mimic a real environment.
    $rootScope.$digest();
  }
};
});
```

Doing a shallow watch on $rootScope.data

The shallow watch is called whenever a top-level object is changed in the data object. This is less expensive because the application does not have to traverse *n* levels of data. Refer to the following code:

```
app.service('WatchCollection', function($rootScope) {
  return {
    run: function() {
      $rootScope.$watchCollection('data', function(n, o) {
    });
  $rootScope.$digest();
  }
};
});
```

During each individual test, we get each watcher service and call its `run` function. This fires the watcher on initialization, and then we push another `test` object to the data array, which fires the watch's `trigger` function again. That is the end of the test. We are using `jsperf.com` to show the results. Note that the `watchCollection` function is much faster and should be used in cases where it is acceptable to shallow watch an object. The example can be found at `http://jsperf.com/watchcollection-vs-watch/5`. Refer to the following screenshot:



This test implies that the `watchCollection` function is a better choice to watch an array of objects that can be shallow watched for changes. This test is also true for an array of strings, integers, or floats.

This brings up more interesting points, such as the following:

- Does our directive depend on a deep watch of the data?
- Do we want to use the `$watch` function, even though it is slow and memory taxing?
- Is it possible to use the `$watch` function if we are using large `data` objects?

So far, the directives that have been used in this book have used the `watch` function to watch data directly, but there are other methods to update the view if our directives depend on deep watchers and very large data sets.

Directives can be in charge

There are some libraries that believe that elements can be in charge of when they should request data. Polymer (`http://www.polymer-project.org/`) is a JavaScript library that allows DOM elements to control how data is requested, in a declarative format. This is a slight shift from the processes that have been covered so far in this chapter, when thinking about what directives are meant for and how they should receive data. Let's come up with an actual use case that could possibly allow this type of behavior.

Let's consider a page that has many widgets on it. A widget is a directive that needs a set of large `data` objects to render its view. To be more specific, lets say we want to show a catalog of phones. Each phone has a very large amount of data associated with it, and we want to display this data in a very clean simple way.

Since watching large data sets can be very expensive, what will allow directives to always have the data they require, depending on the state of the application? One option is to not use the controller to resolve the Big Data and inject it into a directive, but rather to use the controller to request for directive configurations that tell the directive to request certain `data` objects. Some people would say this goes against normal conventions, but I say it's necessary when dealing with many widgets in the same view, which individually deal with large amounts of data.

This method of using directives to determine when data requests should be made is only suggested if many widgets on a page depend on large data sets.

To create this in a real-life example, let's take the `phoneService` function, which was created earlier, and add a new method to it called `getPhone`. Refer to the following code:

```
this.getPhone = function(config) {
  return $http.get(config.url);
};
```

Now, instead of requesting for all the details on the initial call, the original `getPhones` method only needs to return `phone` objects with a `name` and `id` value. This will allow the application to request the details on demand. To do this, we do not need to alter the `getPhones` method that was created earlier. We only need to alter the data that is supplied when the request is made.

[It should be noted that any directive that is requesting data should be tested to prove that it is requesting the correct data at the right time.]

Testing directives that control data

Since the controller is usually in charge of how data is incorporated into the view, many directives do not have to be coupled with logic related to how that data is retrieved. Keeping things separate is always good and is encouraged, but in some cases, it is necessary that directives and XHR logic be used together. When these use cases reveal themselves in production, it is important to test them properly.

So far, this book has not been focused on tests, which care about how data is retrieved and whether that data is correct. The previous tests in the book use two very generic steps to prove business logic. These steps are as follows:

- Create, compile, and link DOM to the AngularJS digest cycle
- Test scope variables and DOM interactions for correct outputs

Now, we will add one more step to the process. This step will lie in the middle of the two steps. The new step is as follows:

- Make sure all data communication is fired correctly

AngularJS makes it very simple to allow additional resource related logic. This is because they have a built-in backend service mock, which allows many different ways to create fake endpoints that return structured data. The service is called `$httpBackend`.

Testing bbPhoneDetails

To showcase how to use `$httpBackend`, we have created tests for the `bbPhoneDetails` directive. The `bbPhoneDetails` directive makes requests for its own information. This information could be very large, which means special precautions need to be taken when requesting for many phones on the same page. This potentially large data is being separated by individual requests for each individual directive.

The `bbPhoneDetails` directive has a small set of requirements. Refer to the following requirements:

- Ability to request for data based on a configuration object
- If this configuration object changes, then request for new information
- Handle all error cases, with regards to requests, correctly

To write tests that prove these requirements, we start by creating a simple describe block that contains all of the services we will need. This describe block also contains our first look at how to use the `$httpBackend` service.

```
/*
  These tests showcase how directives can communicate with remote
    resources to accomplish their desired views.
*/
describe('bbPhoneListApp Demo', function () {
  'use strict';

  var scope, $compile, $httpBackend;

  beforeEach(module('bbPhoneListApp'));
  beforeEach(inject(function (_$rootScope_,
    _$compile_,_$httpBackend_) {
    scope = _$rootScope_;
    $compile = _$compile_;
    $httpBackend = _$httpBackend_;

    $httpBackend.whenGET('test-phone.json')
    .respond({
        "age": 1,
        "id": "xxx-xxx-xxxx",
        "imageUrl": "testPhone.jpg",
        "name": "Amazing Phone",
        "snippet": "This is a Super Duper Phone"
        });

    $httpBackend.whenGET('test-phone2.json')
    .respond({
        "age": 2,
        "id": "yyy-xxx-xxxx",
        "imageUrl": "testPhone2.jpg",
        "name": "Cool Phone",
        "snippet": "This is a Super Amazing Phone"
        });
```

```
      $httpBackend.whenGET('error.json')
      .respond(404);
    }));

    beforeEach(function(){
        scope.configObj = {url: "test-phone.json"};
        successPhoneLinkFn = $compile('<div bb-phone-details
          config="configObj"></div>');
        errorPhoneLinkFn = $compile('<div bb-phone-details
          config="configObj"></div>');
      });
```

The `$httpBackend` service is being injected as any normal service. This is made possible because we have included `angular-mocks.js` in our grunt setup. You can refer to `https://github.com/joshkurz/Black-Belt-AngularJS-Directives/blob/master/karma.conf.js`.

The preceding described block is the parent-level closure that will hold all of our tests. The first `beforeEach` block calls the `bbPhoneListApp` module to inject its context into the scope of the test. The second `beforeEach` block contains the most important piece of code, with regards to how `bbPhoneDetails` accomplishes its requirements. This is the `$httpBackend` service. The third `beforeEach` clause defines the `link` functions that are used, which contain the compiled directives.

In the following tests, we will be using the `bbPhoneDetails` directive to make different requests. These different requests will expect different responses depending upon the actual request. To accomplish this functionality, we are using the `whenGET` method provided by the `$httpBackend` service. This method takes a string as a variable that will match a request that can be made in an `it` clause. If a match is made, then it will respond with the specified data, which will serve as the constant that will prove our tests are successful.

There are more `$httpBackend` functions available for specific testing cases. These can be found at `https://docs.angularjs.org/api/ngMock/service/$httpBackend`.

Refer to the following test case, which proves our first requirement:

```
it('should contain the correct scope parameters based upon the
configuration file', function(){
    successPhoneLinkedDOM = successPhoneLinkFn(scope);
    //apply needed, because the directive is watching the config
      for changes.
    //the directive's watch would never fire if this apply was not
      present.
    scope.$apply();
    //flush function will execute the $httpBackend functions that
      have
    //successfully matched. This will throw an error if nothing
    //matches.
    $httpBackend.flush();
    var phoneScope = successPhoneLinkedDOM.isolateScope();
    expect(phoneScope.phone.age).toBe(1);
    expect(phoneScope.phone.id).toBe("xxx-xxx-xxxx");
    expect(phoneScope.phone.imageUrl).toBe("testPhone.jpg");
    expect(phoneScope.phone.name).toBe("Amazing Phone");
    expect(phoneScope.phone.snippet).toBe("This is a Super Duper
      Phone");
});
```

This test proves that the `bbPhoneDetails` directive is making the correct requests based on the configuration object it is using as input. Since we are using the `flush()` function, provided by `$httpBackend`, we can show that specific requests are being made. This is because all requests that are made during the execution of this test will not execute their respond function until the `flush()` function is fired. In this specific case, the directive requests details for the `test-phone.json` file. The data is retrieved and used correctly inside the directive, and this is proven by testing each attribute of the phone for accuracy.

The next test case that we will write will be for the second logical pathway that this directive could use, depending on how the request it makes plays out. This will be the error scenario that could take place if a file was requested for, which did not exist. Refer to the following test case:

```
it('should contain a phone object that has only an error value',
  function(){
    scope.configObj.url = "error.json";
    errorPhoneLinkedDOM = errorPhoneLinkFn(scope);
    scope.$apply();
    $httpBackend.flush();
    var phoneScope = errorPhoneLinkedDOM.isolateScope();
    expect(phoneScope.phone.error).toBe('no file exists');
```

```
        expect(phoneScope.phone.age).toBe(undefined);
    });
```

The error scenario is proven by using the `$httpBackend` service to return a 404 error code when a specific request is made. The directive should handle this error and any other error correctly.

> Since we are using the AngularJS promise system inside of the directive, we can ensure that if a 404 error is handled correctly, then all other error scenarios will work.

The last requirement that must be met occurs when the directive's `configuration` object changes in any way. This should force a new request that will subsequently update the directive's scope objects with the new data. Refer to the following test case:

```
    it('should request for new data when the config file changes',
      function(){
        var successPhoneLinkedDOM = successPhoneLinkFn(scope);
        scope.$apply();
        $httpBackend.flush();
        scope.configObj.url = 'test-phone2.json';
        //force the directive to go through a digest cycle, which should
    fire a watch function
        //which should request for new data.
        scope.$apply();
        $httpBackend.flush();
        var phoneScope = successPhoneLinkedDOM.isolateScope();
        expect(phoneScope.phone.age).toBe(2);
        expect(phoneScope.phone.id).toBe("yyy-xxx-xxxx");
        expect(phoneScope.phone.imageUrl).toBe("testPhone2.jpg");
        expect(phoneScope.phone.name).toBe("Cool Phone");
        expect(phoneScope.phone.snippet).toBe("This is a Super Amazing
          Phone");
    });
```

This last test proves that the directive updates itself and makes new requests whenever its `configuration` object changes. The test is essentially the first test minus any `expect` functions plus more logic that changes `configObj.url` and checks to make sure that the updates were made correctly. The key to these tests is to make sure to call `flush()`, so that all the pending requests can be processed accordingly.

There are more details with regards to these specific tests, which can be found in the Black Belt repo at `https://github.com/joshkurz/Black-Belt-AngularJS-Directives/blob/master/directives/BigData/tests/bbPhoneDetails.spec.js`.

Now that the requirements have been laid out correctly, let's move on to writing the directive in order to see how we can actually make these tests pass.

Writing the bbPhoneDetails directive

Now, we can create a directive that uses an `isolated` scope and takes a `configuration` object that can be watched for changes. Any time that the `configuration` object changes, we can go ahead and make a request for the large data set. Refer to the following code:

```
app.directive('bbPhoneDetails', ['phoneService',
  function(phoneService){

    function link(scope,element,attrs,controller){

      scope.$watch('config', function(config){
        phoneService.getPhone(config).then(function(request) {
          scope.phone = request.data;
          },function(){
            scope.phone = {error: 'no file exists'};
            });
        },true);

      }

    return {
      restrict: 'A',
      templateUrl: function(tElem,tAttrs){
        return tAttrs.templateUrl || 'phoneDetails.tpl.html';
        },
      scope: {config: '='},
      link: link
      };

}]);
```

The `link` function used by `bbPhoneDetails` watches its `configuration` object for changes. Once it makes the change, it then calls the `getPhone` method provided by the `phoneService` function. This method returns a `promise` object, which is created by the `$http` service. This `promise` object is then resolved once the data has either been requested successfully or an error has occurred. The final result of all the directives working in unison is the same as when all the data is requested at once. The difference is that now the data requests can be scaled in manageable sections. A live example can be found at `http://plnkr.co/edit/mBm7zjGIrBt6GLn9Mo0r?p=preview`.

Remember, the objective of this section is to show how directives do not have to watch huge amounts of data. This is about scaling; even though the demo does not have a large amount of data being used, it does take this possibility into consideration. In some cases, separating the XHR requests into smaller chunks is faster than requesting for all of the data at once and helps create faster digest cycles. This also yields a better user experience, as the user does not have to wait for all of the data before the page finally loads. Now, they can see a portion of the page with lite details, and then they can be shown some type of loading message that the details are being obtained for each widget.

Working with D3

There are many external libraries that are meant for graphing and displaying large amounts of data in a nice consolidated, organized view. Some examples of libraries are `flot.js`, `datatables.js`, `heatmap.js`, and many more. We are going to describe how to work with one of the most popular visualization libraries.

D3 is very popular and has many different features. AngularJS and D3 work wonderfully together if used correctly. D3 has wonderful view-level techniques built inside that work perfectly with the data-binding abilities of AngularJS.

AngularJS watches the model for changes and calls functions depending on whether the data has changed. The directives written in this section will focus on calling the `D3` function to perform most of the very intrinsic DOM alterations. The unique portion will be how these directives are tested for accuracy in an AngularJS context and how they know when to call their D3 related functions.

This section will use a different approach to testing than those used in any of the previous chapters. We are going to introduce Protractor to test the `D3` directives. Since all of the directives created in this section will require some type of live data feed, we are going to showcase how to get this real data into the view while testing.

Protractor is an End-2-End testing framework built on top of the Selenium driver to allow AngularJS applications a simple way to test bindings and check whether interactions are working as expected. More details can be found at `https://github.com/angular/protractor`.

The YouTube views bar chart

In the Black Belt demo, there is a media element page. This media element page has a video player that is fed by a bootstrap `typeahead` directive. The `typeahead` directive calls a function in the controller that hits YouTube's API upon every keystroke. The returned data is a set of data objects that contain information about videos, in relation to what was typed into the `typeahead` directive.

Let's build a simple bar graph widget to show a visualization of this YouTube data and put it into the `uiTypeahead` drop-down list. This will help our users decide which video they would like to watch.

This bar chart is going to rely on D3 for all of its major DOM manipulations. The purpose of the example is to create a real-life example that uses live data. This directive is not in charge of the requests because the `uiTypeahead` directive takes care of that.

The `uiTypeahead` directive works with promises in a similar fashion as the `bbPhoneDetails` directive. The details of this directive can be found at `https://github.com/angular-ui/bootstrap/blob/master/src/typeahead/typeahead.js`.

All the bar chart has to do is know when it should update its DOM with scaled bars that resemble how many views each video has in relation to the other results. To build the YouTube bar chart, we need to alter the `uiTypeahead` template in order to render and feed our `bbBarChart` directive. This will allow the `uiTypeahead` directive to communicate its data to our `bbBarChart` directive. The communication will be through an `isolate` scope set inside of `bbBarChart`.

This is as simple as adding one extra `li` element to the `typeahead` template, as shown in the following code:

```
<li>
  <div bb-bar-chart data="matches" set-the-model="selectMatch">
</li>
```

Now, we actually need to write the directive. The directive will be a combination of specific D3 code and common AngularJS code that controls when the D3 code should be called.

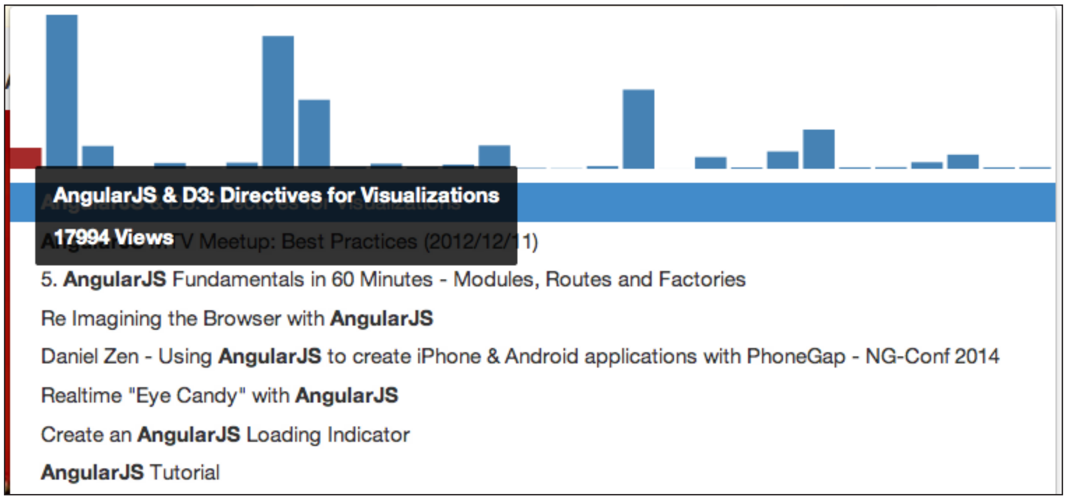
The most relevant piece to this chapter, which is the `watch` function, will be shown. The rest of the code can be found at `https://github.com/joshkurz/Black-Belt-AngularJS-Directives/blob/master/directives/BigData/bbStockChart.js`. Refer to the following code:

```
    angular.module('AngularBlackBelt.BigData',
     ['AngularBlackBelt.BigDataCharts'])
   .directive('bbBarChart', [ function(){

         function link(scope,element,attrs){

         //setting up the bar chart svg element
         var svg = d3.select(element[0])
         .append("svg")
         .attr("width", w)
         .attr("height", h);

         function redraw(data){
           svg.selectAll('*').remove();
           // redrawing the directive with d3 specific DOM
             manipulation code.
           //when we are calling this function we are also adding
           //event handlers that use the scope.setTheModel function
           //so we can communicate with the typeahead function and
             set
           //its internal model
           svg.selectAll("rect")
           .on('mouseover', tip.show)
           .on('mouseout', tip.hide)
           .on('click', function(event,clickData){
             scope.setTheModel(clickData);
             scope.$apply();
             });
         }

       scope.$watch('data', function(newData) {
           var graphData = [];
           angular.forEach(newData, function(dataItem) {
               var stats = dataItem.model['yt$statistics'];
           if (stats) {
             graphData.push({
               label: dataItem.label,
               value: parseInt(stats.viewCount, 10)
               });
           } else {
             graphData.push({
                 label: dataItem.label,
                 value: 0
                 });
             }

           if (graphData.length>0) {
             **redraw**(graphData);
             }
           }, true);}

         return {
           restrict: 'A',
           scope: {data: '=', setTheModel: "="},
           link: link
           };

   }]);
```

The `bbBarChart` function watches a set of data for changes. Specifically, it watches data coming from YouTube. This data is not in the exact format we need, to showcase which video has more views in a bar chart, so we massage the data a bit before we pass it to the `redraw` function.

Massaging data before a DOM is created is a normal part of working with client-side technologies. Sometimes, it is not possible to change the server, so the data must be altered before we can show it in a view of our choice. In this specific case, we are just taking the title and the number of views from the `data` object to be used in the `bbBarChart` directive, which renders the data to the view.

D3 takes care of the dirty work. This makes it possible to have a dynamic bar chart that scales perfectly every time we update our search parameters and new data is fed into the directive. This demo can be found at `http://angulardirectives.joshkurz.net/dist/#/mediaelement`. Refer to the following screenshot:


AngularJS & D3: Directives for Visualizations

17994 Views

AngularJS & D3: Directives for Visualizations

AngularJS-TV Meetup: Best Practices (2012/12/11)

5. AngularJS Fundamentals in 60 Minutes - Modules, Routes and Factories

Re Imagining the Browser with AngularJS

Daniel Zen - Using AngularJS to create iPhone & Android applications with PhoneGap - NG-Conf 2014

Realtime "Eye Candy" with AngularJS

Create an AngularJS Loading Indicator

AngularJS Tutorial

E2E tests for bbBarChart

To test the bar chart correctly, we are going to employ Protractor. This is a new library that is specifically used to write E2E tests for AngularJS. It works with the Selenium Web Driver and by default, works with Jasmine as well. This will allow us to spin up a browser window and actually type `AngularJS` into the input and make sure that all is working correctly, all with one command.

Once Protractor is set up, all we have to do is write a simple test that queries YouTube and checks to makes sure that `bbBarChart` is creating the correct output based upon what is being typed in. In our controller, we have specified a maximum results limit when querying YouTube. This limit is 30, and this is how we will ensure that the bar chart creates the correct number of bars. This test will also prove that by clicking on a specific bar, we are able to load YouTube videos into the `bbMediaPlayer` directive. Refer to the following test:

```
describe("bbBarChart live data interaction", function () {
    beforeEach(function() {
      browser.get('/dist/#/mediaelement');
      });

  it("should contain a typeahead element and a mediaelement which
    communicate via a barChart", function () {
      var typeahead = element(by.model('result')),
      bars = $$('.menuBar'),
      mediaelementSource = $$('.youtubeSourceObj');

      expect(bars.count()).toEqual(0);
      expect(mediaelementSource.count()).toEqual(0);
```

```
        typeahead.sendKeys('AngularJS');
        expect(bars.count()).toEqual(30);
        bars.first().click();
        expect(mediaelementSource.count()).toEqual(1);
        });

});
```

This is a simple test that will prove our `d3BarChart` directive is working and is also working with other directives properly. Some of the individual expectations could be proven in a unit test and would probably be better suited for a unit test. This was a brief explanation of Protractor and how it can be used. Using E2E tests in this manner can be very powerful. It is also a great spectacle to see your site running in an automated fashion.

The stockTicker directive

Now, let's keep this D3 train rolling, but let's speed it up a bit with data that is updated by sockets, rather than by user input. Say, we want to create a directive that works with live ticker updates to the stock market. This would be very useful if we had users who are interested in specific stocks. To accomplish this, we are going to use a JavaScript library called PubNub (`http://www.pubnub.com/`).

One of the most intrinsic portions of the `bbStockTicker` directive is actually not the directive itself. It is going to be the `pubnubService` function. The `pubnubService` function is going to take care of all of our subscriptions and ticker updates that we care about. This is going to be our central hub for communication with the sockets that we are subscribed to.

To accomplish this, we are going to create a service. This service will initialize the `pubnub` object that will be used inside our controllers, which will update our directive's data. The `pubnub` service will also be in charge of subscribing to each ticker that we care about. The tickers will be specific to each view, which means that the service does not care about which ticker is being subscribed to because it could be any ticker. Refer to the following code:

```
angular.module('AngularBlackBelt.demo/BigData',[])
.service('pubnubService', ['$timeout', function($timeout){

  var self = this,
  unsubscribed = {};

  self.pubnub = PUBNUB.init({
      subscribe_key : 'demo',
      publish_key   : 'demo'
```

```
        });

      self.pubnubStockData = {};

      self.subscribeToTicker = function(ticker){
        delete unsubscribed[ticker];
        self.pubnub.subscribe({
            channel : ticker,
            message : $.throttle(3000, function(update,data){
            if(!unsubscribed[ticker]){
              $rootScope.$apply(function(){
                self.pubnubStockData[ticker] = update;
                });
            }
        })
      });
    };

      self.unsubscribeToTicker = function(ticker){

        delete self.pubnubStockData[ticker];
        unsubscribed[ticker] = true;
        self.pubnub.unsubscribe({channel: ticker});

      };

    }]);
```

The service allows our controllers to easily subscribe to any ticker they please without reusing code. One thing of interest in the controller is the `throttle` method that is being used. Ben Alman, the creator of Grunt, wrote this `throttle` function, which can be found at `https://github.com/cowboy/jquery-throttle-debounce`. We are using this function to ensure that the `message` function callback is only called every three seconds because PubNub sends us information very often, which causes way too many `digest` calls. These `digest` calls can be a source of slowness in our app, which is something that we do not want.

The next step is to actually subscribe to the tickers of our choice. To do this, we just run through an array of tickers and subscribe to each. Refer to the following code:

```
    .controller('BigDataCtrl', ['$scope', 'pubnubService',
      function($scope, pubnubService){
      $scope.tickers = ['ORCL', 'ZNGA', 'EA', 'F', 'FB' , 'TRI'];
      for(var tic in $scope.tickers){
        pubnubService.subscribeToTicker($scope.tickers[tic]);
        }
      $scope.stockData = pubnubService.pubnubStockData;
    }])
```

The controller uses the `pubnub` service to subscribe to six tickers with very little code. This is a nice, easy way to utilize the `pubnub` library. This type of implementation can be used for any publish/subscribe design pattern implemented in any AngularJS application.

The next step is to write the directive, but first let's come up with a short list of requirements, as follows:

- The stock chart should update itself whenever its data set changes
- There should be a way to add and remove individual stock symbols

Now, the directive needs to be written. Since we are using D3, there will be a little bit more code specific to D3 in the directive, which is not specifically relevant to this chapter, so we will leave this out. The full directive can be found at `https://github.com/joshkurz/Black-Belt-AngularJS-Directives/blob/master/directives/BigData/bbStockChart.js`. Refer to the following code:

```
angular.module('AngularBlackBelt.BigDataCharts', [])
.directive('bbStockChart', [ function(){

    function link(scope,element,attrs){

      var limit = 60 * 1,
      duration = 750,
      now = new Date(Date.now() - duration),
      color = d3.scale.category20(),
      max = 20,
      x, y, line, svg, axis, paths;

      var width = element.width(),
      height = 500,
      groups = {};

      function resetGroups(tickers){
        //reset the groups the directive
        //uses to create the lines
      }

      function renderGraph() {
        //tick the graph over and redraw the lines
        //this function uses D3 to update the new data
```

```
          //as a new value on the line.
          //we are leaving the D3 code out of this example.
          }

        var killLengthWatcher = scope.$watch('tickers.length',
          function(newVal){
          //reset the element completly
          element.html('');
          //reset the max value so we can rescale
          max = 20;
          //reset the current active groups
          resetGroups(scope.tickers);
          });

      //whenever the data changes we are calling renderGraph which moves
      //the line over with the new data points.
      var killWatcher = scope.$watchCollection('data', renderGraph);

      scope.$on('$destroy', function(elem){
          killWatcher();
          killLengthWatcher();
          scope.data = null;
          scope.tickers = null;
          });
      }

      return {
        restrict: 'A',
        scope: {data: '=',tickers: '='},
        link: link
        };

  }]);
```

The `bbStockChart` directive watches its data for changes. We are using the `watchCollection` function to do this because the ticker pushes new objects to each ticker's array, which will be picked up by the shallow `watch` function. As we have seen, this is much faster than using the normal `$watch` function. Once the data changes, which will be every time PubNub pushes some new data to a `ticker` object, the D3 graph will tick over one data point. This is done by calling some specific D3 logic to reset and rescale the graph based on a shifted array of data.

The directive also watches the `ticker` array. So, whenever a new ticker is added or removed, the directive will update itself with the correct data. The ability to manipulate which `data` objects are being used in the visualization is one of the simplest benefits of AngularJS. These simple steps make the directive responsive to any data changes that are made inside the controller's logic.

Again, the logic is not super advanced, but the overall output of this directive and of all the pieces working together is a nice working component that updates itself automatically. This can be seen at `http://angulardirectives.joshkurz.net/dist/#/stockchart`.

E2E tests for bbStockChart

The E2E tests that have been written for the `bbStockChart` prove that the stock chart updates itself whenever something changes in its subscribed groups. This will be done by actually clicking buttons in our view, which subscribe and unsubscribe tickers to live pubnub data. We are also going to prove that the directive is working with real data by making sure that each path in our chart is using the correct classnames, which correlate to the data that the pubnub service is subscribed too. Refer to the following tests:

```
describe("bbStockChart live data interaction", function () {
  beforeEach(function() {
      browser.get('/dist/#/stockchart');
      });

  it("should contain a d3 chart", function () {
      var elements = $$('.chart');
      expect(elements.count()).toBe(1);
      });

  it("should contain 6 path elements with the correct class
    names", function () {
      var orcl = $$('.ORCL'),
      znga = $$('.ZNGA'),
      ea = $$('.EA'),
      f = $$('.F'),
      fb = $$('.FB'),
      tri = $$('.TRI');
```

```
        expect(orcl.count()).toBe(1);
        expect(znga.count()).toBe(1);
        expect(ea.count()).toBe(1);
        expect(f.count()).toBe(1);
        expect(fb.count()).toBe(1);
        expect(tri.count()).toBe(1);
        });

    it('should add/remove GOOG to the graph when the add/remove
      button is clicked', function() {
        var googAddBtn = element(
          by.repeater('ticker in addTickers').
          row(0)),
        googRemoveBtn = element(
          by.repeater('ticker in tickers').
          row(6)),
        goog = $$('.GOOG');

        expect(goog.count()).toEqual(0);
        googAddBtn.click();
        expect(goog.count()).toEqual(1);
        googRemoveBtn.click();
        expect(goog.count()).toEqual(0);
        });
  });
```

When using Protractor, there is some special functionality to learn. This functionality deals with selecting elements in the view to test. There are many different ways to select elements inside of an AngularJS application with Protractor. In this example, we are using `$$` and the `element.by` function to get items of interest. The elements that are retrieved are tested for correctness, and if all is good, the tests pass.

D3 works wonderfully with AngularJS, depending upon the implementation details. AngularJS offers many ways to separate logic that is relative to how data is requested. Depending on the set of given requirements, certain steps should be taken to bring data into the view. If this is done correctly, no matter which library is used to visualize the data, it will work in collaboration with AngularJS.

Summary

Using live data in HTML5 applications is a must for production-level apps. What is important, is the data and how this data is displayed. There are many different ways to harness large amounts of data on the server, and these methods are becoming more commonplace in mom and pop start-up companies. This means that the deciding benefactor is no longer who can yield the most data, but what can be done with this data.

AngularJS offers many ways to request data and feed the view layer data that can be transformed into different visualizations. These methods include using combinations of services, directives, and controllers to manage how data should be requested and when it should be requested. Usually, the directive's job is to determine when its data changes and to update its view accordingly.

There are special cases when dealing with data at scale, which deem it ok for directives to watch the `configuration` objects that tell the directive how to request data. This method is handy; however, it should not be the first solution of a directive's architecture. Having the controller resolve the data into the view separates the logic, which overall makes the app simpler and easier to work with.
If a directive must request its own data, then it should be tested properly.

There are many ways to visualize data, and many libraries are built specifically for this. These libraries can be used in association with directives to accomplish wonderful components that help users visualize data in many different ways. AngularJS can help drive many visualization tools by allowing various ways to feed the view with large amounts of data. This was made apparent in this chapter.

Protractor allows AngularJS developers an easy way to write E2E tests in a similar fashion as unit tests. These tests use the Selenium Web Driver to support some amazing things, all powered by the console. It is highly encouraged to write E2E tests when working with applications that deal with live data.

7
Optimization and Code Quality

The demand for HTML5 applications has become greater and greater in the past few years. To create the utmost preferment and extendable website, a strong focus needs to be put on optimization and quality code. AngularJS helps us do both with its design pattern constructs, which offer the ability to reuse code in many different parts of an application.

AngularJS takes advantage of many different design patterns and allows you to make an organized application which can be easily extended and optimized. Some of the high-level design patterns include but are not limited to modules, factories, prototypes, singletons, and facades. These design patterns are used throughout the core of AngularJS, and their implementation details have been made public for use in our applications. This allows any application to take advantage of these basic design patterns simply and efficiently.

The main advantage of being able to organize HTML5 applications with proper design patterns is the ability to easily implement efficient and reusable code. Directives offer different techniques that can clean up code and make logical executions much faster. AngularJS creates a simple mechanism to separate the view and model, which allows for increased focus on each individually.

This chapter is broken down into two parts. The first part is about how AngularJS's internal designs allow us to write better code. The second part of the chapter is about optimization in an AngularJS context and how directives play a huge role in this optimization.

AngularJS code quality

AngularJS offers many ways to write good quality JavaScript and HTML. Each language has its own specific implementation techniques that allow better performance and overall better organization. Being able to write orthogonal HTML applications relies upon the separation of the model and the view.

HTML is a domain-specific language that is rigid and built for a specific purpose. Today's Web is moving faster than what any W3C board can keep up with, and because of this, the Web must create more specific domain languages. AngularJS has opened a new vantage point in the reality of what an HTML5 application is. This reality showcases HTML that is built specifically for the application it lives in.

The importance of templates

HTML and CSS have no real design pattern constructs. There are ways to organize and define websites that follow strict nomenclature and design standards, but there is no way to utilize facades or other widely used design patterns in HTML. This is changing as the Web moves forward and HTML becomes more advanced due to what developers are able to achieve with it.

Using directives that focus on templates can drastically reduce redundancy and the total development effort as they allow a more forward approach to the actual implementation structure of the view. By focusing on clean, organized HTML, the abilities to reduce LOC and increase the overall readability are revealed.

The most important directive that provides templating possibilities is `ngInclude`. This directive is very important and deserves special attention. The ability to separate logical portions of an application into separate files is important. Reading and digesting complex information is difficult enough, without the added complexity of unstructured, unorganized code.

By this point, we should already know how to use `ngInclude` or other directives that use templates. This section is not about how to use them, but more about why it is important to use them. To showcase the difference made by using `ngInclude`, let's take a look at the following simple example:

```
<div class="navbar navbar-inverse navbar-fixed-top"
  role="navigation">
<div class="container">
  <div class="navbar-header">
  <button type="button" class="navbar-toggle"
    ng-click="showMobileNav = !showMobileNav;"
    data-target=".navbar-collapse">
  <span class="sr-only">Toggle navigation</span>
```

```
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        </button>
      <a class="navbar-brand" href="#">Black Belt Directives</a>
      </div>
        <div class="collapse navbar-collapse" ng-class="{'in':
          showMobileNav}">
        <div ng-include="'directives/demo/templates/bbNavMenu
          tpl.html'"></div>
        </div><!--/.nav-collapse -->
        </div>
      </div>
```

This example shows how clean HTML can be by using `ngInclude`. Creating DOM elements that have an apparent declaration of where the template lives in the application is beneficial because now we can easily navigate directly to the `bbNavMenu` template and make any necessary edits.

Another great use case that can be solved by `ngInclude` is to replace redundant code with a template. It is advised to use a template whenever there are chunks of HTML that have a similar structure. If we follow through with this practice, all future updates to a site will be much less invasive and less prone to incidental errors.

The application's DOM structure is very important and is overlooked by some developers. AngularJS has built many tools into its framework to allow a perfect, concise view construction. Applications that can be updated and redesigned faster are the applications that will ultimately survive longer in any production environment.

Necessary DOM manipulations

DOM manipulation functions are the most expensive out-of-the-box methods that ECMAScript offers. This is because in order to accomplish DOM manipulation, the browser actually fires low-level C++ code to accomplish any set of given tasks. Saving the amount of DOM manipulations performed can be a very large optimization booster.

AngularJS separates the model and the view. This statement cannot be said enough number of times in a chapter about optimization and organization. For many years, JavaScript business logic and DOM manipulation have been munged together to create websites in order to provide expected results. Although this process works, it is not optimal and causes more unforeseen issues than the structured AngularJS approach.

AngularJS offers the ability to allow much less reflows in an application. It is a known fact that reducing the amount of reflows in an application can cause a significant increase in performance. More information about reflows can be found at `https://developers.google.com/speed/articles/reflow?hl=ru`.

Almost all DOM alterations are based on model changes that occur in memory. AngularJS allows applications to use more optimized solutions to control DOM manipulation executions and minimize reflow. This can be done by only touching the DOM when the model is stable. A stable model means that all the business logic has completed execution. Once the model is stable, then, and only then, do we adjust the DOM.

Directives serve as the separation between the model and the view. This has been explained in many different ways throughout this book; however, its purpose has not been fully portrayed. AngularJS uses modularized encapsulation to provide the ability to logically separate HTML5 applications into their associative categories. This separation, if used correctly, can bring out the best that JavaScript has to offer.

As in any project, the overall code structure is up to the developer. Using AngularJS does not automatically make an application good; however, it does shed light on the proper pathways.

Optimization of the directives

User experience in an application is one of the biggest factors, if not the biggest, to its overall grade. No application should ever be given a passing grade if it is not responsive and has a sluggish feel to it. A JavaScript application has the chance to be on either ends of the speed spectrum because of the wide range of implementation options. There can be more than a few potential bottlenecks in a JavaScript application, and even more in an AngularJS application, which should be noted and approached correctly.

The biggest issue with any AngularJS application that uses the 1.2.0 and above branch is the digest cycle.

The digest cycle is an AngularJS-specific issue and is the cause of most speed-related issues. This is because of the thread-blocking nature of the working of the digest cycle and the issues when working with a lot of data.

Tools for monitoring performance

There are multiple monitoring tools available in each browser that we can take advantage of. Becoming familiar with the development tools available in Chrome, Firefox, and Safari is highly recommended as it can help show the main points in an application's logic. Chrome offers specific tools for AngularJS development, which allows the monitoring of the digest cycle and other specific AngularJS attributes to ensure optimum performance. Batarang, built by Brian Ford and Lukas Ruebbelke, is the name of the custom Chrome plugin. This tool really helps get a higher-level perspective of the internals of an AngularJS app.

The digest cycle

AngularJS performs data binding by using a method called dirty checking. Data binding is broken down by individual bindings, which are also individual `watcher` functions. A `watcher` function is a function that is added to a scope's `watcher` array. A scope is a JavaScript object which has prototype methods that allow the organization and communication of data within an AngularJS application. Each scope has its own array of `watcher` functions called `$$watchers`. Each `watcher` object in the array has a `closure` function that has the knowledge of what changes have occurred to its specific data set.

Running a scope's `$digest` function affects that scope and all of its children. Each `watcher` function in a scope's `$$watchers` array is in charge of checking if the data it cares about has changed. This entire `$digest` process is the slowest part of AngularJS and can cause very significant performance-related issues if not cautiously maintained. Knowing the details of the digest cycle can help when trying to understand how to write fast directives.

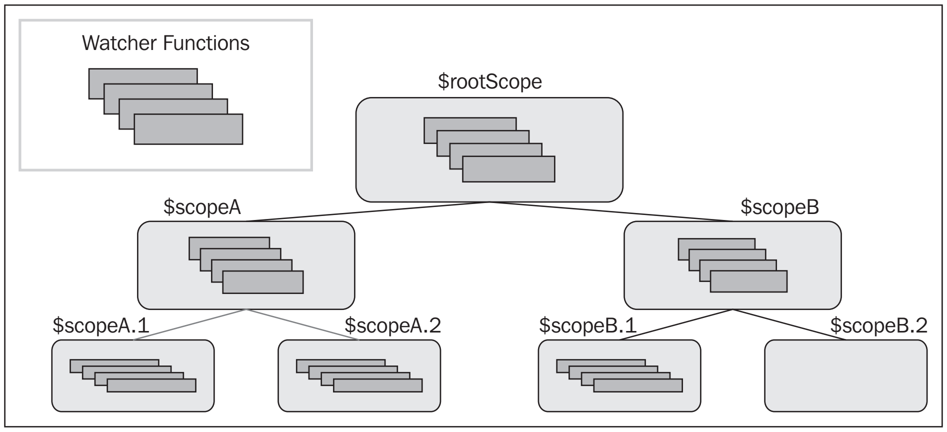
The latest AngularJS code base has incorporated one-time expressions that allow for bindings to occur only once.

At a high-level, the digest cycle is the process of watching data for changes and running a function when the data does actually change. Some core directives automatically set up the `watcher` functions, which can cause unknown slowness to beginners. Another issue to shed light on is on the fact that custom `watchers` can be created manually from any location where a scope is available. Each `watcher` function that gets created is pushed to its associated scope's `watcher` array and is run any time its scope calls its `$digest` function.

There are many core directives that automatically set up `watchers` on their model. The `ngRepeat`, `ngIf`, `ngSwitch`, `ngHide`, and `ngModel` directives set up `watchers` on the data that is used to build their view. These are just a few of the core directives that set up `watchers` behind the scenes. Using brackets `{{}}` in AngularJS also sets up the `watcher` functions for every expression found inside these brackets.

To create one-time expressions in the 1.3 AngularJS branch, all that is needed is `::` before any expression.

Creating all of these `watchers` is how AngularJS provides a magical appearance to newcomers who have just started using the framework. Almost every core directive sets up some kind of dynamic `watcher` on its elements' data. Since the core allows these `watcher` functions to look at variable expressions, the speed of the implementations are up to the developer. Let's take a bird's-eye view of an application and see what happens when a scope calls the `$digest` function. Refer to the following diagram:



In the previous screenshot, we have a tree of scopes that have parent-child relationships. Almost every scope has its own array of `watcher` functions that will fire during every digest. Let's say `$scopeA` calls its `$digest` function. This would mean that `$scopeA.1` and `$scopeA.2` will also call their `$digest` functions, subsequently firing every `watcher` in the parent-child relationship.

Usually, `$apply` is called, which fires `$digest` from the root element and puts every scope into its `$digest` cycle. This function calls all `watchers` at least twice.

Once a scope's `$digest` function is called, all of its children will also fire their associated `$digest` functions. Many times, the digest cycle is initialized by calling `scope.$apply`, which is the same as calling `$rootScope.$digest`. The difference between the two is that `$scope.$apply` calls each scope's `digest` function a minimum of two times. This is why the digest cycle is the biggest cause of optimization concerns.

The `$apply` function calls every scopes's `$digest` function twice for stabilization purposes, as a previous `$digest` could have triggered a dirty flag. If this process triggers an endless amount of `$digest` functions, AngularJS will throw the infamous `$digest() iterations reached. Aborting!` error.

The `watcher` functions can be set from anywhere inside of an execution context in which an AngularJS `scope` object is available. However, most use cases call for directives to set up `watchers` because a directive must know when and how to update the view. The more the `watcher` functions on each scope, the slower an application will be. There are a number of ways to speed up applications by creating directives that accomplish their tasks without setting up the `watcher` functions, by not doing repetitive DOM manipulation, or not running unnecessary digests. These techniques are a part of the optimization techniques that we will cover.

Less bindings yield faster results

The amount of bindings created on a single scope is a major source of failure for the overall responsiveness of an AngularJS application. Directives can help fix this by only setting up `watchers` when necessary. The following are the two scenarios that require some type of data binding:

- Template needs to be initially interpolated with data and watched for changes
- Template just needs its initial value and will never be updated

Interpolation is AngularJS's process of taking data from the `scope` objects and interjecting it into a template by evaluating a template string onto a scope's matching data set.

To accomplish one-time binding expressions in pre 1.3 branches, there are multiple workarounds. The advantages of setting up one-time bindings is immensely important to application views that do not need to change once they are initialized.

There are a few open source libraries that one-time binding expressions in pre 1.3 branches, there are offer functionalities based on this one-time data binding philosophy, which can be found at `https://github.com/Pasvaz/bindonce` and `https://github.com/abourget/abourget-angular`. Both libraries attempt to accomplish this one-time data binding need that AngularJS lacks. The bindonce repository is a more thorough implementation and provides many of the directives needed for this type of functionality. Its major fault is a lack of tests.

Solving the problem with the bbOneBinders directive

Let's look at a simple example of what we are trying to accomplish. Think about a list of places that have an image and a title. Places do not change their values, so there is no need for them to have bindings that watch for their values to change.

Here we have a list of places. The places in the list can change, but the places themselves will not. So, it is not important to keep bindings attached to the place's values. These unnecessary bindings can slow down an application dramatically depending upon the number of items in a list. Refer to the following code:

```
<div ng-app="bbOneBinders" ng-controller="oneBindCtrl">
  <div ng-repeat="place in places">
    <div bb-one-bind-text="place.title"></div>
    <a bb-one-bind-href="place.src"><img class="sampleImg"
      bb-one-bind-src="place.src"></a>
  </div>
    <div bb-one-bind-text="places[0].title"></div>
    <a bb-one-bind-href="places[0].src"><img class="sampleImg"
      bb-one-bind-src="places[0].src"></a>
</div>
```

This list is relatively simple to achieve without any custom directives. This example is not about whether it can be done, but rather about how to do something in the most efficient manner. The preceding example is going to generate a list of geographic places. These places have attributes that will never change, which means that we do not need to watch whether they change. The only piece of data we want to use is the initial value. This causes a need to update its associated DOM element, and then never have to change the element again. The `ngRepeat` directive will provide the list with the ability to change the list of places if need be and it will also allow you to set the places by any value set after the initial digest.

The example also uses a second set of HTML elements, which are highlighted in the preceding code example. This set of HTML elements must not only set their associated DOM, but also need to be able to wait for this data to actually be available. This is because the data may not be there at the initial load, and there needs to be a way to wait for it. A live example of the demo can be found at `http://jsfiddle.net/joshkurz/nZJEp/2/`.

To accomplish this task, we are going to create a set of directives that work with a multitude of different DOM element values. These values will initially watch the data for changes, and once the `watcher` function receives the first value, it will call its associated jQlite method and then immediately remove its `watcher` function.

The bbOneBinders directive

This block of code creates ten directives. Each directive performs very similar tasks, which is why it was chosen to write them all in a condensed fashion. The `bbOneBind` directives are created inside a `forEach` loop. This loop iterates over a list of objects that contain relative information for each directive. Then, inside of the loop, there is a simple logic that names each directive and sets their functionality as a specific link function. Refer to the following code:

```
angular.forEach([
    {tag: 'Src', method: 'attr'}, {tag: 'Text', method: 'text'},
    {tag: 'Href', method: 'attr'}, {tag: 'Class', method:
      'addClass'},
    {tag: 'Html', method: 'html'}, {tag: 'Alt', method: 'attr'},
    {tag: 'Style', method: 'css'}, {tag: 'Value', method: 'attr'},
    {tag: 'Id', method: 'attr'}, {tag: 'Title', method: 'attr'}],
      function(v){
    var directiveName = 'bbOneBind'+v.tag;
    oneBinders.directive(directiveName, function(){
        return {
          restrict: 'EA',
          link: function(scope, element, attrs){
            var rmWatcher = scope.$watch(attrs[directiveName],
              function(newV,oldV){
                if(newV){
                  if(v.method === 'attr'){
                    element[v.method](v.tag.toLowerCase(),newV);
                  } else {
                    element[v.method](newV);
                    }
                  rmWatcher();
                  }
                });
            }
          };
        });
  });
```

This 25-line block of code creates almost the exact same parity as the popular bindonce library that we mentioned previously. The only difference is that here we are not offering any directive that needs to use transclude to perform its tasks. We wanted to keep this example as simple as possible and showcase the really important directives that need to be bound only once to increase performance.

At first, it may be obvious that these directives set a `watcher` function. This `watcher` function is run one time, and then removed from its scope's `$$watchers` array. This allows the data to be asynchronously set and still keeps the original design concept of binding once. The result is a set of directives that run very fast in comparison to normal directives that use regular data binding.

The bbOneBinders tests

The tests for these directives are written in the same fashion as the directives. They call almost all of the describe blocks inside of a loop, which loop over an array of strings. These tests check to make sure that the initial binding worked as planned and that there are no `watchers` on the `$scope` object. Pretty simple, but writing tests in this fashion can go a long way. Refer to the following code:

```
describe('Creating The bbOneBind* directive', function () {

    angular.forEach(['text','src','href','id','class'
      ,'alt','value','title'], function(v){
      var oneBindNode;
      //Should create bbOneBind*
      beforeEach(function(){
          scope.testValue = 'tester+tester+tester' + v;
          oneBindNode = $compile('<div bb-one-bind-'
            + v + '="testValue"></div>')(scope);
          scope.$apply();
          expect(oneBindNode).not.toBe(undefined);
          });

      it('should have the correct text for the oneTime directive',
        function() {
        if(v === 'text'){
          expect(oneBindNode[v]()).toBe('tester+tester+tester' + v);
        } else if(v === 'class'){
          expect(oneBindNode.hasClass('tester+tester+tester'))
```

```
              toBe(true);
          } else {
            expect(oneBindNode.attr(v)).toBe('tester+tester+tester');
          }
        });

    it('should not have any watchers on the scope', function() {
      expect(scope.$$watchers.length).toBe(0);
      });
      });

  });
```

This next set of tests is not run inside of a loop. This is because the individual directives, `bbOneBindHtml` and `bbOneBindStyle`, use special test values to prove that they work. In order to avoid cluttering the previous tests with conditional logic, it was decided to place the individual directives in their own describe block. Refer to the following code:

```
  describe('Creating bbOneBindHtml and bbOneBindStyle directive',
    function () {

      it('should set the correct html to the element and destroy the
        watchers', function() {
          scope.testValue = '<p>No Bindings</p>';
          var oneBindHtmlNode = $compile('<div
            bb-one-bind-html="testValue"></div>')(scope);
          scope.$apply();
          expect(oneBindHtmlNode.html()).toBe('<p>No Bindings</p>');
          expect(scope.$$watchers.length).toBe(0);
        });

    it('should set the correct style to the element and destroy the
        watchers', function() {
        scope.testValue = {width: '100px', height: '200px'};
        var oneBindStyleNode = $compile('<div
          bb-one-bind-style="testValue"></div>')(scope);
        scope.$apply();
        expect(oneBindStyleNode.css('height')).toBe('200px');
        expect(oneBindStyleNode.css('width')).toBe('100px');
        expect(scope.$$watchers.length).toBe(0);
        });

  });
```

These tests prove that the `bbOneBind` directives work as expected.

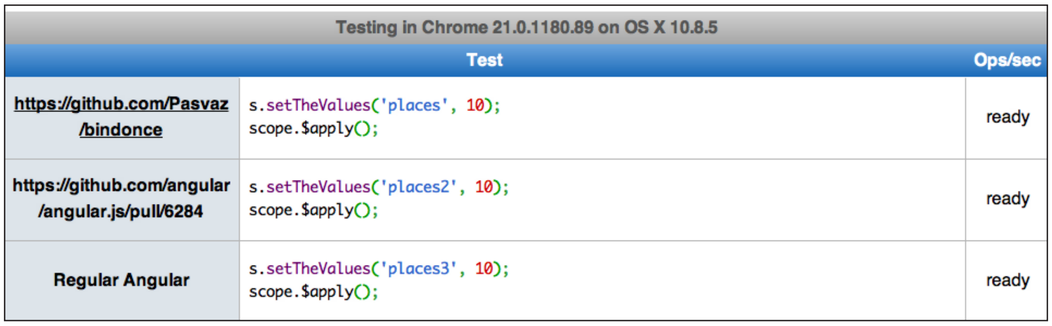
Now, let's look at another type of test. We are going to do an overall performance test to see if these directives actually work better than pure AngularJS, and for fun, let's test against the bindonce library as well. We will test the example that we linked to on JSFIDDLE, except that the tests will not have images in them.

> If all of the directives use the same DOM, then the DOM creation of each will be a part of the control group.

The goal of this next test is to prove which experimental group, directives that set up watchers, performs better under stress. Each of the directives are placed inside of their own `ngRepeat` directive. This `ngRepeat` directive will update whenever its associated array is appended to. We will append each individual array during each test. This will be our benchmark, and we will see which directive can bind more and have more operations per second. Refer to the following code:

```
<div id="myApp" ng-app="OneBinders" ng-controller="oneBindCtrl">
  <div bindonce ng-repeat="place in places">
  <a bo-href="place.src"><span bo-text="place.title"></span></a>
  </div>
  <div ng-repeat="place in places2">
  <a bb-one-bind-href="place.src"><span
    bb-one-bind-text="place.title"></span></a>
  </div>
  <div ng-repeat="place in places3">
  <a ng-href="{{place.src}}">{{place.title}}</a>
  </div>
</div>
```

The JavaScript that we run for each test just pushes three objects to the designated array, which in turn fires `ngRepeat` to create the new DOM elements. The new DOM elements are linked to the proper scope and create their initial bindings each time we run the tests in a loop to increase the amount of operations. Refer to the following screenshot:

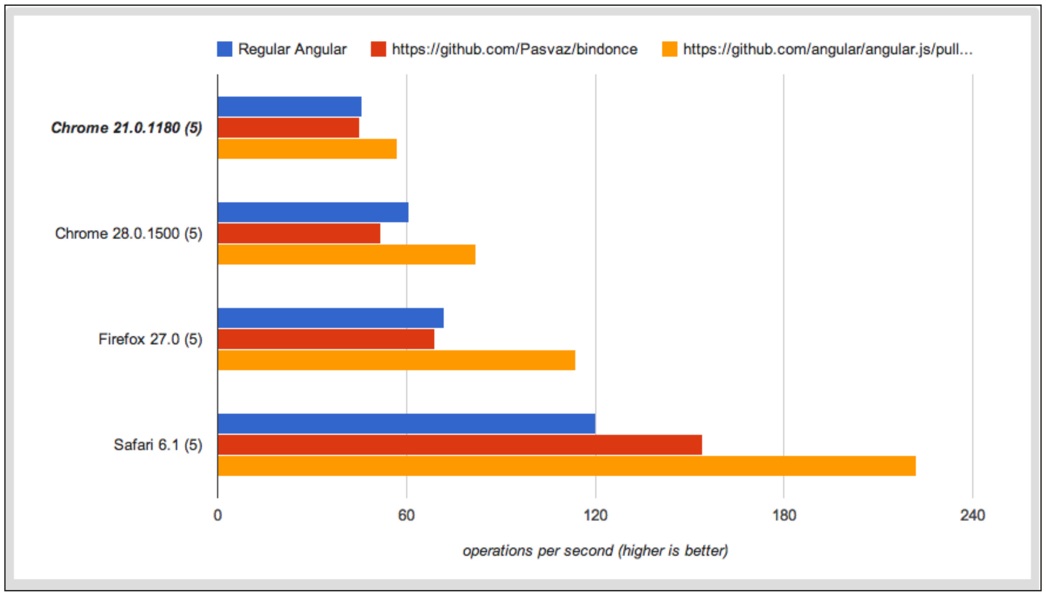


Testing in Chrome 21.0.1180.89 on OS X 10.8.5

	Test	Ops/sec
https://github.com/Pasvaz/bindonce	`s.setTheValues('places', 10);` `scope.$apply();`	ready
https://github.com/angular/angular.js/pull/6284	`s.setTheValues('places2', 10);` `scope.$apply();`	ready
Regular Angular	`s.setTheValues('places3', 10);` `scope.$apply();`	ready

The results were almost as expected except for the fact that the bindonce library performs slower than pure AngularJS.

My assumption as to why this happens is because the bindonce approach is overly complicated. All the bindonce directives require a controller and have many other options defined on their definition objects. These parameters take more time to initialize during the initial linking phase. So, keep your directives simple and to the point. Using simple solutions is usually always the best approach. The live test can be found at `http://jsperf.com/ngbindonce`. Refer to the following screenshot:



There are various ways to accomplish a bind once use case. This was a simple example to showcase the customization abilities that AngularJS offers in terms of optimizing the digest cycle. AngularJS v2.0 and even AngularJS v1.3 will offer solutions that offer this type of functionally as boilerplate or will provide a declarative solution to this problem.

Summary

This chapter was an overview of how to write healthy, fast AngularJS code. We made sure to cover the importance of organizing views correctly. We went over the `$digest` cycle in detail so as to understand why it is important to make sure that it is fast. We showed how directives reduce redundancy and allow us to implement design patterns of our choice inside of AngularJS. We went over a few techniques that directives can perform directly to speed up AngularJS, and we also mentioned why AngularJS separates the view and the model.

Almost all performance concerning AngularJS comes by way of a slow digest cycle. The developer is in charge of making sure that their application is built to their specifications. These specs state that if an application is going to show many elements on a page, make sure that those bindings are being taken care of correctly. This can be done with pagination, using combinations of `ngSwitch` or `ngIf`, creating custom directives that create fewer bindings, or with newer syntax offered by post 1.2.x or a higher core.

There are many more ways to speed up an Angular application than those mentioned in this chapter. Another great optimization technique available is to make sure that all declared filters are very fast, but even this is proof that `watchers` are the source of most of all issues in AngularJS. If you keep the amount of `watchers` down in an application, then your speed will dramatically increase and so will the number of users.

Overall, having a well-constructed application that is fast and well written will go a long way. This chapter covers many different techniques that can be used to write better directives and template views. Using these techniques will make future directives more readable and faster, which will benefit future applications and their writers.

8
Directives and Animations

We know that directives are the only place where DOM manipulation should be performed inside of an AngularJS application. This fact can give us the assurance that a properly structured AngularJS application will have readable and testable logic. Now that the basics of DOM manipulation have been covered, we can create more advanced use cases. What if we wanted to add animations? Would this mess with our perfectly structured, handcrafted directives?

The answer is no. AngularJS has created a wonderful module that allows for simple, easy animation integrations with directives. This is accomplished by separating animation-specific logic from basic DOM manipulation logic. The separation creates less complexity and allows for more focus to be placed on different types of animations.

This chapter is a brief overview of how AngularJS animations work and how to implement them. We will cover public functions that trigger the JavaScript animations, and we will show how to utilize the advantages of CSS-based animations. This is a brief overview of how to implement animations in core and custom directives.

The process of mastering AngularJS animations would require a book of its own. Animation is such a new subject, and it grows and changes every day. The topics in this chapter follow the approaches in the stable `AngularJS 1.2.*` branch. In `AngularJS 1.3.*` and higher, some implementation details could be and will be, changed.

AngularJS is moving at the speed of light, and animations are the fastest moving piece. This is because they are new and their implementation is not fully complete. The idea is solid, but it needs time to fully mature before it can be considered 100 percent.

Currently, there are three sets of animation categories that can be implemented by the `$animation` service. These categories are organized by the type of DOM manipulation they achieve. Each different category modifies CSS classes during its individual animation process. These CSS classes are specific to the type of animation that is being undertaken.

The following are the specific categories of animations available as hooks in AngularJS:

- Adding or removing an HTML element from the DOM:
    - `ng-leave, ng-leave-active`
    - `ng-enter, ng-enter-active`
- Adding or removing CSS classes to an element. The `*` represents a wildcard, which means this could be any classname:
    - `*-add, *-add-active`
    - `*-remove, *-remove-active`
- Moving DOM elements from one position to another:
    - `ng-move, ng-move-active`

Some would say that staggering events could possibly be considered the fourth DOM manipulation category. Staggering events are built from many singular DOM manipulations and are not a part of this basic category set.

The element's classname is always the key to telling the `$animate` service where to look for its specific animation logic. The animation logic can be declared in JavaScript, CSS, or both. The ability to declare animations in different languages reveals great power. There are different situations that call for different types of animations to be used. CSS animations are better for simple to moderately complex animations, and JavaScript animations are better for very advanced complex animations.

The classes that are added to an element are relevant to the `$animate` method being used. These functions are logically named after their purpose. The public methods, `addClass`, `removeClass`, `enter`, `leave`, and `move`, are all related to DOM manipulation. These functions call any specified custom JavaScript functionality and add their associated classes at specified times during the animation life cycle.

Providing animations

Animations in AngularJS are specified by CSS classnames. This allows for clean declarations of what elements should be animated using their classes to associate their corresponding animations. There are several built-in directives that utilize animations out of the box in AngularJS. These directives will serve as a good starting point to learn how to properly write animation code, or the lack of.

Very little work needs to be done to allow for AngularJS to animate core directives. The `ngAnimate` module should be added to the app's dependencies, and the required CSS declarations should be added in accordance with the `ngAnimate` specifications. This is all that needs to be done to allow many of the core directives to use animation. Let's take a look at an example of this.

In the following code snippet, we have a normal `ngRepeat` directive that iterates over a list of players:

```
<div class="well container">
    <button class="btn" ng-click="addPlayer()">Add Player</button>
    <table class="table table-hover">
        <thead>
            <tr>
                <th>#</th>
                <th>Name</th>
                <th>Powers</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="player in players">
                <td>{{$index}}</td>
                <td>{{player.name}}</td>
                <td>
                    <div ng-repeat="power in player.
powers">{{power}}</div>
                </td>
                <td>
                    <button class="btn" ng-click="removePlayer($index)
">Remove</button>
                </td>
            </tr>
        </tbody>
    </table>
</div>
```

The live example of this code can be found at `http://jsfiddle.net/joshkurz/2uJzu/1/`.

Now, we want to add animations to this, so let's write some JavaScript. Wait! Oh yeah, I forgot; we don't have to. Never mind; let's just add `ngAnimate` and some special CSS and watch it fly.

So, first let's add `ngAnimate` to our app's dependencies as follows:

```
<script>
  var app = angular.module('animationDemo', ['ngAnimate'])
  //code omitted
</script>
```

Now, we can add the CSS that we want `ngAnimate` to use while it adds and removes the DOM elements from our list of players, as shown in the following code:

```
<style>
.player-repeat {
    line-height:40px;
    list-style:none;
    box-sizing:border-box;
}

.player-repeat.ng-move,
.player-repeat.ng-enter,
.player-repeat.ng-leave {
  -webkit-transition:all linear 0.2s;
  transition:all linear 0.2s;
}

.player-repeat.ng-leave.ng-leave-active,
.player-repeat.ng-move,
.player-repeat.ng-enter {
    opacity:0;
    max-height:0;
}

.player-repeat.ng-leave,
.player-repeat.ng-move.ng-move-active,
.player-repeat.ng-enter.ng-enter-active {
    opacity:1;
    max-height:40px;
}
</style>
```

Last but not least, we need to add the `player-repeat` class to our `<tr>` element so that when `ngAnimate` adds its associated classes, based on the function that `ngRepeat` is calling, the transition will take effect. A live example can be found at `http://jsfiddle.net/joshkurz/2uJzu`.

We have successfully added animations to our application without writing any extra JavaScript. Whoo Whoo. That's a win in my book. There are other directives in the AngularJS core library that offer this type of plug and play as well. Some use different classes, such as `ng-show` and `ng-hide`. We will go over how these classes actually trigger animations and how `ngAnimate` knows how to animate them correctly.

CSS-based animations

Many core directives rely on the `$animation` service to perform DOM modification. This is true whether or not `ngAnimate` is a dependency in an application. The directives that use `$animate` can be animated with just CSS and do not need to have custom logic implemented. Each core directive has its own DOM-related purpose, which is related to the `$animate` function that it calls. Usually, core directives either add and remove elements or add and remove classes to these elements.

The core directives that use `$animate` use the service so that the developer does not have to write any extra code to achieve basic animations. These basic animations usually include fading, rotating, blinking, or various other types of fluid movement. The `$animation` service enables these animations by adding and removing certain classes to an element that are supposed to be defined in CSS. If they are not defined in CSS, then the animate service will just do the relevant DOM manipulation without any delay in the execution time.

The classes that the `$animate` service adds and removes on the element are based on the method that is called. Each function adds a set of classes that represent the state of the animation based on the delay that is specified in the CSS. Yes, the `$animate` service knows how long the CSS transition or keyframe animations are defined for.

The `$animate` service calculates an element's computed style to determine different declarations that have been made in CSS. This is very important because this is how the `$animate` service knows how long to wait before firing subsequent methods that are waiting to be executed. The overall process is the base of how `ngAnimate` begins its execution.

Working with ngClass and transitions

The following is a basic example of how to add a custom CSS animation to an element; it only uses directives provided by the core to trigger the `$animate` service to do its dirty work:

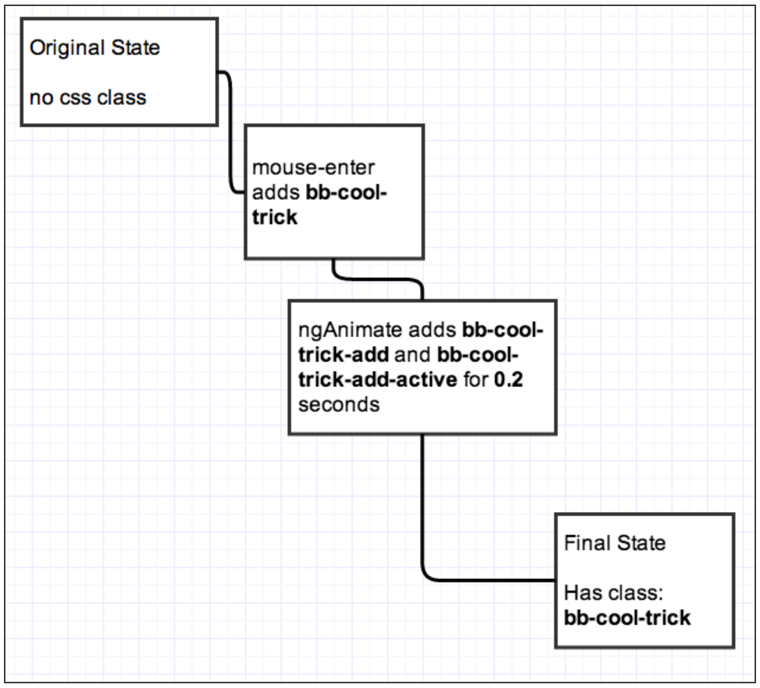
```
<html>
  <div class="bb-table-div">
        <table class="table">
```

```
            <thead>
              <th>Name</th>
            </thead>
            <tbody>
              <tr
ng-class="{'bb-cool-trick': superhero.hasEffect == true}"
ng-mouseenter="superhero.hasEffect = true;"
ng-mouseleave="superhero.hasEffect = false;"
ng-repeat="superhero in superheroes">
                 <td>
                  {{superhero.name}}
                 </td>
              </tr>
            </tbody>
          </table>
      </div>
</html>
```

There is nothing special about this code. The only piece of code relative to the `$animate` service is the `ngClass` directive. This directive uses the `$animate` service to add and remove classnames, which will subsequently call the `animate` service to look for calculated styles on the element that translate into animations. Every time the mouse is entered or taken away, the `ngClass` directive will add or remove the `.bb-cool-trick` class. This will call our defined animations, if we have any, as follows:

```
<style>
.bb-cool-trick-add{
    background-color: skyblue;
}

.bb-cool-trick-add-active{
    font-weight: bold;
}

.bb-cool-trick {
    font-size: 16px;
   -webkit-transition:all linear 0.2s;
    transition:all linear 0.2s;
    padding:10px;
}
</style>
```

The following is a flowchart that describes the animation actions from start to finish:


Original State
no css class
mouse-enter adds **bb-cool-trick**
ngAnimate adds **bb-cool-trick-add** and **bb-cool-trick-add-active** for **0.2** seconds
Final State
Has class: **bb-cool-trick**

The main CSS class is `.bb-cool-trick`. This class could have settings inside of it that change the background color, but we don't actually need the `$animate` service for that. The relevant CSS classes that allow us to create a functionality, which would not be possible without `ngAnimate`, are the `.bb-cool-trick-add` and `.bb-cool-trick-add-active` classes. These classes are only part of the element during the transition, so we can do cool things while the transition is going on. In this case, we make the whole background color sky blue and the font bold. Once the transition is complete, the background will return to its normal state, but the `.bb-cool-trick` class will still be on the element unless the mouse pointer is moved away from the page.

As you can see from the preceding CSS, no `.bb-cool-trick-remove` class is defined. This means that we will only animate the element when the mouse pointer moves into the page, but we want to still make sure to call the `removeClass` method so the style does not stay on the element. We call the `removeClass` method by setting `player.hasEffect` to `false`, which subsequently tells `ngClass` to remove our class from the element. The final output is a table that has rows, which have a very neat, split-second hover effect.

Working with ngClass and animations

We can take the previous example one step further and show the details of each super hero when the mouse enters the row. To do this, we will use the same method as before; however, this time we will use keyframe animations to alter the element's styles.

> Keyframes are commonly used in CSS3 to create animations. The `ngAnimate` module of AngularJS works perfectly with keyframe stages and can detect the total animation delay given to each element based on their class.

The following section will show you how to properly use keyframes with the `$animate` service.

To add each superhero's powers to the tables, we will create an extra `div` element in the table column element. This `div` element will have an `ng-hide` class in it when the `superhero.hasEffect` variable is not true, as shown in the following code:

```
<tr ng-class="{'bb-cool-trick': superhero.hasEffect == true}" ng-
mouseenter="superhero.hasEffect = true;" ng-mouseleave="superhero.
hasEffect = false;" ng-repeat="superhero in superheroes">
              <td>
                {{superhero.name}}
                <div ng-class="{'bb-show-powers': superhero.hasEffect
== true, 'ng-hide': !superhero.hasEffect}">
                    <ul>
                        <li ng-repeat="power in superhero.powers">
                            {{power}}
                        </li>
                    </ul>
                </div>
              </td>
            </tr>
```

Once the `superhero.hasEffect` variable equals `true`, the `.bb-show-powers` class will be added to `div`. During the same execution, the parent row element will receive the `.bb-cool-trick` class. This means that the animation delay we set on `bb-show-powers` will wait for the parent element to complete before it runs. No matter how short the animation is on the child element, if the parent animation is longer, the child element will have to wait for it to be completed to run its finished function.

This timing is based on the CSS code, so be careful to make sure that the delays are set up correctly. For the following example, we have set the same delays for both the parent transition and child animations:

```
.bb-show-powers {
  animation:         super-cool-animation 0.2s;
  font-size: 30px;
  background-color: skyblue;
}
@keyframes super-cool-animation {
  0%   { opacity: 0; font-size: 10px;}
  50%  {opacity: 0.5; font-size: 20px;}
  100% { opacity: 1; font-size: 30px;}
}
```

> It's always best practice to use all of the browser's vendor namespaces in front of each keyframe and animation declaration. We are omitting these for the sake of simplicity.

The final effect of this addition to the example is best experienced firsthand. Refer to `http://angulardirectives.joshkurz.net/dist/#/animations` for more details.

Working with ngIf and transitions

Let's go back to *Chapter 4, Compiling the Advantages*, where we built the recursive `treeNode` directives. A tree-style menu is a perfect use case for an animation. The menu should animate when it is expanded or collapsed. This is a natural animation that people will expect to see when they are navigating within our application.

The best part about the new animation requirement is that we don't even have to edit the directive to achieve the desired animation. Overall, this allows a faster development time when integrating animations into applications, and it also shows the logical separation of how animations are implemented in AngularJS.

The techniques used to animate the `treeNode` directives will be slightly different from the previous example. There are differences because now the `$animate` service is able to actually add and remove DOM elements.

We created two different `treeNode` directives that accomplish the exact same task. Each of these directives will be animated in the same way. They both use the `ngIf` directive to accomplish their conditional logic, which ultimately results in a dynamic menu based on data model. The `ngIf` directive creates and removes DOM elements based on the evaluation of a `$scope` expression.

Once `ngAnimate` is included into the application, all that is needed to achieve the desired animation is the addition of relevant CSS classes. This is true for any core directive that uses the `$animate` service. There are many core directives that fire animation events that add CSS classes to the element during the specified DOM manipulation events. We can take advantage of these CSS classes with any of the three available animation options. For the `treeNode` directive, we are going to use CSS transitions to show how to wire up animations with the `ngIf` directive.

The original templates used for the `treeNode` directives use a CSS class that represents each parent-level `ul` element. This CSS class is the base class that all of the animation logic will be defined from.

For the sake of simplicity, in the following code, we are going to show the `treeNode` directive that uses only transclusion:

```
            <div tree-node-no-template>
                <ul class="list-group">
                  <li class="list-group-item">
                    <span class="btn" ng-show="node.children && !node.
show" ng-click="node.show=!node.show">[+]</span>
                    <span class="btn" ng-show="node.children && node.show"
ng-click="node.show=!node.show">[-]</span>
                    {{node.name}}
                  </li>
                  <li ng-if="$parent.node.show" class="list-group-item
childNode" ng-repeat="node in node.children" ng-transclude></li>
                </ul>
              </div>
```

This HTML describes a directive that uses a recursive technique to render its DOM. The `ngIf` directive is the short circuit that tells the compiler when to add and remove elements. Once an element is added or removed, a class is appended to the element that has the `ngIf` directive specified on it. In this case, the element has a `childNode` class attached to it. The `childNode` class will be the parent CSS class that we will use to define the animations, as follows:

```
.childNode.ng-enter, .childNode.ng-leave {
 -webkit-transition:all 0.2s;
  transition:all  0.2s;
}

.childNode.ng-leave.ng-leave-active,
.childNode.ng-enter {
  opacity:0;
  max-height:0;
}

.childNode.ng-leave,
.childNode.ng-enter.ng-enter-active {
  opacity:1;
  max-height:80px;
}
```

Here, we specify that we want the element to have `opacity` of `0` and `max-height` of `0px` when it is first being entered or when it is actively leaving the element. We also specify that when the element actively enters or when it first leaves the DOM, we want the `opacity` to be `1` and the `max-height` to be `80px`. This will give the menu the appearance of collapsing upwards and downwards as it is being navigated.

This is all we have to do to get animations to work with our directive. Wiring up core directives to use animations is easy, and wiring up custom directives to use animations is just as simple.

JavaScript-based animations

There is a large argument going on in the community. On one side, there are the CSS animators and on the other, the JavaScripters. Both sides believe that their version of animation techniques is better than the other. The truth is that it depends on the specific use case.

Luckily, AngularJS has the built-in ability to allow for both. The implementation details, for JavaScript animations, are based around the same aspects of the CSS-based animations in AngularJS. Everything is based around CSS classes. The CSS classes that are added to an element can be a selector that is the key to an animation closure that holds that class's specific animation logic.

AngularJS animation factory functions are built in the exact same way as services, factories, directives, and other modules in AngularJS. They are used by injecting the `$animate` service into a directive and calling the `service` functions on an element, which will trigger an associated animation. The animation will only be triggered if the focused element, which is passed into the animation service, has a class associated with it that is also the name of an animation factory function. The function can define any animation functions that it needs to accomplish its animation with. These functions will ultimately do a DOM manipulation either before the animation is completed or after. The difference depends upon the method that is called.

Let's say that we have a simple element that we want to cycle through animations with a constant interval once it is appended into the DOM. This can be done very easily, as shown in the following code:

```
app.directive('glowingDiv', ['$animate', function($animate){
   return {
      restrict: 'AC',
      scope: true,
      templateUrl: 'directives/demo/animations/animateMe.tpl.html',
      link: function(scope, element, attrs){

        var parentNode = element.parent();
        scope.addElement = function(){
            var toBeAnimatedNode = angular.element('<div
class="animateMe">Hey Animate Me</div>');
            $animate.enter(toBeAnimatedNode, parentNode, element);
        };
      }
   };
}]);
```

This is a super simple directive. All it does is append a new element to the directive's element when a `scope` function is called. The relevant piece of code shows how the element is appended. The `$animate` service calls the `enter` function, which takes the new element, the parent, and the optional element that need to be appended to. This is the same as calling `element.append` on the directive element.

The `animateMe` class, which is part of `toBeAnimateNode`, is the key that tells AngularJS to animate the element when it enters DOM. An animation function must be created to allow the animation to be executed.

The following is the animation factory function that describes the type of animation that needs to be executed when an element with the `'.animateMe'` class is entered into DOM:

```
app.animation('.animateMe', function(){
  return {
    enter: function(element, done){
        TweenMax.fromTo(element, 0.7, {
            boxShadow: "0px 0px 0px 0px rgba(0,255,0,0.3)"
        }, {
            boxShadow: "0px 0px 20px 10px rgba(0,255,0,0.7)",
            repeat: -1,
            yoyo: true,
            ease: Linear.easeNone,
            onComplete: done
        });
    }
  };
});
```

The `enter` function is defined inside of the animation module. The `ngAnimate` module does not force all functions to be defined inside of a module. There are more available functions that can be called, but we are only worried about the `enter` function. There are an infinite number of animations that can be defined on an element. This creates a wonderful separation of logic inside of our directive.

We use TweenMax to provide the JavaScript animations to the element. The logic is simple and to the point. When the element is entered into DOM, it cycles between the two animations provided by TweenMax. The example can be viewed at `http://angulardirectives.joshkurz.net/dist/#/animations`.

Custom effeckt.CSS animations

There are a couple of wonderful open source CSS animation libraries. One of these libraries can be found at `https://github.com/h5bp/Effeckt.css/`. This library has created many types of animations that are based purely on CSS. To accomplish these animations, the library adds and removes different CSS classes to an element and adjusts data attributes based on the type of animator that we need to execute.

The JavaScript that is used in this open source library is pretty robust. Some of the examples are pretty advanced. One of the most impressive examples is a scrollable list that does many different types of animations to its elements depending on the direction of the scroll and the elements that are visible in the view port.

To wire this example up in AngularJS, we create a directive that attaches itself to a list, which is repeated by `ngRepeat`. This directive's purpose will be to detect a scroll event and calculate the class to be added to the element. The logic that we create to provide the ability to detect the class to be added will be an Angular factory.

This factory determines the direction of the list that is being scrolled and whether or not the element is in the view port. Refer to the following code:

```
app.factory('scrollDirectionFactory', function(){

    return function(){

        var lastScrollTop;

        return {
            checkElement : function(elTop, elHeight, scrollHeight,
parentHeight) {
                var velocity;
                if (scrollHeight > lastScrollTop) {
                    velocity = 'down';
                } else {
                    velocity = 'up';
                }
                lastScrollTop = elTop;

                if (elTop - elHeight < scrollHeight && elTop +
elHeight > scrollHeight - parentHeight) {
                    return true;
                } else if (elTop >= scrollHeight) {
                    if (velocity === 'up') {
                        return 'past';
                    } else {
                        return 'future';
                    }
                } else if (elTop <= scrollHeight) {
                    if (velocity === 'up') {
                        return 'future';
                    } else {
                        return 'past';
                    }
                }
            }
        };

    };

});
```

This `factory` function simply looks at different variables that determine where the element is located in the list as it is being scrolled. There are three different return values that describe the state of the element. The directive uses this `factory` function to know which `$animate` function to call and what class to pass it to.

Let's first take a look at a test that proves that our directive is working as expected. This test dispatches a scroll event on a precompiled `effecktNode` object. Once the event is dispatched, the directive should call `addClass` five times because there will be five `child` directives in the list. Since we are testing this code in karma, `elTop` and `scrollTop` will always be `0`. This means that based on the logic of our scroll direction, the class being added will always evaluate to `past`. Refer to the following code:

```
it('should call the addClass method when the element is scrolled',
function(){
        effecktNode[0].dispatchEvent(scrollEvent);
        expect($animate.addClass.callCount).toBe(5);
        expect($animate.addClass.mostRecentCall.args[1])
          .toBe('past');
});
```

This directive allows each element in its list to be animated in different ways, as follows:

```
app .directive('bbCustomEffecktList', ['$animate',
'scrollDirectionFactory', function ($animate, scrollDirectionFactory)
{

    var scrollService = scrollDirectionFactory();

    return {
        restrict: 'AC',
        link: function (scope, element, attrs) {

            var elHeight = element[0].offsetHeight;

            function animateList(event) {

                var scrollTop = event.currentTarget.scrollTop +
                  elHeight,
                    children = element.find('li'),
                    childTop,
                    childHeight;

                for (var i = 0; i < children.length; i++) {

                    var childEl = angular.element(children[i]);
                    childTop = children[i].offsetTop;

                    if(!childTop || !childHeight){
                        childHeight = children[i].offsetHeight;
                    }

                    if (isIn =scrollService.checkElement(childTop,
                      childHeight, scrollTop, elHeight)) {

                        if (isIn === true) {
                            $animate.addClass(childEl, 'normal');
                        } else if (isIn === 'past') {
                            $animate.addClass(childEl, 'past');
                        } else if (isIn === 'future') {
                            $animate.addClass(childEl, 'future');
                        } else {
                            $animate.addClass(childEl, 'future');
                        }
                    }
                }
            }

            element[0].addEventListener('scroll', animateList);
        }
    };
}])
```

The `bbCustomEffecktList` directive runs a loop of the element's `child` directive every time that element is scrolled. Once the scroll event is fired, we detect the class that should be added to the element. The class that is added to the element will trigger a JavaScript animation that is defined in a relevant animation factory function. Refer to the following code:

```
app.animation('.effeckt-list-item', function() {
  return {

    addClass : function(element, className, done) {

       function realDone(){
         if(className === 'normal'){
           element.removeClass('past');
           element.removeClass('future');
         } else if(className === 'future'){
           element.removeClass('normal');
           element.removeClass('past');
         } else {
           element.removeClass('normal');
           element.removeClass('future');
         }
         done();
```

```
          }

          if(className === 'future'){
            TweenMax.from(element, 1.5, {opacity: 0, rotation:-360,
              transformOrigin:"left 50% 200", onComplete: realDone});
          } else if(className === 'normal'){
            TweenMax.to(element, 0.5, {opacity: 1, left: 0, rotation:
              0, onComplete: realDone});
          } else if(className === 'past') {
            TweenMax.to(element, 1.5, {opacity: 0, rotation:360,
              transformOrigin:"left 50% -200", onComplete: realDone});
          }
        }
      };
    });
```

There are three different types of animations that can be executed according to this animation function. Each animation is determined based on the class that is added to the element. There are a couple of important takeaways from this function. The most important is the `done` function being called after every animation is complete.

The `done` function is the most important piece of code that any AngularJS JavaScript animation can call. This function tells AngularJS that the animation is complete, which fires the DOM close callback function. This is crucial to the overall animation process. Without calling the `done` function, there will inevitably be issues with the `$animation` code, which is related to the DOM elements not acting in an expected way.

The other important takeaway from this module is how the classes are removed in the `realDone` function. The `realDone` function holds custom logic that needs to be executed once the animation is complete. We just remove the classnames, even if the class is not present. The key is that anything can be accomplished inside of a custom `done` function. This allows for specific logic to be accomplished, which ensures that animations act accordingly. The final example can be viewed on the animation example on the black belt demo site.

Summary

AngularJS has integrated a seamless animation library into its core. The ability to separate all animation logic from individual directives is very orthogonal and nonevasive. All animations in AngularJS are based around CSS classes. The two main types of animations that can be achieved are as follows:

- CSS-based animations
- JavaScript-based animations

Each of these types of animations has their own specific use case. CSS animations are best used for simple, data-driven animations that need to offer fast-appealing transitions. JavaScript animations are best for more advanced types of animations. These advanced animations work seamlessly with animation factory functions.

To support animations in an AngularJS application, all that is needed is the `ngAnimate` module and custom animation declarations. These declarations can either be placed in JavaScript or CSS. This allows for a very nice decoupling between DOM and animation logic.

9
Conclusion

Directives offer developers many ways to accomplish functionalities in a simple, efficient, and testable manner. Many of these different techniques are common sense, and some must be discovered by research and practice. The overall point of a directive is to hide DOM-related complexity and create a solid medium between the model and the view.

Directives are the glue that connects DOM and the model together in MV* methodologies. In AngularJS-specific terms, this means connecting the scope together with the template views. Once the two are working together in unison, the application gets the ability to keep the model as its "source of truth".

If an application can accomplish the architectural feat of only using one model for the view and the controller, then that application should have a much better chance to succeed in today's market. A large part of the complexity inside an application comes by way of editing different sets of data, which achieve ultimately the same result. AngularJS gives us the ability to design and architect organized application logic.

There are many ways to represent a model and many ways to edit it from inside of a directive. The directive definition object was created to allow different ways to instruct a directive on how it should display and edit its associated model or models.

A directive's building blocks

The different options for definition objects and how to use them correctly is what this entire book is about. Some of the options are simple and can be covered in a few paragraphs, but some of the options are very detailed and deserve much more coverage. A directive's use case should be broken down to decide what definition objects need to be used for a given directive.

In *Chapter 2*, *Building a Stopwatch Directive*, we created a stop watch directive to show how to use a combination of these directive options. The following is a code snippet of the stopwatch's definition object:

```
return {
    restrict: 'EA',
    scope: {options: '='},
    controller: 'stopwatchCtrl',
    transclude: true,
    compile: function(tElem, tAttrs){
        //code omitted
    }
};
```

This object uses many different options, each of which accomplish specific tasks that are crucial to the stopwatch's functionality. The `restrict` option simply means that the AngularJS compiler will only look for stopwatch elements or attributes on elements as it traverses DOM. The stopwatch can be declared on any element inside of DOM, which means that it is not bound to any certain DOM structure. This is because of the powerful `transclusion` option that allows a directive to use either its entire HTML or just its inner HTML. The `controller` and `scope` options allow the stopwatch to communicate with its surrounding elements and ultimately help render the view with the correct elapsed time value. The final option is `compile`. The `compile` option is intrinsic and has much complexity involved with it. A `link` function could be used here instead, which is what the `compile` function returns. Essentially, the `compile` function is for optimization purposes and simple DOM transformations, and the link function is for DOM manipulation in correlation with an AngularJS scope object.

Third-party libraries

Common third-party libraries such as jQuery, work wonderfully with AngularJS. Many third-party libraries focus on fast DOM-level manipulation. This DOM-level manipulation has been notoriously coupled with business logic that it does not need to be coupled with.

Anytime a DOM-related method is needed, a directive is also needed. The transition from a normal jQuery mindset can be difficult to achieve in the beginning. One of the easiest ways to transition ideology is to make sure that you only use DOM-related methods inside of directives.

There are many wonderful, third-party libraries that accomplish complicated tasks with relatively no extra work. These libraries can be very important in the time line of a project, as there is no point in reinventing the wheel. Any third-party library can be incorporated into an AngularJS application.

The community has written many great directives that prove integration examples with third-party libraries or just explain how to use pure custom directives. It is important to be able to write custom directives that accomplish specific use cases. Every project has their own specialties and custom nuances that require specific attention and custom logic. Once the possibility of wrapping any third-party library into AngularJS has been mastered, more complex directives can be written.

The compile cycle

To understand directives fully, it is imperative to understand the internal processes of the AngularJS compile cycle. The compile cycle is an advanced topic that when broken down into its finite elements becomes simple and digestible. The overall concept of the compile cycle is a depth-first search of an element and its `child` elements that collects directives and runs their associated, defined instructions. These instructions are synchronously called on the DOM element at specified times in the depth-first search. This process is broken down by smaller processes, which allows for an overall simple function that converts the entire DOM into an AngularJS application.

The `compile` function must be run at the beginning of an application's life cycle and can be run anytime thereafter. Directives should always be in charge of compiling new elements into the DOM, unless dealing with callbacks from third-party libraries that are wrapped with a directive themselves. Directives that compile and create dynamic DOM in association with a scope object are powerful tools that create the ability to provide many use cases with simple and effective logic.

Testing directives

Another great way to show non-JavaScript experts how an application's logic should work in certain situations is to allow them to read the application's tests. AngularJS has been built with a **Test Driven Development** (**TDD**) mindset. This book has made sure to showcase this philosophy by testing each directive thoroughly. The tests have been discussed in detail and each decision has been broken down and explained.

Almost all directives should be tested. The thoroughness of the tests depends on the complexity of the logic created inside of the directive. All business logic that is declared inside of a directive should be tested. The creation of DOM elements should be tested as well. There are different methods of testing directives, but each method should ensure that the directive accomplished its use cases with many different inputs and environment variables.

There are multiple types of tests that are needed to prove different directives. Some of the more intrinsic tests are actually called integration tests. These tests bring in multiple modules at once and test their involvements with each other. These types of tests can be very important to ensure the health of an application.

Integration tests usually deem some type of communication.

Directive intercommunication

There are many different ways to accomplish successful communication between different modules and directives. This book discusses the different communication methods in detail, with a plethora of different examples and diagrams to help illustrate the details.

To help display all of the examples created in this book, the website `http://angulardirectives.joshkurz.net/` was created so that the live examples can be viewed in action. This website has all of the directives that we have written as examples to showcase their uses and final output. Many examples of directive communication, templated directives, and third-party libraries integrated with directives have been covered in this book.

Two examples were created to showcase how to display data that is communicated via HTTP. One is a Bootstrap's Typeahead directive (by the AngularUI Bootstrap team `https://github.com/angular-ui/bootstrap`) that queries YouTube videos while incorporating a D3 bar graph of the results. The other example is a D3 line chart that communicates with a socket library that pushes data to the view and updates the line chart. The important details of these directives are how they receive their data and update their views. There are different methodologies of how a directive should watch data and call its main function.

Quality and performance

The performance and organization of a website is something that directives play a huge role in. Directives offer different possibilities of facades and other design patterns inside of HTML markup. Directives can also drastically increase or decrease the performance in an AngularJS site. This is because they play directly with the digest cycle, which determines whether an item has changed or not.

AngularJS uses dirty checking to accomplish data binding. This data binding consists of many watchers that check variables for changes. These watchers can drastically slow a page down as they are very expensive in large numbers. There are ways to make sure the amount of watchers does not exceed an amount that causes "jank" in a page.

In this book, we discussed in detail a custom set of directives that only allow for bindings to occur once. This set of directives was tested with `http://jsperf.com/` to show how higher performance can be achieved using techniques of this nature. We also discussed where AngularJS could be heading in future and how these performance issues will be solved at the core.

Creating views that use minimal amounts of HTML can be a big time saver. AngularJS provides many ways to create quality templates that have focused, to-the-point logic and contain only relative markup. Some even say that AngularJS is the new HTML6, but the reality is that the Web is heading in a direction at a speed that only gets faster. The speed and momentum of the Web will force HTML6 to be more agile and allow for valid customizations to occur to a specific domain's core elements.

Animations

Once performance and organization have been taken care of and the site is running smoothly, it is time to add some sugar to it. The sugar that we have to offer is some sweet animations that work perfectly with core and custom AngularJS directives. Using CSS as the base of all of the animations, we are able to achieve many simple and advanced animations. The animation library that has been created by AngularJS is changing quickly, but the idea of being able to use JavaScript or CSS animations is never going to change.

The examples regarding animations in this book were basic, but showcased some powerful techniques. We created a directive that alternates in a glowing fashion, as well as a list that mimics `Effeckt.css` animations with no third-party libraries. Animations with AngularJS are intuitive and fun once the initial learning curve is overtaken.

Summary

Directives offer so many advantages to an application. Some of the biggest advantages include data binding, templating options, and cross-scope communication. A directive offers the ability to organize all DOM-related logic in a manner that ensures that all the DOM logic has been created and can safely be manipulated or have event handlers attached.

AngularJS development processes are new and different from normal JavaScript frameworks. The advantages of using AngularJS to create rich web applications are immense and have opened up new doors. Many developers are using these doors across the world to create amazing applications with new, interesting directives at their core. Directives are the main reason AngularJS is so popular, and they will continue to be that reason.

Index