

IN TRACTION

AngularJS Directives



Amit Gharat

Table of Contents

1. Introduction
2. [AngularJS Directives: A forerunner of Web Components](#)
3. [AngularJS Fundamentals: A Crash Course](#)
4. [Testing the Beast: Unit testing and E2E testing](#)
5. [Learning to extend HTML with the Directive API](#)
6. [Understanding Scope in Directives for better context](#)
7. [Crafting directives to handle Complex Scenarios](#)
8. [Bringing directives to Life](#)

AngularJS Directives in Traction

AngularJS is a super-heroic JavaScript Framework that makes writing single page applications a breeze. It allows you to expand HTML vocabulary by creating custom tags with a help of directive definition object. AngularJS bundled with a set of directives like ngRepeat, ngModel, ngView which make it really easy to build an application but many a times the existing directives fell short considering the need and complexity you are dealing with. And you face the problem head-on when it comes to build directives from ground ups. This book teaches you to extend AngularJS by writing custom directives and motivates to understand the purpose of testing by facilitating Test Driven Development through out the book.

AngularJS Directives in Traction helps you understand how built-in directives work and teaches you to build custom directives on your own. Ultimately, Angular Directives will be the magic revealed for you.

Why published it for free?

Kindly go through the post, <https://amitgharat.wordpress.com/2015/07/28/angularjs-directives-in-traction/>

AngularJS Directives: A forerunner of Web Components

This chapter covers

- The core features and benefits of AngularJS
- What problems AngularJS Directives solve and why they are worth learning
- How AngularJS Directives mimic Web Components – the Future Standard

If you are reading this book then you might be aware of AngularJS and quite curious to know about what it can offer. In case you are a JavaScript developer or jQuery addict and would like to port some of your jQuery plugins to AngularJS, then this book is for you. Pat yourself on the back for making the right choice and choosing to invest your precious time into the precious to make yourself productive to write your next ambitious application or component.

In this chapter, I'll give you an overview of features and benefits of AngularJS as a whole, as well as describe how AngularJS Directives are helping to shape the future of web applications. We'll start off with understanding the magic behind AngularJS popularity and then get to know directives straightaway.

3Ds of AngularJS – data binding, dependency injection, and directives

AngularJS is built around the belief that declarative programming should be used for building user interfaces and wiring software components. By declarative means that it adapts and extends traditional HTML to better serve dynamic content through two way data binding that allows for the automatic synchronization of data and User Interfaces. With this, AngularJS reduces imperative DOM manipulation (like we do with jQuery) and improves testability by facilitating separation of concerns.

HTML is great for declaring static documents, but it falters when we try to use it for declaring dynamic views in web applications and we often end up with spaghetti code – a mixture of HTML and JavaScript, which is inexpressive, unreadable, and hard to extend further. Other frameworks work around the HTML's shortcomings by abstracting away HTML, CSS, and JavaScript in terms of UI Widgets or by providing an imperative way for manipulating the DOM. But none of these solutions address the problem developers face while developing dynamic applications. Instead, AngularJS lets you write applications in a declarative manner that not only speeds up the development process but also makes it easy to test, extend, and maintain as applications grow large. AngularJS came into existence to make the web better and browsers smarter, but it did so before it would be supported natively in terms of Web Components. In fact, it truly allows us to write composable and reusable components declaratively to facilitate code abstraction.

Going forward we'll take a look at what makes AngularJS so special.

Making HTML dynamic with Data Binding

Who does not like to give commands and have followers to follow them? Data Binding is based on the same notion wherein data models are like commands and are being constantly followed by Views to be updated automatically. Data Binding is a very old concept (new to JS) notably used in Adobe Flex. It is one of the core features of AngularJS that makes the overall framework magical and helps us reduce lines of code (LOC) by doing the heavy lifting itself so we do not have to do DOM manipulation manually and repeatedly.

To understand it better, let us look at a simple but real-world example. Let's say we have a notification panel in our application to show messages that needs to be updated every time a new message arrives. In JavaScript, we would probably do as follows.

```

<html>
<head>
  <title>Updating Notification Panel in Pure JS</title>
</head>
<body>
  <div id='notification'>No Message found.</div>

  <script type="text/javascript">
    document.getElementById('notification').innerHTML = 'Learning Angular Data Binding';
  </script>
</body>
</html>

```

There could be many places in your application from where you could update the notifier and have to use a long-lasting and painful DOM APIs depending on the complexity of the DOM you are dealing with. You may have to write lot of repetitive code for every HTML element that has to be updated dynamically from several places which is cumbersome and unmanageable as your application grows.

What if there were a simpler and better way to handle this? Well, in AngularJS, any type of data (primitives and objects) is a model and once bound to the DOM will update it automatically whenever the associated model changes. So the same thing can be accomplished using data binding in AngularJS as shown in the following listing.

```

<html ng-app="App">
<head>
  <title>Updating Notification Panel in AngularJS</title>
  <script src="http://code.angularjs.org/snapshot/angular.js"></script>
</head>
<body>
  <div ng-bind="notification || 'No Message fund.'"></div>

  <script type="text/javascript">
    angular.module('App', []).run(function($rootScope) {
      $rootScope.notification = 'Learned Angular Data Binding, Today';
    });
  </script>
</body>
</html>

```

With this approach, you are assured that the notification panel will automatically update whenever the notification model changes. Just imagine how easy is it to enable two-way data binding with ngBind directive.

You might be curious to ask, Will it update the model as well if I dynamically change the content of the `<div>` ? Nope. That is because it's a one-way data binding we've set up in this case. The two-way data binding can be achieved with Form Controls such as Input, Textarea, Select, Checkbox, Radio, and so on.

```

<html ng-app="App">
<head>
  <title>Two way Data Binding with Form Controls</title>
  <script src="http://code.angularjs.org/snapshot/angular.js"></script>
</head>
<body>
  <input type='checkbox' ng-model='choose' />
  <div ng-bind="choose"></div>

  <button ng-click="choose=!choose">Toggle Checkbox</button>

  <script type="text/javascript">
    angular.module('App', []);
  </script>
</body>
</html>

```

We can update `<div>` 's content by either ticking the checkbox or directly updating the model in JavaScript as seen before.

So directives let us make HTML dynamic by applying directives so that HTML can interact the way we want it to be.

I'm pretty sure you must be grinning by now considering the possibilities of what you can do with this new learning.

Managing dependencies with Dependency Injection

Dependency Injection (DI) is a software design pattern that deals with managing dependencies. This allows the removal of hard-coded dependencies and makes it possible to change them, whether at run-time or compile-time. A dependency can be resolved either by creating (using the new operator) or passing a reference of the already created object where it's needed. The first approach is not optimal because we then need to hard-code the dependency, thereby making it difficult to modify later or even test. The second option is viable, as we do not have to instantiate the dependency ourselves; we can just hand it over. This makes testing easy as we can mock the dependency at will without knowing how it was created. To get more clarity on the subject, let us look at a trivial example given as follows.

In JavaScript, we often define a Class and create an instance of it to access its properties or methods. In this example, `sayHi()` class is dependent on `hello()` class. But, the way we've resolved the dependency by creating an instance of `hello()` class manually is not ideal. If, in case, the constructor definition of the `hello()` class changes in future, it will affect our class, `sayHi` as well.

```
function hello(who) { this.name = 'Hello ' + who; }
function sayHi(obj) { return obj.name; }

var objHello = new hello('AngularJS');
console.log(sayHi(objHello));
```

What if we could instantiate all the dependencies in one place and just pass the references wherever needed? This will loosely couple all the dependencies, and any modifications later will not affect components that were dependent on it. That's how AngularJS manages dependencies using the Dependency Injection (DI) system built-in:

```
var App = angular.module('DI', []);
App.service('Hello', function() {
  var self = this;
  self.greet = function(who) {
    self.name = 'Hello ' + who;
  }
});

App.controller('SayHi', function(Hello) {
  Hello.greet('AngularJS');
  console.log(Hello.name);
});
```

In this example, we have a class named `Hello` defined via service method in AngularJS. Then we have injected the same as a dependency in the controller, `SayHi`, which calls greet method with a parameter – whom you wish to greet. Just pay attention to how the dependency has been resolved without creating an instance of `Hello` implicitly. Off-course D.I. internally instantiates the class and stores its instance into the same object passed before.

This approach makes it really easy to mock all dependencies during testing so we can test `SayHi` in a complete isolation by mocking the service `Hello` used with.

FYR, Testing code would look something like:

```
describe('Resolve Dependencies', function() {
  var SayHi, Hello;

  beforeEach(module('DI'));
  beforeEach(inject(function($controller) {
```

```

    Hello = {
      greet: function(who) {
        Hello.name = 'Hello ' + who;
      }
    };

    SayHi = $controller('SayHi', {Hello: Hello});
  }));
});

```

That's it! AngularJS will automatically resolve dependencies while testing and use the mocked service instead of the original one.

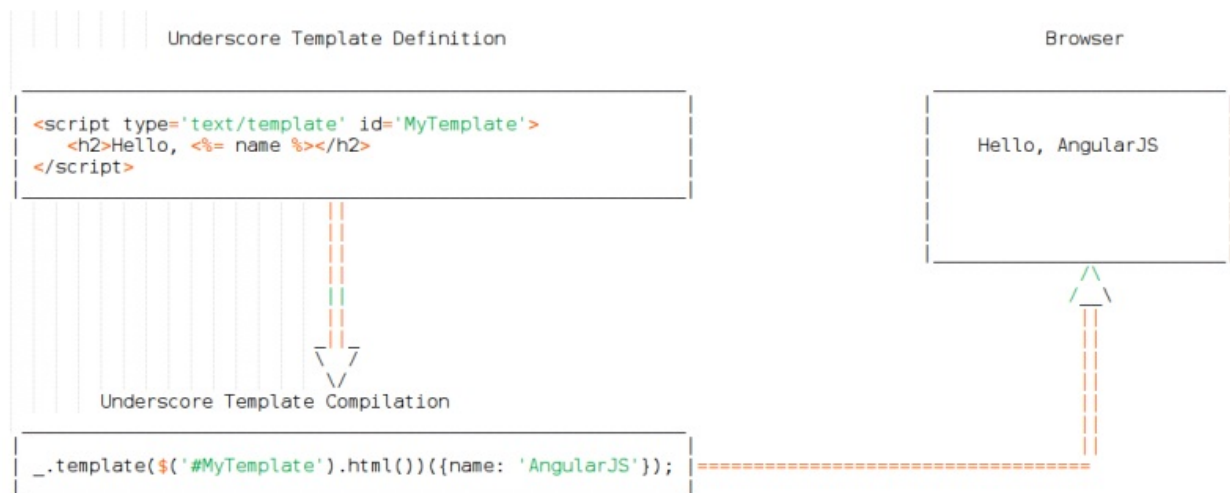
Scaffolding reusable markups as HTML Template

The purpose of JavaScript Templating is to generate HTML that makes sense based on structured data. Such templating libraries take JSON data and compile it into a function which can then be used to generate the final HTML by passing in values for placeholders. There are two types of templating libraries:

- String-based
- DOM-based

In string-based templating, we wrap an HTML into a pair of `<script>` tags, compile, and interpolate it with JSON data to get the final HTML string – which can be injected in the page later. Underscore.js, Handlebars.js, Mustache.js are some of the well known examples of this category. Here is how templating works in Underscore.js. It introduces a special method `_.template()` to feed in the template that generates a compiled function. The compiled function then takes JSON or JS objects to interpolate the template.

As per the following figure, the moment we update the model, each time the whole template will be recompiled and re-rendered in the browser.



The `<%= name %>` expression is a syntax in Underscore.js to create a placeholder for name. The trade-off of this approach is that as the template becomes larger, the amount of stuff that needs to be re-rendered grows, which is slow and wasteful.

In contrast, AngularJS uses HTML(DOM) as a template where only specific parts of the template updates when the model changes. The benefit of this approach of updating only the fragments of the HTML document is that it does not re-render the whole template every time but just updates those parts of the template that were associated with the models. What this means is that it introduces various directives such as `ngBind`, `ngBindHtml`, and so on to achieve this. It supports `{{ }}` as a templating syntax as an alternative to `<%= name %>` in underscore but you could use `ngBind` built-in directive as well to achieve the same as shown in the following listing. Create *angular-template.html* in *ch01/* directory as:

```
<html ng-app="App">
<head>
<title>HTML/DOM Template in AngularJS</title>
<script src="https://code.angularjs.org/snapshot/angular.min.js"></script>
<script type="text/javascript" src="../js/ch01/angular-template.js"></script>
</head>
<body>
<span>Hello, <span ng-bind='name'></span></span>
</body>
</html>
```

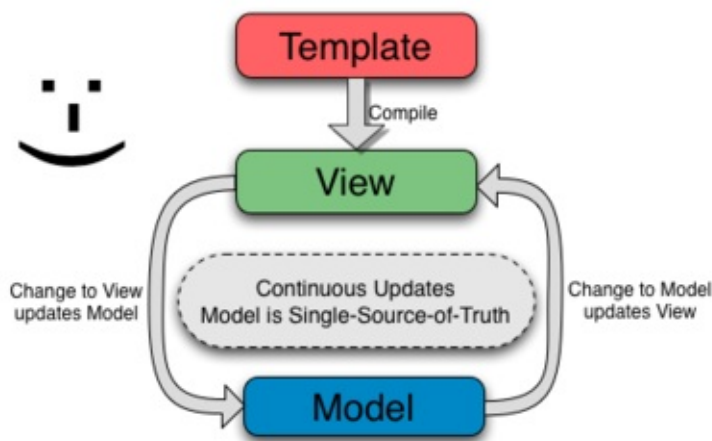
Then add following snippet in `js/ch01/angular-template.js` so:

```
var App = angular.module('App', []);

App.run(function($rootScope) {
  $rootScope.name = 'AngularJS';
});
```

When AngularJS bootstraps, it compiles the HTML and feeds in the actual data replacing the placeholder, `ng-bind='name'`. All this is done by AngularJS in the background and hence it looks very natural and easy to write or manage.

More specifically, in this example, only `span` tag will be updated if `name` updates to something else later, unlike string-based templates, where the entire template string has to be interpreted in order to replace just the `name` expression. Let's see how AngularJS does it:



As you can notice that all the pieces such as a template, data model, and view remain the same as in string based templates but the very difference lies in the compilation of the template which only happens once. Furthermore, data binding takes care of updating relevant parts of the view (instead of entire view) upon model mutation.

Deep Linking AngularJS applications to maintain browser history

Back in the old days, every hyperlink used to refresh the page in order to load it. AJAX (Asynchronous JavaScript and XML) opened the door to load content asynchronously without refreshing a page at all. Developers found a profound way to write applications without a single refresh and hence coined a term, Single Page Applications. With AJAX, web applications can send data to, and retrieve data from, a server asynchronously. Google made a wide deployment of standards-compliant, cross browser AJAX with Gmail and Google Maps. But while doing so, many of the applications were breaking the browser's back button – there was no longer a way to go back to the previous hyperlink visited.

AngularJS does not break the back button, thanks to Routing. It comes with a routing module named `ngRoute`, which enables you to programmatically generate URLs based on route definition. As given in the following example, we've injected `ngRoute` as a dependency (thanks to DI) that provides access to the `$routeProvider` service to define our routes.

Let us look at the following route example as:

```
angular.module('ngRouteExample', ['ngRoute'])
.config($routeProvider) {
  $routeProvider.when('/Book/:bookId', {
    templateUrl: 'book.html',
    controller: BookCtrl
  })
};
<div ng-view>Template loads here</div>
```

In this example, I've set up a simple route to load book details by its Id. When user opens <http://myapp.com/Book/2> in the browser, AngularJS's internals notify the `$routeProvider` service to match the URL format with each of the route definitions and attempts to match the URL format (`/Book/:bookId`) with the URL value (`/Book/2`) in the defined route configuration list (where `:bookId` acts like a parameter). As expected, it will match with the preceding route definition and load `book.html` partial thereafter into the `div` element.

To make it work, AngularJS provides a special directive named `ngView` which renders a template of the current route into the main layout. Whenever current route changes, the view updates as well according to the route definition. It does not look like a big deal to set up routes, but it is an extremely important and powerful module in AngularJS arsenal, especially in the era of Single Page Applications.

Testing is !important

AngularJS is written with testability in mind, and there is no reason why one should not write test cases for any snippet. In fact, there is nothing you can write in AngularJS that cannot be testable. If one cannot, then he is doing it wrong. I would recommend every AngularJS developer to write testable code. The same approach we're going to follow throughout the book.

You as a developer might have gone through a situation (many times) wherein an application stops working or behaves abnormally after fixing a bug or implementing a feature if you've not set up an automated testing environment. And it's very difficult and time consuming to go through all the test cases manually to check if anything is broken every time you do any updates. To solve such problems, the AngularJS team has built many features, libraries, and tools to make testing a **First Class** citizen. There are two ways to do testing in AngularJS: unit testing and end to end testing.

UNIT TESTING

Unit Testing, as the name implies, is all about testing individual units of code i.e. functions. To make unit testing fundamentally more accurate, we have to follow Unix Philosophy – write programs/functions that do one thing and do it well. There are many unit testing frameworks, such as Jasmine, Mocha, Chai, Qunit, and so on, but Jasmine has been used at its core so we're going to use the same for all our examples in this book.

If we were to write a unit test for the example we saw before *ch01/angular-template.html*. We would probably do as follows by creating *tests/specs/ch01/angular-template-unit.js* so:

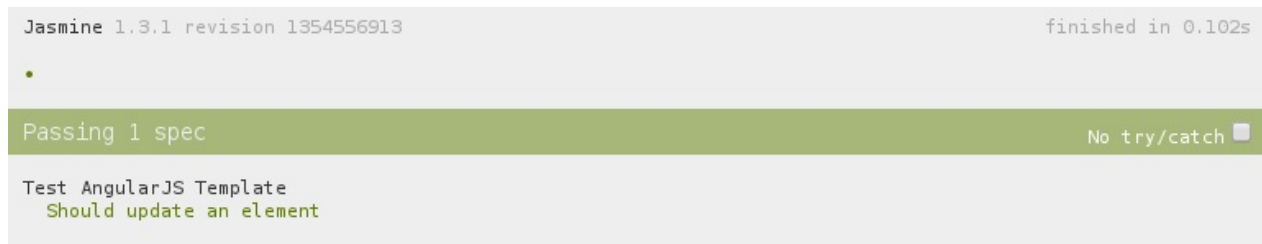
```
describe('Test AngularJS Template', function() {
  beforeEach(module('App'));
  var element, scope;

  it('Should update an element', inject(function($rootScope, $compile){
    $rootScope.name = 'AngularJS';
    element = angular.element(
      '<span>Hello, <span ng-bind="name"></span></span>'
    );
    element = $compile(element)($rootScope);
    $rootScope.$digest();

    expect(element.text()).toBe('Hello, AngularJS');
```

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

In this simple test snippet, we've set up a Jasmine framework to run the test. In Jasmine, `describe()` block allows you to group related tests cases and `it()` block lets you write the actual test to perform. In this case, we've injected two dependencies, namely, `$rootScope` and `$compile`. The `$rootScope` is single root `$scope` that every AngularJS application has. Later we create a DOM in memory using `angular.element()` – an alias for the jQuery (`$`) object and compiled it by passing in the `$rootScope` to the `$compile` service – that replaces placeholders with actual values on the scope. Finally, we make sure that the element contains the valid text with `expect()` – an expectation in Jasmine is an assertion which implements comparison between the actual value and the expected value.



```
Jasmine 1.3.1 revision 1354556913 finished in 0.102s
•
Passing 1 spec No try/catch
Test AngularJS Template
  Should update an element
```

At the end, Jasmine will show you above display if the test succeeds.

END TO END TESTING

No matter how good programmer you are, you cannot presume that every piece of code you write will work as expected on various browsers; hence, functional testing is extremely important, and there is a need for End to End test frameworks.

Protractor is an end to end test framework specifically for AngularJS applications built on top of (a wrapper for) WebDriverJS – Selenium Web Driver which is a standard web applications testing tool used everywhere. The installation of Jasmine with Karma Test Runner and Protractor with Selenium Driver is covered in detail in Chapter 3. Luckily Protractor supports Jasmine like syntax to write test cases so that you can grasp it in no time. Let us quickly check out how to write test cases in Protractor, so create `tests/specs/ch01/angular-template-e2e.js` as:

```
describe('Test AngularJS Template', function() {
  var str;

  it('should greet AngularJS', function() {
    browser.get('ch01/angular-template.html');
    str = element(by.binding('name')).getText();
    expect(str).toEqual('AngularJS');
  });
});
```

In the `describe()` block, we are creating an instance of protractor and loading a page to perform tests on. As soon as we run protractor on a command line, `browser.get()` will fetch the specified html page in real browsers and perform a lookup for the binding, `{{name}}` using `by.binding()` syntax. Finally we're just matching the actual value with the expected one. If everything goes well then this is what you'll see in the command line.

```

-----
PID: 13844 (capability: chrome #1)
-----

Starting selenium standalone server...
Selenium standalone server started at http://192.168.42.56:37341/wd/hub
.

Finished in 2.74 seconds
1 test, 1 assertion, 0 failures

Shutting down selenium standalone server.

```

So AngularJS framework is extremely conscious about testing every bit of AngularJS applications, even directives are no exception!

Meet AngularJS Directives

By now we have covered AngularJS in a nutshell and you probably excited to get to know AngularJS directives. So, here we go. HTML is all about elements, but as the web grew bigger, developers started building complex applications so the existing elements fell short to fill the gap and we chose to rely on markups with lots of JavaScript. The basic idea is that the current set of elements we have in HTML always perform the same role and have the same set of properties/options applied to it but the web today has to do lot more work and HTML can't keep up with the rate of change.

Browsers have engrossed the specification required to analyze the behavior of HTML elements so they understand the language and figure out the layout when a web page loads. But they are not able to interpret the meaning of custom HTML element which is not part of the specification they know. In such condition, the browsers simply consider it as a text content ignoring the element altogether.

AngularJS Directives API teaches the browsers a new trick and helps them to treat custom elements as native ones. Also, it gives developers flexibility and low-level access to create custom elements based on their function. Hence directives are the integral part of the AngularJS framework and the only thing that makes AngularJS stand out from the crowded list of JavaScript frameworks these days. Let us see what are directives made up of.

What are directives

AngularJS Directive is nothing but the way to expand HTML vocabulary. Having said that you can extend existing HTML element or create a new one by encapsulating custom behaviors atop. That means, we can write custom elements, attributes, and so on in AngularJS that browsers can understand using AngularJS Directives API. Here is a non-intuitive example of Accordion widget in Twitter Bootstrap as:

```

<html>
<head>
  <title>Accordion in Twitter Bootstrap</title>
  <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
  <script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
  <script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
</head>
<body>
  <div class="panel-group">
    <div class="panel panel-default">
      <div class="panel-heading">
        <h4 class="panel-title">
          <a data-toggle="collapse" href="#accordion1">
            Accordion Header #1
          </a>
        </h4>
      </div>
      <div id="accordion1" class="panel-collapse collapse in">
        <div class="panel-body">Accordion Body #1</div>
      </div>
    </div>
  </div>

```

```

    </div>
  </div>
  <div class="panel panel-default">
    <div class="panel-heading">
      <h4 class="panel-title">
        <a data-toggle="collapse" href="#accordion2">
          Accordion Header #2
        </a>
      </h4>
    </div>
    <div id="accordion2" class="panel-collapse collapse">
      <div class="panel-body">Accordion Body #2</div>
    </div>
  </div>
</div>
</body>
</html>

```

However, it does fulfill the requirement of having a fancy looking accordion widget in the browser but at the expense of imperative markups that you dare to fiddle. The end-users who will end up using this UI widget no longer need to go through the pain of understanding nitty-gritty things in their way just to use the component. What they really want is the minimal, comprehensible, and declarative piece of code that makes sense to them. Up until, it was not possible to have that level of encapsulation in order to hide the complexity behind the curtain and only show things that matter to the end-users. Here is the same example converted into a declarative style using AngularJS directives API for you to get a sense of what AngularJS directives can do:

```

<html ng-app="App">
<head>
  <title>Accordion in AngularUI Bootstrap</title>
  <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
  <script src="https://code.angularjs.org/snapshot/angular.js"></script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/angular-ui-bootstrap/0.10.0/ui-bootstrap-tpls.js"></script>
  <script type="text/javascript">
    angular.module('App', ['ui.bootstrap']);
  </script>
</head>
<body>
  <accordion>
    <accordion-group heading="Accordion Header #1">
      Accordion Body #1
    </accordion-group>
    <accordion-group heading="Accordion Header #2">
      Accordion Body #2
    </accordion-group>
  </accordion>
</body>
</html>

```

Now the thing is that you will find this piece of markups much more comprehensible than the one that we saw earlier. Note that, even it looks human readable by now, it still generates the same blob of markup behind the scene but hides it upfront to make it declarative and fun to interact with.

Even though this book is all about directives and we are going to explore all the configurations required to define directives in subsequent chapters, following is the blueprint for your reference.

```

angular.module('MyModule', []).directive('directiveName', function () {
  return {
    restrict: 'A',
    template: '<div></div>',
    templateUrl: 'directive.html',
    replace: false,
    priority: 0,
    terminal: false,
    scope: false,
    compile: function (tElement, tAttrs, transclude) {
      return {

```

```

    pre: function preLink(scope, iElement, iAttrs, controller) { },
    post: function postLink(scope, iElement, iAttrs, controller) { }
  },
  link: {
    pre: function preLink(scope, iElement, iAttrs, controller) { },
    post: function postLink(scope, iElement, iAttrs, controller) { }
  }
  transclude: false,
  controller: function($scope, $element, $attrs) { },
  require: 'otherDirectiveName'
};
});

```

These are the parameters the directives API supports to compose a directive, so here is a quick tour:

Restrict: Restricts the directive to a specific DOM declaration style. Can be an element or attribute by default.

Template: Stuffs the element binded to the directive with an extra content to be injected dynamically.

TemplateUrl: Same as templateUrl but the template will be loaded asynchronously via AJAX or XHR.

Replace: Replaces the element with the template added by template or templateUrl option.

Priority: Configures the order in which directives are applied or compiled.

Terminal: Freezes the further compilation of directives on the element, if set to true.

Scope: Different types of scope strategies depending on the requirement.

Compile: Deals with transforming the template or element.

Link: Responsible for registering DOM events and bindings variables on the scope to be used within the template.

Transclude: Brings the content of the element into the template. Works even if replace set to true.

Controller: Allows to share scoped methods or properties with other directives (siblings or child directives).

Require: Injects the controller of a sibling or parent directive in order to use shared scoped method or properties.

You will notice that the blueprint is quite huge and complex but trust me its worth learning. So the takeaway is that the directive is:

- The place for Imperative DOM Manipulation
- The home for binding DOM events and unbinding them later
- The technique to facilitate code reuse with semantic and composable widgets
- The ability to write or use code declaratively
- And ultimately the way to extend HTML vocabulary

Now let us see how directives actually work behind the scene in the following section.

How directives work

In short, AngularJS when bootstrapped, looks for all the directives built-in as well as custom ones, compiles them using built-in compiler, and links with the associated scope by registering listeners or setting up `$watcher`s resulted in 2 way data bindings between the scope and the element.

If this is overwhelming for you, here is the smallest directive that you can possibly ever write. However, it does not do anything but just there to explain how directives get registered in AngularJS. We have defined a custom directive named

noop that can be used as an attribute/element as shown:

```
<html ng-app="App">
<head>
  <title>Basic Directive: noop</title>
  <script src="../../bower_components/angular/angular.js"></script>
  <script type="text/javascript">
    var App = angular.module('App', []);

    App.directive('noop', function() { return {}; }); #B
  </script>
</head>
<body>
  <div noop></div>
</body>
</html>
```

Few things to note here is that:

- The `ngApp` directive is an important built-in directive you'll always need per application to bootstrap the same.
- We then register our directive named `noop` that means no operation as it does not alter or modify the associated element or node. However, this will still be registered by AngularJS because of the default properties set internally such as `restrict` (as an attribute/element by default) in the DOM and `priority` (0 by default) to set the order of execution. This further added into an injector – built-in service that can be used to retrieve object instances to check if it is a directive or an existing HTML node during DOM compilation that we'll see next.
- Finally we use it as a custom attribute/element in the DOM and you'll see just a `div` element in the DevTool. However, you must be nagged by a question that how does AngularJS do it?

That's quite simple than you think. All it does is the compilation of the HTML tree to find directives to activate them. On a high level, it is quite similar to how browser parses HTML and displays the output. Take an example of an anchor element (`<a>`) that turns a normal text into an underlined one with blue font color by default. As discussed before that the browser has a specification of the anchor element, it just performs the required operations to attach those behaviors. Similarly there is a flow of how AngularJS brings directives to life:

- First it finds where to start the compilation process and that's why AngularJS needed a unique marker to begin with. The `ngApp` directive serves that purpose. The element or node (including all of its child nodes) on which it is applied will be owned by AngularJS for further operations such as directives registration, data bindings, and so on. Thus AngularJS does not bootstrap without it. Many beginning AngularJS developers often forget to add the `ngApp` directive until they realize why their directive did not work.
- Once it knows the root node, it traverses the DOM to find sub nodes (elements, attributes, classes, and comments), and maps each with the injector service to check its definition exists or not. This process helps AngularJS to collect all the directives per node and to return their compile functions.
- The Compile function per directive is executed in order to apply behaviors on the respective node. Prior to this step, directives do not exist at all for the browser. This step creates a new or shared scope per directive (depending on its definition) needed to create watchers and set up data bindings in terms of Link functions.
- Finally the link functions are executed in a reverse order than that of the compile functions to register DOM events or update bindings used on elements or within their templates. There is a reason for the reverse ordering which we'll cover in Chapter 4. By this step, the directives are fully activated, probably manipulated the DOM, updated the bindings, attached the DOM events, and so on.

By now, we know that AngularJS activates directives but what does it mean? Let us slightly update the `noop` directive to see it in action as:

```
App.directive('noop', function() {
  var count = 1;

  return {
    compile: function(cElement) {
```

```

    cElement.html('I am noop {{counter}}');

    return function link(scope, element) {
        scope.counter = count;
        count++;
    }
};
});

```

We have added the compile function which is basically used to manipulate the associated element and update its content. Note that the double curly notations are very common data binding syntax used in AngularJS, you can also use built-in directives replacing the same such as `ngBind`, `ngBindTemplate`, and `ngBindHtml`.

As learned before, the compile function generates a link function that gives access to scope object and you are good to go with binding few scope variables to enable two way data binding in this phase. We have set the `counter` variable that updates the binding `{{counter}}` placed earlier in the DOM. You should see `"I am noop 1"` in the browser now.

You probably think that why there is a need for compile and link functions? Why not combine them? The only reason for the existence of the compile function is performance. To find out, let us update the `noop` directive in HTML as:

```
<div noop ng-repeat="item in [1, 2, 3, 4, 5]"></div>
```

Here we basically repeat the same div element five times and each time the `noop` directive will be linked to a new instance of the DOM. And you will see `"I am noop 1"`, `"I am noop 2"`, and so on as the counter increments. Imagine there was no compile function, the text will be injected five times which is unnecessary because we are just repeating the same old element. Also, the linking would have happened at the same time enabling watchers and degrading the performance because the watchers are reevaluated on every digest cycle to make sure views are upto date. Now by separating the two, compiling all directives first and linking them later altogether is not only performant but also cost effective because the compile function of the `noop` directive with `ngRepeat` will be triggered once whereas the linking function will be called for each iteration.

Now that we know how directives are executed so that browsers could understand their meaning without relying on HTML specification, let us look at why to use directives. Is it the ultimate answer to life, the universe, and everything?

Why use directives

By now we pretty much realized the problems that AngularJS directives are trying to solve but is it the ultimate remedy? The answer is Yes! Because the HTML language is expressive enough to create complex UI widgets and also extensible so that we, the developers, could fill in the gaps with custom tags. Probably in the near future, it will be possible through a new set of standards called Web Components.

Web Components are a collection of standards which are working their way through the W3C and landing in browsers soon. In fact they allow us to bundle markup and styles into custom HTML elements. It will allow web developers to use their HTML, CSS, and JavaScript skills to build custom elements or widgets that can be reused easily and reliably irrespective of what frameworks you use.

Even Google engineers have tried to replicate web components as Polyfills (plugins that provide technologies that developers expect the browser to provide natively) in terms of [Polymer Project](#) – a Javascript library consisting of several polyfills for various specs of Web Components APIs. Also, Mozilla has been hard at work – making web components a reality through their own JavaScript library named [X-Tag](#) to bring custom element capabilities to all modern browsers. Similar to Polymer, it also allows us to easily create elements to encapsulate common behavior or use existing custom elements to quickly get the behavior you're looking for.

In a nutshell, AngularJS Directives is the closest thing to Web Components available at this time and we should ride this

wave to get ourselves ready for the future of Web Development.

When use Directives

Apart from setting the foundation for your next ambitious web application, the AngularJS framework and directives provide lots of benefits to improve the overall development process. With the ability to extend HTML, one could design and build an entire application on top of custom elements that would be more manageable and testable. Other benefits are:

Cleaning up the jquery spaghetti code

jQuery is a great library serving to bridge the gap between different browsers by working around various browser quirks efficiently and helping developers to focus more on building things than fixing browser issues. However, if you do not use jQuery wisely then you will soon end up writing unmanageable spaghetti code. There are many common things we often do in most web applications such as performing an action on every keystroke, adding CSS classes or removing them off an element, dynamically checking/unchecking checkboxes, binding custom events, and so on. All this is not possible without having a unique identifier to the element you are targeting in terms of ID, CSS class, data attributes, and so forth.

Take a look at the following jQuery snippet in which we simply bind a click handler on the button that posts content on Google Plus.

```
$('.post-to-gplus').click(function() {
  share();
  $(this).hide();
  $('#chk_gplus').prop('checked', true);
  $('#message').show();
});
```

The problem with this piece of code is that we have to wait until the page loads completely in the browser before we bind the click event on the button. Also the approach would not scale if we use multiple instances to share various posts. What if we make it declarative and less cluttered to handle multiple instances without modifying a bit. Here is the AngularJS version of the same.

```
<button ng-click='share(); shared = true;' ng-hide='shared'>
  Post to Google Plus
</button>

<input type='checkbox' ng-checked='shared'>Posted

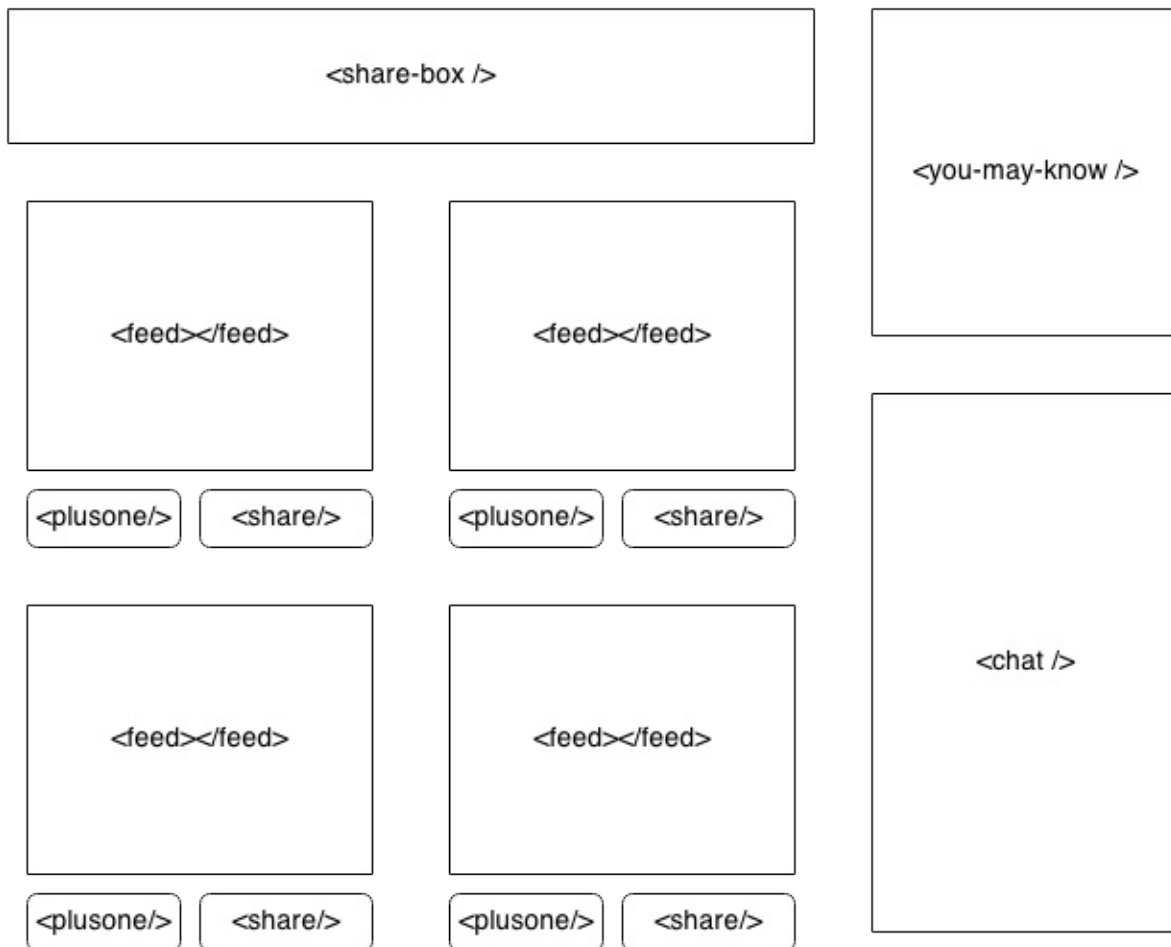
<div class='message' ng-show='shared'>
  Your post has been shared successfully!
</div>
```

A few things to note here is that there is no imperative DOM manipulation involved. Post sharing, we just set the variable, shared to true to update the view automatically. We've then used AngularJS built-in directives, ngHide, ngChecked, and ngShow to achieve exactly the same thing, although declaratively. Notice we also got rid of the ID and Class associated to make it scalable by encapsulating everything into a custom directive.

Assembling an application like lego bricks

Writing a huge application is not a one-man show so we often work with groups of people. While working with teams, maintaining a monolithic application might sabotage the development and probably affect the schedule as well. It even makes it harder to test the whole application that way. The best way is to divide the whole application into modules/components and let each team take care of it by facilitating separation of concerns. They can work on each component independently, test it in isolation, and make it ready to be assembled later.

Suppose you are building the next generation social network named Friendular to beat Google Plus and Facebook. You can break the application into different blocks as shown in the following figure. Each of these blocks could be developed separately and does one thing only.



As you can see the entire application is built on top of tailor-made custom directives just like Lego bricks. This alleviates the pain points and helps to catch potentials bugs/issues at an early stage.

Supporting designers to take a plunge into markups

Developers often have to work closely with web designers to knit a well-designed application so it's the developer's responsibility not to mess too much with the HTML. But you can end up making a mess because of your choice of framework or library, thereby preventing designers from diving into the markups and avoiding making any further changes. Even template libraries are not much of help sometimes.

Here is a simple example to render a list of to-do items. We just iterate over a collection of to-dos and render them as an ordered list.

```
var html = '<ol class="todo-list">';

for (var I = 0; I < todos.length; I++) {
  html+= '<li class="' + todos[I].done ? 'done' : 'undone' + '">' + todos[I].todo + '</li>';
}

html+= '</ol>';
$('body').html(html);
```

This makes it really hard for any designer to modify it if he is not aware of JavaScript or jQuery. Even though it generates a string of HTML, it is not the markup designers care about.

AngularJS's HTML as a template strategy pays off perfectly for both designers and developers in such scenario. So this will change to:

```
<body>
<ol>
  <li ng-repeat='todo in todos' ng-class='{true: "done", false: "undone"}[todo.done]'\>
    {{todo.todo}}
  </li>
</ol>
</body>
```

The `ngRepeat` directive in AngularJS lets us iterate over an array just like for loop in JavaScript. The `ngClass` directive allows us to conditionally apply CSS classes so in this case, if the todo's state is pending then it will remove the done CSS class and add undone instead – and vice versa. This will save the developer time and effort as they do not have to cherry pick the changes done somewhere else by the designer because the designer now can directly modify the same markup without getting into trouble anymore.

Gelling up well with jquery plugins

Even though AngularJS has a stripped-down version of jQuery built-in named jQLite, it does not stop you from using the original jQuery library. The jQuery ecosystem has grown rapidly because of jQuery's ability to extend it with your own function by exposing a public API using `jQuery.prototype` or `$.fn`.

AngularJS does not discourage developers from using jQuery but helps them instead to use it in the angular world in a new way. There are thousands of jQuery plugins available online and new ones coming every day, so porting them all to the AngularJS paradigm is time consuming but would definitely reduce the lines of code which is debatable.

AngularJS enforces declarative than imperative DOM manipulation and to a certain extent the declarative approach solves many problems. But, when you use any jQuery plugin the imperative syntax is inevitable, after all, jQuery is a DOM manipulation library. Still you can encapsulate the jQuery initialization code into its own angular directive to keep it simple. So this trivial jQuery plugin:

```
$(document).ready(function() {
  $('#mytab').fancyTabs();
});
```

Will change to more readable, manageable, and declarative markup ever.

```
<fancy-tabs id='mytab'></fancy-tabs>
```

There are lots of tidbits to jQuery plugins when it comes to AngularJS so we'll cover all those throughout the book.

Summary

We've peeked at the AngularJS arsenal and discussed the differences and similarities between AngularJS Directives (present) and Web Components (future). We've also realized the purpose of AngularJS existence and benefits it offers to Web Developers/Engineers to build powerful applications with ease. AngularJS really speeds up development and encourages us to reuse the saved time/effort to write tests. Also, learning AngularJS Directives now is a way to be ready for Web Components before they become reality.

The next chapter will unpack the AngularJS internals and things you should know about it before writing your first real world

directive.

AngularJS Fundamentals: A Crash Course

This chapter covers

- Usage of Module Design Pattern in Angular ecosystem
- Different ways of bootstrapping angular
- Difference between various services
- How to define, update, and delete models
- Various ways to enable two way data bindings

In the previous chapter, we have discussed the various ways that the angular directives can be helpful and how learning them will actually keep us using the future standards, today.

This chapter provides a quick tour of AngularJS internals – the things you need to know to understand the rest of the book. The purpose of the chapter is to make you aware of angular jargons that will be used through out the book. There are many things about angular that are confusing and new to anyone who comes to angular first time as it's not just another MVC framework you know already. Although, It's difficult to cover the entire framework in a single chapter so this chapter is more inclined to help you focus on important bits only.

Choosing AngularJS

Every developer has an ability to build amazing web applications but the choice of framework may hinder your dreams and knock you down even before you know it. The Angular Team has gone through these setbacks before while building large applications in-house and hence picked up the best practices to create yet another but extraordinary JavaScript framework to help developers spend time on what to build rather than how to. However, Angular has not invented any of the cool stuffs that it is famous for. So If you have an experience on other programming languages then some of these things may not be new but quite exciting for you to use them in JavaScript. Let's begin.

Managing complexity with Modules

Over the years we've seen small and large applications were built in a monolithic fashion. The main problem with this approach is that it makes it extremely difficult to debug and manage. The module design pattern overcomes these limitations offering privacy for properties and methods while exposing a public API.

Angular leverages the same pattern to separate the functionality of an application into independent and interchangeable modules such a way that each module holds everything necessary to do one thing and thereby improves maintainability and testability.

Understanding Modules

In Angular, every application has a main module acts as a backbone for the entire application and you can create as many sub-modules to serve the main module. In fact, a sub-module is a main module until we tie it up with another module. With modules, we can divide the whole application into pieces, functionality wise or component wise, and plug them later to make up the final application. Let's dive into defining our first module.

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

First thing to note here is that angular allows to name the module that suits us. It can be numbers, alphabets, and special characters. The `[]` as a second parameter allows us to pass in dependent modules i.e. sub-modules. Empty `[]` means no dependencies.

Module loading and dependencies

Remember that sub-module is itself a module so we can define them the same way as main module. Sub-modules can have other modules as dependencies further.

```
angular.module('SubModule1', []);
angular.module('SubModule2', []);

angular.module('MainModule', ['SubModule1', 'SubModule2']);
```

When you set up a dependency, all constructors/methods defined on the module come under the same umbrella so that you can invoke them without thinking about sub-module they belong.

Bootstrapping an Angular application

Defining a main module as seen before is not just enough to call angular for a spin. There is a process to initiate angular in an application. You can bootstrap angular either automatically or manually.

Declarative Initialization

Angular initializes automatically when `DOMContentLoaded` event is triggered or `document` is ready. You can tell angular to bootstrap the application using `ngApp` directive. We basically have to apply `ng-app` directive on `html` or any other tag you prefer. At this time, angular starts looking for `ngApp` directive starting from `<html>` tag all the way down to its last child. As soon as it finds it, it will load the associated module defined, and finally compile the DOM to update all bindings.

```
<html ng-app="MyApp">
  <head>
    <script src="http://code.angularjs.org/1.2.14/angular.js"></script>
    <script>
      angular.module('MyApp', []);
    </script>
  </head>
  <body>
    1 + 2 = {{ 1 + 2 }}
  </body>
</html>
```

In this example, we are doing simple arithmetic. The module name used in `angular.module` and in `ng-app` should be same to make it work. This is a widely used approach basically.

Imperative Initialization

Sometimes we need to have a control over when angular bootstraps in case of delaying the initialization process, you can use a manual initialization approach instead. If you've come from a jQuery background then the following example might be familiar to you. This approach may be useful while using angular in conjunction with other frameworks during the migration to AngularJS to control when angular kicks in.

```
<html>
  <head>
    <script src="http://code.angularjs.org/1.2.14/angular.js"></script>
    <script>
      angular.element(document).ready(function() {
        angular.module('MyApp', []);
        angular.bootstrap(
          angular.element(document).find('html'),
          ['MyApp']
        );
      });
    </script>
  </head>
</html>
```

```

    </script>
  </head>
  <body>
    1 + 2 = {{ 1 + 2 }}
  </body>
</html>

```

Few things to note here:

- angular has a strip-down version of jQuery named **jQLite**. It has a minimal set of methods jQuery supports and sometimes may differ in functionality as well.
- `angular.element` is the new `$` (if you do not include `jquery.js` before `angular.js`). Angular will use `$` internally if its available.
- Note `angular.element()` does not support selectors unlike `$` and thats why we'd to use `angular.element(document).find('html')` instead of `$('html')` in jQuery (Remember, jQLite differs from jQuery).
- Later we call `angular.bootstrap()` to begin the manual initialization.

This approach is extremely useful If you want to use angular in conjunction with some other frameworks in order to port part of your application on AngularJS during migration.

Configuring the Module

Once we define an angular module, we have to decorate it in different ways while writing an application. Angular module provides access to various construction functions or interfaces through which we can register new controllers, services, filters, directives, etc. They are:

- Value
- Constant
- Factory
- Service
- Provider
- Filter
- Config Phase
- Run Phase

Let's dive into them in detail.

Value

AngularJS provides an easiest way to manage a pre-instantiated Object to register which can be injected later. We can register anything as a value— that can not be decorated. That means it's not possible to inject it into methods/functions/services during a configuration of application. By doing so ito prevents accidental instantiation of services before they have been fully configured. It can be a string, a number, an array, an object or a function. You can define it as key & value pair and as many.

Every application has some sort of configurations on top of which an application behaves differently on development or production environments. Imagine you have an application that talks to a server to authenticate users and fetch their details, you might want to hit a staging server during development but a production server when it goes live. So `.value()` function is really handy to store such information. In JavaScript, you may use standard variable to do so such as:

```
var LoginService = 'http://myapp-dev.mydomain.com/loginService';
```

In angular, you can write the same as:

```
angular.module('myApp', [])
  .value('LoginService', 'http://myapp-dev.mydomain.com/loginService');
```

The real benefit here is that you can prevent polluting global namespaces unlike the JavaScript approach. In addition to it, such values can be injected in other parts of the application as a dependency.

Lets say we have to build a search engine application that uses various search engines APIs available to perform searches. For now we'll just use Google API to do lookups.

```
angular.module('SEA', [])
  .value('search_engine', 'google.com')
  .config(function() {
    // can not be decorated during the configuration
  })
  .run(function($rootScope, search_engine) {
    $rootScope.message = 'Searching in http://' + search_engine + ' for the sake of knowing...';
  });
```

Now that we've injected the value as a dependency in `run()` method to bind it to a model named `message` to be used in the DOM.

```
<div>{{message}}</div>
```



Constant

Unlike values, Constant lets us register values which can be injected into a module's configuration function to be decorated. Imagine you want to limit login attempts to 3 (maximum) which is a standard way to stop any kind of suspicious activities (probably by hackers) on the web or you might want to use Pi value for mathematical calculations in your application, the angular constant provides a better interface to do that. In JavaScript, you would do:

```
var MAX_LOGIN_ATTEMPTS = 3;
var Pi = 3.14159265359;
```

In angular, you would do:

```
angular.module('myApp', [])
  .constant('MAX_LOGIN_ATTEMPTS', 3)
  .constant('Pi', 3.14159265359);
```

The benefit is same as `.value()` but it can be injected during configuration phase that we'll see shortly. You can use Constant the same way as Value in HTML and you'll see exactly what we saw earlier in a browser with the following snippet.

```
angular.module('SEA', [])
  .constant('search_engine', 'google.com')
  .config(function(search_engine) {
    // can be decorated so perform few searches in the background
  })
  .run(function($rootScope, search_engine) {
    $rootScope.message = 'Searching in http://' + search_engine + ' for the sake of knowing...';
  });
```

NOTE: Both value and constant do not allow you to update its value and have it reflected elsewhere

Factory

The Factory is the most widely used service type because of its simplistic nature. You can set it up using **Object Literal Pattern** to avoid polluting the global name-space and return it as an object. Imagine you are building a mobile application and want to figure out whether its an iPhone or iPad or android device to toggle certain features of the application. You could have following in JavaScript:

```
var isIpad = navigator.userAgent.match(/iPad/i) != null;
var isIphone = navigator.userAgent.match(/iPhone/i) != null;
var isAndroid = navigator.userAgent.match(/Android/i) != null;
```

In angular, you can wrap such nifty global variables in a factory service named, Device as:

```
angular.module('myApp', [])
  .factory('Device', function() {
    return {
      isIpad: navigator.userAgent.match(/iPad/i) != null,
      isIphone: navigator.userAgent.match(/iPhone/i) != null,
      isAndroid: navigator.userAgent.match(/Android/i) != null
    }
  });
```

Which you can use easily within angular application as `Device.isIpad`, `Device.isIphone`, or `Device.isAndroid`. Keeping all useful helper methods in one place makes your code modular and easy to scale.

The same way, in the following example, we've abstracted the logic into a factory and injected it to call `find()` method to get the same message.

```
angular.module('SEA', [])
  .constant('search_engine', 'google.com')
  .config(function(search_engine) {
    // can be decorated so perform few searches in the background
  })
  .factory('LookupFactory', function(search_engine) {
    return {
      find: function() {
        return 'Searching in http://' + search_engine + ' for the sake of knowing...';
      }
    }
  })
  .run(function($rootScope, LookupFactory) {
    $rootScope.message = LookupFactory.find();
  });
```


The example is not hard to understand as we've just replaced the `run()` method with factory. You can even take it further by adding setters/getters methods to `LookupFactory` to override the search engine used.

Service

The Service is slightly different than Factory in terms of how it's defined but works much the same way. Factory return an object or instance but Service returns a Class instead which is instantiated internally by Angular after being injected. We can easily convert `Device` factory method into an angular service in case you prefer it as:

```
angular.module('myApp', [])
  .service('Device', function() {
    this.isIpad: navigator.userAgent.match(/iPad/i) != null;
    this.isIphone = navigator.userAgent.match(/iPhone/i) != null;
    this.isAndroid = navigator.userAgent.match(/Android/i) != null;
  });
```

The angular service produces a class unlike factory that gives it an object orientated flavor. Similarly, we can turn `LookupFactory` into `LookupService` so:

```
angular.module('SEA', [])
  .constant('search_engine', 'google.com')
  .config(function(search_engine) {
    // can be decorated so perform few searches in the background
  })
  .service('LookupService', function(search_engine) {
    this.find = function() {
      return 'Searching in http://' + search_engine + ' for the sake of knowing...';
    };
  })
  .run(function($rootScope, LookupService) {
    $rootScope.message = LookupService.find();
  });
```

It gives you an object oriented flavor with `this` keyword so its up to you what you prefer, service or factory. In my understanding, use Factory to combine helper methods in one place and use Service to create a full-fledged function that does one thing only, such as `AuthService`.

Provider

Ever wonder what if we could change the default search engine on the fly before performing searches. Of -course we can introduce setters in Factory/Service to do so but its not going to help during the configuration phase. That's where **Provider** shines. Imagine you publish a Service named `AuthService` on the web for other developers to use that authenticates a user using Google Auth API. But you want to make it flexible to support Facebook Auth API or Github Auth API without touching the core, you might expose methods to configure which Auth API to use or what action to take post login. Angular Providers give you that level of flexibility so that developers who are going to use it can configure it on the fly.

The Provider expects `$get` function that will expose all methods (`find()` in our case) defined into it after its instantiation. We can have setter methods (i.e `setSearchEngine`) also to be available during the configuration phase to easily change the default search engine to *bing.com*.

```
angular.module('SEA', [])
  .constant('search_engine', 'google.com')
  .provider('LookupService', function(search_engine) {
    var currentSearchEngine = search_engine;

    return {
```

```

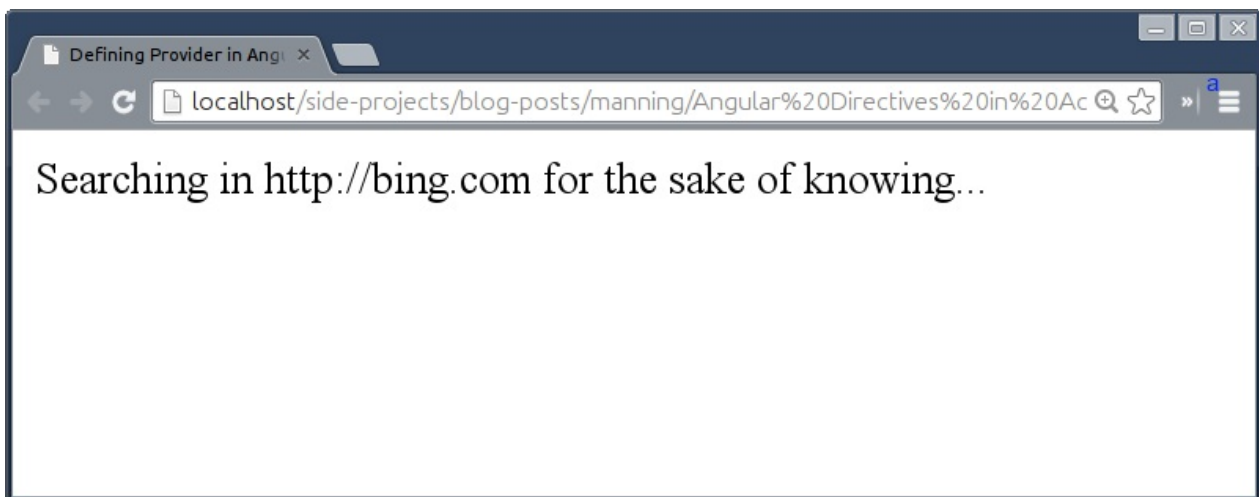
    $get: function() {
        return {
            find: function() {
                return 'Searching in http://' + currentSearchEngine + ' for the sake of knowing...';
            }
        };
    },

    setSearchEngine: function(se) {
        currentSearchEngine = se;
    },
};

})
.config(function(LookupServiceProvider) {
    LookupServiceProvider.setSearchEngine('bing.com');
})
.run(function($rootScope, LookupService) {
    $rootScope.message = LookupService.find();
});

```

Please note that the methods returned by `$get` can not be called during configuration unlike those which are outside. Although, It's a bit confusing for you to understand why have we named the provider as `lookupService` ? That is because the provider ultimately creates a service. Finally this is what we'll see in the browser.



Notice that *google.com* has been replaced by *bing.com* on the fly.

Filter

Filters are used to format data for display to the user. Imagine you want to support French along with English language in an application and you might store all the translations in a locale file as:

```

var enToFr = {
    'Hello, World': 'Bonjour, monde'
};
function translateIt(key) { return enToFr[key]; };

```

And you would use it in JavaScript as:

```
translateIt('Hello, World');
```

The same thing we can achieve in angular in a better way with filter constructor as:

```
angular.module('myApp', []).filter('translateIt', function() {
```

```
var enToFr = {
  'Hello, World': 'Bonjour, monde'
};
return function(key) {
  return enToFr[key];
};
});
```

Then you can use it in a Controller or Service by injecting a `$filter` built-in service to invoke our filter as:

```
$filter('translateIt')('Hello, World');
```

The real benefit here is that this also works in a DOM out of the box, so you could do:

```
<div>{{"Hello, World" | translateIt }}</div>
```

Let us see how we can use it in our Search Engine application. The link to *bing.com* is not clickable but we can easily convert it into a hyper-link using a built-in angular filter named, `linky`. The Linky filter comes under `ngSanitize` module which is a security layer to avoid any unsafe HTML rendered into Angular applications.

We'll have to inject `$filter` as a dependency which will give you access to various built-in filters in angular.

```
angular.module('SEA', ['ngSanitize'])
  .run(function($rootScope, LookupService) {
    $rootScope.message = $filter('linky')(LookupService.find());
  });
```

NOTE: You need to include `angular-sanitize.js` and inject `ngSanitize` module in the main module of your application. Also take a moment to go through other built-in filters (<http://docs.angularjs.org/api/ng/filter>).

This will turn *bing.com* into a weblink as shown below.



Apart from data formatting, Angular filters also provide high level of code abstraction because sometimes flooding services with utility functions is a bad practice. Let's write a custom filter to make the message Camel Cased.

In the following example, we are using a regular expression to match a first character from the each word in a message having a blank space before it or the word itself is at the beginning then make it uppercase.

```
angular.module('SEA', ['ngSanitize'])
  .filter('CamelCase', function() {
    return function(message) {
      return message.replace(/(?:^|\s)\w/g, function(match) {
        return match.toUpperCase();
      });
    };
  })
  .run(function($rootScope, LookupService) {
    $rootScope.message = $filter('CamelCase')(
      $filter('linky')(LookupService.find())
    );
  });
```

Using the filter constructor, you can actually expand the range of built-in filters and use them the same way. Then we've passed the output of `linky` to the `CamelCase` filter. The below figure shows how it works.



Angular filters API is a great example of abstracting a huge but crucial implementation into tiny testable pieces.

Config Phase

Angular bootstrapping process is divided into two phases. One of them is a configuration phase in which various providers such as `$rootScopeProvider`, `$locationProvider`, `$routeProvider`, and so on can be injected to be configured further. You can even inject custom providers.

Imagine you're using some sort of server side programming language that uses `{{}}` as a templating syntax and you want to use something else for angular data binding to avoid conflicts. The best place to tell angular to use some other interpolation syntax is the configuration phase. Let's see how:

```
angular.module('SEA', [])
  .config($interpolateProvider) {
    $interpolateProvider.startSymbol('[[');
    $interpolateProvider.endSymbol(']]');
  })
  .run(function($rootScope) {
    $rootScope.look = 'Look, It still works...!';
  });
```

The `$interpolateProvider` exposes two methods to change the symbols, namely `startSymbol` and `endSymbol`. In this example, we've changed the symbols to `[[` and `]]` respectively.

```
<div>[[look]]</div>
```

In addition to this, we can even register all the construction functions we saw earlier inside `config` using `$provide` provider and `$filterProvider` for filters without changing the implementation a bit.

```
angular.module('SEA', ['ngSanitize'])
.config(function($provide, $filterProvider) {
  $provide.value('search_engine_value', 'google.com');

  $provide.constant('search_engine_constant', 'google.com');

  $provide.factory('search_engine_factory', function(search_engine_constant) {
    return {
      find: function() {
        return 'Searching in http://' + search_engine_constant + ' for the sake of knowing...';
      }
    };
  });

  $provide.service('search_engine_service', function(search_engine_constant) {
    this.find = function() {
      return 'Searching in http://' + search_engine_constant + ' for the sake of knowing...';
    };
  });

  $provide.provider('search_engine_provider', function(search_engine_constant) {
    var currentSearchEngine = search_engine_constant;
    return {
      $get: function() {
        return {
          find: function() {
            return 'Searching in http://' + currentSearchEngine + ' for the sake of knowing...';
          }
        };
      },
      setSearchEngine: function(se) {
        currentSearchEngine = se;
      },
    };
  });

  $filterProvider.register('CamelCase', function() {
    return function(message) {
      return message.replace(/(?:^|\s)\w/g, function(match) {
        return match.toUpperCase();
      });
    };
  });
});
```

The following figure demonstrates the usage of all of these instances all together.



The configuration phase is the best place to replace and extend any part of the AngularJS core.

Run Phase

The second phase involved in the angular bootstrapping process is Run which is the closest thing to the main method in AngularJS. Because of which it runs after all the services are configured. It also resembles `$(document).ready()` event in jQuery as angular uses the same event internally to bootstrap itself which we saw in the beginning of the chapter. Let us look at the following example wherein we modify the model within controller which was defined in a run block as:

```
<html ng-app="myApp">
<head>
  <script src="http://code.angularjs.org/snapshot/angular.min.js"></script>
  <script type="text/javascript">
    var App = angular.module('myApp', []);

    App.run(function($rootScope) {
      $rootScope.name = 'AngularJS';
    });

    App.controller('MyCtrl', function($scope) {
      $scope.name = 'AngularJS, My precious!!!';
    });
  </script>
</head>
<body ng-controller="MyCtrl">
  {{name}}
</body>
</html>
```

The run method brings up the `$rootScope` which is a root `$scope` every angular application has. Other scopes become child scopes of it. All methods and properties assigned to `$rootScope` can also be read/write from child scopes and available anywhere within an application because of prototypical inheritance in JavaScript but not the other way around. If you run this example in a browser, you'll be amazed to see *AngularJS, My precious!!!* instead of *AngularJS*. But beware of using `$rootScope` to bind models unless its really necessary because `$rootScope` is a global object and thereby makes everything bound to it, global. As you may know that global variables do not get Garbage Collected by the browser VMs (Virtual Machine) and free the memory used. This might leak memory unless you manually do the cleanup. However, Child scopes will be destroyed unlike `$rootScope` when it's not required by Angular itself. Last year, I was working on a large application and mistakenly I used `$rootScope` to store all the data which was not consumed by the application all the time but it was still hanging around in memory even though it was not required. Later I figured that the `$rootScope`'s global nature causing the issue which I fixed later using Controllers.

Building better Data Model

Being an MVW framework, a model is an essential part of angular applications. Unlike other MVC frameworks where models contain the interactive data as well as a large part of the logic surrounding it in terms of conversions, validations, computed properties, and so on. Models in angular are just there to hold data without much fuzz and also provide an API to access and manipulate that data.

Defining a Model

Data models are just plain old JavaScript objects (POJOs) and do not require any getter/setter methods to instantiate. Any type of data, primitives or objects can be used as models in angular and hence makes it easy to set up. Let's learn by example.

```
angular.module('Modeling', [])
  .run(function($rootScope) {
    $rootScope.book = {
      name: 'Angular Directives in Action',
      status: 'reading'
    };
  });
```

In this very simple example, we've defined our first model named `book`, which is a normal JavaScript object – the only difference is that it's bound to `$rootScope` to be used inside angular view later. You can update the model the way you update any variable in JavaScript.

In addition to it, you can also instantiate a model at the DOM level using a special directive named `ng-model` and you can even fill it up using `ng-init` directive. This mainly used with all kinds of form controls such as:

```
<input type='text' ng-init="book = {'name': 'Angular Directives in Action', 'status': 'reading'}" ng-model='book.name' />
```

With this our `book` model will be set with default book data and will show the book's title in the input.

Listening for models mutation

It's very common to perform some actions when model changes and angular is no different. In many cases, angular by default takes care of model's mutation and updates the associated view automatically. But sometimes you need a control over when that happens, so `$watch` is the answer for that. A `$watch` is an extremely simple API to create a listener for a model – it takes a model name to watch and a callback to call post update. The callback returns an updated value to perform a check on and churn out a particular action.

```
angular.module('Modeling', [])
  .run(function($rootScope) {
    $rootScope.book = {
      name: 'Angular Directives in Action',
      status: 'reading'
    };

    $rootScope.$watch('book', function(newVal) {
      if (newVal.status === 'read') {
        console.log('Well Done..!');
      }
    });

    $rootScope.book.status = 'read';
  });
```

As soon as we update the book's `status`, the watch will trigger, and the “*Well Done..!*” message will be logged to the console.

Using a Controller to segregate models

To understand what a controller is and why we need to pay attention to it while scaffolding an angular application, we'll go through an interesting example. Suppose, you start a small company alone. In early days, you were excited about the progress as everything was well and good - New customers flocking every month, money flowing in, and the business starts to roll. After few months, you realized that you can not handle the load alone and to work effectively you need more people to join in.

You hire 10 people, all reporting back to you. The move of hiring more people was right so you could focus on business side than dealing with customers or marketing. By then the business grows a lot and you soon realize that 10 people are not enough to handle the growing customer base properly. This time the team of 10 people grows to 100 – all communicating directly to you. After few years, the company was stuck with the last milestone, could not expand anymore, and employees were unmotivated too. You soon figured that hiring more people is not the solution but to manage well in order to improve performance .

Immediately you schedule a meeting with all the employees and decided to split the whole team into sub-teams of having 10 people each. You recognize the great talent from each team and let them lead their respective teams. Then you name all the teams such a way that each name serves some purpose and also keeps team members motivated and focused on their Job. The significant benefit of this scheme is that only 10 team leaders are now reporting you on a daily basis. It's not that you get less information about the progress now but a concise and summarized report on what we are doing and where we are heading. With this, in couple of years, your company became structured and well organized, people were happy and rejuvenated to do their best, and wealth piled up. Awesome..!

Well, the intention behind the story was to tell you the tale of building a large application in AngularJS, and how you can smartly use controllers to manage and scale it. In the story, the company referred to an application itself and you were the `$rootScope` . The ten people you hired initially were the methods/functions used during bootstrapping. Later you saw the miserable condition of the `$rootScope` and the application as the load on the `$rootScope` increased because all methods/properties were global. Finally you split the whole application into different controllers (i.e. teams) to manage various parts of it by naming them descriptively. This leads the application to perform excellently and later scale well.

The take away is that do not rely heavily on the global scope but leverage child scopes with angular controllers.

How do I use Angular Controllers?

AngularJS controller is nothing but a plain JavaScript constructor function which exposes a `scope` . However, the scope object inherits root scope because of JavaScript prototypical nature. Basically, in JavaScript, when you look up any property or method on the object and if it does not exist there then it goes up the prototype chain until it finds. So, all the methods/properties defined on the `$rootScope` can be accessed via `$scope` . The angular controller creates some sort of barricade between the root scope and child scopes in a way that you can read from the outside world but can not expose to it. Having said that, you can have a method of the same name in more than one controller without breaking a thing.

Angular introduces a special directive named `ngController` . While angular bootstraps, it will hook up the scope created by the controller definition with the DOM. This makes sure that all the methods or properties used in the DOM act within the controller's fence. Let's look at an example. As we have already envisaged of running a successful company, for this example, we'll define two controllers, namely, `BODCtrl` and `CEOCtrl` for board of directors and chief operating officer respectively. And we'll display their job roles and the responsibilities they carry as follows:

```
var App = angular.module('CtrlDemo', []);

App.controller('BODCtrl', function($scope) {
  $scope.person = {
```



```

    'role': 'Board of Directors',
    'aka' : 'Trustees',
    'job' : 'for everything good for the organization'
  };
});

App.controller('CEOCtrl', function($scope) {
  $scope.person = {
    'role': 'CEO',
    'job' : 'for total management of an organization'
  };
});

```

Notice, in both the controllers, we have defined the same model named person. Then we'll declaratively specify the controllers in HTML to display the person details as follows:

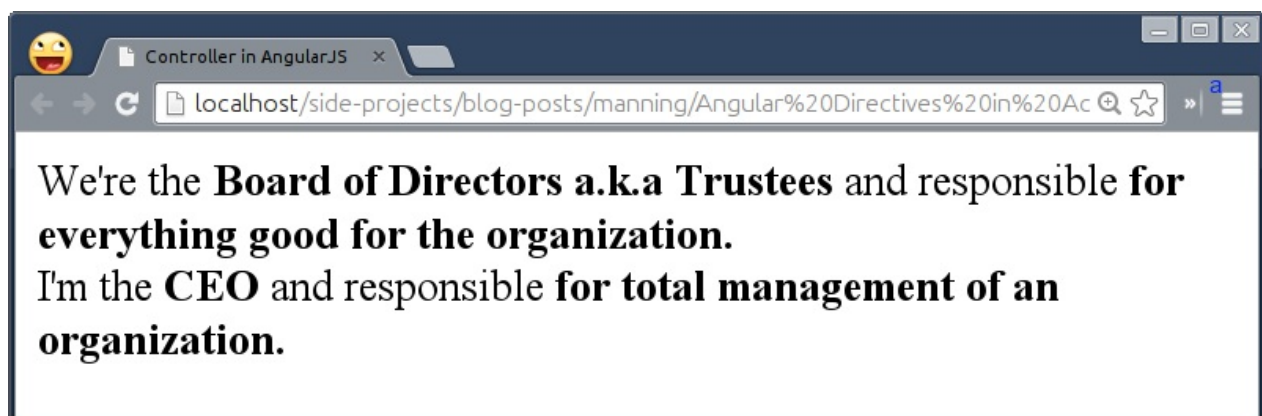
```

<body ng-app="CtrlDemo">
  <div ng-controller="BODCtrl">
    We're the
    <b>{{person.role}} a.k.a. {{person.aka}}</b>
    and responsible
    <b>{{person.job}}.</b>
  </div>

  <div ng-controller="CEOCtrl">
    I'm the
    <b>{{person.role}}</b>
    and responsible
    <b>{{person.job}}.</b>
  </div>
</body>

```

The code is pretty straight forward to understand as we've simply specified the controllers and using their respective models to display the details. At the end, this is what you'll see in the browser.



Many few people know that the CEO is not the king of the company and even he has to report to someone i.e. board of directors. So let's try nesting both the controllers and let the CEO tell who he reports to.

```

<body ng-app="CtrlDemo">
  <div ng-controller="BODCtrl">
    We're the
    <b>{{person.role}}</b>
    and responsible
    <b>{{person.job}}.</b>

    <div ng-controller="CEOCtrl">
      I'm the
      <b>{{person.role}}</b>
      and responsible
      <b>{{person.job}}</b>
      but reporting to
    </div>
  </div>

```

```

        <i>{{person.role}}</i>
      </div>
    </div>
  </body>

```

What we intend to have is the `BODCtrl`'s role in the highlighted section but the controller's confining scope does not allow us to access it inside `CEOCtrl`. To solve such problems and many more because of this restrictive behavior of the controller, AngularJS v1.2 came up with an easy fix to infiltrate with **Controller As** Syntax. This allows us to alias the controller at the directive level (in HTML). So our HTML would look like:

```

<body ng-app="CtrlDemo">
  <div ng-controller="BODCtrl as bod">
    We're the
    <b>{{person.role}} a.k.a. {{person.aka}}</b>
    and responsible
    <b>{{person.job}}.</b>

    <div ng-controller="CEOCtrl">
      I'm the
      <b>{{person.role}}</b>
      and responsible
      <b>{{person.job}}</b>
      but reporting to
      <i>{{bod.person.role}}</i>
    </div>
  </div>
</body>

```

You can have a unique name-space for the controller at the DOM level and the `$scope` becomes this giving you an object oriented flavor. So you can access person details through `bod` here. Let's update our `BODCtrl` to fix the demo as:

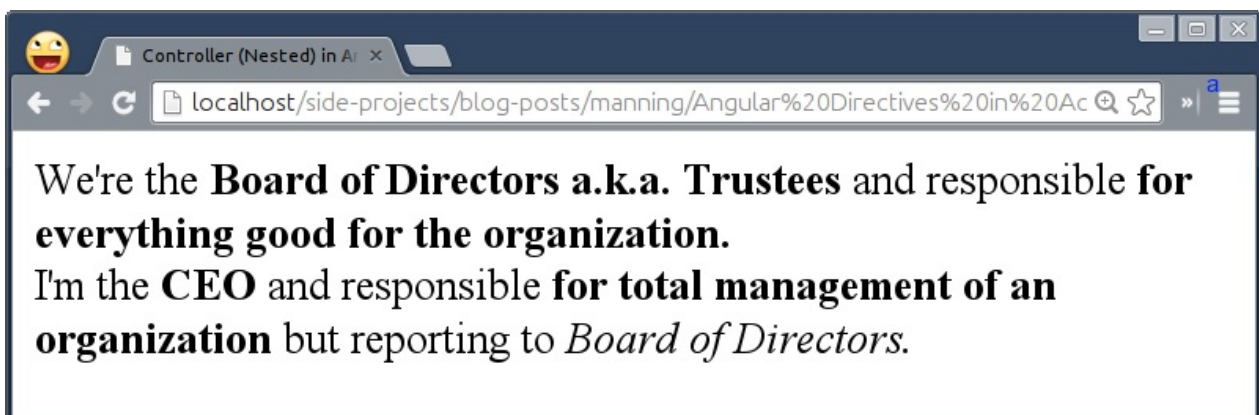
```

App.controller('BODCtrl', function($scope) {
  $scope.person = {
    'role': 'Board of Directors',
    'job' : 'for everything good for the organization'
  };

  this.person = $scope.person;
});

```

Alternatively you could bind the person object directly to this instead of `$scope`, but then we would have to replace all the instances of `person.*` with `bod.person.*` in the DOM. Use what you prefer. This will update as:



We saw how a controller is an essential ingredient in the MVC framework like AngularJS and let's us structure an application nicely. Now let us crack Data Binding.

Auto updating Views using 2-way data binding

Two way Data Binding is where angular shines a lot and that too without any glue code. It let's you use POJOs (Plain Old JavaScript Objects) to enable the binding as there is no complex setter/getter methods involved. The double curly `{{expression}}` syntax is very basic and widely used in many angular applications but this sometimes leads to **FOUC**. The FOUC stands for the Flash Of Unstyled Content that means you will see `{{model}}` in the App until angular compiles it to show the actual value. Additionally, you can either use `{{model || "?"}}` to show `?` before the model evaluates or apply `ng-cloak` CSS class on the element containing the expression to avoid FOUC. The `ng-cloak` CSS class normally keeps the element hidden until the expression evaluates and then reveals it by removing the CSS class. But that's extraneous and can be circumvented.

Eschewing FOUC

There are better ways to handle FOUC issues elegantly. It is preferable to use `ngBind` instead of `{{expression}}` which is almost similar to the `ngCloak` approach but less verbose. So you can use `ngBind` as follows:

```
<b ng-bind='person.role'></b>
<b>a.k.a</b>
<b ng-bind='person.aka'></b>
```

Please note that you can not use the double curly notation with `ng-bind` as it takes an expression to be evaluated not the already evaluated expression as a string.

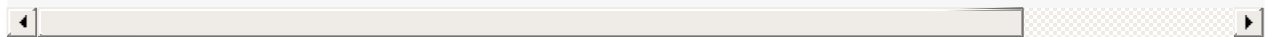
Unfortunately, `ngBind` can not even take more than one expressions so we'd to use multiple `ngBind` helplessly. However, `ngBindTemplate` directive can be useful in such scenarios. Let's see how:

```
<b ng-bind-template="{{person.role}} a.k.a. {{person.aka}}"></b>
```

As you can see, to support multiple expressions unlike `ngBind`, it has to leverage double curly notations.

To display BOD's details, we'd used many bindings. In order to reduce down these bindings to one, we may store the entire statement into a single model but neither `ngBind` nor `ngBindTemplate` evaluates HTML tags so you will see `Board of Directors` instead of **Board of Directors**. So, `ngBindHtml` directive is here to help.

```
$scope.statement = "We're the <b> Board of Directors a.k.a Trustees</b> and responsible <b>for everything good for the
```



And use the model, statement by replacing the existing markups as:

```
<span ng-bind-html="statement"></span>
```

You should look at how many bindings you are using and can they be reduced or even completely avoided as it is one of the optimization techniques to consider while writing slick and fast applications in Angular.

Handling events in Angular with the ease of native ones

While interacting with an application, we as a user, often perform many actions that present us with desire outputs. We press buttons, insert text into input boxes, drag something and many more but one thing we may not know is common here is that all these actions trigger events. That is, pressing button triggers a Click event, inserting text triggers a Keypress

event, dragging something causes a Mousemove event to trigger, and so on. HTML and JavaScript since a very beginning are rich to handle such events and provide built-in handlers such as `onClick`, `onKeyPress`, `onMouseMove`, and so forth but these events do not co-operate well with angular because it does not get notified as these events trigger outside the context of angular. Although angular provides public APIs to workaround this but it's not natural to work with.

Hence AngularJS team decided to have its own set of events as wrappers around the native ones. A `click` event is the most common one so let's see how to trigger it in angular as:

```
<b ng-bind-template="{{person.role}} a.k.a. {{person.aka}}" ng-click="person.role='BODs'"></b>
```

The `ngClick` handler can take either an angular expression or a function binded to `$scope`. The advantage of using `ngClick` over `onClick` is that it triggers a `$digest` loop (being in the angular's context) that updates the person's role. The same with `ng-submit` event to replace the native event, `onSubmit` to catch the form submission action and add our own behavior.

TIP: Please take a moment to go through all the events angular supports, <http://docs.angularjs.org/api/ng/directive>

Styling elements in angular way

Similar to events in angular, you can conditionally apply CSS classes on any element. Applying `active` or similar CSS class on an element to toggle its state is an ordinary task in most applications. So following jQuery snippet,

```
$('#button').click(function() { $(this).addClass('active'); });
```

can be rewritten in angular as:

```
<button ng-click="isPainting=true" ng-class="{true: 'active'}[isPainting]">Paint</button>
```

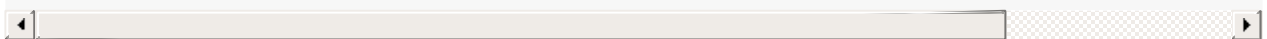
In this example, we are relying on a model named, `isPainting` and using it in a condition to apply active CSS class on the button. If the paint mode is not on then the class will be removed automatically. In case you want to use a different CSS class if `isPainting` is false then you can extend it as:

```
ng-class="{true: 'active', false: 'inactive'}[isPainting]"
```

We might want to apply the color to the button which a user is using while painting. In that case, `ngClass` would not be much helpful because the color would be any random color and creating different CSS classes per color is not the right approach.

An `ngStyle` directive comes really handy to tackle this issue. `ngStyle` lets you apply inline CSS styles conditionally.

```
<button ng-click="isPainting=true;color='red';" ng-class="{true: 'active'}[isPainting]" ng-style="{background-color: color}">
```



We again rely on a different model named `color` to set the background color on the button. If you change the model to something else later then angular will repaint the background with the new color.

Listing a List

At some point, every application has to show a list of collection in a tabular format but angular application is really good at

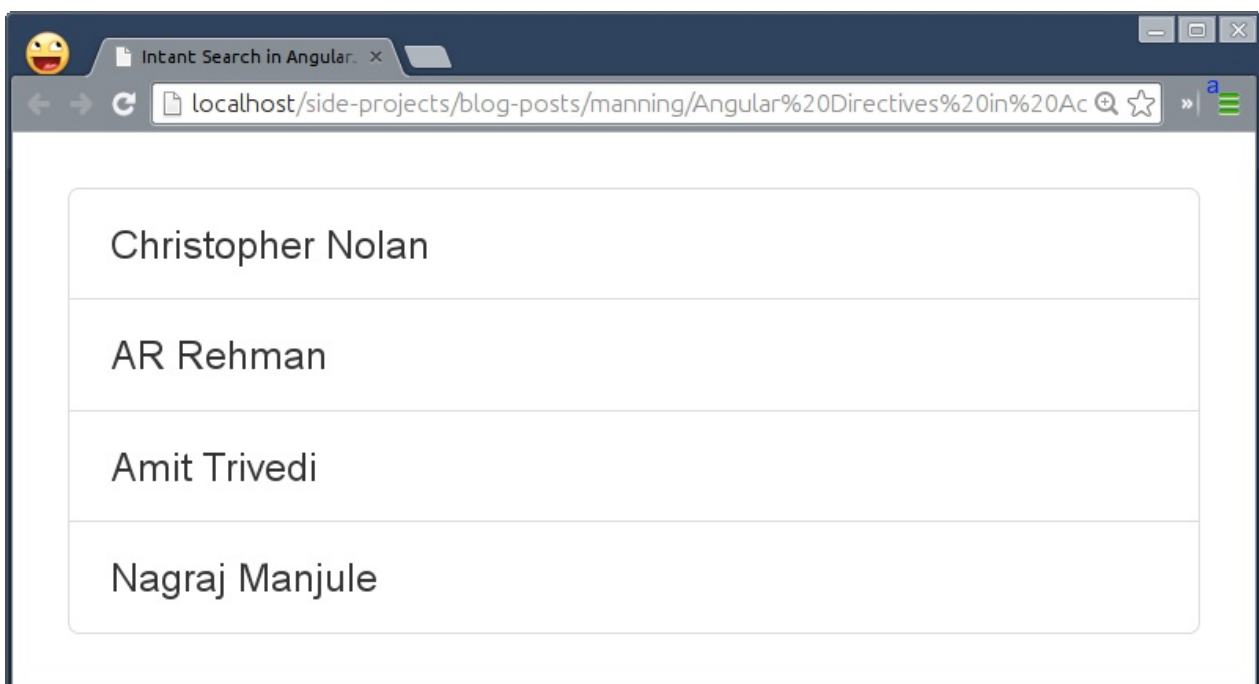
it. Angular has a special directive named `ngRepeat` that let's you iterate over a collection and it's even well optimized to render a large collection. Here is my favorite non-techies defined in a controller:

```
App.controller('InstantCtrl', function($scope) {
  $scope.favorites = [
    'Christopher Nolan',
    'AR Rehman',
    'Amit Trivedi',
    'Nagraj Manjule'
  ];
});
```

And we'll use it in HTML as follows:

```
<div ng-controller="InstantCtrl">
  <ul class="list-group">
    <li class="list-group-item" ng-repeat="favorite in favorites" ng-bind="favorite"></li>
  </ul>
</div>
```

Here `favorite` refers to each item in the collection. You will see the following output.



Adding an instant search functionality to a collection to quickly filter out the data, always gives me the creeps and so may you. It's horrified and time consuming to implement it in pure jQuery or JavaScript from the scratch. In contrast, angular makes it way simpler than ever using built-in `search` filter. We'll first add an input to type the search criteria:

```
<div ng-controller="InstantCtrl">
  <input type='text' ng-model="search" />

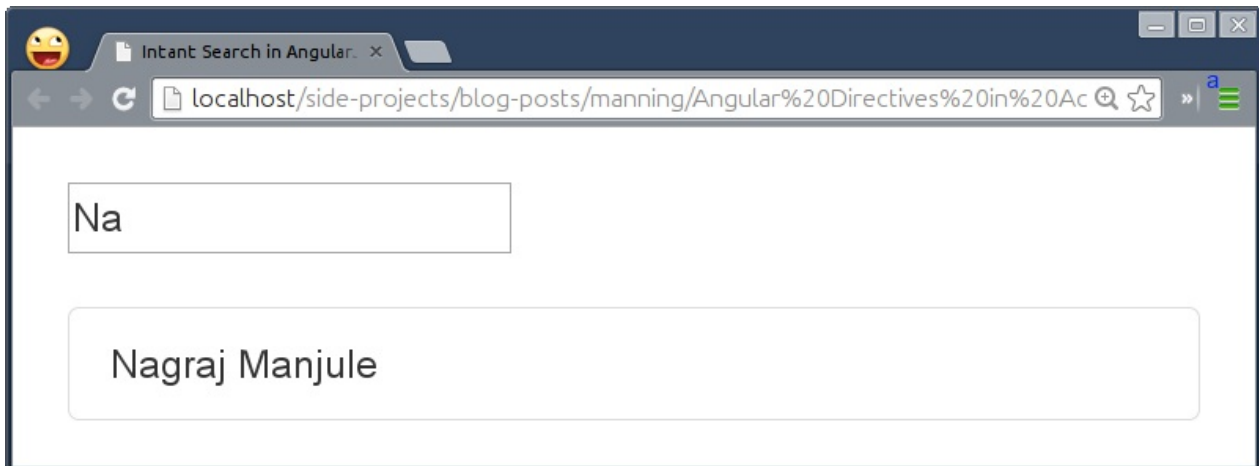
  <ul class="list-group">
    <li class="list-group-item" ng-repeat="favorite in favorites" ng-bind="favorite"></li>
  </ul>
</div>
```

Note that `ng-model` directive binds any form control (input in this case) to a property on the `$scope`. It is same as defining `$scope.search = ''` in the controller.

As we saw in earlier sections that angular let's us define custom filters but it also has built-in filters to simplify developer's life. The search filter named `filter`, yeah funny, is one of them that returns a subset of a collection based on an expression passed. Let's use it and update the HTML as:

```
<li class="list-group-item" ng-repeat=favorite in favorites | filter: search" ng-bind=favorite"></li>
```

Passing the model, `search` will filter out the collection as:



So iterating over a collection and rendering the data in HTML was never easier than this. Also custom and built-in filters power us to extend its behavior.

Summary

We made it! We've learned the pretty much all the basic but important stuff about the nuts and bolts of AngularJS. We've understood how angular provides various ways to structure small as well as large applications which make it really easy to scale later. We also found out different ways to enable data bindings through rich set of built-in directives. In short, Angular is not just a framework but a toolkit that can be customized and extended as we wish.

The next chapter will be about testability.

Testing the Beast: Unit testing and E2E testing

This chapter covers

- The importance of testing to build workable applications
- Seamless Unit Testing with Karma Test Runner
- Benefit of End to End testing with Protractor
- Leveraging all these to help us write testable directives

We have already learned enough about how AngularJS helps developers to keep JavaScript code aside from the DOM to make it easy to test but this chapter will take our learning to the next level and teach us how to write testable AngularJS applications and directives seamlessly. In this chapter, we'll set up an established development environment first. Then we'll get over the intricacies of testing with Karma Test Runner and Protractor. We'll also learn how Protractor facilitates an end to end testing strategy to automate manual testing the user would do instead. At the end of chapter, we'll configure Karma and Protractor to suite our testing needs for all the examples from the book.

We are going to use Jasmine to write assertions for both Unit and E2E testing. I would recommend to go through the documentation on how to write assertions in Jasmine (<http://jasmine.github.io/2.0/introduction.html>) and Protractor (<https://github.com/angular/protractor/blob/master/docs/api.md>) as we are not going to cover them here in detail.

Setting up the Testing Environment

We as creators often spend our precious time and put huge efforts to build something amazing. No matter how experienced you are or which company you work with, every application you craft is ambitious for you and your growth. But those ambitions may get crushed if you do not test your application beforehand and may run into last-minute bug on launch day or miss important deadlines too. It's not that people ship stuff without testing; however many of us do manual testing that means fiddling with the application from a user's perspective. Although there is nothing wrong in it but developers are humans and humans are not perfect unlike automatons. Moreover, manual testing is time consuming for developers to spend their time doing it.

Imagine you are working on some application along with your team and you recently introduce a feature that somehow break the existing functionality; but you are not aware of it until you release the new implementation to Quality Control (QC) and they notify you. The worst case scenario would be that if even QC team did not notice the bug and hit your users badly. So automated testing flow helps us to focus on shipping than worrying about broken things. Here we could just offload the tasks to a system that does it for us. Many such problems can easily be rectified by automated testing before being deployed on production. According to **Julie Ralph** who is one of the core AngularJS developers and Head of Protractor (E2E testing framework for AngularJS) once said that, *"Testing is about gaining confidence that your code does what you think it should do"*.

In this section, we'll see how easy is it to set up the testing environment to test AngularJS code. We'll begin with the basic installation of prerequisite softwares for a testing environment to run on. Let's get started.

Installing Git

I presume you must have used Git before but if not, Git is a free and open source distributed version control system designed to handle everything from small to large projects. I can not imagine working alone or in a team without using Git at work and home on my personal projects. Although Git is not the requirement for testing AngularJS directives, but we'll going to need it in the last chapter wherein we'll publish all the directives to the world. We'll host them on Github – an online project hosting for Git – for other developers to access them via Bower. If you are not fond of command line, visit git-scm.com website to download the installer to install the latest version of Git on your system. Following instruction might be helpful for command line fanatics.

On a Mac, again the easiest way to do this is using Homebrew (<http://brew.sh>). Run the following command in the Terminal after installing Homebrew:

```
brew install git
```

On Windows, the easiest way to install Git is to run the Windows installer from <http://git-scm.com/downloads>.

On Linux, run the following command in the shell:

```
sudo apt-get install git
```

Post installation, make sure Git is properly installed by running `git --version` command which should tell you the installed version.

Installing NODE.JS and NPM

Node.js is a platform built on V8 – a JavaScript Engine which powers Google Chrome browser. That means Node.js does not run in a web browser but as a server side JavaScript language. Whereas NPM is a NodeJS package manager written entirely in JavaScript, and runs on the NodeJS platform. Meaning we can use it to install packages/programs written in NodeJS. We do not need to install it separately as it is bundled and installed automatically with the environment. NPM runs through command line and manages dependencies for an application which we'll see shortly. To install NodeJS, visit nodejs.org (<http://nodejs.org>).

On a Mac, you could either run the installer from the preceding link or run following command to install it using Homebrew package manager. This will install both Node and NPM.

```
brew install npm
```

On Windows, simply download the installer from nodejs.org (<http://nodejs.org>).

On Linux, you could download the source and compile it manually using following commands. Uncompress the downloaded source first. Then run:

```
./configure  
make  
sudo make install
```

You could also install through Linux package manager.

```
sudo apt-get install npm
```

Assuming you have successfully installed node.js, to make sure just fire up a terminal and run following command to check the version installed as:

```
node --version
```

We are good to go if it shows the installed version without any error. Also note that any of these (Apache2, Nginx, WAMP, XAMPP, and so on) web servers is required for Protractor tests to work as it can not handle local files. So install any web

server before moving on.

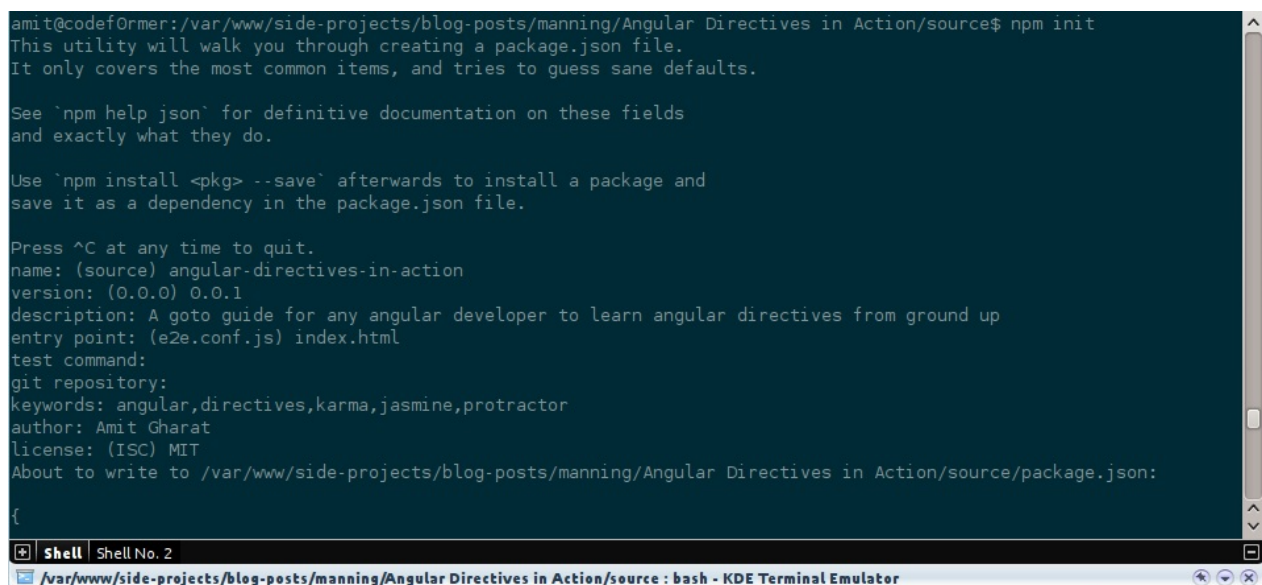
Now let's create a directory for our project somewhere in the root directory of the web server you have installed probably in `C:/Program Files/Apache Group/Apache2/htdocs` or `/var/www/` as:

```
mkdir angular-directives-in-traction
cd angular-directives-in-traction
```

Then create a `package.json` file by running following command in the terminal as:

```
npm init
```

You will be bombarded with few questions by the command such as *name of the package*, *description*, *version*, *entry point*, *test command*, *git repository*, *keywords*, *author*, and *license*. These information will be important for us to publish AngularJS directives later. For now, name and author information is enough to move on as shown:



```
amit@codefOrmer:/var/www/side-projects/blog-posts/manning/Angular Directives in Action/source$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (source) angular-directives-in-action
version: (0.0.0) 0.0.1
description: A goto guide for any angular developer to learn angular directives from ground up
entry point: (e2e.conf.js) index.html
test command:
git repository:
keywords: angular,directives,karma,jasmine,protractor
author: Amit Gharat
license: (ISC) MIT
About to write to /var/www/side-projects/blog-posts/manning/Angular Directives in Action/source/package.json:
{
```

NOTE: To get more specifics of NPM's package.json handling, please go through <https://www.npmjs.org/doc/json.html>

Installing Bower

Like NPM is a package manager for NodeJS, Bower is for the web. It is created by Twitter which offers a generic, unopinionated solution to the problem of front-end package management. Up until, developers used to visit a website to grab an updated version of a jQuery plugin or any library but not now. Just run bower command to install, update, and remove any dependency. For that, first install bower with:

```
npm install -g bower
bower --version
```

This command will install `bower` globally.

As we have bower in place, let's install necessary AngularJS modules for all the examples we have seen so far in previous two chapters. First create `bower.json` as:

```
bower init
```

Please enter *project name*, *description*, and so on when asked. At the end, it will create `bower.json` which is really useful to encompass references of external modules or jQuery plugins. You can check out the following figure to fill in the answers.



```

amit@codefOrmer:/var/www/side-projects/blog-posts/manning/Angular Directives in Action/source$ bower init
[?] name: angular-directives-in-action
[?] version: 0.0.1
[?] description: A goto guide for any angular developer to learn angular directives from ground up
[?] main file: index.html
[?] what types of modules does this package expose?
[?] keywords: angular,directives,bower,karma,jasmine,protractor
[?] authors: Amit Gharat
[?] license: MIT
[?] homepage:
[?] set currently installed components as dependencies? Yes
[?] add commonly ignored files to ignore list? Yes
[?] would you like to mark this package as private which prevents it from being accidentally published to the regist
[?] would you like to mark this package as private which prevents it from being accidentally published to the regist

```

Now that bower has been set up, let us install following dependencies locally instead of loading them directly via CDN as seen in previous chapters:

```

bower install --save angular
bower install --save angular-sanitize
bower install --save angular-mocks
bower install --save jquery#1.10.2
bower install --save bootstrap

```

You may be wondering, what does `--save` do? Well, all these bower packages will be referenced under dependencies section in `bower.json`. This is useful in case you are working in a team and to install the bower packages required for your application to work on other systems, you can just run following command to resolve dependencies automatically:

```

cd angular-directives-in-action
npm install

```

You can take a look at `bower.json` to find records of all the packages that are downloaded under `bower_components/` directory.

```

amit@codef0rmer:/var/www/side-projects/blog-posts/manning/Angular Directives in Action/source$ bower install --save
angular && bower install --save angular-sanitize && bower install --save angular-mocks && bower install --save jquery
y#1.10.2
bower cached      git://github.com/angular/bower-angular.git#1.2.18
bower validate     1.2.18 against git://github.com/angular/bower-angular.git#*
bower cached      git://github.com/angular/bower-angular.git#1.2.18
bower validate     1.2.18 against git://github.com/angular/bower-angular.git#1.2.18
bower cached      git://github.com/angular/bower-angular.git#1.2.18
bower validate     1.2.18 against git://github.com/angular/bower-angular.git#>=1
bower cached      git://github.com/angular/bower-angular-sanitize.git#1.2.18
bower validate     1.2.18 against git://github.com/angular/bower-angular-sanitize.git#*
bower cached      git://github.com/angular/bower-angular-mocks.git#1.2.18
bower validate     1.2.18 against git://github.com/angular/bower-angular-mocks.git#*
bower cached      git://github.com/jquery/jquery.git#1.10.2
bower validate     1.10.2 against git://github.com/jquery/jquery.git#1.10.2
bower cached      git://github.com/jquery/jquery.git#2.1.1
bower validate     2.1.1 against git://github.com/jquery/jquery.git#>= 1.9.0
amit@codef0rmer:/var/www/side-projects/blog-posts/manning/Angular Directives in Action/source$ cat bower.json
{
  "name": "angular-directives-in-action",
  "version": "0.0.0",
  "authors": [
    "Amit Gharat a.k.a. codef0rmer"
  ],
  "description": "A goto guide for any angular developer to learn angular directives from ground up",
  "main": "index.html",
  "keywords": [
    "angular",
    "directives",
    "grunt",
    "bower",
    "karma",
    "jasmine",
    "protractor"
  ],
  "license": "MIT",
  "ignore": [
    "**/*.**",
    "node_modules",
    "bower_components",
    "test",
    "tests"
  ],
  "dependencies": {
    "angular": "~1.2.18",
    "angular-sanitize": "~1.2.18",
    "angular-mocks": "~1.2.18",
    "iscroll": "~5.1.1",
    "jquery": "1.10.2",
    "bootstrap": "~3.1.1",
    "animate.css": "~3.1.1"
  }
}

```

We have successfully installed the prerequisite softwares such as Git, Node, NPM, and Bower on top of which our testing stack will run. Now let us set up Karma and Protractor in following sections.

Unit Testing with Karma

First thing to note here is that Karma is not a framework to write unit tests. It is essentially a tool which spawns a web server that executes source code against test code for each of the browsers connected i.e. captured. A browser can be captured in two ways:

1. Manually by visiting the URL where the Karma server is listening (typically <http://localhost:9876>)
2. Automatically by letting Karma know (via `unit.conf.js`) which browsers to start when Karma is run.

Once specified browsers are captured, the results for each test against each browser are examined and displayed via the command line to the developer such that they can see which browsers and tests passed or failed in real-time. We'll set it up soon but here is how it looks to get you excited:

```

amit@codef0rmer:/var/www/side-projects/blog-posts/manning/Angular Directives in Action/source$ npm run test-unit
> angular-directives-in-action@0.0.0 test-unit /var/www/side-projects/blog-posts/manning/Angular Directives in Action/source
> karma start unit.conf.js

INFO [karma]: Karma v0.12.16 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 37.0.2008 (Linux)]: Connected on socket ysunRxfovuGaVY5B2CH6 with id 59853766
Chrome 37.0.2008 (Linux): Executed 1 of 1 SUCCESS (0.092 secs / 0.091 secs)

```

Karma also watches all the files, specified within the configuration file, and whenever any file changes, it triggers the test run by sending a signal to the testing server to inform all of the captured browsers to run the test code again. Each browser then loads the source files inside an Iframe, executes the tests and reports the results back to the server. The server collects the results from all of the captured browsers and presents them to the developer in the terminal. The main goal of Karma is to bring a productive testing environment to developers.

TIP: If you are more interested to learn how karma works internally then Vojta Jina's thesis about Karma for unit testing web applications is <https://github.com/karma-runner/karma/raw/master/thesis.pdf> worth a read.

Installing Karma

Karma installation is extremely easy with NPM. You just have to run following commands in the terminal:

```
npm install karma --save-dev
npm install karma-chrome-launcher --save-dev
npm install karma-jasmine --save-dev
```

Similar to `--save` command line option used with bower, `--save-dev` option records a reference of the installed NPM package under `devDependencies` section in the `package.json` file. Again, this is to make it easy to resolve `devDependencies` on different systems.

In case, you get `EACCESS error` because of missing write permissions on the `node_modules/` directory. Try once again with `sudo`:

```
sudo npm install karma --save-dev --allow-root
sudo npm install karma-chrome-launcher --save-dev --allow-root
sudo npm install karma-jasmine --save-dev --allow-root
```

Notice it must have created karma directory inside `node_modules/`. It has also installed `karma-chrome-launcher` to enable Karma to boot Google Chrome browser by locating the installed location of Chrome on your system to run tests. The `karma-jasmine` package is an adapter for Jasmine so we do not have to load it manually while testing.

Okay, we can now test karma, just fire following command in a terminal:

```
./node_modules/karma/bin/karma start
```

If you find *Karma v0.12.16 started*, just clap on..! However, typing such lengthy command all the time kinda sucks and so you might find it useful to install `karma-cli` globally. Remember all your NPM packages reside in `node_modules/` directory unless installed globally with `-g` option:

```
npm install -g karma-cli
karma start
```

This allows us to run karma from anywhere but will use the local version if exists.

Configuring Karma

It's time to configure Karma to serve our project well. Karma needs to know about our project structure in order to run tests and this is done via a configuration file. So let us create a Karma configuration file with:

```
karma init unit.conf.js
```

Beware that you will again get bombarded with quick questions but keep your chin up and stay with me. Please use exact answers given below to set it up properly.

1. Which testing framework do you want to use? **Jasmine**
2. Do you want to use Require.js? **No**
3. Do you want to capture any browsers automatically? **Chrome**
4. What is the location of your source and test files? **js/*.js**
5. Should any of the files included by the previous patterns be excluded? **Press Enter!**
6. Do you want Karma to watch all the files and run the tests on change? **Yes**

This will create `unit.conf.js`, do check it out. You could see `js/*.js` as mentioned above under files section:

```
files: [
  'js/*.js'
],
```

If you remember we had written Unit as well as End-to-End (E2E) tests in the first chapter to validate AngularJS templating so it is time to test them in real browsers. Replace files option in `unit.conf.js` as:

```
files: [
  'bower_components/angular/angular.js',
  'bower_components/angular-mocks/angular-mocks.js',
  'js/ch01/angular-template.js',
  'tests/specs/ch01/*-unit.js'
]
```

The `angular-mocks` module allows us to inject and mock AngularJS services into unit tests. In addition, it also extends various core AngularJS services so that they can be inspected and controlled in a synchronous manner within test code.

After bower components, we have added the actual JavaScript code and then test files. For the sake of convention, we will use the same file name for the test with the suffix `-unit.js` throughout the book. Now as we have everything in place, we can see if the tests are working or not using following command:

```
karma start unit.conf.js
```

The command will fire up Google Chrome and validate our tests by showing the following output in the terminal:

```
INFO [karma]: Karma v0.12.16 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 36.0.1976 (Linux)]: Connected on socket lrgRQrSoNdT9DqcbCx-j with id 78497101
Chrome 36.0.1976 (Linux): Executed 1 of 1 SUCCESS (0.081 secs / 0.078 secs)
```

Keep updating the `unit.conf.js` file for any new examples to test.

Extending Karma

Karma is fully customizable in a sense that it can be molded to accommodate our application structure. We'll go through

some but essential configurable options here. Open `unit.conf.js` in an editor and update it as per the instructions given below.

BASEPATH

It is essential to tell Karma where is the configuration file located. This path is prepended to all relative paths defined under files section. In case the karma configuration file is not in the root of the application but the source and test files are, then you may have to change `basePath` to say, `../../..` depending upon directory structure you have followed. We'll keep it empty in our case as:

```
basePath: ''
```

FRAMEWORKS

Karma also needs to know about the testing framework used to write unit tests. This option let us use a list of testing frameworks we want to use. As we saw earlier that Karma, by default, comes with Jasmine adapter so we will stick with Jasmine so:

```
frameworks: ['jasmine']
```

In case you are interested to use qunit then first install karma-qunit node module using npm command in the terminal.

```
npm install karma-qunit --save-dev
```

And later update frameworks section from `['jasmine']` to `['qunit']` in the `unit.conf.js` file.

TIP: There are various testing frameworks karma supports, please visit <https://www.npmjs.org/browse/keyword/karma-adapter> to install them if needed.

FILES

This option takes a list of source files and test files to load in the browser while testing by Karma. You can even target files with patterns or regular expressions. That is:

```
files: [
  'js/**/*.js',
  'tests/specs/ch01/*-unit.js'
]
```

The first entry does a recursive lookup with `/**/` for all JavaScript files inside `js/` and child directories. The second one matches all the file names ending with `-unit.js`.

PORT

This refers to the port where karma server runs on. The default port is `9876` but you can change it to anything that not in use. We'll stick to the default one as:

```
port: 9876
```

Nevertheless, handy option to keep port number easy to remember with custom one.

BROWSERS

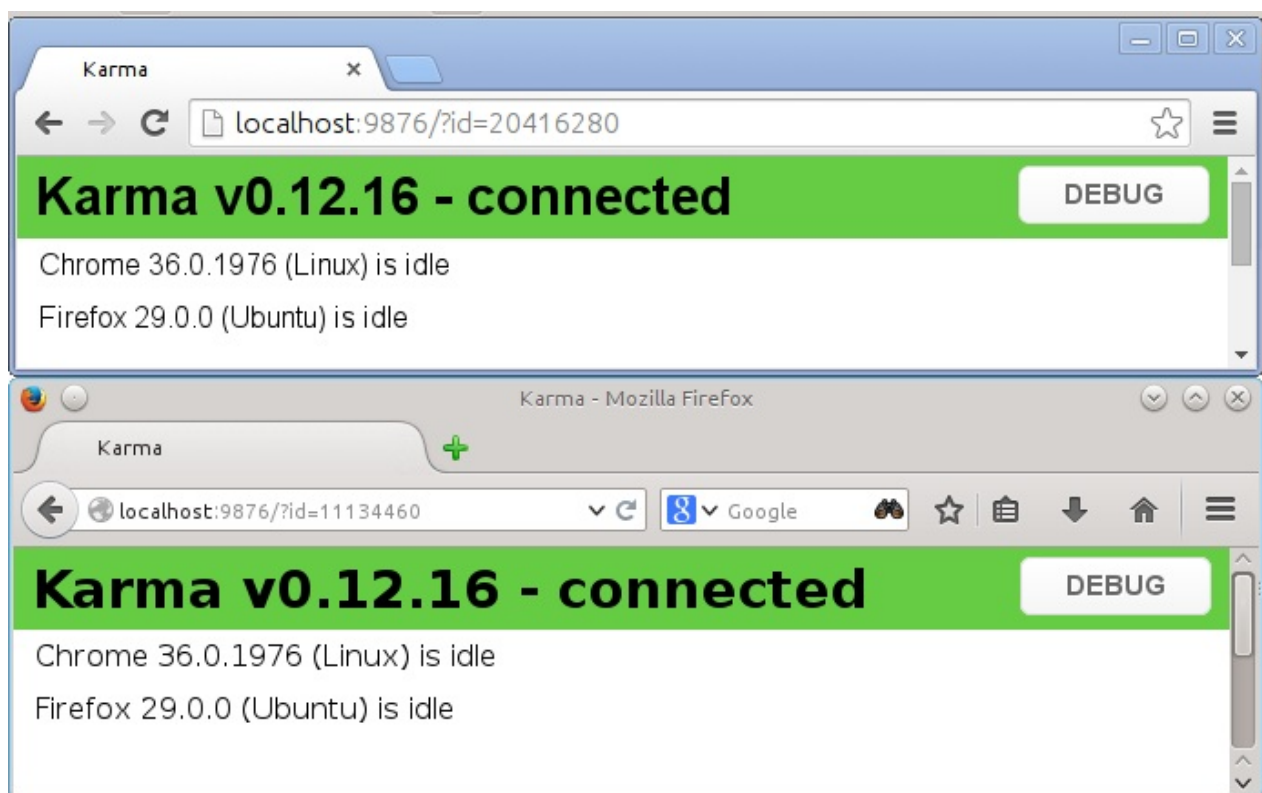
We often do testing on multiple browsers but Karma uses Google Chrome browser, by default, to run all the tests. However, we can easily add a bunch of browsers to run the same tests. For instance, to automate the same for Firefox, we need to install Firefox launcher (similar to Chrome launcher) with following command first:

```
npm install karma-firefox-launcher --save-dev
```

And update browsers section in `unit.conf.js` as:

```
browsers: ['Chrome', 'Firefox'],
```

If we restart karma again with `karma start unit.conf.js` command in the terminal then we'll see both browsers running the same tests as shown in the following figure.



TIP: Please visit <https://www.npmjs.org/browse/keyword/karma-launcher> to install other kinds of launcher including IE, Safari, Opera, and so on.

Note that the karma server starts at port `9876` and then immediately loads the same URL in Google Chrome and Firefox to capture it in order to run tests atop. In addition, you can even load the same URL in Internet Explorer to run tests against it.

SINGLERUN

This is useful for continuous integration mode. We are going to keep captured browsers alive so that tests will be run every time we make changes to our files with:

```
singleRun: false
```


If set to `true` then karma will capture all targeted browsers, run the tests, and exit.

Definition: Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

AUTOWATCH

This enables/disables watching files listed in files section and executing tests whenever any file changes. Useful only if `singleRun` option is set to `false`.

TIP: There are other options exist to fully customize Karma that we are not covering here because of space constraint, so feel free to go through them at <http://karma-runner.github.io/0.10/intro/configuration.html>.

In addition to all of these options in `unit.conf.js`, you can override some of them on the fly while starting Karma as shown:

```
karma start unit.conf.js --browsers Chrome,Firefox --port 1337 --singleRun true --autoWatch false
```

That's it..! We are done with Karma installation.

Integration Testing with Protractor

Like Jamine is to Unit Testing, Protractor is to E2E (End to End) or Integration Testing. When applications grow in size and complexity, it becomes unrealistic to rely on manual testing to verify the correctness of new features, catch bugs and notice regressions. End to end tests are made to find these problems.

Protractor is an End to End test framework for AngularJS built on top of WebDriverJS a.k.a Selenium. Selenium-Webdriver is a browser automation framework. Tests are written with the WebDriver API, which communicates with a Selenium server to control the browser under test. It runs tests against your application running in real browsers, interacting with it as a user would. Visit <http://angular.github.io/protractor/#/infrastructure> to know more about how it works under the hood.

Installing Protractor

Let's quickly install protractor locally. Additionally you could install protractor globally with `-g` flag if you wish as:

```
npm install protractor --save-dev
```

Protractor installs `webdriver-manager` by default which is a part of Selenium that controls browsers to run automated tests to mimic the behavior of a real user to test applications. Then we will run the Selenium installation script to download the chromedriver required to run Selenium itself and build a start script:

```
./node_modules/protractor/bin/webdriver-manager update
```

Please note that Selenium has a dependency on Java, so you must have Java (<http://java.com>) installed on your system. Now we are good to start the standalone version of Selenium.

Configuring Protractor

Similar to Karma, protractor also relies on the configuration file. The configuration file specifies how the runner should start webdriver, where your test files are, and global setup options. Protractor already has a reference configuration file that we can use and update accordingly. Let's copy it in the root directory of our project:

```
cp ./node_modules/protractor/docs/referenceConf.js e2e.conf.js
```

It uses Jasmine as a default framework to write assertions.

Extending Protractor

Protractor is also fully configurable like Karma. So let's check with some of the important configurable options and update them.

BASEURL

Use to define an absolute base URL of an application. In our case, it would be:

```
baseUrl: 'http://localhost/side-projects/blog-posts/manning/Angular Directives in Action/source/'
```

Please change it accordingly. This will be prepended to specs.

SPECS

This is similar to files section in Karma configuration consisting of test files to run. Update `e2e.conf.js` with following:

```
specs: [
  'tests/specs/**/*.e2e.js'
]
```

Please note that we are going to use similar convention for protractor tests as well. We'll use the same name as original file with the suffix `-e2e.js`. And `/**/` targets chapter-wise directories i.e. ch1, ch2, ch3, and so on.

SUITES

Running all the tests often is time consuming and you might want to just test the current file being modified. In such scenario, restricting the tests by chapter would be big time saver. With suites options, we can group related tests in order to run them separately. Update `e2e.conf.js` as:

```
suites: {
  ch1: 'tests/specs/ch01/*.e2e.js'
}
```

Here we basically created a chapter-wise suites to run tests independently if needed. With this, we can run all tests for the first chapter only (excluding the rest) as:

```
./node_modules/protractor/bin/protractor e2e.conf.js --suite ch1
```

CAPABILITIES

This controls a browser on which you want to run tests. By default, it is chrome.

TIP: You may be interested to go through a specification of DesiredCapabilities and their content at <https://code.google.com/p/selenium/wiki/DesiredCapabilities>.

MULTICAPABILITIES

Although the capabilities section is helpful but you might want to run more than one instance of webdriver on the same tests, use this option instead. This takes an array of capabilities. If this is specified, capabilities section will be ignored. For instance, let's add Firefox as a capability so:

```
multiCapabilities: [{
  'browserName': 'chrome'
}, {
  'browserName': 'firefox'
}]
```

Try running the protractor command to see tests running in both the browsers. It also supports safari, opera, and ie.

FRAMEWORK

Protractor for now only supports Jasmine and Cucumber assertion frameworks. We'll stick to Jasmine because even AngularJS uses it for its own tests.

ONPREPARE

This is really useful option to handle more complex operations such as exporting the test report in XML or taking a screenshot of the window to analyze errors. It takes a callback to run when protractor is ready and before the specs are executed. For now, we'll simply resize the browser window as:

```
onPrepare: function() {
  browser.driver.manage().window().setSize(500, 500);
}
```

TIP: It is recommended to go through useful docs added by the Protractor Core Team at <https://github.com/angular/protractor/tree/master/docs> to know more about other options that we have not covered because of space constraint.

With this modifications in place, our testing environment is ready. In the following section, we'll write a simple but real-world directive and test it with Karma as well as Protractor. Let's get started.

Learning to test by Real World example

Imagine your application have a profile page for each user to showcase his/her user information along with a nice profile picture. And it is very likely that the profile picture might fail to load because of some technical server issue which would affect the application layout. As different browsers treat missing images differently, it will probably give a bad impression to your users so its always better to handle errors more gracefully. Let us do that.

Create a `placeholder.html` file in `ch03/` directory as:

```
<html ng-app="PlaceholderApp">
<head>
  <title>Placeholder Directive</title>
  <script type="text/javascript" src="../../bower_components/angular/angular.js"></script>
  <script type="text/javascript" src="../../js/ch03/placeholder.js"></script>
  <link rel="stylesheet" type="text/css" href="../../bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body ng-controller="ProfileCtrl">
```

```



</body>
</html>

```

As you can see, we have two profile images, one that works, and another fails to load. Both are having a special directive applied on them, `onImageLoad` to load the placeholder image if the profile image fails to load.

Then we'll define the directive in `js/ch03/placeholder.js` as:

```

describe('Chapter 3: ', function() {
  beforeEach(module('PlaceholderApp'));

  var element;

  it('Should replace the default image with placeholder If it fails to load', inject(function($rootScope, $compile) {
    // If loads
    element = angular.element('');
    element = $compile(element)($rootScope);
    expect(element[0].getAttribute('src')).toEqual('http://lorempixel.com/100/100/people/1/');

    // If fails, use placeholder image
    element = angular.element('');
    element = $compile(element)($rootScope);
    waitsFor(function() {
      return element[0].getAttribute('src') === 'http://lorempixel.com/100/100/cats/';
    });
    runs(function() {
      expect(element[0].getAttribute('src')).toEqual('http://lorempixel.com/100/100/cats/');
    });
  }));
});

```

The second model named `userImageFail` has a broken url which will definitely fail to load. That will trigger an error callback on the image to replace it with a placeholder image that will eventually fire up a load event in order to make it visible. This trick is used to avoid FOUC (flash of unstyled content) problem when the original image does not load (especially in IE). Though this directive looks quite simple but very useful in real world applications.

Testing with Karma

Now that our directive is ready, it's time to unit test the same to make sure it works as expected. Simply create a new Jasmine test file in `tests/specs/ch03/placeholder-unit.js` as:

```

describe('Chapter 3: ', function() {
  beforeEach(module('PlaceholderApp'));
  var element;

  it('Should replace the default image with placeholder If it fails to load', inject(function($rootScope, $compile) {
    element = angular.element('');
    element = $compile(element)($rootScope);
    expect(element[0].getAttribute('src')).toEqual('http://lorempixel.com/100/100/people/1/');

    element = angular.element('');
    element = $compile(element)($rootScope);
    waitsFor(function() {
      return element[0].getAttribute('src') === 'http://placekitten.com/100/100';
    });
    runs(function() {
      expect(element[0].getAttribute('src')).toEqual('http://placekitten.com/100/100 ');
    });
  }));
});

```

Please note that `waitsFor` is a Jasmine's way to track an asynchronous call and perform an action once it is completed. When we load an invalid image URL, it would take some time to trigger the error callback. As this delay is not consistent, we can not randomly wait for 1 or 2 seconds before the placeholder directive takes an action on it. So `waitsFor` provides a better interface for pausing the control flow until the current assertion has been resolved. As soon as it returns true, Jasmine reaches to `runs()` block, where it validates the `src` attribute of the image.

Now, update the file section in `unit.conf.js` to verify the preceding spec as:

```
files: [
  'bower_components/angular/angular.js',
  'bower_components/angular-mocks/angular-mocks.js',
  'js/ch01/angular-template.js',
  'tests/specs/ch01/*-unit.js'
  'js/ch03/placeholder.js',
  'tests/specs/ch03/*-unit.js'
],
```

Then run Karma on the command line with `karma start unit.conf.js` command and you will notice that the test is passing as expected.

Testing with Protractor

Writing test cases in Protractor is quite similar to unit testing with one difference is that it will be executed in real browsers and the way your users would perceive it.

First, create `placeholder-e2e.js` in `tests/specs/ch03/` directory as:

```
describe('Chapter 3:', function() {
  it('should replace the default image with placeholder If it fails to load', function() {
    browser.get('ch03/placeholder.html');

    // If loads
    expect(element.all(by.tagName('img')).first().getAttribute('src')).toEqual('http://lorempixel.com/100/100/people/1/');

    // If fails, use placeholder image
    var imgElement = element.all(by.tagName('img')).last();
    browser.wait(function() {
      return imgElement.getAttribute('src').then(function(src) {
        return src === 'http://lorempixel.com/100/100/cats/';
      });
    }, 60 * 1000);
    expect(imgElement.getAttribute('src')).toEqual('http://lorempixel.com/100/100/cats/');
  });
});
```

The `browser.wait()` command is quite similar to `waitsFor()` block in Jasmine. It takes a condition to evaluate, timeout in milliseconds to wait for the condition to be true (60 seconds in our case), and an optional message to show if timed out.

To test the same, update the suite option in `e2e.conf.js` as:

```
suites: {
  ch1: 'tests/specs/ch01/*-e2e.js',
  ch3: 'tests/specs/ch03/*-e2e.js'
},
```

And run the following command in the terminal to see it passing.

```
./node_modules/protractor/bin/protractor e2e.conf.js --suite ch3
```

We've just scratched the surface here covering Karma and Protractor, so I will recommend to go through the API documentation to learn more.

Automating tests with “npm run”

So far, we have seen long commands to run unit and End-to-End tests. The `npm run` command is perfectly adequate to trim them down to make it easy to remember and save us few keystrokes every time we run them. You can run following command to see all the available options with npm:

```
npm --help

Usage: npm <command>

where <command> is one of:
  add-user, adduser, apihelp, author, bin, bugs, c, cache,
  completion, config, ddp, dedupe, deprecate, docs, edit,
  explore, faq, find, find-dupes, get, help, help-search,
  home, i, info, init, install, isntall, issues, la, link,
  list, ll, ln, login, ls, outdated, owner, pack, prefix,
  prune, publish, r, rb, rebuild, remove, repo, restart, rm,
  root, run-script, s, se, search, set, show, shrinkwrap,
  star, stars, start, stop, submodule, t, tag, test, tst, un,
  uninstall, unlink, unpublish, unstar, up, update, v,
  version, view, whoami
```

For instance, `npm test` command may trigger test scripts if available, but internally it fires `npm run test` where test is some sort of scripts or process we want to invoke. That means we can extend these options to run customized scripts. If we look at `package.json` created at the beginning of the chapter, you may find following in the scripts section:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
}
```

So, let's update it to create two new scripts for Karma and Protractor as:

```
"scripts": {
  "test": "karma start unit.conf.js --singleRun true && npm run test-e2e",
  "test-unit": "karma start unit.conf.js",
  "test-e2e": "protractor e2e.conf.js"
}
```

First one runs both unit (single run) and End-to-End tests together which is really useful to do a quick test drive. The other two scripts are there to run them separately. You can use them as:

```
npm test
npm run test-unit
npm run test-e2e
```

All these commands basically make their respective scripts little handy to use but essentially do the same thing.

Summary

In this chapter, we have learned about testing units of source code with Jasmine framework and End to End testing of applications in real browsers using Protractor. We have successfully set up a testing environment suitable for this book and to the companion suite but can easily be extended further. We have also automated Unit testing with Karma Test Runner to save our time by running all the tests in the background in no time.

However, writing tests can be challenging and time consuming but *“Minutes now can save Hours later”*. The next chapter will be hands-on directives to extend HTML vocabulary.

Learning to extend HTML with the Directive API

This chapter covers

- Various ways of using directives in HTML
- Difference between `template` and `templateUrl` options
- Benefit of prioritizing execution of multiple directives on element
- Purpose of terminating execution of directives
- Unit and E2E testing of sample directive
- Learn to use a jQuery plugin in AngularJS context

Forget about all HTML elements for a moment! The one element that allured me to atleast try out HTML a decade ago was `<marquee>` element. The reason behind this is that it was the only HTML element animating stuff on and off the page at that time when developers had to use Flash or other technologies to create animation for the web. However, it can only scroll text horizontally or vertically but things have changed since then; now CSS3 and advanced JavaScript capable to do high quality animations seamlessly without any third party plugins.

In this chapter, we'll resurrect the good old marquee element, give it super powers using CSS3 animations, and learn how AngularJS Directives API helps into all this.

Gaining an Insight into directives

Through a long but inspiring journey so far, we have come to a moment where we will actually get our hands dirty with directives API. Considering the fact that we can use directives before Web Components come into existence and widely supported, we are now enough prepared to write our first directive.

In simple words, **AngularJS Directives API** is a way to extend HTML vocabulary. By means of Directives API is that AngularJS has it's own reasoning behind how one should write custom element and it does not adhere to the standards of Web Components for now. But It might cohere in future as the standards evolve so hopefully one day you could use directives along with Web Components.

If you take an example of jQuery and realized how it was built out of necessity but after so many years even though all major browsers have implemented complex DOM querying using CSS3 or JavaScript selectors as a part of a Selectors API natively, developers are still happy to use jQuery and they will not go away from it anytime soon. The reason behind the wide adoption over native implementation is that jQuery did not just solve one specific problem, instead it created the ecosystem around it in terms of Plug-ins, UI frameworks, and handling cross-browser quirks without burdening developers. AngularJS is on the same path to make it a rudimentary choice for developers when it comes to build web applications and extend HTML. At the time of writing this chapter, few working examples of AngularJS data bindings with polymer had also emerged that can give us a clear sign where it is heading.

To begin with, we'll leverage `<marquee>` tag initially as a scroller to move text to avoid getting overwhelmed with CSS3 animations in the beginning but add them to move words up and down to create a wave-like effect by eliminating the marquee tag altogether at the end of the chapter.

We'll call it `<super-marquee>`. Let us first create an HTML file named *restrict.html* in *ch04/* directory.

```
<html ng-app="MarqueeApp">
  <head>
    <script type="text/javascript" src="../../bower_components/angular/angular.js"></script>
    <script type="text/javascript" src="../../js/ch04/restrict.js"></script>
  </head>
  <body>
    <div super-marquee>
```

```
</body>
</html>
```

For the record, the `super-marquee` element will have multiple words wrapped in `span` tags to float around individually using CSS3 keyframe animations that we'll just change `margin-top` from `0px` to `100px` in order to move them up and down, alternatively. To set up the base animation first, add following into the `head` tag:

```
<style>
  @-webkit-keyframes animate { to { margin-top: 100px; } }
  @-moz-keyframes animate { to { margin-top: 100px; } }
  @-ms-keyframes animate { to { margin-top: 100px; } }
  @keyframes animate { to { margin-top: 100px; } }

  div.wave > span {
    float: left;
    text-align: center;
    padding-right: 40px;
    -webkit-animation: animate 1.5s ease-in alternate infinite;
    -moz-animation: animate 1.5s ease-in alternate infinite;
    -ms-animation: animate 1.5s ease-in alternate infinite;
    animation: animate 1.5s ease-in alternate infinite;
  }
</style>
```

This will move all the texts vertically but we need an alternate movement to create a wave-like motion so let's use varying delay as follows. Add following in the above style tag as:

```
div.wave > span:nth-child(1) { }

div.wave > span:nth-child(2) {
  -webkit-animation-delay: .4s;
  -moz-animation-delay: .4s;
  -ms-animation-delay: .4s;
  animation-delay: .4s;
}

div.wave > span:nth-child(3) {
  -webkit-animation-delay: .8s;
  -moz-animation-delay: .8s;
  -ms-animation-delay: .8s;
  animation-delay: .8s;
}

div.wave > span:nth-child(4) {
  -webkit-animation-delay: 1.2s;
  -moz-animation-delay: 1.2s;
  -ms-animation-delay: 1.2s;
  animation-delay: 1.2s;
}
```

This will move all the words at the same time but with different speed that will create a wave-like effect. Now that we have setup the base for the example to work, let's begin to play with Directives API.

Restricting the Usage to enforce HTML semantics

The most obvious application of directives API is defining new HTML vocabulary. The directives API is so thorough that it allows us to define specific directive declaration style to suite our need. It enables us to use directives as attributes, elements, CSS classes, and HTML comments in the DOM.

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



```
<!-- directive: angular -->
```

For a huge name such as following, you can split the words using hyphens while declaring and camelcase it while defining a directive so:

```
<div the-quick-brown-fox-jumps-over-the-lazy-dog></div>
```

```
App.directive('theQuickBrownFoxJumpsOverTheLazyDog', function() { ... });
```

Pay attention to **the directive's definition in JavaScript** and **declaration in HTML** as both are different. In order to maintain any directive syntactically valid, we should not declare our directive as `theQuickBrownFoxJumpsOverTheLazyDog` in the DOM. However, it may be correctly rendered by browsers (not by AngularJS) but will not be validated by HTML/XHTML validators. The problem is that different browsers (or even different versions of the same browser) will make different guesses about the same illegal construct; worse, if your HTML is really pathological, the browser could get hopelessly confused and produce a mangled mess, or even crash. In contrast, the directive definition returns a function and one can not use hyphenated function name in JavaScript which prevents us from using `the-quick-brown-fox-jumps-over-the-lazy-dog` while defining the same.

NOTE: Always use hyphen while declaring a directive in HTML and camelCase while defining it in JavaScript. This is the biggest point of confusion for beginners and might be time-saver for you. In case you are interested to know how does HTML/XHTML validation work, read the FAQ (<http://validator.w3.org/docs/help.html>) maintained by W3C.

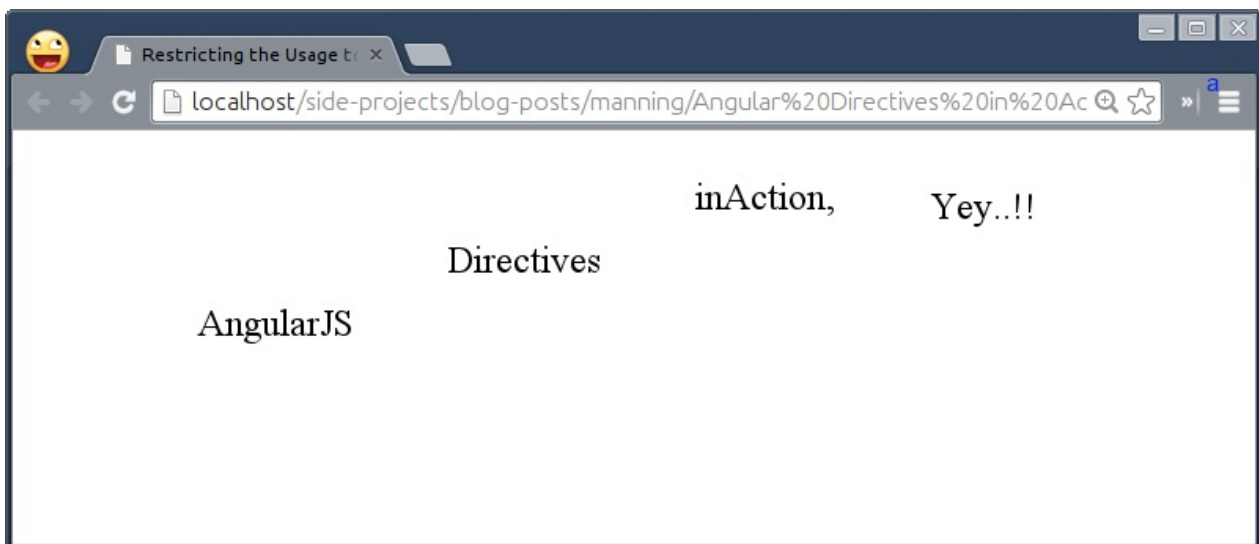
Moving on we'd applied a custom attribute on a div element earlier so let us write the directive for the same in *js/ch04/restrict.js* as follows:

```
var App = angular.module('MarqueeApp', []);

App.directive('superMarquee', function() {
  return {
    compile: function(element) {
      var scrollingText = '<marquee>\n
        <div class="wave">\n
          <span>AngularJS </span>\n
          <span>Directives </span>\n
          <span>inAction, </span>\n
          <span>Yey...!!</span>\n
        </div>\n
      </marquee>';

      element.html(scrollingText);
    }
  };
});
```

Do not worry about compile function for now as it will be covered in detail in later chapters. For now, consider it as a wrapper to put up a logic for the directive. Here we've just a string with each word wrapped in `span` tags which ultimately go within a `<marquee>` tag. Later we insert the markup in the element itself using a jQuery `.html()` method. With this, our directive must start to roll text in the browser as shown:



That's just an HTML attribute but let us look at other ways of using the `restrict` option and their pros and cons.

HTML ELEMENT

Showing off an element of your own name is more exciting than anything else. With `restrict` as an element option in directives, you can actually extend the existing HTML tags offering by making the markups more meaningful. You can add following option in order to use `superMarquee` directive as an element:

```
App.directive('superMarquee', function() {
  return {
    restrict: 'E',
    link: function() { ... }
  }
});
```

And then use it in the DOM so:

```
<super-marquee></super-marquee>
```

Note that the Internet Explorer versions prior to 9 do not go well with custom elements so you need to take care of that with additional prescriptions.

TIP: Please visit <http://docs.angularjs.org/guide/ie> to take necessary majors in order to run your application smoothly on older IE.

NOTE: Please note that AngularJS 1.2 will continue to support IE8, but the core team will not address any pending issues specific to IE8 or earlier. Also, AngularJS 1.3 is dropping support for IE8.

Here are the pros and cons to decide when to use the option.

Pros:

- Beneficial for main directive (I prefer to call it a King Directive) that can encompass other directives to make up the whole functionality. For example, an accordion widget that we saw in the first chapter which had a main directive, `accordion`, as follows:

```
<accordion>
  <accordion-group heading="Accordion Header #1">Accordion Body </accordion-group>
  <accordion-group heading="Accordion Header #2">Accordion Body </accordion-group>
```

```
</accordion>
```

- More expressive. For example, `<tabs><tab>Tab 1</tab><tab>Tab 2</tab></tabs>` is more readable than `<div tabs><div>Tab 1</div><div>Tab 2</div></div>` from developer's perspective.

Cons:

- Can not use multiple element level directives together and hence need to rely on Class or Attribute level directives to extend behavior. For example, we need use `enable` attribute to toggle the tab's state as: `<tabs><tab enable='false'></tab></tabs>`.
- Require extra efforts to handle older versions of Internet Explorer.
- Can not be validated against HTML/XHTML validators.

AngularJS supports this declaration style for many built-in directives such as `form`, `input`, `ngInclude`, `ngSwitch`, and so on.

HTML ATTRIBUTE

To avoid adding unnecessary complexity to support IE8 or older, you can restrict the directive as an attribute which is a default option and hence if you had noticed that we did not specify it for `superMarquee` directive earlier. This is the most common option used in many open source AngularJS applications, libraries, and widgets. You can specify it as `restrict:'A'` in the directive definition. Similar to Element level directives, this also has few advantages without any disadvantages.

Pros:

- Do not have to rely on other directives to pass in configurations to alter the behavior. For example, `<div tab>Tab 1</div>` can be easily extended as `<div tab='{enable: false}'>Tab 1</div>`.
- Can be used with Element and Class level directives.
- Works in all the browsers (even older I.E.) with ease.
- Can be validated against HTML/XHTML validators with `data-` prefix.

AngularJS uses this declaration style as well for many built-in directives like `form`, `ngIf`, `ngClass`, `ngInclude`, and `ngSwitch`.

CSS CLASS

You can also defined the directive as a CSS Class using `restrict:'C'` option. There is no added advantage over element and attribute levels, instead, it is recommended to use E or A option over C.

Pros:

- Can be used with element and attribute level directives to attach behaviors.
- Sometimes it's possible to utilize CSS classes than applying extraneous attributes if they justify the purpose of the directive. The best example is the dropdown menu functionality of AngularUI Bootstrap where simply adding a class (`.dropdown-toggle`) activates the menu (`.dropdown-menu`) as shown:

```
<span class="dropdown">
  <a class="dropdown-toggle">My Dropdown Menu</a>
  <ul class="dropdown-menu">
    <li>One</li>
    <li>Two</li>
  </ul>
</span>
```

Cons:

- Class level directives may be hard to differentiate from the actual CSS classes in the DOM so use sparingly.

HTML COMMENT

Directives as HTML comments do not make sense in the first place but they are really handy in places where the DOM APIs limit the ability to create directives that spanned multiple elements such as `<table>`, ``, ``, and so on. If you try to use any of the restrict options that we learned so far within `<table>` element then browser will automatically take it out while rendering, so the following approach would fail.

```
<table>
  <tr><td>SuperMarquee Live Demo</td></tr>
  <div super-marquee></div>
</table>
```

But if you define the directive as an HTML comment and replace it with scrollingText as:

```
App.directive('superMarquee', function() {
  return {
    restrict: 'M',
    compile: function(element) {
      var scrollingText = '<marquee>\n
        <div class="wave">\n
          <span>AngularJS </span>\n
          <span>Directives </span>\n
          <span>inAction, </span>\n
          <span>Yey...!!</span>\n
        </div>\n
      </marquee>';

      // If element is a comment
      if (element[0].nodeType === 8) {
        element.replaceWith('<tr><td>' + scrollingText + '</td></tr>');
      } else {
        element.html(scrollingText);
      }
    }
  }
});
```

And update the table so:

```
<table>
  <tr><td>SuperMarquee Live Demo</td></tr>
  <!-- directive: super-marquee -->
</table>
```

Then this will load `super-marquee` within the table itself as shown in the following figure.



Now comment level directives do not have pros except the one we just saw with `<table>` but has many cons as given below:

- Can not be used with element, attribute, and class level directives.
- Can not use multiple comment level directives together to extend behavior.
- Extra precaution to be taken to avoid removing HTML comments during a `build` process via grunt, gulp, and so on.

So the important things to note that:

- We can use all the options together in any order while defining a directive with `restrict: 'EACM'` option and free to use any declaration style in HTML.
- Use element or attribute level options often. As AngularJS 1.3 has dropped IE8 support so `restrict: 'EA'` is the default restrict option now.
- Restrict to attribute or class in case of adding just a behavior without any configuration. For example, the `ngIf` directive that can only be used as an attribute because it takes an expression to add or remove the corresponding element when the expression evaluated to `true` or `false` respectively.
- Avoid using comment level directive as AngularJS 1.2 introduces `directive-start` and `directive-end` as a better solution to the problem that could be solved earlier by the comment level directive. So that we can use `super-marquee-start` and `super-marquee-end` directives as an alternative to the comment level `superMarquee` directive.

Now that we know how to define directives, let us find out how to add content into the directive in the next section.

Feeding the minimal Template inline

We saw earlier that the marquee text was stored in a variable named `scrollingText` which we were injecting into the DOM conditionally. That's what exactly `template` option is for. The `template` option can consist of HTML markups that will be produced when the directive is executed by AngularJS compiler. It can also hold other directives or AngularJS data binding expressions. Let's update our directive in `restrict.js` as:

```
App.directive('superMarquee', function() {
  return {
    restrict: 'EACM',
    template: '<marquee>\n
      <div class="wave">\n
        <span>AngularJS </span>\n
        <span>Directives </span>\n
        <span>inAction, </span>\n
        <span>Yey..!!</span>\n
      </div>\n
    </marquee>'
```

```
};
});
```

As you can see, we've just moved the content from `compile` to `template` so all types of directives will still work except the comment one. That is because when the directive is processed, the template is inserted into the element on which the directive was declared and DOM APIs do not allow to inject the markups inside the HTML comment.

Is it a bug in AngularJS directives API or DOM API? Nope, It's not a bug at all from either side because that's how DOM APIs work but AngularJS let's you fix this problem with `replace` option which we'll see later.

You can even pass a function in the `template` option that should return a string as a template. Note that you get access to `element` and `attrs` to do some activities or perform actions on the element or the template. So here we are just grabbing the text from the `data` attribute and injecting it inside the `div` as shown.

```
template: function(element, attrs) {
  return '<marquee><div class="wave">' + attrs.text + '</div></marquee>';
}
```

And then we can update the declaration to change the default text in the DOM itself as:



Although it looks weird to add `span` tags in a string as an attribute that could be added as the element's content and will be fixed with `transclusion` in Chapter 6. For now, we'll live with it.

Note that the template is:

- Useful to replace the directive's content or the directive itself.
- Expecting a single root element in the template. For example ' `<div>one</div><div>two</div>` ' is an invalid template which should be used as ' `<div>one <div>two</div></div>` '.
- Compiled after getting injected so that bindings are evaluated and child directives are applied with the correct scope.
- Convenient to use if the markup is short and readable.

But what if the template is too complex? Let us find out in the next section.

Feeding an external template using templateUrl

It's daunting to update the template in JavaScript and will also be difficult to scale later, so it would be great if we put it inside a separate HTML file and read it via XHR/AJAX. The `templateUrl` option is our savior for the same.

Here we basically move the entire markup into a separate HTML file named `templateUrl-partial.html` and AngularJS internally initiates an XHR request to read the file content asynchronously, stores it in `$templateCache` for later use, and finally injects it into the element.

Additionally, if you do wish to avoid XHR request altogether (preventing latency) to quickly load the template, you can wrap the template in pair of `<script>` tags somewhere in the body tag in `restrict.html` as follows:

```
<script type="text/ng-template" id="templateUrl-partial.html">
  <marquee>
    <div class="wave">
      <span>AngularJS </span>
      <span>Directives </span>
    </div>
  </marquee>
```

```

    <span>inAction, </span>
    <span>Yey..!!</span>
  </div>
</marquee>
</script>

```

Please note that the `type` attribute should have `text/ng-template` specified and the `id` attribute is a must as it will be used as a `templateUrl` instead. The template will be cached by the `$templateCache` service (but not compiled) when AngularJS bootstraps. The `$templateCache` service is a very simple service similar to HTML5 `localStorage` that takes key and value as a template to cache but only persisted during the life-cycle of the application. That means reloading a page will result in clearing the `$templateCache` container.

Now replace the directive definition in `restrict.js` with:

```

App.directive('superMarquee', function() {
  return {
    restrict: 'EACM',
    templateUrl: 'templateUrl-partial.html'
  };
});

```

When the directive is compiled, the `templateUrl` will be used as a key to look up in the `$templateCache` store to find the respective template to load from the cache. Otherwise it will trigger an XHR request to load the same physically.

In case you want to avoid the template in a `<script>` tag or in an external file and even do not want to bloat the directive definition with the template option, you may use `$templateCache` store to cache the template manually as shown:

```

App.run(function($templateCache) {
  $templateCache.put('templateUrl-partial.html', 'Isolated but Cached');
});

```

This approach has been used by AngularJS UI library to keep all the templates used by various widgets at one place. Other benefits are:

- Allows us to override the existing template of any widget with an external template (using above approach) in order to extend it without touching the directive. Although, this is not possible with the template option.
- Enables lazy loading of the template so that the child directives/expressions will be evaluated only when the directive gets compiled. This keeps the template inert reducing active `$watchers` (that may cause performance issues and will be discussed in Chapter 7) if the directive is not activated yet.

Now that the `superMarquee` directive is rising and shining but still can not be used as a comment with HTML table because of the `template` and `templateUrl` options. Let us fix that in the next section.

Replacing the associated element

By default the template is injected into the directive but sometimes the base element on which the directive is applied needs to be replaced by the template itself. This option specifies where the template should be inserted. The default option value is `false`. Otherwise, it will replace itself with the template.

In earlier section, we noticed that after using `template` and `templateUrl` options, the comment level directive stopped working because of the DOM limitation to update the HTML comment with the template. You can now use `replace:true` option to obviate that glitch. Let's update our directive definition to specify the option as:

```

App.directive('superMarquee', function() {

```

```
return {
  restrict: 'EACM',
  replace: true,
  templateUrl: 'templateUrl-partial.html'
};
});
```

Things to note that:

- Can only be used with `template` or `templateUrl` option.
- All the attributes exist on the directive element are copied and applied on the root element of the template before replacing, so all DOM events or sibling directives will work as expected.
- It is very rarely used option and will be removed (currently deprecated in AngularJS 1.3) in the major release of AngularJS 2.0 because this has difficult semantics (e.g. how attributes are merged with the root element of the template) and leads to more problems compared to what it solves. Also, with Web Components, it is normal to have custom elements in the DOM.

What happens when we use multiple interdependent directives and how their order of execution affects the functionality? Well, let's see that in the next section.

Prioritizing invocation irrespective of placements

When there are multiple directives defined on an element and especially one directive is dependent on other, sometimes it is necessary to specify the order in which the directives are compiled. We, developers, are smart enough to mess up with the placement of the directives and break a thing or two. Would not it be great to automate this by making directives smarter to invoke in a specific order without worrying about how they were declared in HTML? The answer to that is a `priority` option. AngularJS compiler sorts directives by priority before applying them on elements.

With reference to the priority option, each directive can have one of the two phases i.e. Compile or Link (will be covered in Chapter 6 in more detail). The compile phase (`compileFn`) is useful if you want an element to be processed (or perform any operation on it) before being injected into the document. Whereas a link phase (`linkFn`) is there to link the element with a scope associated. You can override either of it. In fact, the `compileFn` returns the `linkFn` . So it's the compile phase not the link phase that gets prioritized with the priority option so directives with higher numerical priority are compiled first but linked or processed last.

Some of the built-in directives use priorities to dictate the order of execution when used in conjunction with others and override either `compileFn` or `linkFn` depending on their purposes. For example, the `ngRepeat` directive holds a higher priority (that is 1000) than other directives such as `ngInclude`(400), `ngInit`(450), `ngController`(500), `ngIf`(600), `ngSwitch`(800) and so on. Suppose we use `ngInclude` directive along with `ngIf` on the same element as shown:

```
<div ng-include="'do-not-load-me.html'" ng-if="false"></div>
```

For those who are not familiar with `ngInclude` directive, it is basically used to fetch, compile, and include an external HTML fragment via XHR/AJAX request whereas `ngIf` directive removes or recreates a portion of the DOM tree based on an expression. So because of the higher priority the XHR call to fetch `do-not-load-me.html` will be prevented by `ngIf` that removes the respective element before the call is made. This makes more sense instead of loading the fragment first by `ngInclude` and then removing it later with `ngIf`. Same is applicable to `ngController` that has less priority than `ngIf`.

Similarly the `ngInit` directive allows us to evaluate an expression in a current scope from within the DOM itself which needs to be processed after `ngController` directive (if used on the same element) which creates a new scope, so that the values will be initialized (by `ngInit`) on the controller's scope not the parent scope or `rootScope`.

Going forward with `superMarquee`, we've hard-coded the text in `span` tags in the template, so let's add them dynamically

that will make it easy to change the scrolling text later. The template may have text with or without span tags. We'll write a new directive named `looper` that will wrap each word in a `span` tag as follows. Add following in *restrict.js* as:

```
App.directive('superMarquee', function() {
  return {
    restrict: 'EACM',
    replace: true,
    template: '<marquee><div class="wave">AngularJS Directives inAction, Yey...!!</div></marquee>',
    compile: function() {

    }
  };
});

App.directive('looper', function() {
  return {
    restrict: 'AC',
    compile: function(element) {
      if (element.find('span').length) return;

      var DOM = '';

      element
        .text()
        .split(' ')
        .forEach(function(word) {
          if (word !== '') {
            DOM+= '<span>' + word + '</span>';
          }
        });

      element.children().html(DOM);
    }
  };
});
```

It's time to try both directives in the DOM, let's update *restrict.html* so:

```
<div super-marquee looper></div>
```

Unfortunately, this will not work as intended as `super-marquee` will execute after `looper`. But, Why?? Because the default priority for any directive is 0, if not specified. When multiple directives on the same element have the same priority, they will be sorted by `name` in ascending order i.e. a directive named `apple` will be processed before `google`. So if you rename the directive's name from `looper` to `wrap`, our example will work as expected. Give it a try by changing it in both *restrict.js* and *restrict.html*. However, this approach is not reliable so we'll set a proper priority without renaming any of them.

Now set a `priority: 1` to `looper` and `priority: 2` to `superMarquee` and try re-ordering the directives in the DOM so:

```
App.directive('superMarquee', function() {
  return {
    restrict: 'EACM',
    replace: true,
    priority: 2,
    template: '<marquee><div class="wave">AngularJS Directives inAction, Yey...!!</div></marquee>',
    compile: function() {

    }
  };
});

App.directive('looper', function() {
  return {
    restrict: 'AC',
    priority: 1,
    compile: function(element) {
      if (element.find('span').length) return;

    }
  };
});
```

```

var DOM = '';

element
  .text()
  .split(' ')
  .forEach(function(word) {
    if (word !== '') {
      DOM+= '<span>' + word + '</span>';
    }
  });

element.children().html(DOM);
}
});
});

```

As both directives have compile methods, the `superMarquee` directive will be compiled before `looper`.

NOTE: If you replace the compile methods with link (empty) methods (and log their ordering in console) for both the directives then you'll notice that `looper`'s link method will be called after `superMarquee`'s link method. The reason for reversing the priority order of Link methods will be covered in Chapter 6 as it is a bit overwhelming to be discussed in here.

Now, you will see it working irrespective of directives' placements as shown in the following figure:



So the takeaways are:

- The default priority is always 0 for any directive, if not set.
- The priority can be negative as well.
- The priority is used to sort the directives before their compile functions are called.
- The directives with higher numerical priority are compiled first but linked last. However, it is reversed for element transcluded directives (which enable inclusion of the element on which the directives were applied that will be covered in Chapter 6) such as `ngIf`, `ngRepeat`, and so on wherein the directives with lower numerical priority are compiled first but linked last because the compiler only transcludes directives with low priority (those are not yet linked or processed) than the current directive. So the processing of element transcluded directives happen from high (600 - `ngIf`) to low (400 - `ngInclude`), whereas for normal directives, it goes from low (1 - `looper`) to high (2 - `superMarquee`).
- Multiple directives on the same element with the same priority will be sorted by name in ascending order.
- Multiple directives on the same element with the same name as well as priority will be sorted by their placements in the DOM.
- Always consider Compile and Link phases before setting the priority on any directive.
- Finally, It is very rarely used but can be helpful only if a directive is manipulating the DOM or creating a new scope and thereby affecting other directives such as `ngIf/ngInclude` or `ngInit/ngController` example that we saw earlier.

As the priority option is pretty important to dictate the order in which directives are compiled, sometimes it's essential to stop the compilation of directives further to avoid any clash but how?

Terminating invocation to stop further compilation

A `terminal` option is often used along with the `priority` option to prevent further compilation of the directives on an element. If `terminal: true` is set then the current priority will be the last set of directives which will execute, others are ignored without getting compiled.

The `looper` directive does not continue wrapping words into span tags if they exist. But we do not validate if total number of span tags equal total number of words to allow continuation. So let's create a new directive named `validate` to remove existing `span` tags if not matched and pave the way for `looper`. Although we could have this logic in the `looper` directive itself but we'll have a separate directive to do that for the sake of separation of concerns. Add following in *restrict.js*:

```
App.directive('validate', function() {
  return {
    restrict: 'AC',
    priority: 2,
    compile: function(element) {
      var $span = element.find('span'),
          totalSpans = $span.length,
          totalWords = element.text().split(' ').length;

      if (totalSpans !== totalWords) {
        for (var i = 0; i < totalSpans; i++) {
          angular.element($span[i]).replaceWith($span[i].textContent + '&nbsp;');
        }
      }
    }
  };
});
```

Note that we're using `priority: 2` here because we want it to be compiled before `looper` but after `superMarquee` and hence update the priority of `superMarquee` to 3 as:

```
App.directive('superMarquee', function() {
  return {
    restrict: 'EACM',
    replace: true,
    priority: 3,
    template: '<marquee><div class="wave">AngularJS Directives inAction, Yey...!!</div></marquee>',
    compile: function() {

    }
  };
});
```

With this change, the `looper` directive will always run last and any new directive has to have either 1 or higher priority. Update the `looper` directive definition in *restrict.js* as:

```
App.directive('looper', function() {
  return {
    restrict: 'AC',
    priority: 1,
    terminal: true,
    compile: function(element) {
      if (element.find('span').length) return;

      var DOM = '';

      element
        .text()
```

```

        .split(' ')
        .forEach(function(word) {
            if (word !== '') {
                DOM+= '<span>' + word + '</span>';
            }
        });

        element.children().html(DOM);
    }
};
});

```

Please note that the terminal option is always used along with priority and compile functions. That is because if we use linking functions for `superMarquee`, `validate`, and `looper` with priority 1, 2, and 3 respectively then they will run top to bottom unlike compile functions and the order of compilation will be like `superMarquee`, `validate`, and then `looper`. So if we use terminal option on `looper` then both `superMarquee` and `validate` directives will not be compiled because of low priorities, and setting terminal option on `superMarquee` does not make sense because it's `looper` not `superMarquee` should be executed at the end stopping further compilation.

The reason why the terminal option exists is:

- To work with directives that use element transclusion such as `ngIf`, `ngRepeat`, and `ngInclude` where all directives with low priority should be compiled before transclusion happens.
- To restrict using it (with some priority) in conjunction with specific directives only. For instance, `ngNonBindable` directive is used to ignore AngularJS bindings and does not expect to be used with other built-in directives except `ngRepeat` and thus has the same priority as that of `ngRepeat` which is 1000. Custom directives need to have higher priority to work well with it.
- To prevent using it with other directives at all by setting `priority: 0` and `terminal: true`. This approach has been used with built-in `script`, `style`, and `ngOptions` directives that prevent us from applying other directives.

Note that if multiple directives with terminal option are placed on the same element, the higher priority will be used to terminate the execution further. Now that we got the hang of the directive definition object as well as few options of it, its time to upgrade the `superMarquee` directive in the next section.

Making of Super Marquee

As of now our directive relies on the built-in `<marquee>` tag to roll the text which is quite uneasy (even looks poor marquee) so let's throw it away to leverage CSS3 keyframe animations to make it really superb. Look at the new way to use `superMarquee` in the DOM. Create `super-marquee.html` in `ch04/` directory as:

```

<html ng-app="MarqueeApp">
<head>
  <title>Final Extended superMarquee</title>
  <script type="text/javascript" src="../../bower_components/jquery/jquery.js"></script>
  <script type="text/javascript" src="../../bower_components/angular/angular.js"></script>
  <script type="text/javascript" src="../../js/ch04/super-marquee.js"></script>
</head>
<body>
  <super-marquee loop= validate data-text="AngularJS Directives inAction, Yey..!!" data-scrolldelay="10" data-direction="left">
</body>
</html>

```

Please note that we have used `data-*` attributes to distinguish them from custom directives such as `validate` and `looper`. The `data-text` attribute holds the actual text to be rolled, `data-scrolldelay` lets you control the speed of the rolling text in seconds, `data-direction` allows you to set the direction of the rolling text i.e. left/right/up/down, and finally `data-wavedelay` controls the wave movement in milliseconds.

To adjust with the DOM level changes, we need to give our directives an overhaul. Add following in *super-marquee.js* in *js/ch04/* directory as:

```
App.directive('superMarquee', function() {
  return {
    restrict: 'EACM',
    replace: true,
    priority: 3,
    template: function(element, attrs) {
      return '<div class="wave-wrapper"><div class="wave">' + attrs.text + '</div></div>'
    },
    compile: function(element, attrs) {
      if (!attrs.direction || attrs.direction === 'up' || attrs.direction === 'down') {
        element.css('text-align', 'center');
      }

      element.children().css({
        '-webkit-animation': attrs.direction + ' ' + attrs.scrollldelay + 's linear infinite',
        '-moz-animation': attrs.direction + ' ' + attrs.scrollldelay + 's linear infinite',
        '-ms-animation': attrs.direction + ' ' + attrs.scrollldelay + 's linear infinite',
        'animation': attrs.direction + ' ' + attrs.scrollldelay + 's linear infinite'
      });
    }
  };
});
```

Now that we have replaced `<marquee>` tag with `div.wave-wrapper` which is a container to roll the text within, let us set the animation direction, and speed of the rolling text.

Update `looper` directive in *super-marquee.js* as:

```
App.directive('looper', function() {
  return {
    restrict: 'AC',
    priority: 1,
    terminal: true,
    compile: function(element, attrs) {
      var $target = element.children(),
          $span = element.find('span'),
          chars = element.text().split(''),
          totalSpans = $span.length,
          totalChars = chars.length,
          wavedelay = 0,
          DOM = '';

      if (totalSpans === totalChars) {
        angular.forEach($span, function(span) {
          wavedelay += parseInt(attrs.wavedelay, false) || 0.015;
          angular.element(span).css({
            'display' : 'inline-block',
            '-webkit-animation' : 'bump 0.3s ease-in ' + wavedelay + 's alternate infinite',
            '-moz-animation' : 'bump 0.3s ease-in ' + wavedelay + 's alternate infinite',
            '-ms-animation' : 'bump 0.3s ease-in ' + wavedelay + 's alternate infinite',
            'animation' : 'bump 0.3s ease-in ' + wavedelay + 's alternate infinite'
          });
        });
      } else {
        $target.html('');
        chars.forEach(function(char, index) {
          if (char === ' ') {
            $target.append(' ');
          } else {
            wavedelay += parseInt(attrs.wavedelay, false) || 0.015;
            $target.append(
              angular.element('<span/>').html(char).css({
                'display' : 'inline-block',
                '-webkit-animation' : 'bump 0.3s ease-in ' + wavedelay + 's alternate infinite',
                '-moz-animation' : 'bump 0.3s ease-in ' + wavedelay + 's alternate infinite',
                '-ms-animation' : 'bump 0.3s ease-in ' + wavedelay + 's alternate infinite',
                'animation' : 'bump 0.3s ease-in ' + wavedelay + 's alternate infinite'
              })
            );
          }
        });
      }
    }
  };
});
```

```

    });
  }
});
}
};
});

```

This code looks a bit scary but it is not. Instead of wrapping words in `span` tags, we just wrap each character to improve the wave movement. Then we are applying some CSS animations to bump each character by 20px but with varying delay using `wavedelay`.

Next, update the `validate` directive with minor change in *super-marquee.js* as:

```

App.directive('validate', function() {
  return {
    restrict: 'AC',
    priority: 2,
    compile: function(element) {
      var $span = element.find('span'),
          totalSpans = $span.length,
          totalChars = element.text().replace(/\s/g, '').split('').length;

      if (totalSpans !== totalChars) {
        for (var i = 0; i < totalSpans; i++) {
          angular.element($span[i]).replaceWith($span[i].textContent + '&nbsp;');
        }
      }
    }
  };
});

```

Here, we are just comparing total number of `span` tags with total number of characters to remove them if not matched and let `looper` directive to take action. That's All.

Finally add some CSS in *super-marquee.html* so:

```

div.wave-wrapper {
  width: 500px;
  height: 100px;
  line-height: 100px;
  overflow: hidden;
}
div.wave {
  white-space: nowrap;
}
@-webkit-keyframes right {
  0% { -webkit-transform: translateX(500px); }
  100% { -webkit-transform: translateX(-250px); }
}
@-ms-keyframes right {
  0% { -ms-transform: translateX(500px); }
  100% { -ms-transform: translateX(-250px); }
}
@keyframes right {
  0% { transform: translateX(500px); }
  100% { transform: translateX(-250px); }
}
@-webkit-keyframes left {
  0% { -webkit-transform: translateX(-250px); }
  100% { -webkit-transform: translateX(500px); }
}
@-ms-keyframes left {
  0% { -ms-transform: translateX(-250px); }
  100% { -ms-transform: translateX(500px); }
}
@keyframes left {
  0% { transform: translateX(-250px); }
}

```

```

    100% { transform: translateX(500px); }
  }
  @-webkit-keyframes down {
    0% { -webkit-transform: translateY(-50px); }
    100% { -webkit-transform: translateY(100px); }
  }
  @-ms-keyframes down {
    0% { -ms-transform: translateY(-50px); }
    100% { -ms-transform: translateY(100px); }
  }
  @keyframes down {
    0% { transform: translateY(-50px); }
    100% { transform: translateY(100px); }
  }
  @-webkit-keyframes up {
    0% { -webkit-transform: translateY(100px); }
    100% { -webkit-transform: translateY(-50px); }
  }
  @-moz-keyframes up {
    0% { -moz-transform: translateY(100px); }
    100% { -moz-transform: translateY(-50px); }
  }
  @-ms-keyframes up {
    0% { -ms-transform: translateY(100px); }
    100% { -ms-transform: translateY(-50px); }
  }
  @-webkit-keyframes bump {
    from { -webkit-transform: translateY(-10px); }
    to { -webkit-transform: translateY(10px); }
  }
  @-ms-keyframes bump {
    from { -ms-transform: translateY(-10px); }
    to { -ms-transform: translateY(10px); }
  }
  @keyframes bump {
    from { transform: translateY(-10px); }
    to { transform: translateY(10px); }
  }
}

```

You can customize the CSS to adjust the superMarquee's container size, wave movement, and character bump to suit your needs. Here is your little bumpy `superMarquee` is ready.



Go customize the CSS keyframes to make it even wave-er.

Testing of Super Marquee

As decided in the first chapter, we'll stick to the promise to write test cases to make sure things work fine in various browsers. First setup a base for Jasmine in `tests/spec/ch04/super-marquee-unit.js` as:

```
describe('Chapter 4: ', function() {
  beforeEach(module('MarqueeApp'));

  var element;

  // your tests go here
});
```

We already saw the purpose of describe block in the first chapter, so I'll skip it here.

Unit Testing of superMarquee with Jasmine

Let's check if element level superMarquee works or not – simply replace the comment (above) with following snippet as:

```
it('Should compile super-marquee directive as an Element', inject(function($rootScope, $compile) {
  element = angular.element('<super-marquee loop validate data-text="AngularJS Directives inAction, Yey..!!" data-scr
  element = $compile(element)($rootScope);

  expect(element.text()).toBe('AngularJS Directives inAction, Yey..!!');
}));
```

Here we are expecting the value of `data-text` to match with the content of the element. Similarly you can test Attribute, Class, and Comment level directives so I'll leave that to you but will be included in the source code. Then we'll check if the validate directive removes `span` tags conditionally, so let's test that as well so:

```
it('Should validate marquee text If SPANs count do not match with number of chars', inject(function($rootScope, $compile) {
  // If not matched, remove SPANs
  element = angular.element('<div validate><span>A</span> B</div>');
  element = $compile(element)($rootScope);

  expect(element.text()).toBe('A B');
  expect(element.find('span').length).toBe(0);

  // If matched, do not remove SPAN
  element = angular.element('<div validate><span>A </span><span>B</span></div>');
  element = $compile(element)($rootScope);

  expect(element.text()).toBe('A B');
  expect(element.find('span').length).toBe(2);
}));
```

At the end, test the `loop` directive as:

```
it('Should loop through characters to wrap them in SPANs If not available', inject(function($rootScope, $compile) {
  // If text only
  element = angular.element('<super-marquee loop validate data-text="A B" data-scrollldelay="2" data-direction="up"
  element = $compile(element)($rootScope);

  expect(element.text()).toBe('A B');
  expect(element.find('span').length).toBe(2);

  // If empty char is wrapped in SPAN
  element = angular.element('<super-marquee loop validate data-text="<span>A</span><span> </span><span>B</span>" data-direction="up"
  element = $compile(element)($rootScope);

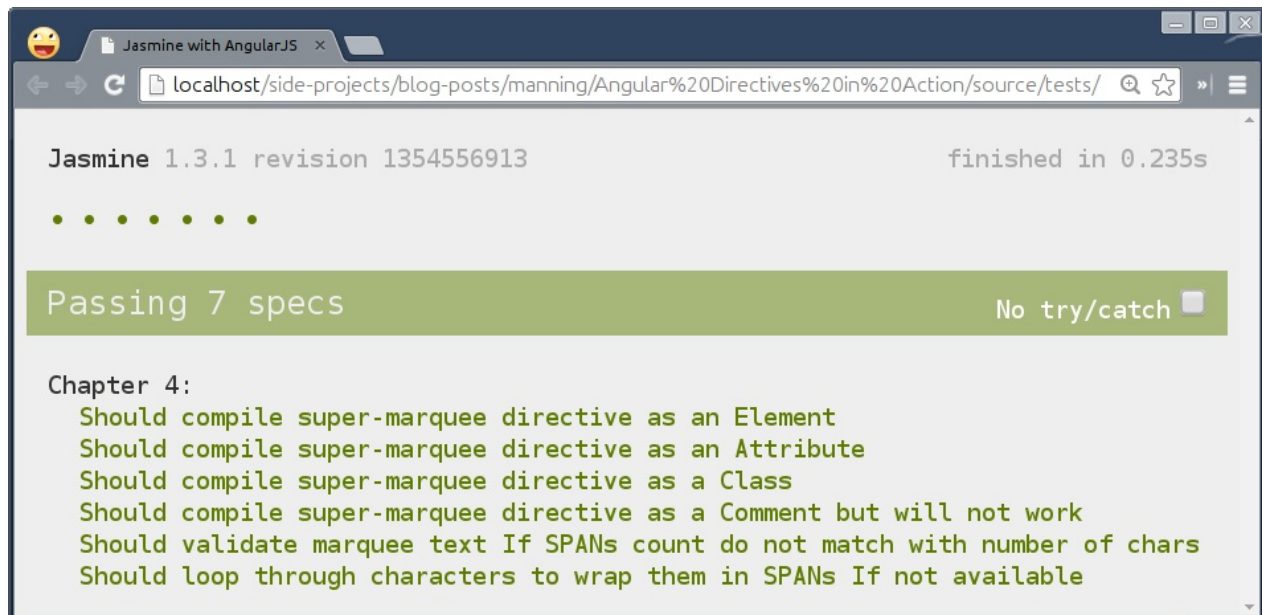
  expect(element.text()).toBe('A B');
  expect(element.find('span').length).toBe(2);

  // If SPANs count do not match chars count
  element = angular.element('<super-marquee loop validate data-text="<span>A</span> B" data-scrollldelay="2" data-direction="up"
  element = $compile(element)($rootScope);
```



```
expect(element.text()).toBe('A B');
expect(element.find('span').length).toBe(2);
});
```

Once we run these test cases in the browser, you will see the following result.



Having all the unit tests passed, we are now relieved from worries and free to update the directive without hesitating a bit. In fact, even if you break something unknowingly, broken tests will let you know about it immediately.

Integration testing of superMarquee with Protractor

Similar to unit testing, the integration testing does not require much effort. It only requires two files, the spec file and the configuration file. As we have already configured it in Chapter 2, we'll go ahead with writing test cases. Create `super-marquee-e2e.js` as:

```
describe('Chapter 3:', function() {
  var ptor = protractor.getInstance(), str;

  it('should activate super-marquee directive', function() {
    ptor.get('ch03/super-marquee.html');
    element.all(by.tagName('div')).each(function(element) {
      expect(element.getText()).toEqual('AngularJS Directives inAction, Yey...!!!');
    });
  });
});
```

Few things to note here that:

- `by` is a collection of element locator/selector strategies which allow you to find elements by CSS selectors, ID, attribute, and tag Name.
- `element.all` returns all the elements that match the locator.
- Later we loop through each matched element and fetch their innerText using `element.getText()` to compare with what we had set in `data-text` earlier.

This test will ensure that `superMarquee` directive works fine in real browsers too.

Using jQuery Plugin in the AngularJS Context

With your learning so far, you will now be able to use any jQuery plug-in available on the web in AngularJS. Let's look at `iscroll` jQuery plugin which is a high performance, small footprint, dependency free, and multi-platform JavaScript scroller which mimics iOS native scrollbar. Let us first install `iscroll` and `twitter bootstrap CSS` framework using `bower`, so run following commands in a terminal as:

```
cd angular-directives-in-traction
bower install --save iscroll
bower install --save bootstrap
```

Here is a list of movies I recently watched that I want you to scroll through for fun. Let us create `iscroll-directive.html` in `ch04/` directory as follows:

```
<html ng-app="App">
<head>
  <title>iScroll Angular Directive</title>
  <script src="../../bower_components/angular/angular.js"></script>
  <script src="../../bower_components/iscroll/build/iscroll.js"></script>
  <link rel="stylesheet" type="text/css" href="../../bower_components/bootstrap/dist/css/bootstrap.css">
  <style type="text/css">
    #wrapper {
      width: 280px;
      height: 202px;
      overflow: hidden;
      position: absolute;
    }
  </style>
  <script type="text/javascript">
    var App = angular.module('App', []);

    App.controller('MainCtrl', function($scope) {
      $scope.movies = [
        'Her',
        'Amazing Spiderman 3D',
        'Silicon Valley HBO',
        'Anchorman 2',
        '300 Rise of an Empire',
        'The Hobbit The Desolation of Smaug',
        'Fandry',
        '12 Years a Slave',
        'Frozen',
        'Ender's Game',
        'Gravity',
        'Riddick',
        'Jobs',
        'The Hobbit The Unexpected Journey',
        'Man of Steel',
        'Turbo',
        'After Earth',
        'Europa Report',
        'Forest Gump',
        'Now You See Me'
      ];

      window.setTimeout(function() {
        new IScroll('#wrapper', {scrollbars: true, mouseWheel: true});
      }, 0);
    });
  </script>
</head>
<body ng-controller="MainCtrl">
  <div id="wrapper">
    <div>
      <ul class="list-group">
        <li class="list-group-item" ng-repeat="movie in movies" ng-bind="movie"></li>
      </ul>
    </div>
  </div>
</div>
```

```
</body>
</html>
```

The code is straight forward to understand as we are just listing down movies collection using `ngRepeat` but then invoking `iScroll`'s constructor on the list to get a pretty scrollbar. The only problem here is that the `iScroll` constructor call is an imperative DOM manipulation in the controller which is an anti-pattern and hence should be avoided at any cost. So we'll remove the following block added earlier from the controller:

```
window.setTimeout(function() {
  new IScroll('#wrapper', {scrollbars: true, mouseWheel: true});
}, 0);
```

Encapsulate the same into its own directive named `iscroll`:

```
App.directive('iscroll', function() {
  return {
    restrict: 'EAC',
    link: function(scope, element) {
      window.setTimeout(function() {
        new IScroll(element[0], {scrollbars: true, mouseWheel: true});
      }, 0);
    }
  }
});
```

As the directive gives access to the original element, we do not have to depend on its id and hence we can reuse the same directive in many places within an application. Next apply the directive on `div#wrapper` as:

```
<div id="wrapper" iscroll>
```

Because of JavaScript single threaded nature, adding a `0ms` delay would actually invoke `iscroll` after a list is rendered by `ngRepeat`. Although it's not safe as you may have to increase the delay depending upon how lengthy your list is. Also the `iscroll` will not update if the content grows/shrinks but do not worry about that for now. We'll learn how to auto update it by watching over a collection of movies in later chapters. However, this technique is good enough for you to use any jQuery plugin or imperative DOM manipulation in AngularJS that does not rely on scopes.

Summary

In this chapter, we have learned about different ways to define directives and their advantages and disadvantages while using them in HTML. Use comment level directives to be used along with other elements that span multiple elements such as `ul`, `li`, `table`, and so on. We saw how template option comes handy to consume a readable chunk of markups but let us load it from an external file when it grows. You can also keep all your templates cached at once place using `$templateCache` service. Many open source AngularJS UI libraries use the same approach. We also realized that the `priority` option automates the order of execution of multiple directives applied on the same element irrespective of their placement (in HTML) that really saves developers effort to remember the ordering and also prevents potential bugs. Directives are the means to prevent imperative DOM manipulation with built-in directives (wherever possible) or use it safely (whenever required) with custom directives. Now you'll be able to understand most of the directives available on the web to improve your knowledge further.

In the next chapter we'll explore scope and what role it plays to make directives configurable.

Understanding Scope in Directives for better context

This chapter covers

- How scope brings newness to the framework like AngularJS
- Learn to write a simple directive that shares the global scope
- Benefit of inheriting the global scope for inbound context
- Sand boxing an isolated but reusable components
- Unit and E2E testing for sample directive
- Extending jQuery plugin with scopes

In the previous chapter, we almost made our hands dirty by playing with easy but crucial options to shape directives. In this chapter, we'll expand our understanding by learning about why and how scopes play an important role in directives and the framework itself. We'll begin with a simple **Spinner** component to choose from a range of values using up/down buttons. We'll then extend it to have a pre-populated list of items to flip through. At the end, we'll create a reusable component and write test cases to make it bulletproof in ever-green browsers. Finally, we'll learn by example how to use angular scopes with jQuery Plugins.

Keeping directives clean with Scope

Being an MVW framework, AngularJS does have a way to define Data Model that consists of application data and business logic. Unlike other frameworks, Model in AngularJS is a plain old JavaScript object and does not require any getter or setter. The getter/setter methods provide a bridge between what is your data and how the framework understands it in order to bind external behavior on it. However, AngularJS does that with Scope. The scope acts as a glue between your data and view resulted in a bi-directional data binding. It also gives access to different methods such as `$watch`, `$apply`, `$digest`, and so on to cater to various needs to build scalable webapps in AngularJS which we will cover in Chapter 7.

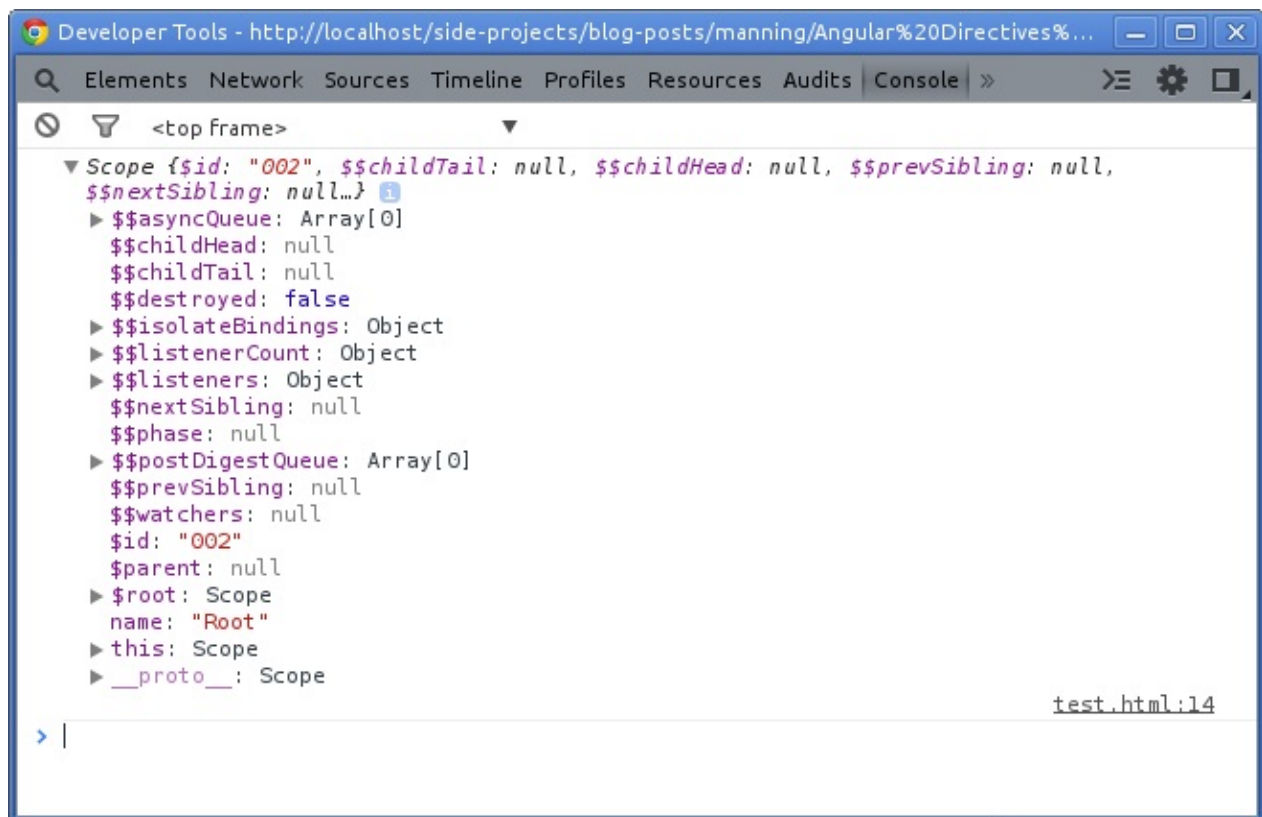
Moreover, **Scope** is an integral part of AngularJS lifecycle to perform data binding and that is why every application built on top of it always has a `rootScope`. The other scopes become child scopes which will be created either automatically by some of the built-in directives (`ngController`, `ngInclude`, `ngSwitch`, and so on) or manually by custom directives. To become proficient in directives, one must understand how scope hierarchy works to help solve issues often raised while working with scopes. Here is a simple example to help us understand how AngularJS maintains a relation between scopes. In this example, I've a model named `name` bound on the `rootScope`.

```
<html ng-app="App">
<head>
  <title>Understanding Scope</title>
  <script src="http://code.angularjs.org/snapshot/angular.js"></script>
  <script type="text/javascript">
    var App = angular.module('App', []);

    App.run(function($rootScope) {
      $rootScope.name = 'Root';
      console.log($rootScope);
    });

    App.controller('FirstCtrl',function($scope){$scope.name = 'First'; });
    App.controller('SecondCtrl',function($scope){$scope.name = 'Second';});
    App.controller('ThirdCtrl',function($scope){$scope.name = 'Third'; });
  </script>
</head>
<body></body>
</html>
```

The browser console will show you the following as follows.

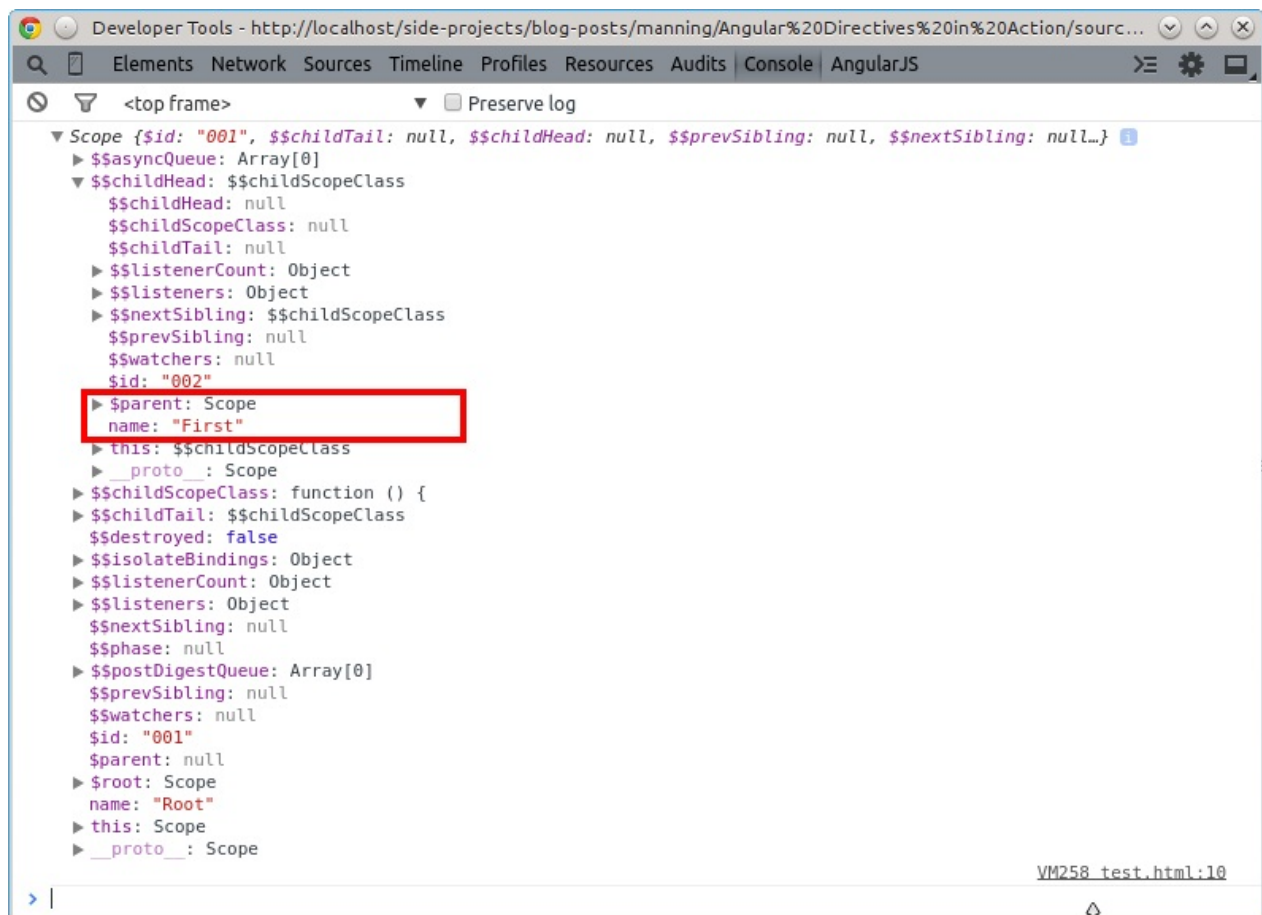


There you could see our model `name` assigned with the same value defined earlier in the run block which is `Root`. Note that public properties in the scope object are prefixed with `$` whereas private ones use `$$`. It is recommended to use only public properties such as `$parent` in the application if needed. You can also notice that the scope object has some weird private properties, `$child`, `$childHead`, `$childTail`, and `$nextSibling` which AngularJS uses to traverse through the scope hierarchy during the digest cycle to find respective `$watchers` for dirty checking.

The `rootScope` is the top most scope in the hierarchy so it will never have any next or previous sibling scopes. As we do not have any child scopes both `$childHead` and `$childTail` are empty as well. Let us update the code to see how hierarchy changes by adding following in the body:

```
<div ng-controller="FirstCtrl"></div>
<div ng-controller="SecondCtrl"></div>
<div ng-controller="ThirdCtrl"></div>
```

The `ngController` directives in HTML will instantiate all three AngularJS controllers defined before in JavaScript. Now after refreshing the page, you will find the information about child scopes under `$childHead` as well as `$childTail` as shown in the following figure.



However, the most important thing to note here is that each of the child scopes refer to their parent scope. As you may know that JavaScript uses Prototypical Inheritance to maintain object hierarchy. In a prototypal system, objects inherit from other objects and that is why we can easily find the parent scope for any child scope by looking at `__proto__` or `$parent` object.

So if we update the `FirstCtrl` controller to following, you will see "Root Root Root" in the console. That's because when `name` is not defined on the current `$scope`, it goes up the scope chain to find the same. Whereas `$scope.$parent` and `$scope.__proto__` both will return the `name` property defined on the parent scope.

```
App.controller('FirstCtrl', function($scope) {
  console.log($scope.name, $scope.$parent.name, $scope.__proto__.name);
  $scope.name = 'First';
});
```

To read the name from the current scope, simply execute the `console.log` statement after setting the `name` property to see "First Root Root" in the console. However, `__proto__` object may not have the correct parent reference information sometimes and may lead to issues while inheriting data models in AngularJS so always remember to use `$parent` object on the scope which handles the parent referencing with ease.

Now that we have understood the scope and prototype gotchas, it is time to dive into AngularJS Directives API again and understand how scope benefits directives. But before moving to that, let us set up the base for Spinner directive and then will write a directive along the way.

First, create `scope-false.html` under `ch05/` directory and put following in it:

```
<html ng-app="SpinnerApp">
<head>
  <title>Scope: false</title>
```

```

<script src="../../bower_components/angular/angular.js"></script>
<script src="../../js/ch05/scope-false.js"></script>
<link rel="stylesheet" href="../../bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body ng-controller="ScopeCtrl">

    <adia-spinner></adia-spinner>

</body>
</html>

```

Now define `ScopeCtrl` in `js/ch05/scope-false.js` which will create a new scope that will be served as a parent scope for the directive as:

```

var App = angular.module('SpinnerApp', []);

App.controller('ScopeCtrl', function($scope) {
    $scope.value = 1;
});

```

As you can see we've used a custom element name `adiaSpinner` to load the spinner widget in HTML, so let us revive it in the following section.

Sharing parent scope: Simple

All the directives we wrote in the previous chapters so far, all had used the scope indirectly. That is, we can tell AngularJS not to create a new scope but to share the exact same parent scope with `scope: false` option. This is also the default option so that the parent scope will be used as a directive's scope by default. Let us explore this option by using it with spinner directive and figure out the use-cases for the same. Add following in `js/ch05/scope-false.js` as:

```

App.directive('adiaSpinner', function() {
    return {
        restrict: 'EA',
        replace: true,
        scope: false,
        template: '\
            <div class="btn-group">\
                <button class="btn btn-success" ng-click="inc()">+</button>\
                <button class="btn disabled">{{spinnerText()}}</button>\
                <button class="btn btn-danger" ng-click="dec()">-</button>\
            </div>',
        link: function(scope) {
            scope.inc = function() {
                scope.value++;
            };

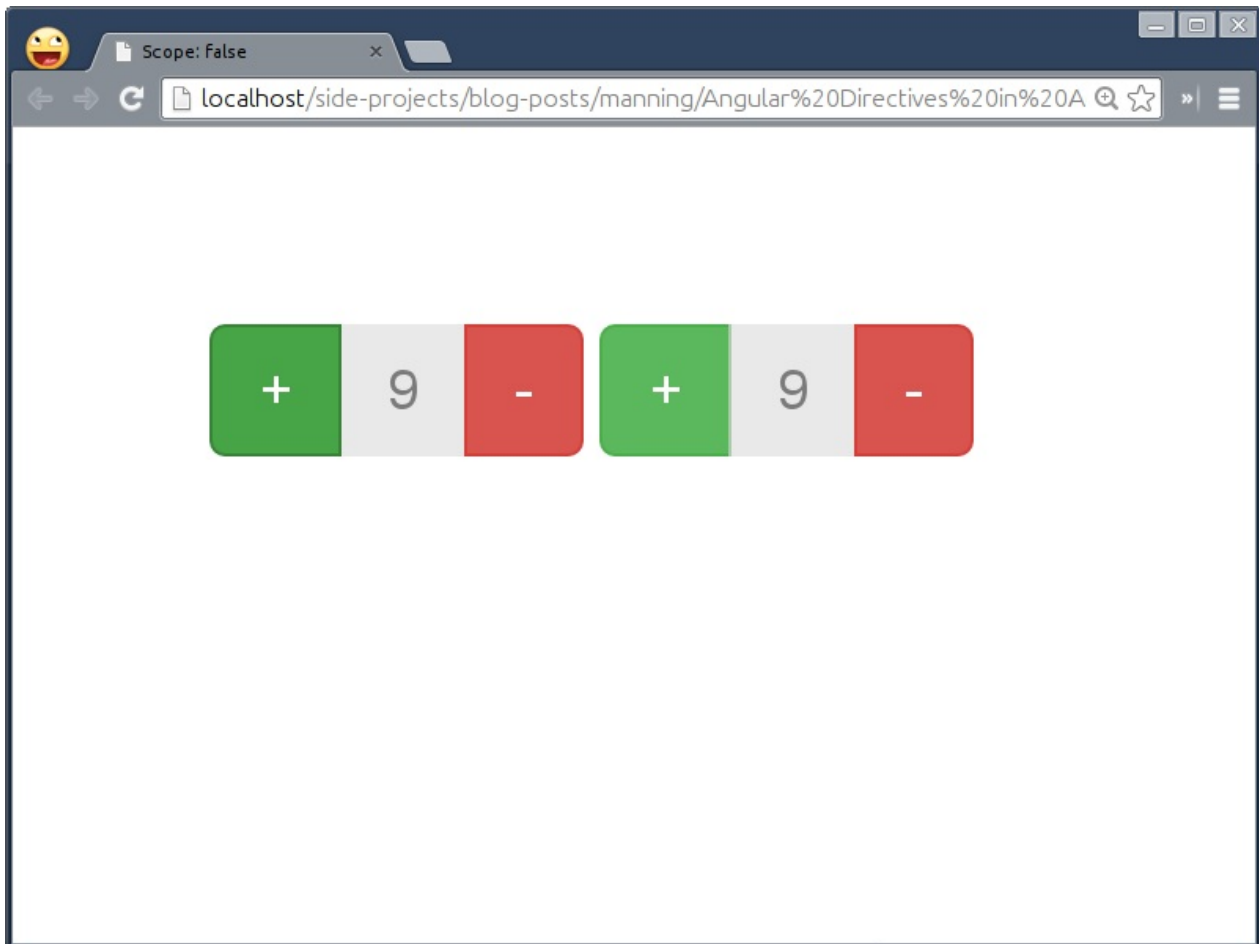
            scope.dec = function() {
                scope.value--;
            };

            scope.spinnerText = function() {
                return scope.value;
            };
        }
    };
});

```

Notice that our directive does not have it's own scope but sharing the parent one instead and hence we could access the value property defined in the controller earlier. The `inc()` method increments the value by 1 whereas `dec()` method decrements it by 1. In addition, the `spinnerText()` method simply displays that value. Now you should see the super simple spinner directive in the browser, click plus/minus buttons to see it in action for yourself.

Over the course of this book, we learned that the real benefit of directives is that they facilitate re-usability of the code by allowing us to re-use the same directive in many places without any side effects. But if you use two instances of `<adia-spinner>` element in HTML, you will be surprised to see that if the first instance increments the value, the other instance also shows the updated value immediately. The following figure focuses on the issue:



However, it is not at all intimidating to see such abnormal behavior because both are sharing the same parent scope which means if you update the value in the controller, both directives will reflect the exact value instantly. On the other hand, if any of the directives modifies the same on its scope, it will be reflected by other directive as well.

That does not mean you should never use this option for any directive, but its just not right for our widget which we will fix in the next section but before that, let us look at a few important aspects you may want to know about it:

- Being a default option, we do not have to mention scope option as `false` explicitly if any directive does not require it.
- This option allows us to use the exact same parent scope (not inherited one) in the directive.
- Any changes to the parent scope will affect other directives using it and vice versa.
- If we use any directive that uses shared scope such as `adiaSpinner` along with other directive on the same element that has its own inherited child scope, the `adiaSpinner` directive will share the inherited scope (created by other directive) instead of the parent one irrespective of the order in which these directives get compiled with priority option enabled.

Overriding the parent scope is crazy, right? But there are few real-world scenarios where you may want to use the shared scope such as:

- If any directive is not dealing with the scope at all but simply manipulating DOM or binding DOM events.
- If any directive is just reading the parent scope values but not overriding them.
- If any nested directive referring to the scope of the parent directive and if its fine to override values on the shared scope. For example, AngularJS UI Bootstrap's datepicker widget (<http://angular-ui.github.io/bootstrap/#/datepicker>)

creates its own scope but is shared across its child directives such as `daypicker`, `monthpicker`, and `yearpicker`.

Similarly, we had used the shared scope for the `iScroll` directive in the previous chapter as its only dealing with the DOM.

Inheriting parent scope: Contextual

In order to solve the problem we faced with the shared scope in the spinner directive earlier, we can have a separate scope but prototypically inherited from the parent scope so that we can read the existing scope values globally but write new or update existing ones locally. Since we are creating a new inherited scope here, other instances of the directive do not get affected when scope value changes. The reason for using a new scope which is inherited from its parent scope is that values binded to the parent scope will be accessed out of the box by the directive without relying on helper properties such as `$parent` or `__proto__`.

By changing the scope option from `false` to `true` fixes the issue for us. Try at your end. Now that the problem has been solved and we are allowed to use the same directive in many places without any side effects, let us add few new features to the spinner directive to take it to the next level. First, add few more parameters to customize the directive to enable `range` type. With range type, we can control how the default value should be incremented or decremented. We can also limit the value to grow/shrink with min and max options. Here is how we can use the same directive with extended options in the DOM as:

```
<adia-spinner data-default="{{value}}" data-min="1" data-max="5" data-interval="2"></adia-spinner>
```

Then update the directive to read them with:

```
App.directive('adiaSpinner', function() {
  return {
    restrict: 'EA',
    scope: true,
    template: '\
    <div class="btn-group">\
      <button class="btn btn-success" ng-click="inc()" ng-disabled="opt.default >= opt.max">+</button>\
      <button class="btn" disabled="{{spinnerText()}}</button>\
      <button class="btn btn-danger" ng-click="dec()" ng-disabled="opt.default <= opt.min">-</button>\
    </div>',
    link: function(scope, element, attrs) {
      scope.opt = {
        type      : 'range',
        default    : parseInt(attrs.default, false) || 1,
        interval   : parseInt(attrs.interval, false) || 1,
        min        : parseInt(attrs.min, false) || 1,
        max        : parseInt(attrs.max, false) || Infinity
      };

      scope.inc = function() {
        scope.opt.default += scope.opt.interval;
      };

      scope.dec = function() {
        scope.opt.default -= scope.opt.interval;
      };

      scope.spinnerText = function() {
        return scope.opt.default;
      };
    }
  };
});
```

No worries...! We have done slight modifications to accommodate extra spinner options as `scope.opt` replacing `scope.value` used before. As we have a separate scope for each instance now, all options will be in their respective context

without affecting parent scope.

Apart from what the spinner directive does now, if we enable it to walk through a set of predefined values such as weekdays or ratings, it will be very useful in real-world applications than just being used as a spinner. So, let us support one more type named `list` to spin a pre-populated list, update the link method of the directive as:

```
link: function(scope, element, attrs) {
  var isList = attrs.type === 'list';

  scope.opt = {
    type      : attrs.type || 'range',
    default   : parseInt(attrs.default, false) || 1,
    interval  : parseInt(attrs.interval, false) || 1,
    min       : parseInt(attrs.min, false) || 1,
    max       : parseInt(attrs.max, false) || Infinity,
    list      : scope.$eval(attrs.list) || []
  };

  if (isList) {
    scope.opt.default = attrs.default ? scope.opt.default : 0;
    scope.opt.min = 0;
    scope.opt.max = scope.opt.list.length - 1;
  }

  scope.inc = function() {
    scope.opt.default += scope.opt.interval;
  };

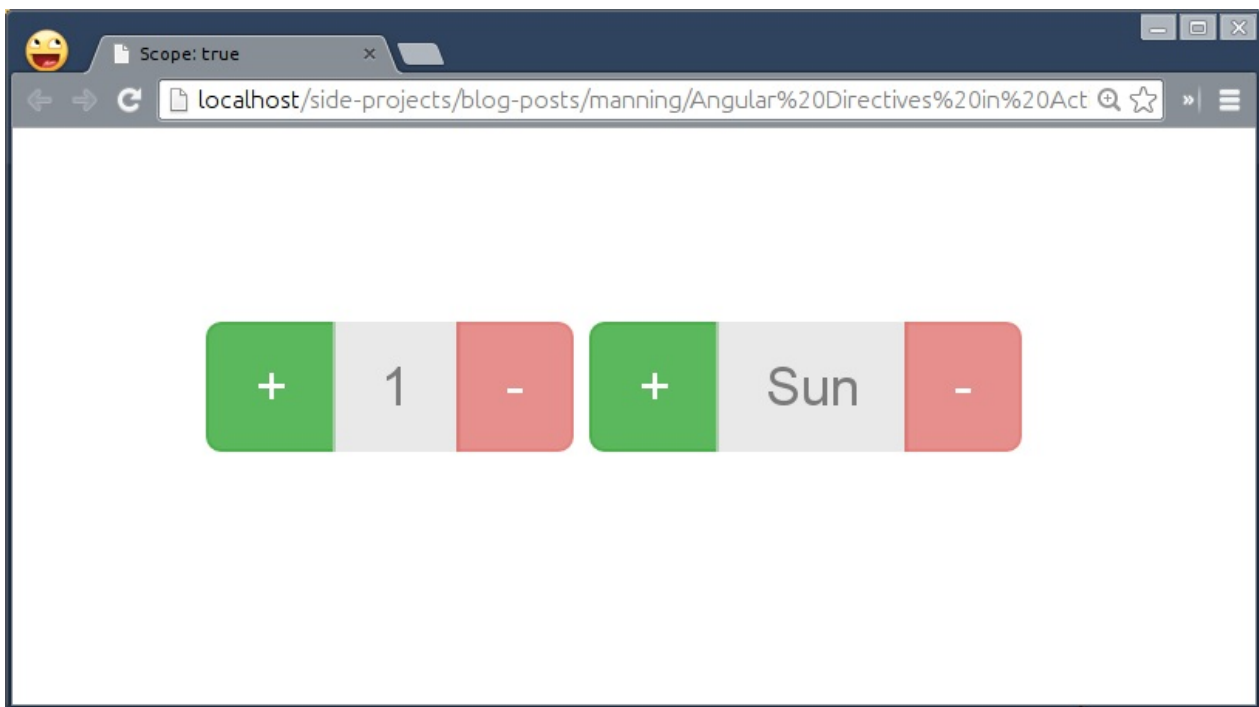
  scope.dec = function() {
    scope.opt.default -= scope.opt.interval;
  };

  scope.spinnerText = function() {
    return isList ? scope.opt.list[scope.opt.default] : scope.opt.default;
  };
}
```

Let us test it out quickly. Add one more instance of the spinner directive next to the current one in HTML as shown:

```
<adia-spinner data-type="list" data-list="['Sun', 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat']"></adia-spinner>
```

There you go! We can now use the same directive to display different set of data in the form of Spinner:



\$evaluating AngularJS expressions

One thing you might have noticed that we have used `scope.$eval` call to transform a list passed as a string attribute into a JavaScript array. The `$eval` method executes AngularJS expressions but not JavaScript expressions on the current scope and returns the result. So `$rootScope.$eval('name')` is same as `$rootScope.name`. Both will return the same value i.e. `Root`. The `$eval` call takes 2 parameters, an expression to evaluate and local value (as JavaScript object) to replace with. However, the second parameter is optional. In addition to that you can also do a lot more cool stuff with it as follows:

- Initializing a variable on the scope. For instance, `$scope.$eval('PI = 3.14')` will bind a model named `PI` with a value of `3.14` on the current scope.
- Performing arithmetic operations such as `$scope.$eval('3 + 4')` would return `7` as a result. We can even replace numbers used with AngularJS models that is `PI + PI` wherein `PI` is defined on the same scope beforehand.
- Overriding existing values of expressions with locals that is `$scope.$eval('minute * 60 + "s"', {minute: 2})` would bind `minute` model on the current scope first and then will be used within the expression to evaluate further. Offcourse, this would return `120s` as a result.
- Evaluating objects or arrays passed as a string or model similar to `scope.$eval(attrs.list)` seen before.

However, `$eval` is synchronous and hence evaluates the expression right away but sometimes it is useful to execute particular expression at later point in time, to do so use the asynchronous variant of `$eval` called `$evalAsync`. Note if the digest cycle is in progress, the expression will be evaluated during the same. Otherwise, it will trigger a new digest cycle on the `$rootScope` after 0ms using `window.setTimeout`. Since the `setTimeout` method in JavaScript has its own call stack and therefore the method passed to it will be run as soon as the current stack is cleared. In simple words, the `setTimeout` call allows browser to finish whatever it has been waiting to finish before executing our method. Having said that our expression will be evaluated atleast once. AngularJS internally uses the same in `$anchorScroll` service to scroll to the related element whenever browser hash/route changes.

\$parsing AngularJS expressions

In contrast to `$eval`, the `$parse` service converts an AngularJS expression into a function which represents the compiled expression that you can evaluate by passing a proper context a.k.a. scope and local variables a.k.a locals in order to override values present in the context/scope. Similar to `$eval`, the `$parse` call takes 2 parameters and returns the result after evaluating the expression on the current scope with locals passed as a second parameter. Like `$eval`, you can use `$parse` as:

```
App.run(function($rootScope, $parse) {
  console.log($parse('PI = 3.14')($rootScope));
  console.log($parse('1 + 2')());
  console.log($parse('PI + PI')($rootScope));
  console.log($parse('minute * 60 + "s"')($rootScope, {minute: 2}));
});
```

However, `$parse` is a built-in service/provider in AngularJS that we have to inject as a dependency before using. Here `$rootScope` passed to `$parse` is the context/scope and second parameter is the locals to override the existing value of minute with “2”. In fact, `scope.$eval` internally calls the same `$parse` service which means `$parse` is an ideal choice if you want to evaluate same AngularJS expression multiple times. The `$parse` call compiles the expression into a function awaiting for the context and locals to be passed in. Hence the `scope.$eval` method may call `$parse` multiple times (for multiple `$eval` calls) which can be avoided by calling `$parse` once to convert the expression into a function that ready to serve the right context/scope later. For the spinner directive, it is fine to use `scope.$eval` over `$parse` as we are parsing `attrs.list` just once.

Now that we realized the benefit of `$parse` service over `$eval`, let us walk through some important details about using a new inherited child scope in a directive:

- We need to explicitly mention to create a new scope for the directive if its required.
- This option creates a new scope but prototypically inherited from its parent.
- Any changes to the new scope created will not affect its parent scope and vice versa but it can be circumvented with a custom `$watch` method.
- This option gets higher priority than shared scope option if multiple directives with different scope strategies are exposed on an element.
- Because of prototypical inheritance, it still allows us to access any model defined on the parent scope seamlessly. Use `scope.$parent` alternatively.

Apart from writing self composed widgets with it, there are few real-world use cases where you might want to use this option:

- If any directive needs a scope to bind few models locally (within the directive's instance) without any side effects such as `ngController` directive.
- If any nested directive requires its state to be maintained separately. For example, an accordion widget can keep multiple panels open at a time and hence need a separate child scope to handle the same. That's not the case with a Tab widget wherein only one tab can be opened at a time. Even the `ngRepeat` native directive uses it over items it repeats for separate context per item.
- The interpolation used in an element such as `<alert type=' '></alert>` will be evaluated in the context of it's own scope. If the `type` model is not found on the current scope, the parent scope will be used for evaluation.

This option is often used than Shared scope as well as Isolate scope which is what we are going to learn about in the next section.

Isolating parent scope: Private

Even though `scope:true` makes the directive contextual with the separate but prototypically inherited scope which has their own problems. Sometimes to build reusable components, it needs a complete isolation to avoid accidentally reading or modifying data in the parent scope and that is what isolate scope provides. The isolate scope differs from normal scope in that it does not prototypically inherit from the parent scope. For instance, imagine an Alert directive whose purpose is just to show error messages, highlight them with different color codes i.e. success, error, info, etc and hide when closed. This kind of component does not have to depend on a parent scope as it can easily evaluate whatever stuffed into it. Here is how we can set an isolate scope to the spinner directive:

```
scope: {
}
```

With this option, the interpolation used in an element such as `<alert type='{{type}}'></alert>` will not be evaluated on the isolate scope but the parent scope. Off-course this does not make sense at all as you may need a controller's scope or `$rootScope` to define the model named `type` to use with the directive. Therefore, the isolate scope allows us to pass an object which defines a set of local scope properties derived from the parent scope. There are three ways to do that:

- `@` or `@attr`
- `=` or `=attr`
- `&` or `&attr`

Let us look at each and see how to use to benefit the spinner directive.

@ for One way Data Binding means a String

We all know that an attribute of an element is always a string and that is why we had to use `parseInt` to parse `default`, `interval`, `min`, and `max` options to integers. As you already guessed the `@` is best suitable for passing string values to the spinner widget, so let us quickly update `scope:true` with:

```
scope: {
  'type': '@'
}
```

In our case, the type will always be a string, either range or list which we can be used as:

```
<adia-spinner data-type='list'></adia-spinner>
```

Additionally you can pass an evaluated expression such as `data-type="{{type}}"` wherein the type property needs to be defined on the parent scope which will be evaluated within the parent context and then passed to the directive as a string attribute. But you can avoid doing so with the two-way data binding syntax explained in the next section.

= for Two way Data Binding means an AngularJS model

Passing other options such as `default`, `interval`, and so on in the form of interpolation does not make sense as we then have to parse them into integers later. What if we could send these options as AngularJS models which will be evaluated on the parent context before being passed to the directive. Well, that's exactly what `=` option does. We can pass:

```
scope: {
  'type'      : '@',
  'default'   : '=',
  'interval'  : '=',
  'min'       : '=',
  'max'       : '=',
  'list'      : '='
}
```

With this we can completely get over `$eval` vs `$parse` paradox of choice and pass either AngularJS models or primitives because AngularJS itself evaluates models on the parent scope before being passed ahead as:

```
<adia-spinner default="1" interval="1" min="1" max="5"></adia-spinner>
```

```
<adia-spinner type="list" list="['Sun', 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat']"></adia-spinner>
```

Most of the options the spinner directive takes are not mandatory, so we can make some of them optional with the question mark (?) as a suffix as shown:

```
scope: {
  'type'      : '@',
  'default'    : '=?',
  'interval'   : '=?',
  'min'        : '=?',
  'max'        : '=?',
  'list'       : '=?'
}
```

In addition to this, the interval option does not look that fancy and may be misleading to our users. We can change that without modifying the underlying directive definition as:

```
scope: {
  'type'      : '@',
  'default'    : '=?',
  'interval'   : '=?step',
  'min'        : '=?',
  'max'        : '=?',
  'list'       : '=?'
}
```

Yes, `=step` let us use step instead of interval in the DOM, so update Spinner instances as follows:

```
<adia-spinner default="1" step="1" min="1" max="5"></adia-spinner>
<adia-spinner type="list" list="['Sun', 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat']"></adia-spinner>
```

Notice the usage of `step` replacing `interval` attribute above. Now we'll update the link method to replace `attrs.*` occurrences with `scope.*` as we do not need them now:

```
link: function(scope, element, attrs) {
  var isList = scope.type === 'list';

  scope.opt = {
    type      : scope.type || 'range',
    default    : scope.default || 1,
    interval   : scope.interval || 1,
    min        : scope.min || 1,
    max        : scope.max || Infinity,
    list       : scope.list || []
  };

  if (isList) {
    scope.opt.default = scope.default ? scope.opt.default : 0;
    scope.opt.min = 0;
    scope.opt.max = scope.opt.list.length - 1;
  }

  scope.inc = function() {
    scope.opt.default += scope.opt.interval;
  };

  scope.dec = function() {
    scope.opt.default -= scope.opt.interval;
  };

  scope.spinnerText = function() {
    return isList && scope.list ? scope.opt.list[scope.opt.default] : scope.opt.default;
  };
}
```

```
}
```

With little modifications, we are ready to see the directive in action.

& for Method means an AngularJS method expression

Imagine we want to use the spinner widget in an online Job Application Form and the values filled need to be saved back into the database when submitted. We'll need a provision to pass a method or callback that will be triggered when spinner updates. That's exactly `&` helps us to accomplish. Let us add optional `onChange` callback to the Spinner directive as:

```
scope: {
  'type'      : '@',
  'default'    : '=?',
  'interval'   : '=?step',
  'min'        : '=?',
  'max'        : '=?',
  'list'       : '=?',
  'onChange'   : '&?'
}
```

In HTML, we can either pass a function call with or without parameters or just mention a function name to be called. The difference between the two is that if a function name is passed, it will be evaluated on the parent context returning the actual function definition which we will then be invoked with `scope.onChange()` call. In contrast, a function call is converted into a function which represents the compiled expression that can be invoked by passing parameters if required. So we'll write a new method in the link function in order to notify of the change. Add following inside the link method of the directive as:

```
scope.inc = function() {
  scope.opt.default+= scope.opt.interval;
  scope.notify();
};

scope.dec = function() {
  scope.opt.default-= scope.opt.interval;
  scope.notify();
};

scope.notify = function() {
  if (typeof scope.onChange({param: {value: scope.opt.default, text: scope.spinnerText()}}) === 'function') {
    scope.onChange()({value: scope.opt.default, text: scope.spinnerText()});
  }
};

scope.notify();
```

This will immediately trigger the callback to save the current spinner state when the directive is compiled as well as when the value updates. Here is how we can use it in HTML:

```
<adia-spinner data-default="1" data-step="1" data-min="1" data-max="5" on-change="getRating(param)"></adia-spinner>
<adia-spinner data-type="list" data-list=["Sun", 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat']" on-change="getWeekday"></
<button class="btn btn-lg btn-primary" ng-click="submit()">Save</button>
```

Please note that the `param` parameter highlighted above should match with the one used inside the notify method. The updated value will be passed as a parameter to a callback method that is `getRating` and `getWeekday` in our case.

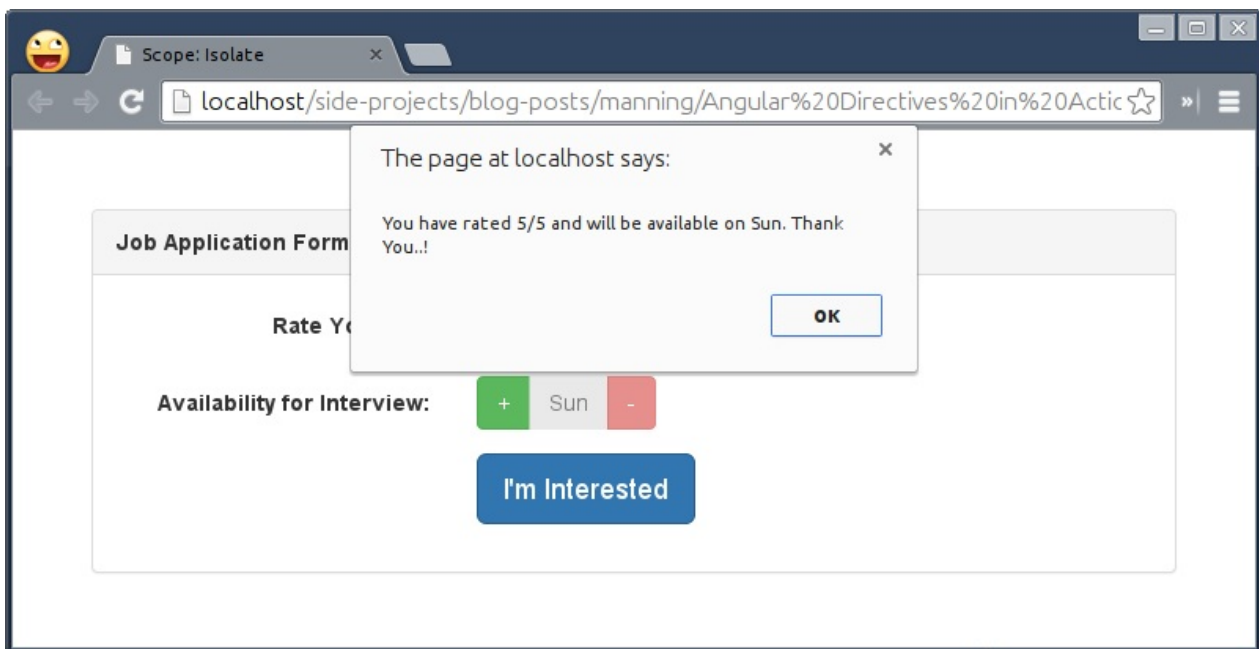
Finally fit the last piece of the puzzle..! Define both the callbacks in `ScopeCtrl1` that we defined at the beginning of the chapter as:

```
App.controller('ScopeCtrl', function($scope) {
  $scope.getWeekday = function(objWeekday) {
    $scope.weekday = objWeekday;
  };

  $scope.getRating = function(objRating) {
    $scope.rating = objRating;
  };

  $scope.submit = function() {
    alert('You have rated ' + $scope.rating.value + '/5 and will be available on ' + $scope.weekday.text + '. Thank You..!');
  };
});
```

Go ahead and check it out..! When submit is pressed, you should see the selected weekday and rating as per following figure.



I'm sure you may be wondering about why there is a need of `ScopeCtrl` to have callbacks defined outside of directives as Isolate scope is all about pluggable components that have no intent to use AngularJS expressions inherently. Well, your assumption is damn right and there is a scope of improvement using `ngModel` on the directive which we'll cover in Chapter 6 after getting the hang of `require` option. For now, let us move forward to write test cases for the same in the next section.

Testing of Spinner

Time for some tests..! Passing tests is a proof that your code actually works and a token of an excellent work done. Let us write Jasmine test cases first by updating `unit.conf.js` and add references `scope-isolate.js` and its test file as shown:

```
files: [
  'bower_components/jquery/jquery.js',
  'bower_components/angular/angular.js',
  'bower_components/angular-mocks/angular-mocks.js',
  'js/ch01/angular-template.js',
  'js/ch04/super-marquee.js',
  'js/ch05/scope-isolate.js',
  'tests/specs/ch01/*-unit.js',
  'tests/specs/ch04/*-unit.js',
  'tests/specs/ch05/*-unit.js'
]
```


Please note that we'll only test *scope-isolate.js* because of space constraint so writing test cases for *scope-false.js* and *scope-true.js* will be pretty straight forward and hence are left as an exercise for you. We'll start off with if the directive takes default options properly or not. So let us add following in *tests/specs/ch05/scope-isolate-unit.js* as:

```
describe('Chapter 5: ', function() {
  beforeEach(module('SpinnerApp'));

  var element, $plus, $minus;

  it('Should compile spinner directive with isolate scope', inject(function($rootScope, $compile) {
    element = angular.element('<adia-spinner></adia-spinner>');
    element = $compile(element)($rootScope);
    $rootScope.$digest();
    expect(element.find('.btn:eq(1)').text()).toBe('1');

    element = angular.element('<adia-spinner data-default="2"></adia-spinner>');
    element = $compile(element)($rootScope);
    $rootScope.$digest();
    expect(element.find('.btn:eq(1)').text()).toBe('2');

    $rootScope.default = 3;
    element = angular.element('<adia-spinner data-default="default" data-step="2"></adia-spinner>');
    element = $compile(element)($rootScope);
    $rootScope.$digest();
    expect(element.find('.btn:eq(1)').text()).toBe('3');
    element.find('.btn:eq(0)').click();
    expect(element.find('.btn:eq(1)').text()).toBe('5');
  }));
});
```

If you remember, we have not used `rootScope.$digest` while testing *superMarquee* directive in the previous chapter but the *Spinner* directive uses data binding to update the spinner value and `$digest` call is required to update the binding with the actual value. The `.btn:eq(0)` and `.btn:eq(1)` jQuery selectors target spinner's plus button and the text respectively. The `toBe` is one of the matchers Jasmine supports to match the actual value with the expected one.

Also the spinner should not increment or decrement the value when it's limit has reached. Let us check if the spinner's plus/minus buttons freeze as expected, so add the following to create one more Jasmine spec as:

```
it('Should not increment/decrement the value when spinner limit is reached', inject(function($rootScope, $compile) {
  element = angular.element('<adia-spinner data-min="1" data-default="2" data-max="3"></adia-spinner>');
  element = $compile(element)($rootScope);
  $rootScope.$digest();

  $plus = element.find('.btn:eq(0)');
  $minus = element.find('.btn:eq(2)');
  expect(element.find('.btn:eq(1)').text()).toBe('2');
  expect($plus.is(':disabled')).toBeFalse();
  expect($minus.is(':disabled')).toBeFalse();
  $minus.click();
  expect($minus.is(':disabled')).toBeTruthy();
  $plus.click().click();
  expect($plus.is(':disabled')).toBeTruthy();
}));
```

Note that `toBeFalse` and `toBeTruthy` are the matchers we can use to replace `toBe(false)` and `toBe(true)` respectively – no added benefits over `toBe` but little handy for matching booleans.

Finally, we'll test if the spinner directive does support `list` type. So update *scope-isolate-unit.js* with:

```
it('Should work with type: list', inject(function($rootScope, $compile) {
  $rootScope.availability = ['Friday', 'Saturday', 'Sunday'];
  element = angular.element('<adia-spinner data-type="list" data-default="2" data-list="availability" on-change="which"');
  element = $compile(element)($rootScope);
```

```

$rootScope.whichDay = function(day) {
  $rootScope.day = day;
};
spyOn($rootScope, 'whichDay').andCallThrough();
$rootScope.$digest();

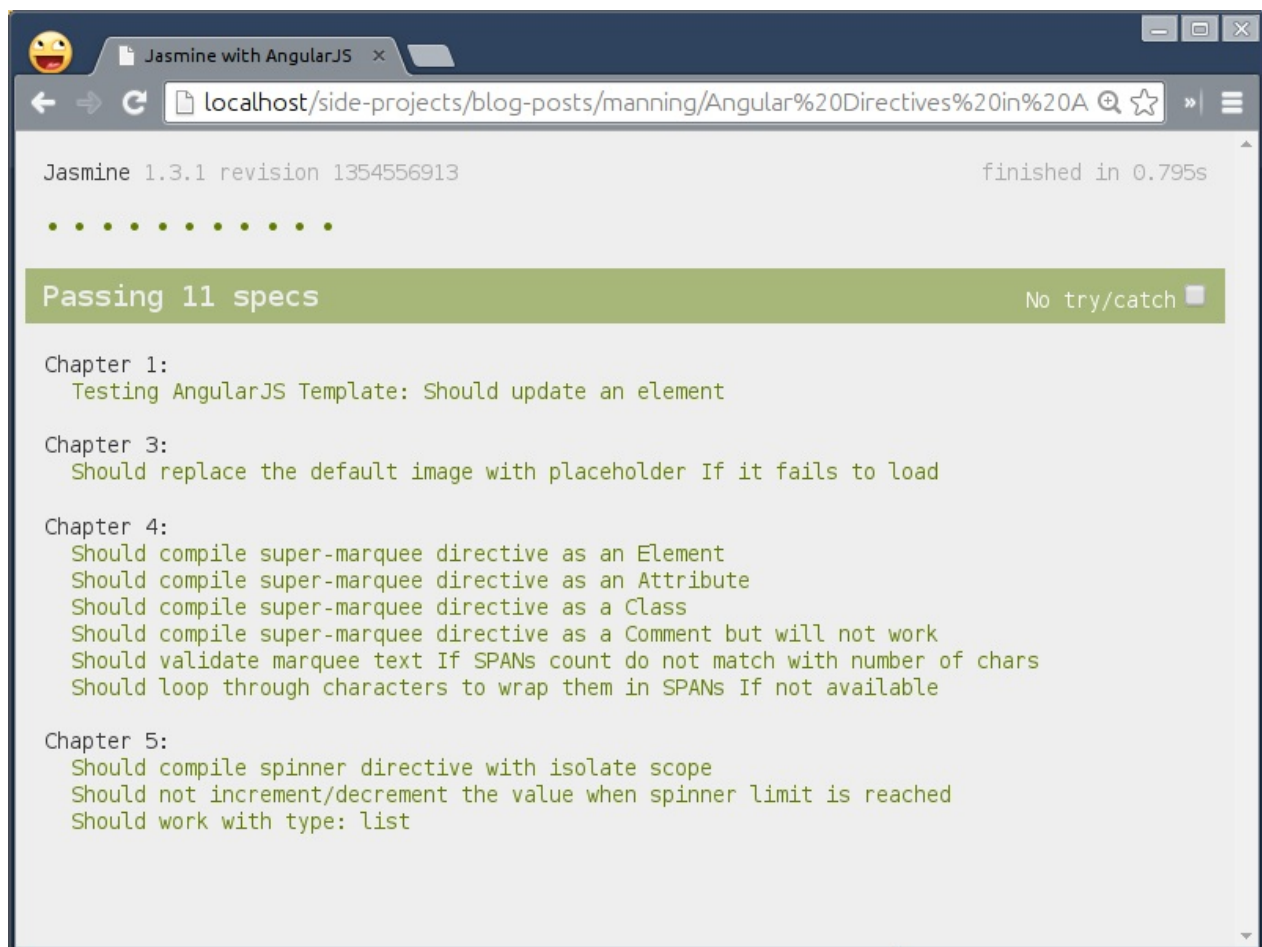
$plus = element.find('.btn:eq(0)');
$minus = element.find('.btn:eq(2)');

expect(element.find('.btn:eq(1)').text()).toBe('Sunday');
expect($plus.is(':disabled')).toBeTruthy();
$minus.click().click();
expect(element.find('.btn:eq(1)').text()).toBe('Friday');
expect($minus.is(':disabled')).toBeTruthy();

expect($rootScope.whichDay).toHaveBeenCalled();
expect($rootScope.day).toEqual({text: 'Friday', value: 0});
});

```

Now that we have the specs ready, just run `"npm run test-unit"` command in the terminal to see all tests passing as:



We are now rest assured that the Spinner directive is bullet proof and works as intended. Let us write E2E tests to do functional testing as well. First update the suites for this chapter in `e2e.conf.js` as:

```

suites: {
  ch1: 'tests/specs/ch01/*-e2e.js',
  ch2: 'tests/specs/ch02/*-e2e.js',
  ch4: 'tests/specs/ch04/*-e2e.js',
  ch5: 'tests/specs/ch05/*-e2e.js'
},

```

And then create *tests/specs/ch05/scope-isolate-e2e.js* as:

```
describe('Chapter 5:', function() {
  it('should activate spinner directive', function() {
    browser.get('ch05/scope-isolate.html');

    element.all(by.tagName('button')).then(function(btn) {
      var $ratingPlus = btn[0];
      var $rating = btn[1];
      var $ratingMinus = btn[2];
      // increment rating
      expect($rating.getText()).toBe('1');
      $ratingPlus.click();
      $ratingPlus.click();
      $ratingPlus.click();
      expect($ratingPlus.isEnabled()).toBeTruthy();
      $ratingPlus.click();
      expect($rating.getText()).toBe('5');
      expect($ratingPlus.isEnabled()).toBeFalsy();
      // decrement rating
      $ratingMinus.click();
      $ratingMinus.click();
      $ratingMinus.click();
      expect($ratingMinus.isEnabled()).toBeTruthy();
      $ratingMinus.click();
      expect($rating.getText()).toBe('1');
      expect($ratingMinus.isEnabled()).toBeFalsy();

      var $availPlus = btn[3];
      var $avail = btn[4];
      var $availMinus = btn[5];
      expect($avail.getText()).toBe('Sun');
      $availPlus.click();
      $availPlus.click();
      $availPlus.click();
      $availPlus.click();
      $availPlus.click();
      expect($availPlus.isEnabled()).toBeTruthy();
      $availPlus.click();
      expect($avail.getText()).toBe('Sat');
      expect($availPlus.isEnabled()).toBeFalsy();
      $availMinus.click();
      $availMinus.click();
      $availMinus.click();
      $availMinus.click();
      $availMinus.click();
      expect($availMinus.isEnabled()).toBeTruthy();
      $availMinus.click();
      expect($avail.getText()).toBe('Sun');
      expect($availMinus.isEnabled()).toBeFalsy();
    });
  });
});
```

This completes the testing for the Spinner directive, so quickly run `npm run test-e2e` command to run them in Protractor. Now let us shift our focus on a jQuery plugin and see how can our newly learned scope related techniques help us to improve the `iscroll` directive which we wrote in the previous chapter.

Using jQuery Plugin the Angular Way

First of all, create a copy of *iscroll-directive.html* from *ch04/* to *ch05/* directory. Then update `div#wrapper` to make `iscroll` directive customizable. Here we can customize the scrollbars visibility and mousewheel support as:

```
<div id="wrapper" iscroll data-scrollbar="true" data-mousewheel="true">
```

And update the directive definition so:

```
App.directive('iscroll', function() {
  return {
    restrict: 'EAC',
    scope: {
      scrollbars: '=?',
      mousewheel: '=?'
    },
    link: function(scope, element) {
      window.setTimeout(function() {
        new IScroll(element[0], {
          scrollbars: angular.isDefined(scope.scrollbars) ? scope.scrollbars : true,
          mousewheel: angular.isDefined(scope.mousewheel) ? scope.mousewheel : true
        });
      }, 0);
    }
  };
});
```

The `setTimeout` JavaScript method was a little hack we had used earlier to delay the `iscroll` instantiation but it will not always work. AngularJS let us allow to write a custom `$watch` method to update the view when model changes which we can leverage here to replace `window.setTimeout` call. Update the link method of the `iscroll` directive so:

```
link: function(scope, element) {
  var myScroll = null;

  scope.$watch(function() {
    if (myScroll) {
      myScroll.refresh();
    } else {
      myScroll = new IScroll(element[0], {
        scrollbars: angular.isDefined(scope.scrollbars) ? scope.scrollbars : true,
        mousewheel: angular.isDefined(scope.mousewheel) ? scope.mousewheel : true
      });
    }
  });
}
```

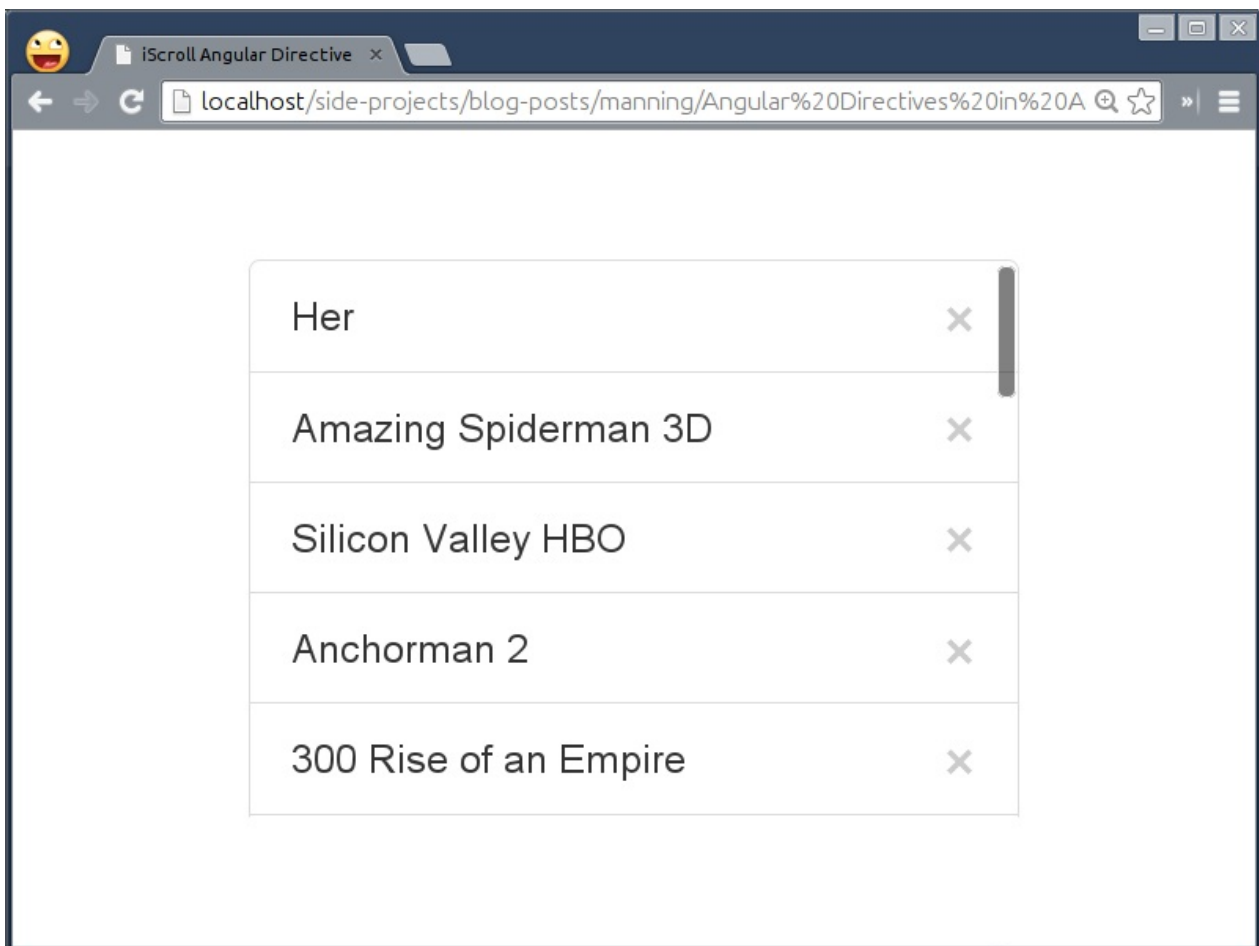
The `$watch` method in AngularJS takes an expression or a JavaScript method to watch over as a first parameter and a callback to trigger as a second parameter. In this case, we are not watching anything that means the callback will be triggered for every `$digest` cycle. That is why to avoid re-instantiation of `iScroll`, we have cached the instance to simply update `iscroll` if its already constructed. To test that out, let us replace the existing `` element:

```
<li class="list-group-item" ng-repeat="movie in movies" ng-bind="movie"></li>
```

with:

```
<li class="list-group-item" ng-repeat="movie in movies">
  <button type="button" class="close" ng-click="movies.splice($index, 1)">&times;</button>
  <span ng-bind="movie"></span>
</li>
```

Here we have just added a close icon next to each movie so that we can see if the `iscroll` updates when any movie is removed from the list.



As soon as we remove any movie, the `$digest` cycle will run to update the `` list, and the iScroll `$watch` will be triggered to update itself. Try at your end.

NOTE: There are various ways to use `$watch` method in AngularJS to resolve common UI problems. How complex or heavy \$watchers could sabotage your application and what you can do to fix such issues, all that will be covered in detail in Chapter 7.

Summary

There we are..!

This chapter has expanded our understanding on why and how scopes play an important role in AngularJS directives and the framework itself. Directive is a place to put any sort of imperative DOM manipulation in AngularJS and it's very likely that we are not going to deal with scope in such directives, so always use simple a.k.a. shared scope in such scenarios. Sometimes we need an easy access to methods or properties assigned on the parent scope but still require own space for a directive to work efficiently without affecting the rest of the application by accidentally overriding models on the parent scope. Use inherited child scope in such cases. Isolated a.k.a. Private scope is simply useful to make reusable components. We also learned that `$parse` over `$scope.$eval` can be used for performance gain while evaluating same expression multiple times. We then used 3 ways to pass data to an isolated directive as attributes, bi-directional AngularJS models, and AngularJS method expressions.

All in all we learned everything about scope in directives to master different scope strategies to improve a simple yet useful Spinner widget. Later we tested the Spinner directive using Karma and Protractor to verify if it works well. Finally we leveraged the newly learned skill to fix the iScroll directive to auto update itself when a list changes.

Crafting directives to handle Complex Scenarios

This chapter covers

- The difference between Compile and Link
- Transcluding an external markup to customize template used by directives
- Use of directive controller to wire up nested directives and how it differs from normal controller
- How directives can be independent but still communicate with each other by requiring a directive controller of parent or sibling directives
- Enabling data binding, validation, DOM updates, and data formatting and parsing to custom controls using `ngModelController`
- Unit and E2E testing for sample directives

In the previous chapter, we saw how different scope strategies are useful to build simple to complex reusable components or widgets in AngularJS.

In this chapter, we'll learn all the useful things (mentioned above) that will help us build component/widget from the scratch in AngularJS. We'll create a complex *Page Thumbnail Pane* (used in Adobe Acrobat) as an exercise to quickly navigate to particular page. We'll also improve the *Spinner* directive as promised in order to get the current state without relying on external scoped callbacks. Then we'll combine *Spinner* directive with *Thumbnail Pane* directive to make it easy to navigate to specific thumb.

Next level Up!

All the things that we learned over the course of this book so far are good enough for us to port any existing component or plugin in the AngularJS world with an extra layer on top of it in terms of AngularJS Directives. Take an example of an *iscroll* directive that we wrote in the previous chapter that really helped us to use *iScroll* jQuery plugin as is in AngularJS context. But sometimes adding this extra layer is unnecessary and bloating the widget which can be avoided by restructuring it entirely using AngularJS Directives API.

Imagine Twitter Bootstrap's Tab component built on top of jQuery which can be completely restructured and written in a short time with two-way data binding and become modular with dependency injection in AngularJS. AngularUI Bootstrap is the perfect example of this wherein it has native AngularJS directives for the same with small footprint and no 3rd party JavaScript dependencies at all. That does not mean, you should always rewrite any existing jQuery plugin or so. But in many cases, you should because AngularJS two way data binding surely reduces the effort you require to make something in other than AngularJS which makes it declarative as well. We'll learn everything that required to make rewriting jQuery plugins in AngularJS possible for you.

Let us create a base template for our Page ThumbViewer example that we'll enhance step by step as we conquer each option in directive definition object. Along the way we'll also build sample directives to strengthen our understanding about each of those options. First of all scaffold Page Thumbviewer component by creating *thumbnail-viewer.html* in *ch06/* directory as:

```
<html ng-app="ThumbApp">
<head>
  <title>ThumbViewer</title>
  <script src="../../bower_components/angular/angular.js"></script>
  <script src="../../js/ch06/thumbviewer.js"></script>
  <link rel="stylesheet" type="text/css" href="../../bower_components/bootstrap/dist/css/bootstrap.css">
  <style type="text/css">
    .thumbnail-container {
      display: inline-block;
      width: 100%;
      overflow: hidden;
    }
  </style>
</head>
<body>
  <div class="thumbnail-container">
    <div class="thumb">
      <img alt="Thumbnail image" data-bbox="120 930 250 980" />
    </div>
  </div>
</body>
</html>
```

```

padding: 10px;
}
img { width: 100%; }
</style>
<script type="text/ng-template" id="thumbviewer.html">
  <div class='thumbnail-container bg-success img-rounded'>
    <div class='thumbnail-wrapper'>
      <!-- Page Thumbs go here -->
    </div>
  </div>
</script>
<script type="text/ng-template" id="thumb.html">
  <div class="thumbnail pull-left"></div>
</script>
</head>

<body style="padding: 10px;" ng-controller="ThumbCtrl">
  <thumbviewer data-size="150" data-gap="0"></thumbviewer>
</body>
</html>

```

Then define a directive in *js/ch06/thumbviewer.js* so:

```

var App = angular.module('ThumbApp', []);

App.controller('ThumbCtrl', function($scope) {
  $scope.thumbnails = [
    { title: 'Page 1', image: '1.png', active: true },
    { title: 'Page 2', image: '2.png' },
    { title: 'Page 3', image: '3.png' },
    { title: 'Page 4', image: '4.png' },
    { title: 'Page 5', image: '5.png' },
    { title: 'Page 6', image: '6.png' },
    { title: 'Page 7', image: '7.png' },
    { title: 'Page 8', image: '8.png' },
    { title: 'Page 9', image: '9.png' },
    { title: 'Page 10', image: '10.png' },
    { title: 'Page 11', image: '11.png' },
    { title: 'Page 12', image: '12.png' },
    { title: 'Page 13', image: '13.png' },
    { title: 'Page 14', image: '14.png' },
    { title: 'Page 15', image: '15.png' }
  ];
});

App.directive('thumbviewer', function() {
  return {
    restrict: 'EA',
    replace: true,
    templateUrl: 'thumbviewer.html',
    scope: {
      size: '=?',
      gap : '=?'
    }
  };
});

```

Make sure to have couple of images in *img/* directory and update the collection `thumbnails` accordingly. Go ahead and see it for yourself in a browser. I'm sure you will see a rounded green block.

Compile begot Link

In the first chapter, we learned a little bit about how AngularJS brings custom elements or attributes to life by compiling them to react to user interactions. But what exactly happens when any directive gets compiled? To understand that, we must know that every directive has one of these methods, i.e. Compile or Link. It is very confusing at first to realize that the compile method itself produces a link function and the directive can have either of two.

To understand why these two phases exist in the first place, we need to take a look at a simple directive called *compileCheck* that will repeat itself over five times using *ngRepeat* built-in directive. Create *ch06/compile-vs-link.html* as:

```
<html ng-app="CVLApp">
<head>
  <title>Compile vs Link</title>
  <script src="../../bower_components/angular/angular.js"></script>
  <script src="../../js/ch06/compile-vs-link.js"></script>
</head>
<body>
  <compile-check ng-repeat="item in [1, 2, 3, 4, 5]"></compile-check>
</body>
</html>
```

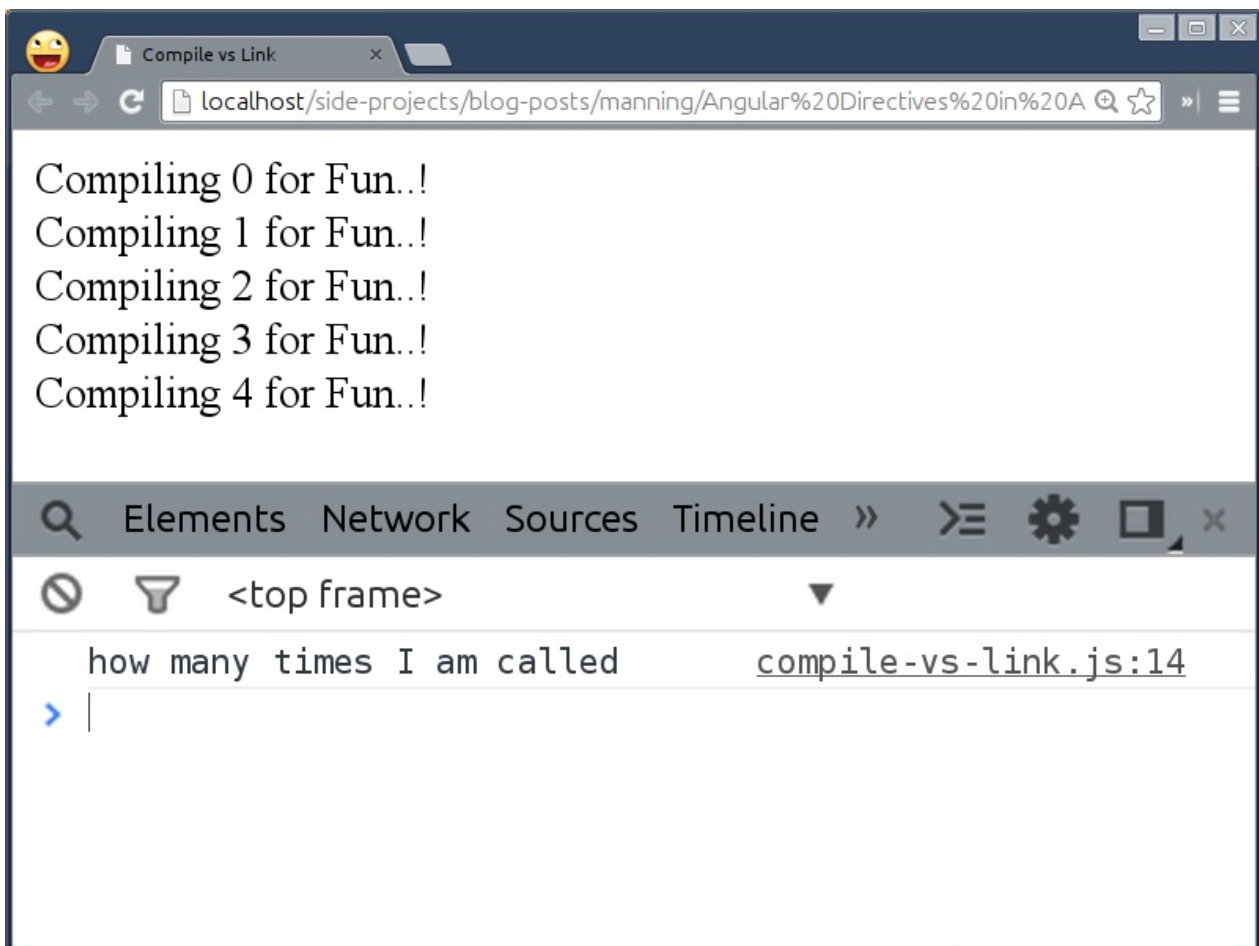
Then create *js/ch06/compile-vs-link.js* as:

```
var App = angular.module('CVLApp', []);

App.directive('compileCheck', function() {
  return {
    restrict: 'EA',
    template: '<div>Compiling {{$index}} for Fun..! </div>',
    compile: function(cElement, cAttrs) {
      console.log('how many times I am called');
    }
  };
});
```

Please note that scope has not been created yet so we can not access it in the compile function. The `cElement` refers to the template element where the directive has been declared, in our case `<compile-check>` and `cAttrs` consists of all the attributes associated with the element such as `ng-repeat` for this particular example.

This simple directive basically dumps the template into the directive's body and then *ngRepeat* just renders it for five times. If you may have thought that the compile method calls for each item in the list then you are surely wrong as shown in the following figure.



As you could see, the compile method calls only once for all but why?

Compile Phase: manipulate the template and performance benefit

The sole reason for having a separate phase to compile all the directives first and then link them altogether is only a performance benefit. Think for a moment that we do not have a compile phase, just a link phase wherein a directive gets compiled as well as linked with the right scope. Imagine we want to update the template to have an hash tag (`#perfatters`) at the end of the sentence in that phase.

Considering the fact that we now have just a link phase, the template will be modified for each instance of the list that it repeats which is redundant. In addition to this, it will interpolate `{{index}}` binding to `0` for the first item as it is now linked to the respective scope right away and will create a watch for the same. The moment you have a watch method set up for any binding, it will be evaluated for each digest cycle on the scope to update the respective view. So when the second item gets compiled and `{{index}}` binding associated with it will also be interpolated to `1`. Now the digest cycle has run again for the new binding to be evaluated. This will drastically reduce performance when a list grows. That's why we have a compile phase in directives.

So to update the template at compile time, we'll add following in the compile phase as:

```
var App = angular.module('CVLApp', []);

App.directive('compileCheck', function() {
  return {
    restrict: 'EA',
    template: '<div>Compiling {{index}} for Fun..! </div>',
    compile: function(cElement, cAttrs) {
      console.log('how many times I am called');
      cElement.find('div').append('#perfatters');
    }
  }
});
```

```
};
});
```

Now the template will be updated first but only once and the transformed template will be used by the repeater later. In short, the compile phase creates a compiled function for each directive waiting for a scope to interpolate bindings similar to the `$parse` service we saw in the previous chapter. Here are some of the checkpoints to consider before using the compile phase in any directive:

- The compile function gives access to two parameters that is the template element and the template attributes. *The parameters can be renamed but the order should not be changed.*
- The template element refers the element on which the directive is applied but the same element will be inserted post compilation is not guaranteed, especially, when used with `ngRepeat` directive which makes a clone of it to repeat over. Suppose our directive binds a click event on the template element, it will not be existed when rendered because `ngRepeat` creates a copy of the element and injects back into the template. The cloning of elements does not carry dynamic DOM events forward unless they are inline such as `onClick`. *That means its not recommended to bind DOM events in this phase.*
- The `cElement` is equivalent to a `$(cElement)` so you do not have to wrap AngularJS elements in `jQuery()` or `$()` constructor because AngularJS contains a minimal version of jQuery named `jQLite` which emulates certain core features of jQuery. *That means you can bind a click event to `cElement` with `cElement.on('click', fn);` instead of `cElement.click(fn);` which will require jquery to be included.*
- The `cAttrs` is an object containing all the attributes and their values associated with the element. The hyphenated attributes are converted to camelCase so that you can access the value of `compile-check` attribute as `cAttrs['compileCheck']`. However, it will return non-evaluated value if it has a data-binding expression because of the unavailability of the scope to evaluate it against.

So, all the complex things that do not require scope can be accomplished in the compile phase for the performance gain. Other benefits are:

- To use different link functions to be produced conditionally. For example, `ngValue` directive that binds the given expression to the value of a form control if the expression is boolean (true/false). Otherwise set up a watcher to evaluate the expression passed, every time the value of the expression changes.
- To use both compile as well as link functions together. For instance, `$parse` or `$interpolate` AngularJS expressions once in the compile phase and get them evaluated in the link phase later by passing an appropriate scope. The built-in directives such `ngClick`, `ngDbclick`, and so on are using the same approach.
- And the most importantly to transform the template element or the template itself.

Moving forward, we'll look at what role the link phase plays and what are best scenarios to use it over compile phase in the following section.

Link Phase: Registering DOM listeners and Setting up \$watch methods

We would start with the question that why can not we bind DOM listeners to an element, `cElement` which is available in the compile function. The reason is that `cElement` and the element available inside a link function are not same. The `cElement` is basically a cloned version of the original element. So when we updated the template with reference to `cElement`, in reality, we modified the cloned element not the actual element that goes into the DOM at the end. For binding DOM events or having a custom `$watch` methods, the link function is the best place.

The link function is further broken down into two phases, `preLink` and `postLink`. Suppose we have nested directives then the flow of all these phases would be like:

```
<parent><child1></child1><child2></child2></parent>
```

```

1. parent compile phase
2. child1 compile phase
3. child2 compile phase
4.1. parent controller
4.2. parent preLink phase
5.1. child1 controller
5.2. child1 preLink phase
5.3 child1 postLink phase
6.1. child2 controller
6.2 child2 preLink phase
6.3 child2 postLink phase
7. parent postLink phase

```

With this you can imagine, if we bind something to a scope inside the parent preLink (4.2.) method, it will be available to all of its children (5.1 – 6.3) because of the way data flows as shown. Now, why do we need such a flow? Because sometimes we might want to access the content of the child directives into the parent postLink (7.) method for further manipulation so the order of execution makes more sense because the child directives should be compiled/linked before gets transforms by the parent directive. Let us try that out.

```

var App = angular.module('CVLApp', []);

App.directive('compileCheck', function() {
  return {
    restrict: 'EA',
    template: '<div>Compiling {{$index}} for Fun..! </div>',
    compile: function(cElement, cAttrs) {
      console.log('how many times I am called');
      cElement.find('div').append('#perfmatters');

      return {
        pre: function(scope, element, attrs) {
          scope.$index = scope.$index + 1;
        },
        post: function(scope, element, attrs) {
          element.find('div').on('click', function(event) {
            alert('You tapped ' + scope.$index);
          });
        }
      }
    }
  };
});

```

In here, we are just updating `$index` used in the template by 1 in the `preLink` method and binding a click handler in the `postLink` method. However, merging both in any of the methods would lead the same output. So it's very rare to use both occasionally but can be useful in case of nested directives. Additionally, you could also do the same with a link function as shown:

```

link: {
  pre: function(scope, element, attrs) {
    scope.$index = scope.$index + 1;
  },
  post: function(scope, element, attrs) {
    element.find('div').on('click', function(event) {
      alert('You tapped ' + scope.$index);
    });
  }
}

```

Now that we understood the difference between link and compile phases, let us quickly walk through some useful bits to keep in mind:

- Unlike compile function, the link function receives 3 parameters, namely, `scope`, `element`, and `attrs`. Feel free to rename them as you wish but the order should be intact.
- If an attribute consists of an AngularJS model that is `data-index='index'`, it can be extracted after the evaluation using `scope.$eval(attrs['index'])` or `$parse(attrs['index'])(scope)` service. If it contains an interpolated string such as `data-index=''`, the observer function, `attrs.$observe` should be used instead that will be covered in Chapter 7.

Most importantly, if a scope is the requirement for your directive, always use the link phase over compile. However, there are other benefits as well you may want to consider before using it. They are:

- The element in the link function refers to the same one that goes into the DOM at the end, so it is safe to bind DOM events in this phase.
- As the scope is available in this phase, we can set up custom watches similar to how we have done in case of *iScroll* directive before.
- Even though the link phase is further divided into two more phases, pre and post. The preLink method is very rarely used but the postLink method is the default link method which is what we have used in Spinner as well as iScroll directives before.

Now it's time to use both phases. Let us see how to apply our understanding to Page Thumbviewer directive in the next section.

Using Link and Compile in Page Thumbviewer directive

We'll first repeat a list of images inside the thumbviewer directive with the help of child directive named `thumb` which we'll write soon. First update the thumbviewer directive in HTML as:

```
<thumbviewer data-size="150" data-gap="0">
  <thumb ng-repeat="thumbnail in thumbnails" active="thumbnail.active">
    <div></div>
    <div class="caption text-center" ng-bind="thumbnail.title"></div>
  </thumb>
</thumbviewer>
```

Then update both the templates used for thumbviewer and thumb directives respectively so:

```
<script type="text/ng-template" id="thumbviewer.html">
  <div class='thumbnail-container bg-success img-rounded'>
    <div class='thumbnail-wrapper' ng-style="wrapperStyle()">
      <!-- Page Thumbs go here -->
    </div>
  </div>
</script>

<script type="text/ng-template" id="thumb.html">
  <div class="thumbnail pull-left" ng-style="thumbnailStyle(this)"></div>
</script>
```

Please note that both the methods passed to `ngStyle` directives return the width to be set on their respective elements. It will be calculated dynamically and may vary based on a total number of thumbnails. For now, we'll hard code the width in the thumbviewer directive as we do not have it but will be available when we maintain a separate collection of all the thumbnails being rendered by the thumb directive. Please note that `$scope.thumbnails` defined in the ThumbCtrl controller will not be helpful in this situation and can not be accessed within the directive as we are using an isolate scope. However, we can pass it as an attribute i.e. `data-thumbnails-count="{{thumbnails.length}}"` on the thumbviewer directive but that will be extraneous because we are already repeating over the same collection using the thumb directive. So we'll utilize what is being passed to `ngRepeat` later in the chapter.

Pay attention to `this` passed as a parameter to the `thumbnailStyle()` method which will carry a separate scope created for

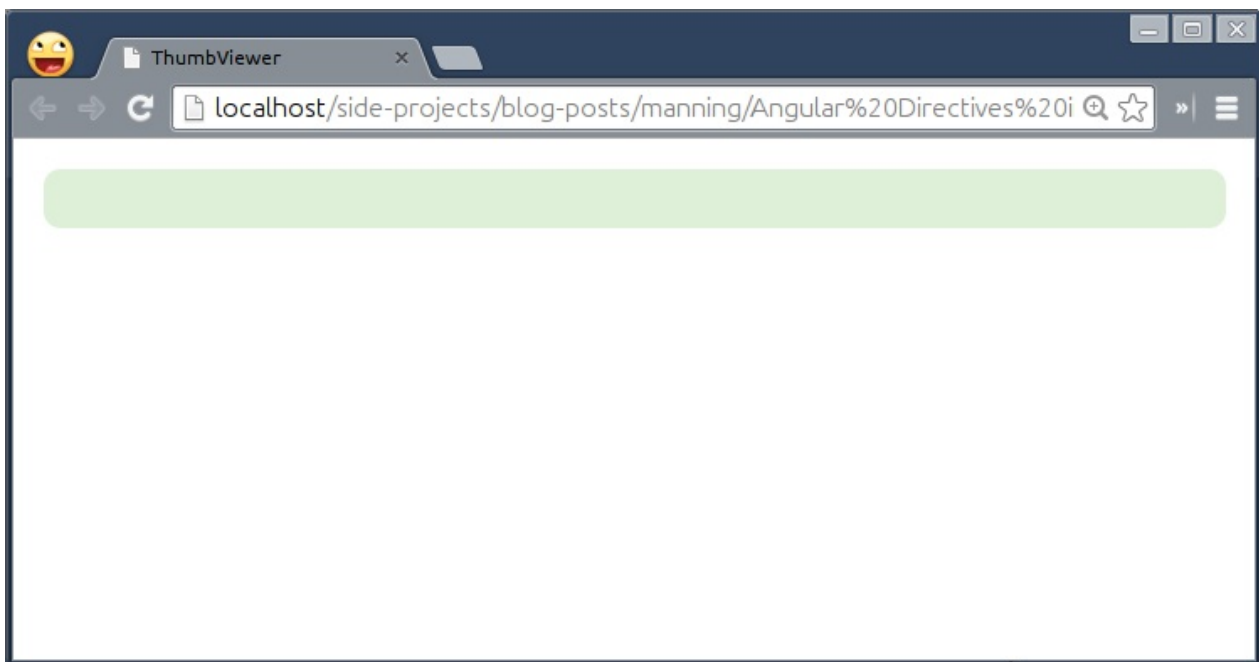
each `<thumb>` directive by the repeater. The scope will be required to figure out the currently selected thumb and animate it back and forth like carousel that we'll see in later sections. For now, add `wrapperStyle()` method in thumbviewer directive:

```
App.directive('thumbviewer', function() {
  return {
    restrict: 'EA',
    replace: true,
    templateUrl: 'thumbviewer.html',
    scope: {
      size: '=?',
      gap : '=?'
    },
    link: function(scope, element, attrs) {
      scope.wrapperStyle = function() {
        return {
          width  : '1000px',
          display : 'inline-block'
        };
      };
    }
  };
});
```

Then we'll write a new directive named `thumb` that is being repeated by `ngRepeat` to create multiple thumbnails. Add following in the same JS file.

```
App.directive('thumb', function() {
  return {
    restrict: 'EA',
    replace: true,
    templateUrl: 'thumb.html',
    scope: {
      active : '=?'
    },
    compile: function(tElement, tAttrs) {
      return function postLink(scope, element, attrs) {
        scope.thumbnailStyle = function(thumb) {
          return {
            width      : 150,
            marginBottom: 0
          };
        };
      };
    }
  };
});
```

As of now, there is no benefit of using compile method in this specific directive but it may be helpful in case we want to manipulate the template for the thumb directive later. Remember we were passing the thumb width and gap via attributes to the thumbviewer directive that will replace the hard coded value `150px` later to auto scale itself in order to accommodate all the thumbnails. Here is what you will see in the browser.



I'm sure you must be disappointed not to see all the thumbnails here that we added into thumbviewer directive in the DOM earlier. But if you recall what we have learned about `templateUrl` and `replace` options in Chapter 4 that the base element gets replaced by the template. So is there any way to include an external content into the template? Let us read on the following section to find out.

Transcluding Away

The `template` or `templateUrl` options are great to encapsulate markups that shape any directive but with a predefined template, the component may not be reusable. There are many real world scenarios wherein an external markup needs to be injected into the template to decorate the directive further. For instance, take an example of an accordion widget that let's you set up a heading for each accordion header but a content to be shown inside the accordion panel may vary depending upon the requirement. So it requires a provision to include varying content in the directive to fulfill the need, `transclude` option is the best deal for that. It allows us to include outside markup into the template smartly. There are two ways to transclude the content:

transclude: true

The easiest way is to just allow a directive to inject an arbitrary markup into the template with `transclude: true` option. This option simply informs a directive to get ready for transclusion. This follows whether an external markup needs to be injected as is or manipulated before getting injected.

Imagine your webapp requires a robust system to provide contextual feedback for typical user actions with flexible alert messages. Basically you need a container to put up error messages and the container itself should take care of its style appropriately. Let us write an alert directive to explore it further. Create *transclude.html* in *ch06/* directory as:

```
<html ng-app="TranscludeApp">
<head>
  <title>Transclude</title>
  <script src="../../bower_components/angular/angular.js"></script>
  <script src="../../js/ch06/transclude.js"></script>
  <link rel="stylesheet" type="text/css" href="../../bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body style="padding: 50px;">
  <alert data-state="warning"><b>Hang on..!</b> Document may take some time to upload.</alert>

  <alert data-state="success"><b>Well done..!</b> You jump to the next level.</alert>
</body>
```

```
</html>
```

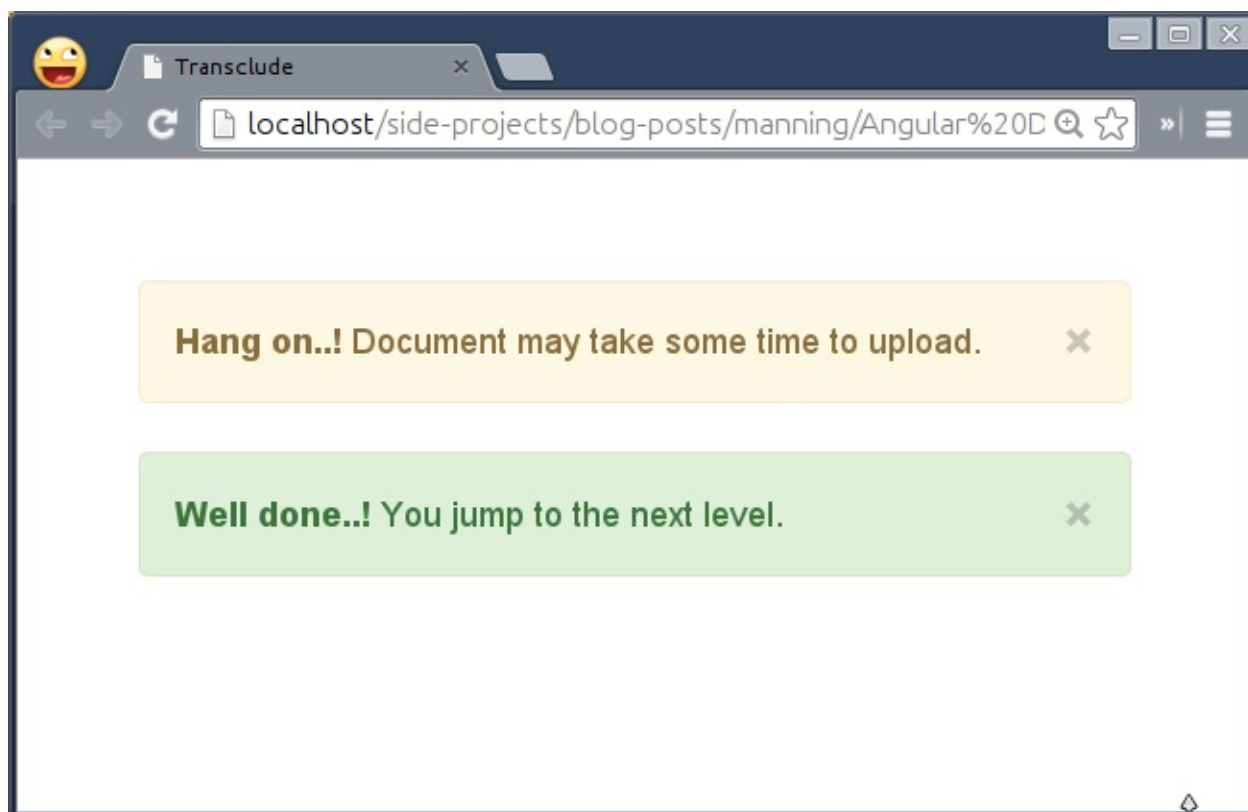
Notice we had created a custom element `alert` to hold an error message that will be transcluded into the template used. We can also set the state of an alert message to have different styles based on the context of the error. Add the following in *js/ch06/transclude.js*:

```
var App = angular.module('TranscludeApp', []);

App.directive('alert', function() {
  return {
    restrict: 'EA',
    replace: true,
    transclude: true,
    scope: {},
    template: function(element, attrs) {
      return '<div class="alert alert-' + ( attrs.state || 'info' ) + '" ng-hide="hidden">\
        <button type="button" class="close" ng-click="hidden = true">\
          <span aria-hidden="true">&times;</span>\
        </button>\
        <p ng-transclude></p>\
      </div>';
    }
  };
});
```

As expected we had set transclude option to true to enable transclusion. AngularJS comes with a special kind of directive named `ng-transclude` to let us control the placement of the transcluded content inside the template. Here we wanted to place an error message into the `<p>` element. At the end of the alert message, we have shown a close icon to hide it manually using `ngClick` and `ngHide` directives.

When the alert directive gets compiled, it's content/body will be transcluded inside the template where `ngTransclude` directive is defined and finally the directive declaration will be replaced by the template. Go ahead and give it a shot.



In case you want to manipulate the content before the transclusion, you can completely get rid of the `ngTransclude`

directive used in the template and introduce a `transcludeFn` method as a 5th argument in a link function. This method gives access to a cloned transcluded element which you can manipulate before injecting it into the template manually unlike `ngTransclude` directive (that does it automatically). Let's update the directive to see it in action so:

```
App.directive('alert', function() {
  return {
    restrict: 'EA',
    replace: true,
    transclude: true,
    scope: {},
    template: function(element, attrs) {
      return '<div class="alert alert-' + ( attrs.state || 'info' ) + '" ng-hide="hidden">\
        <button type="button" class="close" ng-click="hidden = true">\
          <span aria-hidden="true">&times;</span>\
        </button>\
      </div>';
    },
    link: function(scope, element, attrs, NullController, transcludeFn) {
      transcludeFn(scope, function(tContent) {
        element.append(tContent);
      });
    }
  }
});
```

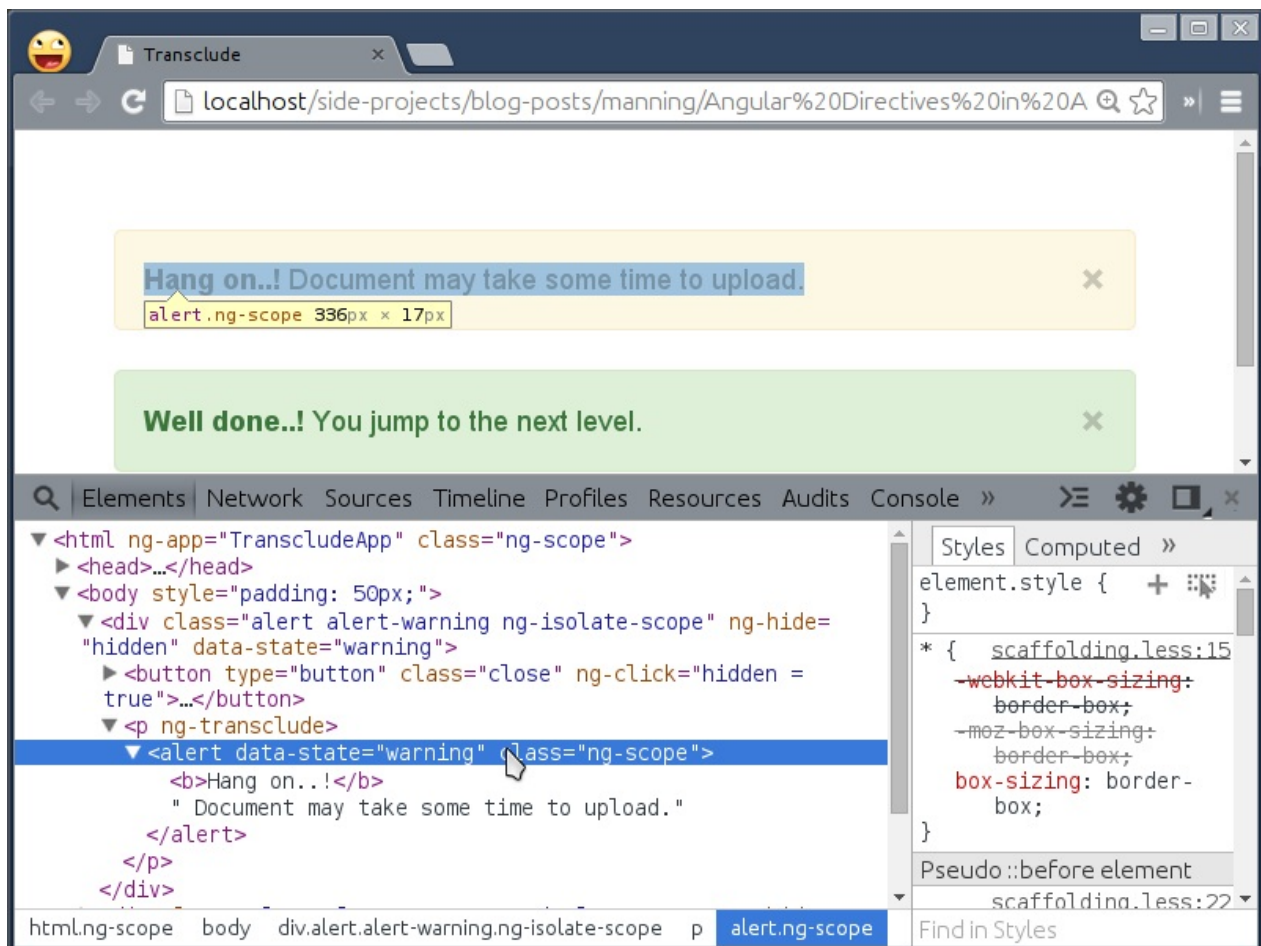
The `NullController` is just a placeholder required to be the 4th argument for transclusion to work. However, there is no controller that we want to use here so just passing `NullController` to avoid syntax errors. We'll learn about controllers more in later sections.

Notice we removed `ngTransclude` directive from the template and have `transcludeFn` method to which we can pass a scope, the content will be evaluated against (to update all bindings if used). However, the scope parameter is optional if no bindings available. This function returns a cloned transcluded element but manipulated which we can then be appended inside the element i.e. `div.alert`. However, we are not manipulating the cloned transcluded content in this example but you got the point – you can add, remove, or manipulate it however you want. You could try (as an exercise) to make the directive smart enough to understand the meaning of the transcluded content to automagically style the alert irrespective of data-state attribute passed. For instance, if an alert message consists of words such as 'sorry', 'failed', 'error', and so on then the alert should apply `alert-danger` class to highlight something has gone wrong.

So the takeaway here is that use this option only if you want to transclude the content of the element on which the directive was binded. The AngularJS internally does not use it but there are many open source widgets built on top of AngularJS find it really useful. Some of the AngularUI Bootstrap's components such as accordion, carousel, datepicker, modal, tab, and so on are using it extensively along with `templateUrl` because this gives flexibility to extend the respective component with external markups. Sometimes transcluding the entire element is the requirement, let us find out how to achieve that in the next section.

transclude: 'element'

This option allows to transclude the whole element along with its content on which the directive was binded. If you change `transclude: true` option to `transclude: 'element'` in the previous example and reload the browser, you will not see any visual change but the underlining template has been modified as shown in the following figure:



You could see that the alert directive itself goes into the template (under `<p>` element) but does not get recompiled further to avoid never ending recursive compilation process – luckily AngularJS takes care of that..! This option is mainly used by the built-in `ngRepeat` directive where it needs to copy the same repeatable element to render it over and over again. Other built-in directives also such as `ngInclude`, `ngIf`, and `ngSwitch` use this option to include, remove, and recreate external or default portion of markups. However, this option is rarely used outside.

NOTE: Alternatively, you could also use `transcludeFn` in a compile phase as well but is now deprecated since angular version 1.2+ and not recommended.

Fixing Page Thumbviewer directive using transclusion

In the previous section on Link vs Compile, the thumbnails were not shown by the thumbviewer directive because they got replaced by the template. We can fix it with transclusion in both the directives. Go head and add `transclude: true` for both as shown:

```
App.directive('thumbviewer', function() {
  return {
    restrict: 'EA',
    replace: true,
    templateUrl: 'thumbviewer.html',
    transclude: true,
    scope: {
      size: '=?',
      gap: '=?'
    },
    link: function(scope, element, attrs) {
      scope.wrapperStyle = function() {
        return {
          width: '1000px',
          display: 'inline-block'
        };
      };
    }
  };
});
```

```

    };
  }
};
});

App.directive('thumb', function() {
  return {
    restrict: 'EA',
    replace: true,
    templateUrl: 'thumb.html',
    transclude: true,
    scope: {
      active : '=?'
    },
    compile: function(tElement, tAttrs) {
      return function postLink(scope, element, attrs) {
        scope.thumbnailStyle = function(thumb) {
          return {
            width : 150,
            marginBottom: 0
          };
        };
      };
    }
  };
});
});

```

Then update the template to inject the arbitrary markup into the template using `ng-transclude` directive as:

```

<script type="text/ng-template" id="thumbviewer.html">
  <div class='thumbnail-container bg-success img-rounded'>
    <div class='thumbnail-wrapper' ng-style="wrapperStyle()" ng-transclude></div>
  </div>
</script>

```

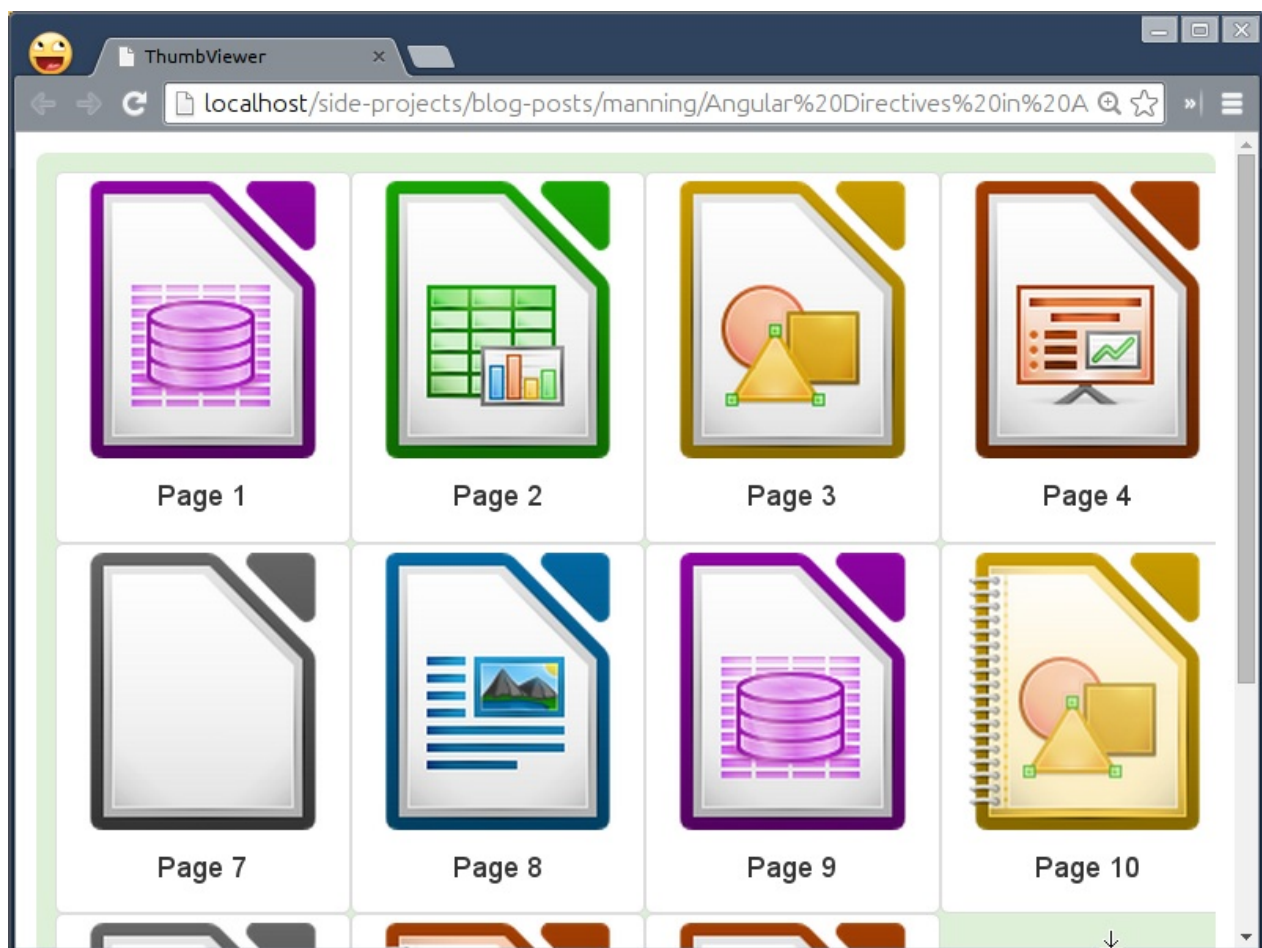
Note, this will transclude all the thumbnails that `ngRepeat` spewed in the DOM. Now update the thumb template to include an image and a caption added for each thumb while declaring it so:

```

<script type="text/ng-template" id="thumb.html">
  <div class="thumbnail pull-left" ng-style="thumbnailStyle(this)" ng-transclude></div>
</script>

```

You will be excited to see thumbViewer quite working as per the following figure.



Controlling behavior and sharing

In Chapter 2, we learned about AngularJS Controller in detail and how it helps to avoid polluting global scope. Directive also has a special type of controller called *Directive Controller*. However, it is not different than the normal controller. In fact, you can inject the normal controller into the directive definition to use it as a directive controller. There are two ways to define Directive Controller on the directive. First is to define it inline as:

```
App.directive('foo', function() {
  return {
    restrict: 'EA',
    template: '<div>{{first}} {{FooCtrl.last}}</div>',
    controller: function($scope, $element, $attrs) {
      $scope.first = 'Foo';
      this.last = 'Bar';
    },
    controllerAs: 'FooCtrl'
  };
});
```

Given the fact that it consumes the same instance of a controller, we are allowed to access everything that a normal controller can do such as `$scope`, `$element`, and `$attrs`. However, these parameters behave slightly different than in a link function. In the directive controller, these are not just parameters (as they are in the link phase), instead we are asking the injector (D.I system) to create instances of all the dependencies of the controller in terms of `$scope`, `$element`, and `$attrs`. Hence it's fine to rearrange them but can not be renamed.

Here `controllerAs` syntax is similar to what we could use in the DOM i.e. `<div ng-controller='FooController as FooCtrl'></div>` which creates an alias for methods/properties assigned on the controller's instance. Since `$scope` and `this` both are not same and that is why have used them differently in the view/template above. The `$scope` is an object associated

with the controller to which we can bind properties/methods whereas `this` is an instance of the controller (which refers to `FooCtrl`). However, `this` should be used to bind properties/methods in the controller definition and `FooCtrl.property` should be used in the view in order to access them.

In addition to it, we define the controller separately as a normal controller and inject it later in the directive definition as follows:

```
App.controller('FooCtrl', function($scope, $element, $attrs) {
  $scope.first = 'Foo';
  this.last = $scope.last = 'Bar';
});
```

Then use it within the directive so:

```
App.directive('foo', function() {
  return {
    restrict: 'EA',
    template: '<div>{{first}} {{FooCtrl.last}}</div>',
    controller: 'FooCtrl',
    controllerAs: 'FooCtrl'
  };
});
```

So it's upto you what syntax to use next time. Given that both the controller as well as link functions are quite similar, what may be functional differences between them?

- The directive controller basically enables inter-communication between different directives. So one way to look at it is as an API controller to define a set of actions, maintain states, and set up a communication bridge across directives.
- Sometimes flooding the link function with utility methods which are not used within a view can be moved into the controller. This will make the link function slim, since it is only used to bind DOM events and set up custom watchers.

NOTE: Directive controller should not be used for any sort of DOM manipulation which is an anti-pattern. Also, properties/methods defined in the directive controller can not be accessed in the link function of the same directive, however, it can be accessed by other directives via require option which we'll learn in following sections.

Meanwhile, Let us fix the thumbViewer directive using the controller option in the next section.

Fixing Page Thumbviewer directive using Directive Controller

Earlier we hardcoded the thumbnailContainer's width to `1000px` and that's why the thumbnails were reflowed to the next row. To solve this problem, we need to calculate the width dynamically. Add a new controller in *thumbviewer.js* as:

```
App.controller('ThumbViewerCtrl', function($scope) {
  this.thumbSize = $scope.thumbSize = $scope.size || 150;
  this.thumbGap = $scope.thumbGap = $scope.gap || 10;
});
```

Here we have cached the thumbnail size and it's gap on the scope object as well as the instance of the controller in order to access it within thumbviewer as well as thumb directives. Now it's time to introduce the controller in thumbviewer directive as follows:

```
App.directive('thumbviewer', function() {
  return {
    restrict: 'EA',
```

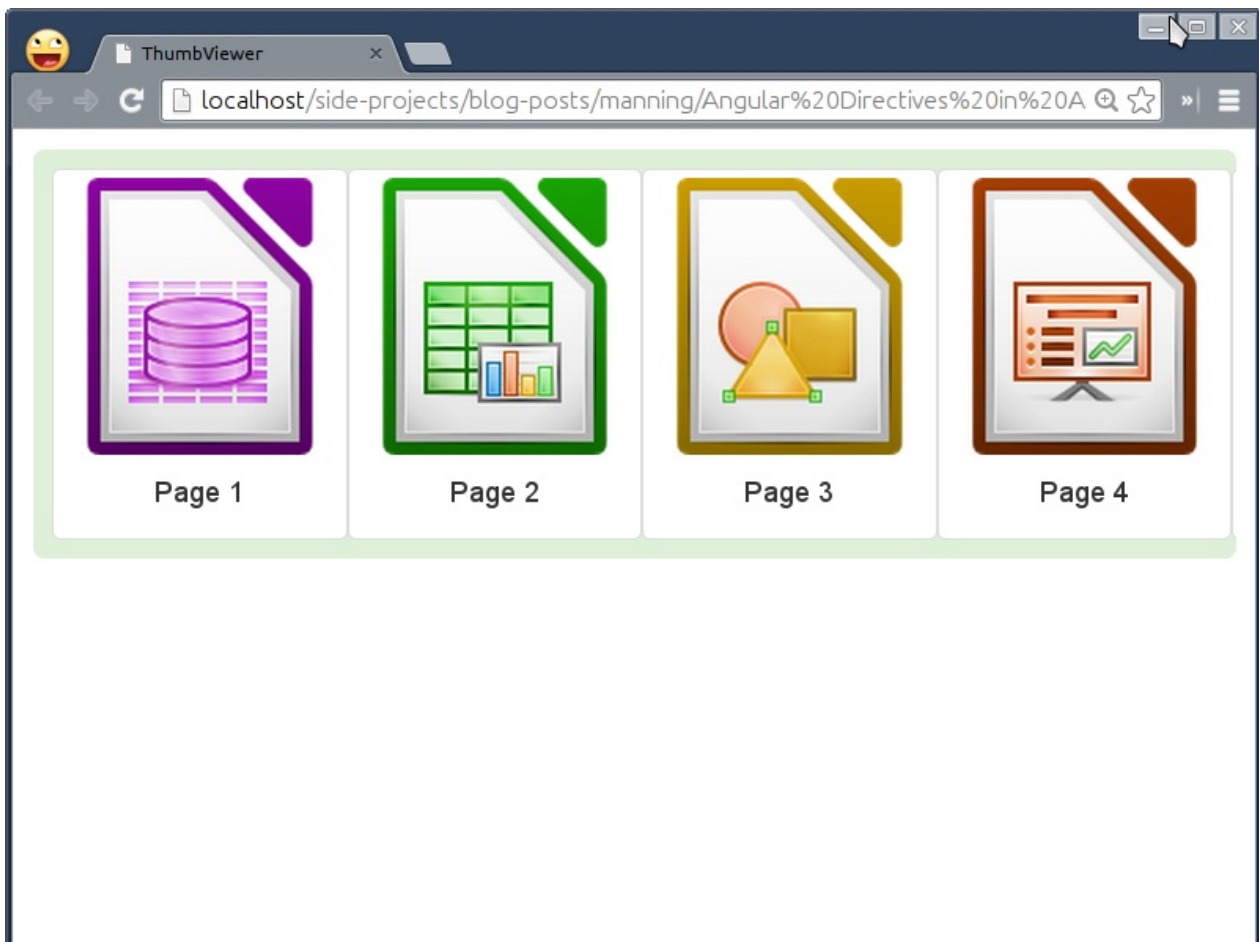
```

replace: true,
templateUrl: 'thumbviewer.html',
transclude: true,
controller: 'ThumbViewerCtrl',
scope: {
  size: '=?',
  gap: '=?'
},
link: function(scope, element, attrs) {
  scope.wrapperStyle = function() {
    var totalSize = scope.thumbSize * 15;
    var totalGap = scope.thumbGap * (15 - 1);

    return {
      width: totalSize + totalGap + 'px',
      display: 'inline-block'
    };
  };
}
};
});

```

For now, we have hardcoded a total number of thumbnails loaded in the `wrapperStyle()` method because that information is not available yet but we'll be obtained via `require` option in the next section. However, this will at least render all the thumbnails in a single row as shown in the following figure:



Let us look at the final piece of the puzzle to access the controller in the `thumb` directive in the following section so that we will be able to dynamically calculate the total number of thumbnails which is currently hard-coded as we saw before.

Enabling intercommunication within directives

Once properties or methods published by the Directive Controller, other directives need to ask for the same to access those

methods and is possible only with `require` option. Once you mention any directive on the `require` option of another directive, its controller is introduced as a 4th parameter in a link function of it. The directive having a directive controller can be binded to the same element or its parent as shown:

```
App.directive('foo', function() {
  return {
    restrict: 'EA',
    require: 'sameElement', // or '^parentElement'
    link: function(scope, element, attrs, Ctrl) {}
  };
});
```

Sometimes there is no hard dependency on the directive controller so we can prefix it with a question (?) sign to make it optional, otherwise an error is thrown in case it is not found. You can even require multiple directive controllers in an array format, i.e. `require: ['someElement', '^someParentElement1', '?^someParentElement2']`.

This also changes the way you access each controller in a link method that is `ctrl[0]`, `ctrl[1]`, and `ctrl[2]` for `someElement`, `someParentElement1`, and `someParentElement2` respectively. Let us see how we can use it with thumbviewer directive.

Fixing thumb directive by require-ing the directive controller

As we know already that the thumbviewer directive has a total number of thumbs hardcoded previously because we were expecting that information from the thumb directive once all the thumbnails are loaded. With `require` option, we now can update thumbs collection in the thumb directive which was defined in the ThumbViewerCtrl controller earlier. First of all update ThumbViewerCtrl as:

```
App.controller('ThumbViewerCtrl', function($scope) {
  this.thumbs = $scope.thumbs = [];
  this.thumbSize = $scope.thumbSize = $scope.size || 150;
  this.thumbGap = $scope.thumbGap = $scope.gap || 10;

  this.addThumb = function(thumb) {
    $scope.thumbs.push(thumb);
  };

  this.selectThumb = $scope.selectThumb = function(selectedThumb) {
    var selectedIndex = $scope.thumbs.indexOf(selectedThumb);

    angular.forEach($scope.thumbs, function(thumb) {
      if (thumb.active) thumb.active = false;
    });
    selectedThumb.active = true;
  };
});
```

Now require thumbviewer directive into the thumb directive definition to access its controller and call `addThumb()` method which we defined in the thumbviewer directive controller before. The moment any thumb directive instance gets compiled, its scope will be added into the collection, `thumbs` – this way we'll have a total number of thumbnails rendered at the end.

```
App.directive('thumb', function() {
  return {
    restrict: 'EA',
    replace: true,
    templateUrl: 'thumb.html',
    transclude: true,
    require: '^thumbviewer',
    scope: {
      active : '=?'
    },
    compile: function(tElement, tAttrs) {
```

```

return function postLink(scope, element, attrs, thumbviewerCtrl) {
    thumbviewerCtrl.addThumb(scope);

    scope.isFirstThumb = function(thumb) {
        return thumbviewerCtrl.thumbs.indexOf(thumb);
    };

    scope.thumbnailStyle = function(thumb) {
        return {
            width      : thumbviewerCtrl.thumbSize + 'px',
            marginLeft : ( scope.isFirstThumb(thumb) === 0 ? 0 : thumbviewerCtrl.thumbGap ) + 'px',
            marginBottom: 0
        };
    };

    scope.select = function(selectedThumb) {
        thumbviewerCtrl.selectThumb(selectedThumb);
    };
};
};
});

```

Note that the link method runs for every thumb item with an isolate scope to maintain active state for each thumb separately. Now that we have a total number of thumbs rendered, we can use this information in the `wrapperStyle()` method to dynamically calculate the width for the same, replacing the hardcoded values used temporarily. So, update it as follows:

```

scope.wrapperStyle = function() {
    var totalSize = scope.thumbSize * scope.thumbs.length;
    var totalGap  = scope.thumbGap * (scope.thumbs.length - 1);

    return {
        width      : totalSize + totalGap + 'px',
        display    : 'inline-block'
    };
};

```

Then bind a click handler on each thumbnail as:

```

<script type="text/ng-template" id="thumb.html">
    <div class="thumbnail pull-left" ng-style="thumbnailStyle(this)" ng-class="{active: active}" ng-click="select(this)">
</script>

```

Note that the `this` parameter passed to `select()` method refers to an isolate scope of each thumbnail. Then we have used the `active` property to apply `active` class on the same to highlight. Finally, we need to apply a border if any thumbnail is selected as:

```

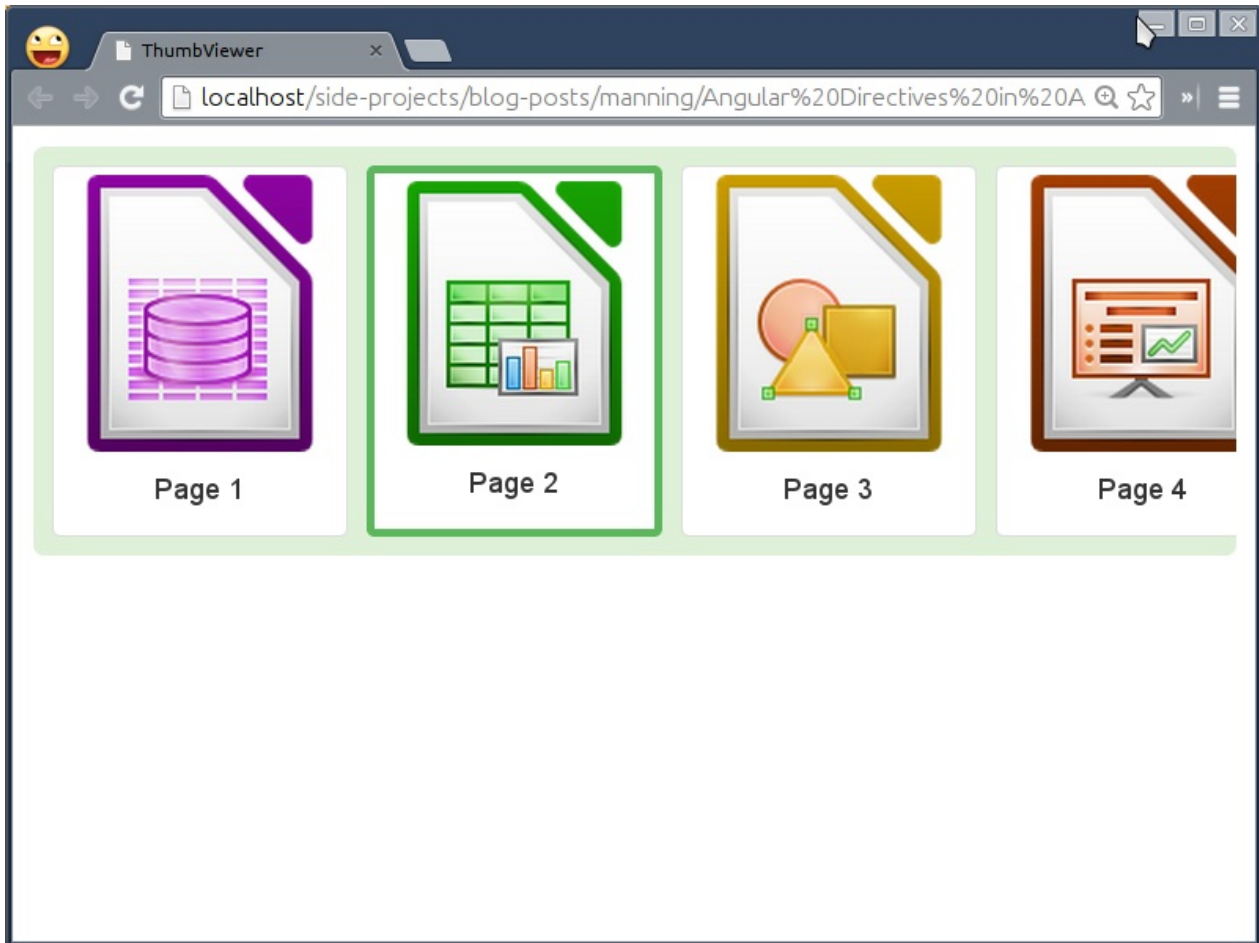
<style type="text/css">
    .thumbnail-container {
        display: inline-block;
        width: 100%;
        overflow: hidden;
        padding: 10px;
    }

    .thumbnail.active {
        border: 4px solid #5cb85c;
    }

    img { width: 100%; }
</style>

```


That's it! Our thumbnailviewer directive is almost ready as shown in the following figure.



You can go ahead and try clicking couple of thumbs and see how it works. Meanwhile, let us walk through some important advices as:

- The only purpose of `require` option is to inject the directive controller into the directive and read/write shared properties/methods.
- This option basically takes a single directive controller as a string and multiple directive controllers in the form of an array.
- Requiring any directive, injects its controller into the linking function as a fourth parameter.
- Directive controller is a place to put business logic of the directive that can be shared as well. However, it will be helpful only if two directives have parent-child relationship or are applied on the same element. To share the logic between different directives that do not have such relationship, AngularJS services should be used.
- The methods/properties defined on this object in the directive controller can not be accessed by the same directive. Use `scope` object instead. That's why we binded `thumbSize` and `thumbGap` properties both on `this` as well as `scope` objects in `ThumbViewerCtrl`, so that it will be read by `thumb` as well as `thumbviewer` directives.

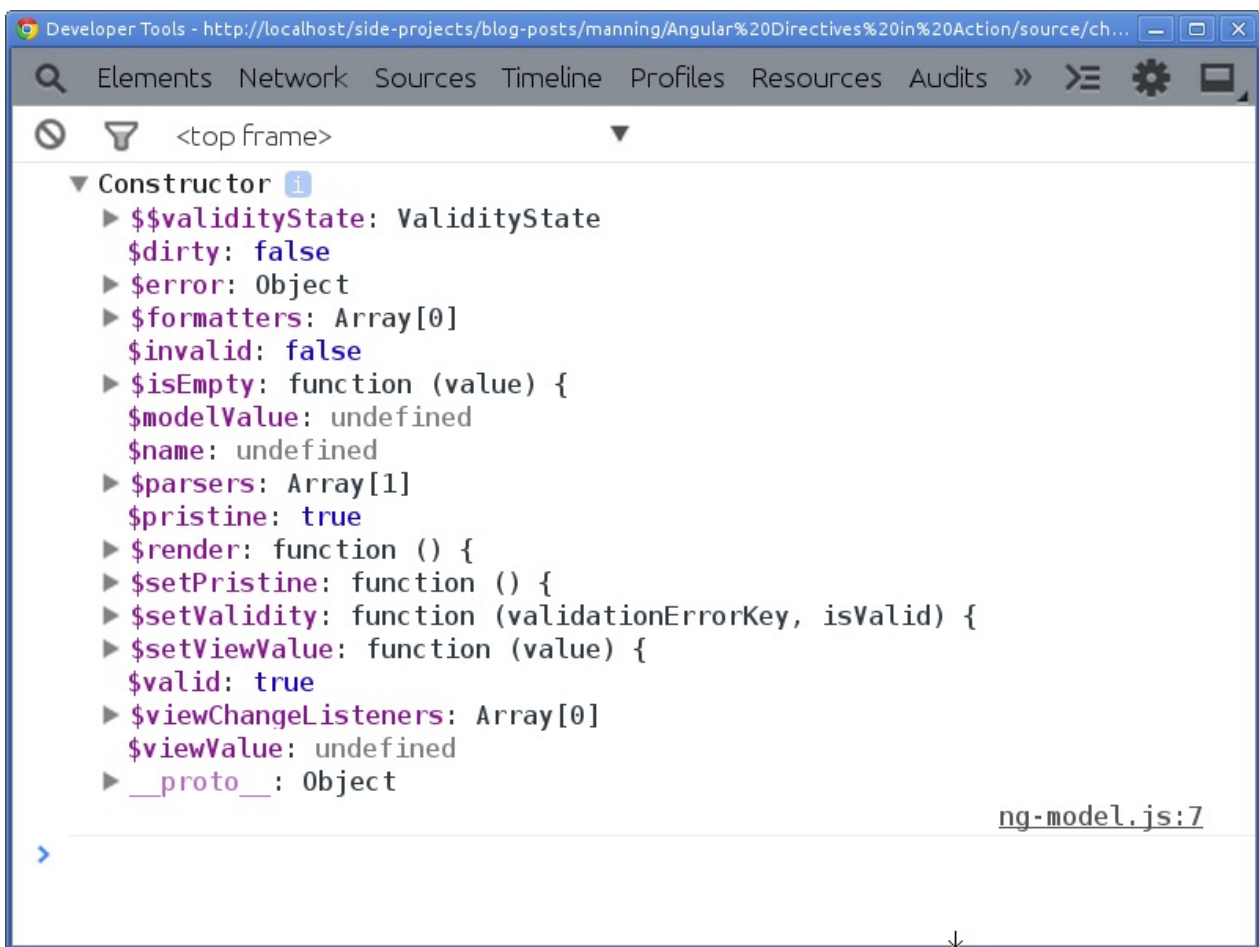
All that means is that `require` option allows us to inject the directive controller of any directive, off-course, including built-in directives. One of them is `ngModel` directive, let us explore.

Requiring ngModelController

Some of built-in directives such as `ngModel`, `ngInclude`, and `ngSwitch` expose directive controllers but are not much extensible except `ngModelController`. That is why we are going to explore `ngModelController` in this section.

As we have learned about `ngModel` in chapter 2 that is mainly useful for binding the view into a model, often used with form elements but you can use it with any directive that holds some sort of value (object or primitives) that is used by other

AngularJS components. The `ngModelController` provides access to various services for data bindings, validations, model formatting and parsing as shown in the following figure.



We'll go through all of them one by one, see their benefits, and use some of them in `thumbviewer` directive if necessary. Imagine there is a Job application form you want candidates to fill in and HTML form is the great example to dig `ngModelController` deep. So, let us quickly create a simple example to see how it works. Create `ch06/ng-model.html` as:

```
<html ng-app="ngModelApp">
<head>
  <title>ngModel Controller</title>
  <script type="text/javascript" src="../../bower_components/angular/angular.js"></script>
  <script type="text/javascript" src="../../js/ch06/ng-model.js"></script>
  <link rel="stylesheet" type="text/css" href="../../bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body>
  <form name="MyForm" role="form">
    <div class="form-group">
      <label class="control-label" for="email">Email address</label>
      <input type="email" name="email" class="form-control" placeholder="Enter email" ng-model="email">
    </div>
  </form>
</body>
</html>
```

The `ngModel` directive optionally relies on the `form` directive so that its controller keeps track of form elements' states that it holds in ways whether user has interacted with the form, whether the form is correctly filled, and so on. Please note that form and form elements should have `name` attribute on them to reassure their states and errors.

\$dirty vs \$pristine

Imagine you want your visitors to get excited to fill in the Job Application form with some fancy message but want to hide it

when they started interacting with the form. That means we want to show an informative message first and hide it later when the form is dirty which we can be done with `$dirty` and `$pristine` respectively. The `$pristine` state turns false when user interacts with the form control and makes `$dirty` state to true. Access both states on the form as, `MyForm.$dirty` and `MyForm.$pristine`.

In addition, it is also possible to find the same for each form control. The only requirement is the name attribute should be applied on it. Assuming the form control has `name='email'`, its states can be accessed as `MyForm.email.$dirty` and `MyForm.email.pristine`.

Now, let us add two alert messages into the form element so:

```
<div class="alert alert-success" ng-hide="MyForm.$dirty"><b>Want to Join a Dream Company?</a></div>

<div class="alert alert-info" ng-hide="MyForm.$pristine"><b>Focus on your strengths to attract more recruiters!</a></div>
```

Here you will see the first message initially and as you type, it will be replaced by the second message. If a value of any of the form controls in the form is modified, the form becomes dirty. By default it is pristine. With `$setPristine()` method, it is possible to bring it to its original condition, however calling the same on an individual form control that is

`MyForm.email.$setPristine()` will set itself to its pristine state but not the form. In reverse, calling it on the form that is `MyForm.$setPristine()` will set all the form controls (including the form) to their pristine states. This is really useful, especially, after form submission when we clear the form.

Moreover, AngularJS also applies `ng-pristine` or `ng-dirty` CSS classes on the form controls and the form element that can be used to specify styles in CSS to emphasize visually.

`$invalid` and `$valid`

Quite similar to the preceding states, this actually validates or invalidates a form control as well as the form. When the form element is compiled by AngularJS compiler, it is passed through the `$validators` pipeline (that we'll see in the later sections) to validate it. This pipeline consists of various validators methods such as `required`, `minlength`, `maxlength`, and so on and each validator should return true when processed, otherwise it invalidates the form control which in turn invalidates the form itself.

Imagine we may want to give immediate feedback to users if they did not fill out details as expected such as wrong email address in this case as:

```
<div class="form-group" ng-class="{ 'has-error': MyForm.email.$invalid }">
  <label class="control-label" for="email">Email address</label>
  <input type="email" name="email" class="form-control" placeholder="Enter email" ng-model="email">
</div>
```

In here, we are just checking if an email address is correct then apply `has-error` class, otherwise prompt users with an error message. AngularJS automatically adds `ng-valid` or `ng-invalid` CSS classes as per the validation on each form control including the form.

Also note that AngularJS automatically validates values based on input type such as email, number, required attribute, and so on but sometimes we need our own logic to validate the input which is covered in the following section.

`$setValidity`

Along with default validations, you can also define a custom set of validations to validate inputs. Using `$setValidity()` method, we can define a type of validation and its state. Imagine you want your users to give instant feedback if they entered already used email address.

Let us write a custom directive named `isUnique` for the same as:

```
App.directive('isUnique', function() {
  return {
    restrict: 'A',
    require: 'ngModel',
    link: function(scope, element, attrs, ngModelCtrl) {
      ngModelCtrl.$setValidity('uniqueEmail', false);
    }
  };
});
```

And apply this directive on our email input element so:

```
<input type="email" name="email" class="form-control" placeholder="Enter email" ng-model="email" is-unique>
```

If the existing email is used, the `ng-invalid` and `ng-invalid-unique-email` CSS classes will be applied on the element by AngularJS. For now, we have by default set the validity to false that will apply `has-error` CSS class on the form element (via `ngClass` condition added before) to prompt users before they even type anything, however we'll fix this problem in later sections with a custom validation method.

`$modelValue` and `$viewValue`

The data models we define using `ngModel` directive has two values in it i.e. `modelValue` and `viewValue`. They are two sides of the same coin. The basic difference between the two is that the `viewValue` is what user sees in the DOM (actual string value in the view) and `modelValue` represents the value in the model (that the control is bound to) but may not be the same as the `viewValue`.

Imagine you want to ask users if they are willing to relocate. Essentially we are going to read/write true or false boolean values but in the DOM we might want to show relevant text instead. So when `$modelValue` is true, the `$viewValue` will be 'Yippi..!' in our case. Let us try that out by adding following in the form as:

```
<div class="form-group">
  <label class="control-label">Willing to Relocate?</label>
  <input type="checkbox" name="relocate" ng-model="relocate" will-relocate> {{MyForm.relocate.$viewValue}}
</div>
```

Here our `ngModel` holds a model named `relocate` and it has `will-relocate` directive applied that will allow us to update `modelValue/viewValue` accordingly. Write `willRelocate` directive as follows:

```
App.directive('willRelocate', function() {
  return {
    restrict: 'A',
    require: 'ngModel',
    priority: 1,
    link: function(scope, element, attrs, ngModelCtrl) {
      ngModelCtrl.$render = function() {
        ngModelCtrl.$modelValue = ngModelCtrl.$modelValue || false;
        ngModelCtrl.$viewValue = ngModelCtrl.$modelValue ? 'Yippi..!' : 'Nope, not ready!';
      };
    }
  };
});
```

The `$render` method is called when the directive gets compiled first time and is responsible to update the view. Here we are overriding it to update the model and view values if not set. Please note that the priority should be more than 0 because

ngModel directive also has a zero priority that resulted in execution of willRelocate directive prior to ngModel and the original `$render` method (defined in ngModel directive) will not be superseded by ours. So providing higher priority leads to reverse the order of execution of both directives. So when the directive gets compiled, the modelValue will be set to false by default which in turn will update the viewValue obviously to 'Nope, not ready!'.

\$parsers

If you are curious, you might have tried toggling the willRelocate checkbox and may be little puzzled to see the value next to it does not change when checkbox is selected. The `$parsers` is a collection of methods to execute as a pipeline whenever view value changes. Remember to return the value in each of the parser method which will be used by the subsequent parser method so that last returned value will eventually populate the model.

Now let us update willRelocate directive to fix the preceding issue:

```
ngModelCtrl.$parsers.push(function(viewValue) {
  var modelValue = viewValue;
  ngModelCtrl.$viewValue = modelValue ? 'Yippi..!' : 'Nope, not ready!';

  return modelValue;
});
```

Here we are implementing our own parsing logic to alter the viewValue. Note that whenever the checkbox is checked or unchecked, both `$modelValue` and `$viewValue` set to true or false respectively and `$parsers` method will be run after that. If the viewValue is true, it will be transformed to it's string representation and returned the original viewValue to populate the model that is true or false accordingly. Mainly, Parsers can be used for validation purpose, so it's time to fix the isUnique directive. Update it as:

```
App.directive('isUnique', function() {
  return {
    restrict: 'A',
    require: 'ngModel',
    link: function(scope, element, attrs, ngModelCtrl) {
      ngModelCtrl.$setValidity('uniqueEmail', true);

      ngModelCtrl.$parsers.push(function(viewValue) {
        ngModelCtrl.$setValidity('uniqueEmail', !(viewValue === 'foo@bar.com'));

        return viewValue;
      });
    }
  };
});
```

Here you could have a collection of email Ids or trigger an XHR request to check the entered email is unique and not available on the DB server.

\$formatters

What if you apply checked attribute on the checkbox, did you see any abnormal behavior? I assumed that the checkbox is checked in your case but the text next to it still says, 'Nope, not ready!'. That is because we had now changed the model value but ngModelController is not notified about the change to update the view accordingly. That's the reason `$formatters` exist which is a collection of methods to execute as a pipeline whenever model value changes. Add following in the link method of isUnique directive:

```
ngModelCtrl.$formatters.push(function(modelValue) {
  ngModelCtrl.$setValidity('uniqueEmail', true);

  return modelValue;
});
```

This snippet will be useful when we reset the form after submission. Also in the `willRelocate` directive so:

```
ngModelCtrl.$formatters.push(function(modelValue) {
  var viewValue = (modelValue || ( !angular.isDefined(modelValue) && element.prop('checked') )) ? 'Yippi..!' : 'Nope, r
  ngModelCtrl.$modelValue = viewValue === 'Yippi..!';

  element.prop('checked', ngModelCtrl.$modelValue);

  return viewValue;
});
```

Similar to `$parsers`, we get the `modelValue` first, mutate it to have `viewValue`, and update the checkbox's state if `$modelValue` is truthy. Finally return the `viewValue` to be used by the subsequent `$formatters` method if available so that last returned value would eventually populate the `$viewValue`. The only difference with respect to `$parsers` is that the `$render` method will be called in this case after all the `$formatters` are run to update the view.

`$viewChangeListeners`

This method triggers every time a view value changes, so it's a good place to perform DOM manipulation. It calls after `$formatters` so there is no much difference between the two but probably useful to keep all change listeners at one place. You can set up a listener as follows:

```
ngModelCtrl.$viewChangeListeners.push(function() {
  // do something interesting
});
```

`$setViewValue`

If you noticed `willRelocate` directive, we have been morphing `$viewValue` through parsers and updating `$modelValue` through formatters based on the view value. However, sometimes both `$modelValue` and `$viewValue` are equal, for instance, email address. To refill the email form control with the default email address, we can use `$setViewValue` method that updates `$viewValue`, runs all the parsers, and then sets `$modelValue`. In reverse, when `$modelValue` changed by `$setViewValue`, it runs formatters, sets the `$viewValue`, and then triggers `$render`. Ultimately, `$setViewValue` method completes the cycle of updating model as well as view values.

`$setPristine`

It is a very common use case to reset the form to it's clean state after form submission. The `$setPristine()` method paves a way for such scenarios. Let us add Apply button at the end of the form to submit the Job application form:

```
<button type="submit" class="btn btn-primary" ng-click="email = '';relocate = false;MyForm.$setPristine();">Apply</butt
```

Although it looks simple, but there are lot of things going on under the hood. Setting an email model to empty string causes to trigger formatters of the `isUnique` directive that validates it. Updating `relocate` to false leads to unchecking the checkbox if checked, running formatters to reset the `$modelValue`, and then rendering both via `$render` call. At the end, we set the form controls to pristine states with `$setPristine` method that simply removes `ng-dirty` class replacing it with `ng-pristine`. In short, it takes form controls back in time as if it just compiled and untouched. Additionally, you can call the same method for each form control individually. Now that we understood enough about `ngModelController`, let us use it with `Spinner` directive to improve it.

Using `ngModelController` with `Spinner` Directive

In the previous chapter, we promised to use `ngModel` with `Spinner` directive to get rid of the external callbacks to get the spinner's value or state. So, first copy `ch05/scope-isolate.html` as `ch06/ng-model-spinner.html` and then update the `ngApp` attribute as:

```
<html ng-app="nmSpinnerApp">
<head>
  <title>ngModel Spinner</title>
  <script src="../../bower_components/angular/angular.js"></script>
  <script src="../../js/ch06/ng-model-spinner.js"></script>
  <link rel="stylesheet" type="text/css" href="../../bower_components/bootstrap/dist/css/bootstrap.css">
</head>
```

This will basically avoid any clash with duplicate modules while testing it with Karma and in Protractor later. Then reference the correct JavaScript under head tag as shown before.

Then apply `ngModel` directive to rating spinner replacing `data-default` attribute as:

```
<adia-spinner ng-model="rating" data-step="1" data-min="1" data-max="5" on-change="getRating()"></adia-spinner>
```

The `getRating()` method does not require any parameter as it now works as just a change listener to perform any action when the rating changes. Now the current rating can be accessed directly as `$scope.rating` even without having the `onChange` listener.

Then update weekday spinner with:

```
<adia-spinner ng-model="weekday" data-type="list" data-list="arrWeekday" on-change="getWeekday()"></adia-spinner>
```

Here `weekday` model represents an index of `arrWeekday` that we are going to use in `ScopeCtrl` later. That's it from the DOM perspective. Now copy `js/ch05/scope-isolate.js` to `js/ch06/ng-model-spinner.js` and update the module so:

```
var App = angular.module('nmSpinnerApp', []);
```

After that require `ngModelController` in the `adiaSpinner` directive and inject the controller in a link method as follows:

```
require: 'ngModel',
link: function(scope, element, attrs, ngModelCtrl) { }
```

Then move `scope.opt` object reference into `$render` method as:

```
var isList = scope.type === 'list';

ngModelCtrl.$render = function() {
  scope.opt = {
    type      : scope.type || 'range',
    default   : ngModelCtrl.$modelValue || 1,
    interval  : scope.interval || 1,
    min       : scope.min || 1,
    max       : scope.max || Infinity,
    list      : scope.list || []
  };

  if (isList) {
    scope.opt.default = ngModelCtrl.$modelValue ? scope.opt.default : 0;
    scope.opt.min = 0;
  }
}
```

```

    scope.opt.max = scope.opt.list.length - 1;
  }

  ngModelCtrl.$setViewValue(scope.opt.default);
};

```

Nothing has changed much, we are simply replacing the occurrences of `scope.default` with `ngModelCtrl.$modelValue` and finally updating the `viewValue` with the help of `setViewValue` call. Then update `inc` and `dec` methods slightly so:

```

scope.inc = function() {
  scope.opt.default+= scope.opt.interval;
  ngModelCtrl.$setViewValue(scope.opt.default);
};

scope.dec = function() {
  scope.opt.default-= scope.opt.interval;
  ngModelCtrl.$setViewValue(scope.opt.default);
};

```

Earlier we used to call `notify()` method to trigger `onChange` callback but as we learned before that the `$setViewValue` triggers `$viewValueChangeListeners` methods in the end whenever view value changes, that means we can keep the `notify()` call at one place as follows:

```

ngModelCtrl.$viewChangeListeners.push(function() {
  if (typeof scope.onChange() === 'function') {
    scope.onChange();
  }
});

```

Also note that the spinner's callback does not require any parameter now as `ngModel` directive takes care of that. Hence the `onChange` callback becomes merely the change event listener for the spinner similar to `onchange` event for select HTML element.

Then we'll have a custom watch method to watch over the list so that the widget will be re-rendered if the list is modified dynamically as:

```

scope.$watch('list', function() {
  ngModelCtrl.$render();
});

```

This is very simplest form of `$watch` expression that re-renders the spinner UI whenever list model changes, we are going to cover in detail in the next chapter. Finally we'll update `ScopeCtrl` to define `arrWeekday` used before in the DOM and their callbacks so:

```

App.controller('ScopeCtrl', function($scope) {
  $scope.arrWeekday = ['Sun', 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat'];

  $scope.getWeekday = function() {
    console.log('Weekday is ' + $scope.arrWeekday[$scope.weekday]);
  };

  $scope.getRating = function() {
    console.log('Rating changed to ' + $scope.rating);
  };

  $scope.submit = function() {
    alert('You have rated ' + $scope.rating + '/5 and will be available on ' + $scope.arrWeekday[$scope.weekday] + ' . 1');
  };
});

```



You are now free to take a look at `ng-model-spinner.html` in the browser – visually it did not change but under the hood a lot..!

Let us quickly check if anything is broken with the changes we have done so far by running e2e tests for the same. I'm not covering unit testing here because of space constraint but will be included in the companion suite. The E2E testing does not require any change except the reference, so copy `tests/specs/ch05/scope-isolate-e2e.js` to `tests/specs/ch06/ng-model-spinner-e2e.js` and change the description as shown:

```
describe('Chapter 6:', function() {
  it('should activate spinner directive', function() {
    browser.get('ch06/ng-model-spinner.html');
```

Now add a new suite in `e2e.conf.js` under suites so:

```
suites: {
  ch1: 'tests/specs/ch01/*-e2e.js',
  ch3: 'tests/specs/ch03/*-e2e.js',
  ch5: 'tests/specs/ch05/*-e2e.js',
  ch6: 'tests/specs/ch06/*-e2e.js'
},
```

Finally run protractor to see tests succeed with following command in the terminal:

```
./node_modules/protractor/bin/protractor e2e.conf.js --suite ch6
```

Be happy to see all the tests in green..! In the next section, we'll use the spinner directive to extend thumbviewer directive.

Improving Page Thumbviewer with Spinner directive

Now it is time to spice up our `thumbViewer` with spinner directive that will assist users to navigate to any thumb easily. First of all, we need to include the spinner JavaScript file into `thumbnail-viewer.html` under head tag as:

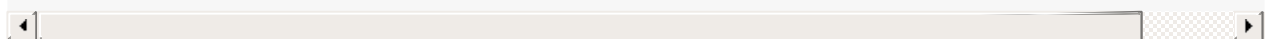
```
<script type="text/javascript" src="../js/ch06/ng-model-spinner.js"></script>
```

Then inject the respective module into `ThumbApp` module in `thumbviewer.js` as:

```
var App = angular.module('ThumbApp', ['nmSpinnerApp']);
```

Now let us inject spinner directive into the thumbviewer template as shown:

```
<script type="text/ng-template" id="thumbviewer.html">
  <div class='thumbnail-container bg-success img-rounded'>
    <div class='thumbnail-wrapper' ng-style="wrapperStyle()" ng-transclude></div>
    <div style="padding-top: 10px;">
      <adial-spinner ng-model="thumbTracker" data-type="list" data-list="thumbList" on-change="updateThumb()"></adial-spinner>
    </div>
  </div>
</script>
```



Few things to note here is that `thumbTracker` model holds the `thumbIndex` of currently selected thumb, `thumbList` is an actual numeric collection of all the thumbs based on which min and max limit will be calculated internally by the spinner directive, and `updateThumb` is a callback to trigger when new thumb is selected.

To find total number of thumbs after all are rendered, we can use AngularJS `$watch` method for the same. Put following in `ThumbViewerCtrl` so:

```
$scope.$watch(function() { return $scope.thumbs.length; }, function(newVal, oldVal) {
  if (newVal || newVal !== oldVal) {
    $scope.thumbList = [];
    for (var i = 0; i < newVal; i++) {
      $scope.thumbList.push(i + 1);
    }
  }
});
```

Go, play with it, If you are curious. However, while changing the spinner's state, thumb's focus does not shift to the new one. To fix it, we'll activate the callback used by the spinner directive i.e. `updateThumb` method in `thumbviewer` directive as follows:

```
scope.updateThumb = function() {
  var selectedThumb = scope.thumbs[scope.thumbTracker];
  if (selectedThumb) {
    scope.selectThumb(selectedThumb);
  }
};
```

Here `scope.thumbTracker` returns the `thumbIndex` by which we find its metadata (via `thumbs` collection) and then pass it to `selectThumb` method which was defined in the controller earlier. You can check on your end to see the focus shifts from thumb to thumb now when spinner's value changes but not the other way around when any thumb is clicked manually. Let us fix that as well by updating the `selectThumb` method in the controller to inform `thumbTracker` about the change as:

```
this.selectThumb = $scope.selectThumb = function(selectedThumb) {
  var selectedIndex = $scope.thumbs.indexOf(selectedThumb);

  angular.forEach($scope.thumbs, function/thumb) {
    if (thumb.active) thumb.active = false;
  });
  selectedThumb.active = true;
  $scope.thumbTracker = selectedIndex;
};
```

While navigating through thumbnails, you will notice that the thumbs which are out of the viewport do get the focus but are not visible. For that we need to adjust the left margin of `.thumbnail-wrapper` element like a carousel, so update `wrapperStyle()` in the link method from `thumbviewer` directive:

```
scope.marginLeft = 0;

scope.wrapperStyle = function() {
  var totalSize = scope.thumbSize * scope.thumbs.length;
  var totalGap = scope.thumbGap * (scope.thumbs.length - 1);
  var scrollLeft = scope.thumbTracker * (scope.thumbSize + scope.thumbGap);
  var leftPos = (scrollLeft + scope.thumbSize + scope.thumbGap * 2 + scope.marginLeft);
  var isInViewport = leftPos !== scope.thumbGap && leftPos < window.innerWidth;

  if (!isInViewport && scrollLeft >= 0) {
    scope.marginLeft = -scrollLeft;
  }
}
```

```

return {
  width  : totalSize + totalGap + 'px',
  display : 'inline-block',
  marginLeft: scope.marginLeft      + 'px'
};
};

```

Refresh the browser to check out the updates..! Although the movement is very quick and unnatural so we'll sprinkle some CSS to smoothen the transition. Add following in *thumbnail-viewer.html*:

```

.thumbnail-wrapper {
  -webkit-transition: margin-left 0.5s ease;
  -moz-transition: margin-left 0.5s ease;
  -ms-transition: margin-left 0.5s ease;
  -o-transition: margin-left 0.5s ease;
  transition: margin-left 0.5s ease;
}

```

It's almost ready..! Try clicking spinner few times to see it in action. Now let us handle the external callback when any thumb is selected so add the same on the thumbviewer directive in the DOM.

```

<thumbviewer data-size="150" data-gap="0" onselect="loadPage">
  <thumb ng-repeat="thumbnail in thumbnails" active="thumbnail.active">
    <div></div>
    <div class="caption text-center" ng-bind="thumbnail.title"></div>
  </thumb>
</thumbviewer>

```

Then update the isolate scope object for thumbviewer directive so:

```

scope: {
  size: '?',
  gap : '?',
  onselect: '&?'
},

```

And update `selectThumb` method in ThumbViewerCtrl:

```

this.selectThumb = $scope.selectThumb = function(selectedThumb) {
  var selectedIndex = $scope.thumbs.indexOf(selectedThumb);

  angular.forEach($scope.thumbs, function(thumb) {
    if (thumb.active) thumb.active = false;
  });
  selectedThumb.active = true;
  $scope.thumbTracker = selectedIndex;

  if (typeof $scope.onselect({param: {thumb: selectedThumb, index: selectedIndex}}) === 'function') {
    $scope.onselect()({thumb: selectedThumb, index: selectedIndex});
  }
};

```

This if-condition block is quite similar to the one we had used with spinner directive which passes the thumb's metadata and its index for further use. You could have `loadPage` method in ThumbCtrl to log the selected thumb as follows:

```

$scope.loadPage = function(param) {
  console.log('Page ' + param.index + ' is loading... Please Wait.');
```

We are good to go now..! Go play with freshly baked Thumbnail Viewer in AngularJS.

Testing of ThumbViewer

So far our diligent effort to make something useful in AngularJS is not commendable unless it works in real browsers. I do not find any possibility to unit test the thumbViewer directives so we'll swiftly test it in Protractor only. Create *thumbviewer-e2e.js* in *tests/specs/ch06* directory and put following test code in it:

```
describe('Chapter 6:', function() {
  it('should activate thumbViewer directive', function() {
    browser.get('ch06/thumbnail-viewer.html');

    var $thumbnailWrapper = element.all(by.css('.thumbnail-wrapper')).first();
    var $thumbnailContainer = element.all(by.css('.thumbnail-container')).first();
    var $thumbs = $thumbnailContainer.$$('.thumbnail');
    var $thumbTrackerButtons = $thumbnailContainer.$$('.button');
    var $thumbTrackerNext;
    var $thumbTracker;
    var $thumbTrackerPrev;

    $thumbs.then(function(thumbs) { expect(thumbs.length).toBe(15); });

    // test cases go here
  });
});
```

The code is self-explanatory so we'll go straight into writing the test cases for spinner used for thumb navigation. Replace the comment above with the following snippet:

```
describe('should able to navigate using spinner', function() {
  $thumbTrackerButtons.then(function(thumbTrackerButtons) {
    $thumbTrackerNext = thumbTrackerButtons[0];
    $thumbTracker = thumbTrackerButtons[1];
    $thumbTrackerPrev = thumbTrackerButtons[2];

    expect($thumbTracker.getText()).toBe('1');

    // Click next button twice to navigate to 3rd thumbnail
    $thumbTrackerNext.click();
    $thumbTrackerNext.click();
    expect($thumbnailWrapper.getCssValue('margin-left')).toBe('0px');
    expect($thumbTracker.getText()).toBe('3');

    // Navigate to 4th thumbnail and verify transition
    browser.sleep(0).then(function() {
      $thumbTrackerNext.click();
      browser.sleep(500).then(function() {
        expect($thumbnailWrapper.getCssValue('margin-left')).toBe('-480px');
        expect($thumbTracker.getText()).toBe('4');
      });
    });

    // Navigate back to the 3rd thumbnail and verify transition
    browser.sleep(0).then(function() {
      $thumbTrackerPrev.click();
      browser.sleep(500).then(function() {
        expect($thumbnailWrapper.getCssValue('margin-left')).toBe('-320px');
        expect($thumbTracker.getText()).toBe('3');
      });
    });

    // Navigate back to the 1st thumb and verify transition
    browser.sleep(0).then(function() {
      $thumbTrackerPrev.click();
      browser.sleep(500).then(function() {
        expect($thumbTracker.getText()).toBe('2');
        $thumbTrackerPrev.click();
        browser.sleep(500).then(function() {
          expect($thumbnailWrapper.getCssValue('margin-left')).toBe('0px');
        });
      });
    });
  });
});
```

```

        expect($thumbTracker.getText()).toBe('1');
    });
    });
    });
    });
    });
    });

```

Why `browser.sleep` and what it does? If you recall, we had used 500ms in CSS transition to last the animation for 0.5 second. So when we navigate to a thumb that requires transition to freeze, we wait for 500ms and then check its position to validate its visibility. The `browser.sleep()` is useful to make the driver sleep for the given amount of time in order to schedule commands later. Notice that we have used so many `browser.sleep(0)` statements to slow down the actions performed by Protractor.

NOTE: We resize the window to 500 by 500 px while running protractor tests as per Chapter 3, so all the assertions will be based on the same resolution. They may change for different resolution. But it's practical to stick to one specific resolution to test all your test cases unless you are testing responsive layout.

As we have tested thumb navigation using the Spinner directive, we could also navigate to specific thumb with manual clicks. Let us test that too. Add following as:

```

describe('should able to navigate using thumbs', function() {
    $thumbTrackerButtons.then(function(thumbTrackerButtons) {
        $thumbTracker = thumbTrackerButtons[1];

        expect($thumbTracker.getText()).toBe('1');

        // Click 2nd and 3rd thumbs
        $thumbs.then(function(thumbs) {
            thumbs[1].click();
            expect($thumbTracker.getText()).toBe('2');
            expect($thumbnailWrapper.getCssValue('margin-left')).toBe('0px');
            thumbs[2].click();
            expect($thumbnailWrapper.getCssValue('margin-left')).toBe('0px');
            expect($thumbTracker.getText()).toBe('3');
        });
    });
});

```

As you already guessed, there is no way to go back to previous thumb which is out of viewport in order to test, Spinner is used for the same which we already tested. In short, you can only go forward with manual thumb navigation and back & forth with the help of Spinner.

Summary

That was handful..!

In this chapter, We have covered all the important topics we need to know to build complex widgets in AngularJS and using directives API from the scratch. We got over our biggest confusion with compile and link and figured that DOM manipulation independent of scope can be done in the compile phase whereas the link phase can be used to read/write scope values and bind DOM events. We also learned about performance optimization that can be done with compile phase over link. We then understood the concept of transclusion to inject an external markups into a static template used by a directive. We can also manipulate the external markups before the transclusion with `transcludeFn` function. We realized that how we can slim down our link method by moving all the methods/properties into the directive controller and have them shared with other directives using `require` option if necessary. We later fix Spinner directive from the previous chapter to leverage `ngModelController` to prevent using external callbacks to store its state. At the end of the chapter, we tested our Page Thumbviewer directive in real browsers with Protractor in no time to gain confidence to release it to the world.

Bringing directives to Life

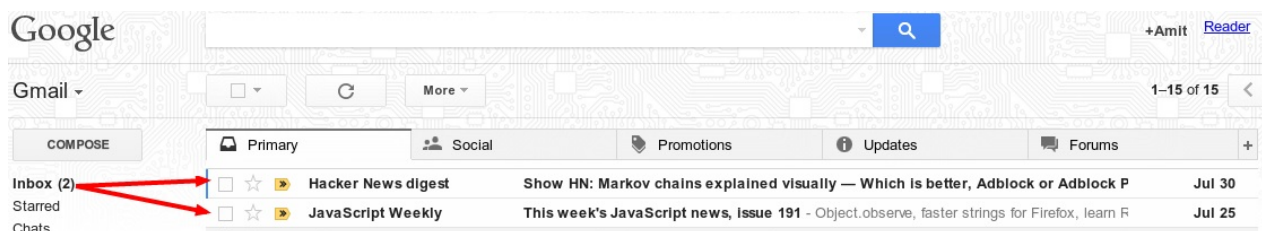
This chapter covers

- How data binding works behind the scene
- \$digest vs \$apply
- Various ways to watch over AngularJS models and the paradox of choice
- How and why to \$observe HTML attributes
- Different ways to speed up digest loop

What is a Mutation and the necessity to keep an eye on

Over the years, developers have been writing webapps in a traditional way. What that means is that you need to take care of view that needs to be updated manually when data model changes in order to flush it to a browser. This involves a lot of efforts and complexity, especially, when multiple views are interdependent.

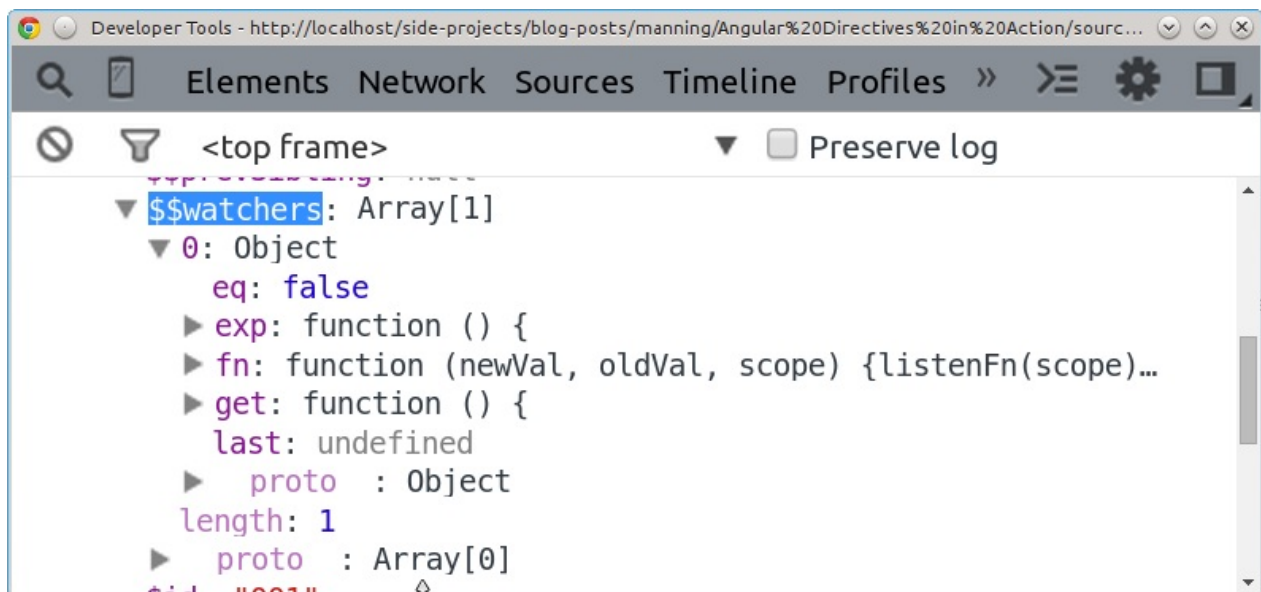
Take an example of Gmail – a web based email application - that keeps track of a total number of unread mails in an In-box on the left hand side and at the same time syncs it up with a list of all the mails shown on the right hand side. The moment we open any unread mail, the unread count goes down by 1 immediately as shown in the following figure.



This is called DOM mutation and there are in-numerous places where we do active DOM manipulation using DOM APIs in JavaScript, jQuery, and other similar libraries. When an application becomes large, it is very cumbersome to manage all the pieces in a sane way. However, there is no wrong using those libraries but definitely we can do better. By eliminating much of the boilerplate code needed, we can focus on building the actual application than scaffolding everything all over again. So AngularJS team has thought through the problem and came up with an idea to implement two-way data binding in AngularJS. This basically offloads the efforts required by developers into a framework itself, saving their time to utilize it for better use.

Understanding \$digest

We already had the bird's eye view of the `$digest` loop in Chapter 1 that takes care of the two-way data binding, so we'll go straight into understanding what does it mean exactly? Well, when any data binding expression such as `{{}}`, `ng-bind`, `ng-class`, and so on is compiled, a watcher gets registered for the same that will be evaluated during the digest cycle and its listener is called or re-run every time the binding's value changes. There are multiple bindings used at a time and hence their listeners are maintained in a collection of objects named `$watchers` which is available per scope. As we all know so far that root scope is the top most scope in any AngularJS application and remaining scopes become children. They further may have child scopes as well thus creating a tree or graph like data structures. Below are the details needed for each watcher to effectively figure out when to call the listener to update the view.



As you could see, each `$watcher` carries five properties pertaining to the watch method whenever it gets registered either manually or automatically using built-in directives. They are:

exp: This holds an expression or value that we passed while registering a watcher.

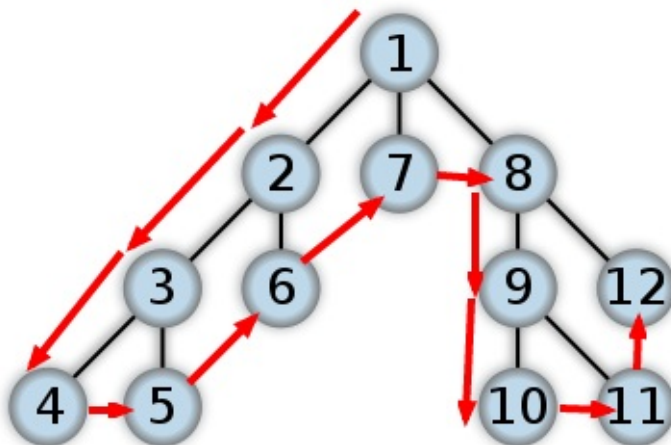
get: Evaluated expression using `$parse` service.

eq: This is an abbreviation for equality check (true/false) that decides whether to compare nested objects/arrays.

fn: A callback listener to trigger if the change in model detected.

last: The last evaluated expression to compare it with the updated value in order to trigger a listener a.k.a `fn` method during the digest.

So, whenever a value, `get` of an expression, `exp` changes, the current value is compared with the previously stored one, `last` and the view is updated by calling the listener, `fn`, if it did not match or is dirty. Please note that the digest loop runs from `$rootScope` to deep down the scope tree using depth-first search algorithm (DFS). The DFS processes vertices first deep and then wide as illustrated below. Imagine each of the vertex (with children) as a scope in a tree and arrows follow the direction the way each scope object is reached to evaluate all of its watchers before moving onto a next scope. However, watchers (leaf nodes) are evaluated recursively.



In order to maintain such a hierarchy by AngularJS, each scope has `$nextSibling`, `$prevSibling`, `$childHead`, and `$childTail` objects for easy traversing during the digest loop and locate the appropriate scope for further processing of

watchers. Hence less as well as light-weight watchers are extremely useful so that the digest loop completes as quickly as possible that makes an application responsive and unobtrusive. However, updating a JavaScript variable (defined with `var`) in AngularJS context does not trigger a digest loop, so what exactly triggers it, you may ask?

\$digest vs \$apply

AngularJS calls `$apply` internally to trigger the digest loop that ultimately updates views as seen before. The purpose of the `$apply` call is to evaluate the expression passed and then run the digest loop on `$rootScope` and all of its descendants to be dirty checked in order to update all bindings. To understand what that means, let's take a look at `ngClick` directive from the AngularJS source as:

```
var ngEventDirectives = {};
forEach(
  ['click'],
  function(name) {
    var directiveName = directiveNormalize('ng-' + name);
    ngEventDirectives[directiveName] = ['$parse', function($parse) {
      return {
        compile: function($element, attr) {
          var fn = $parse(attr[directiveName]);
          return function ngEventHandler(scope, element) {
            element.on(lowercase(name), function(event) {
              scope.$apply(function() {
                fn(scope, {$event:event});
              });
            });
          };
        }
      };
    }];
  }
);
```

Here `ngEventHandler` method is a `postLink` method that will be called for each instance of the directive. The `$apply` call will first trigger the `jqLite` click handler and run a digest loop then after. Let us use `ngClick` in the following example to see how many bindings get dirty checked when `$apply` is called. Create `$apply.html` in `ch07/` directory as:

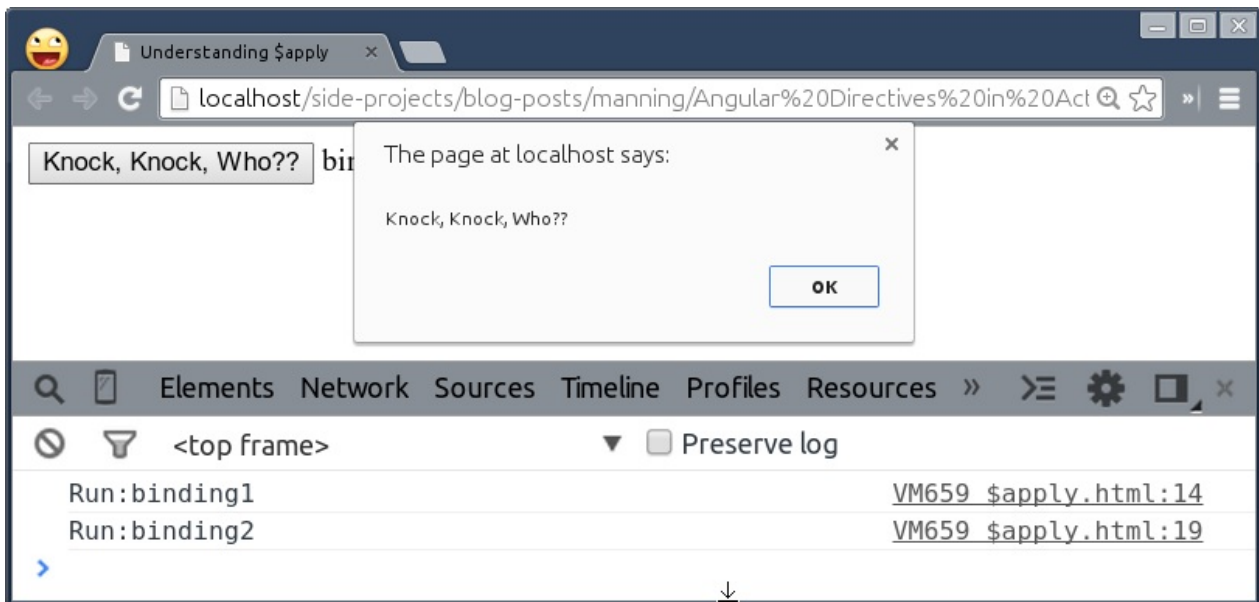
```
<html ng-app="App">
<head>
  <title>Understanding $apply</title>
  <script src="../../bower_components/angular/angular.js"></script>
  <script>
    var App = angular.module('App', []);

    App.run(function($rootScope) {
      $rootScope.alert = function($event) {
        alert($event.target.textContent);
      };

      $rootScope.binding1 = function() {
        console.log('Run:binding1');
        return 'binding1 ';
      };

      $rootScope.binding2 = function() {
        console.log('Run:binding2');
        return 'binding2';
      };
    });
  </script>
</head>
<body>
  <button ng-click="alert($event)">Knock, Knock, Who??</button>
  <span ng-bind="binding1()"></span>
  <span ng-bind="binding2()"></span>
</body>
</html>
```

In this simple example, I've added two dummy bindings but there could be lot of them in real-world applications. When we click the button, you'll notice following in the console:



In fact, you'll never need the `$apply` call for models that change within the AngularJS context. But it will be useful outside the context such as jQuery plugin or JavaScript DOM events which we will see later.

Unlike the `$apply` call that evaluates bindings from root scope to all of its descendants, `scope.$digest` call processes all of the `$watchers` of the current scope and its children only. As a `$watcher`'s listener may change the model, the digest loop keeps iterating the `$watchers` until no more listeners are firing and everything is stable. That is why the digest loop runs atleast twice whenever it is triggered.

Now that `ngClick` method does not modify any scope model, the digest loop is a bit overhead for such a small operation. So we'll write a custom directive to handle the click event and the digest flow. First, apply a custom directive on the button as follows:

```
<button custom-click="alert($event)">Custom Knock, Knock, who??</button>
```

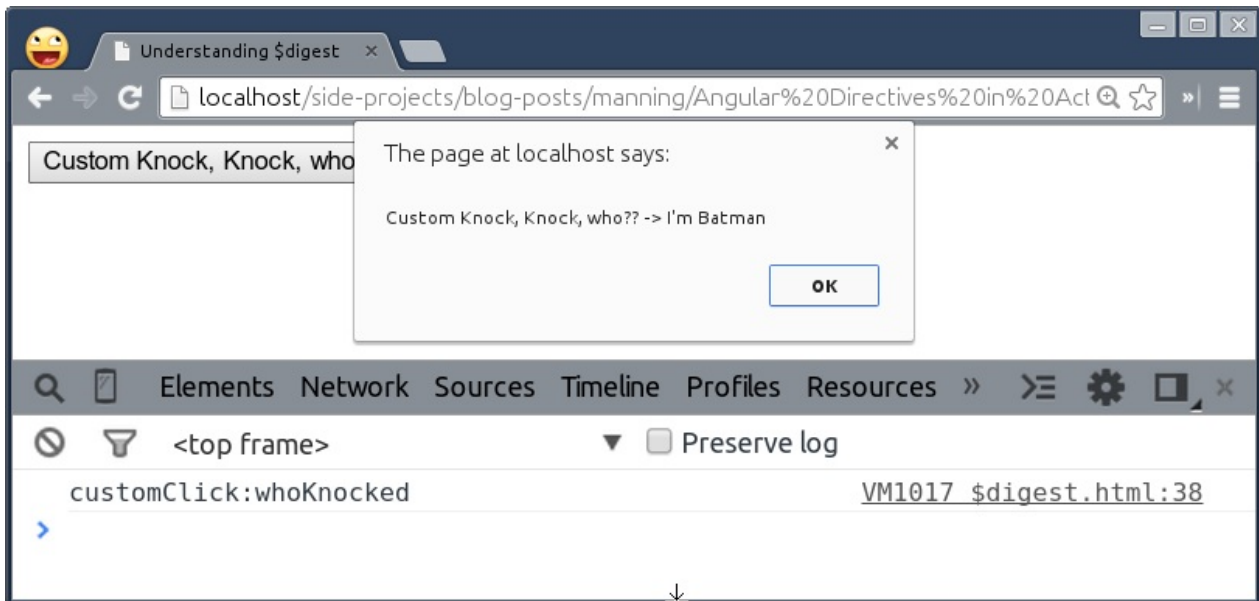
Then define the directive so:

```
App.directive('customClick', function($parse) {
  return {
    scope: true,
    template: 'Custom Knock, Knock, who?? -> {{whoKnocked()}}',
    link: function(scope, element, attrs) {
      var fn = $parse(attrs.customClick);

      element.on('click', function($event) {
        scope.who = "I'm Batman";
        scope.$digest();
        fn(scope, {$event: $event});
      });

      scope.whoKnocked = function() {
        console.log('customClick:whoKnocked');
        return scope.who;
      };
    }
  };
});
```


Note that we've created a new child scope to separate it out from the parent scope and its bindings during digest. When the button is clicked, we update the model, `who`, that triggers the digest loop to update the bindings for the current scope only using `$digest` instead of `$apply`. To our surprise, this only evaluates `whoKnocked` binding, leaving others untouched as shown in the following figure.



The takeaway is that the `$apply` method triggers a digest on the root scope which is a bit scary to think of in the first place because all of the bindings used in an application will be re-evaluated during each digest cycle and some of them may call their listeners as well. Hence its very likely to slow down AngularJS application if we do not know these important bits of information and how to use it at your disposal we'll be covered in the next section.

Unlike `scope.$apply`, AngularJS restricts the digest loop to spawn the current scope and its children only using `scope.$digest` that gives performance benefit sometimes. However, this is not recommended and may have other implications or diverse effect on bindings. For example, a view would have not been updated when the associated data model is modified on the parent scope within the `customClick` directive as the digest loop had spawned just the current scope. So to avoid many similar side-effects, the digest loop always runs from the root scope.

`$scope.$apply` vs `$rootScope.$apply`

This is confusing for lot of developers at first and you might want to know the difference between these two calls. However, to your surprise there is no difference at all. Both will run a new digest cycle on `$rootScope` after the passed functions are evaluated on their respective scopes.

Though this was just for us to understand how AngularJS knows when to run a digest loop and how far. Lets see how to write fast watchers to speed up the digest loop in the next section.

`$watching` Data Models

Throughout this book, we have used many built-in directives to enable two way data binding. But sometimes those are not enough and we are in need of custom watcher that will watch an expression to perform some action further. AngularJS allows us to register a custom listener callback to be executed whenever a watch expression changes and favorably it uses the same technique to register watchers internally. Here is how we can define a watch in AngularJS,

```
$watch(watchExpression, listener, [objectEquality]);
```

The `watchExpression` can be a function that returns a value being watched or a string representing a model binded on a scope. The `listener` callback triggers whenever there is a change in the value of the expression during the digest. Note that the listener gives access to new and old value of the expression that you can compare for further use. The reason it gives new and old value for comparison is that when a watcher is registered with the scope, its listener is called asynchronously to initialize the watcher. In rare cases, this is undesirable because the listener is called when the result expression didn't change. To detect this scenario within the listener, you can compare the `newVal` and `oldVal` before executing the listener logic. The `objectEquality` is false by default. If true, it performs a deep watch over nested objects or arrays which will have adverse effect on performance and memory footprint that will be covered in later sections. Below are couple of ways to register simple watchers and their differences:

```
scope.a = scope.b = 1;

scope.$watch('a', function(newVal, oldVal) {
  console.log('a changed to ' + newVal);
});

scope.$watch('b', function(newVal, oldVal) {
  console.log('b changed to ' + newVal);
});

scope.$watch('a + b', function(newVal, oldVal) {
  console.log('a or b changed to ' + scope.a + ' ' + scope.b);
});

scope.$watchGroup(['a', 'b'], function(newVal) {
  console.log('a or b changed to ' + newVal[0] + ' ' + newVal[1]);
});
```

In this example, we have 2 models named, `a` and `b` on the current scope. First two watchers will be run if the values of `a` and `b` change respectively. However, the third watcher will be run if either of them changes with `newVal` as a sum of both in return which is a bit overdo. So AngularJS 1.3 came up with a better alternative in terms of `$watchGroup` to return changed values separately as shown. The only difference is that the two separate watchers are registered by `$watchGroup` with a common listener callback unlike the third watcher where only one watcher was registered.

Now that we understand the foundation for registering watchers, let us see how to watch over objects or arrays in the following section.

\$watch for objects or arrays

The `$watch` method allows us to keep an eye on a property binded to a scope and perform any operation when it changes. If you remember, we had already used such a watch method for dynamic list required by `ThumbnailViewer` directive in *ch06/ng-model-spinner.html* in the previous chapter. That was:

```
scope.$watch('list', function() {
  ngModelCtrl.$render();
});
```

This essentially watches `scope.list` model on the scope to update the spinner's state when a list grows on the fly with `scope.list = newList`. Notice it will not be triggered when an item is added (by JavaScript array `push` method), removed (by JavaScript array `splice` method), or moved (rearranged). Also, modifying any item or object property with `scope.list[0] = 'Funday'` will not affect the watch. That is because AngularJS keeps a reference of the original watched value so the referenced copy mutates automatically if the original copy changes. As both are same, the listener will not be called in such cases.

The temporary solution to fix the above watch to be called every time when an item is added into or removed from (but not moved) the list and that is to change the watch expression so:

```
scope.$watch('list.length', function() {
  ngModelCtrl.$render();
});
```

With this change, the watcher will store a length of the list (instead of actual collection as before) to compare it during a next digest cycle before calling the listener.

\$watch for object expression with a dot

The watch method is also good at understanding an object expression with a dot in it. Imagine you want to add a little bounce effect to the spinner text in the Spinner directive, you could watch `scope.opt.default` and add animation on every update.

To try that out, first, make a copy of `js/ch06/ng-model-spinner.js` file into `js/ch07/` directory as `watch-spinner.js`. As we are going to use CSS3 animation library called **animate.css**, let us install it quickly by running following commands in a terminal or GitBash as:

```
cd angular-directives-in-traction
bower install --save animate.css
```

animate.css library provides a bunch of cool, fun, and cross-browser animations (CSS classes) purely in CSS3 that you can easily apply to any element to make it dance.

Now create `watch-spinner.html` in `ch07/` directory so:

```
<html ng-app="nmSpinnerApp">
<head>
  <title>Understanding $watch</title>
  <script src="../../bower_components/angular/angular.js"></script>
  <script src="../../js/ch07/watch-spinner.js"></script>
  <link href="../../bower_components/bootstrap/dist/css/bootstrap.css" rel="stylesheet">
  <link href="../../bower_components/animate.css/animate.css" rel="stylesheet">
</head>
<body style="padding: 10px;">
  <adia-spinner ng-model="bounceCount"></adia-spinner>
</body>
</html>
```

This will render a range Spinner by default. Now let us watch over `scope.opt.default` expression by adding following into the link method of the spinner directive:

```
scope.$watch('opt.default', function(newVal, oldVal) {
  if (newVal && newVal !== oldVal) {
    element.find('button').eq(1)
      .addClass('bounceIn animated')
      .on('webkitAnimationEnd mozAnimationEnd MSAnimationEnd oanimationend animationend', function() {
        angular.element(this).removeClass('bounceIn animated');
      });
  }
});
```

What we are doing here is that bouncing the spinner text with `bounceIn` CSS class whenever the spinner value increments or decrements. And at the end, just removing these CSS classes after animation is over to make it ready for the next round. Try at your end to see it bouncing.

Both Object/Array from the previous section and Object expressions from the preceding example are efficient approaches

as they merely keep a reference of the value around for comparison during the digest cycle.

\$watch for objects or arrays deeply

The referenced watcher on `scope.list` we saw earlier does not get invoked if any of its property changes so it is useful to have it deep watched in certain scenarios. The `ngStyle` and `ngClass` built-in directives use deep watchers as both allow you to apply multiple CSS styles and classes on an element, and need to figure out a change in nested objects. You can use `ngStyle` directive in the DOM as:

```
<div ng-style="{left: leftPos, top: topPos}"></div>
```

In this declaration, any of the two properties (`scope.leftPos` and `scope.topPos`) may change and AngularJS has to re-render the view accordingly. In the deep watch call, AngularJS keeps a cloned copy of the value being watched in a memory in order to compare it with the new one. So `equality` check ensures that both values are same including nested objects/arrays within, otherwise it fires a listener callback. Please note that it can be extremely expensive to deeply watch an objects/arrays if not used carefully and may even degrade performance of the application.

Here is how you can convert a normal watch to become a deep watch with third parameter as true so:

```
$watch(watchExpression, listener, true);
```

If you have not seen the source code of `ngStyle` before, that's fine. We'll write a custom directive to mimic `ngStyle` to get yourself an idea how it works. Imagine you are a publisher and want to enable authors (who write for you) to design a book cover for their own books. It's a simple interactive cover builder application that you want to build wherein authors can customize the color of the title and have any font size that suites them for the same. Create *watch-deeply.html* in *ch07/* directory:

```
<html ng-app="App">
<head>
  <title>Understanding $watch</title>
  <script src="../../bower_components/angular/angular.js"></script>
  <link href="../../bower_components/bootstrap/dist/css/bootstrap.css" rel="stylesheet">
  <style type="text/css">
    .title {
      text-align: center;
      color: black;
      font-size: 10px;
      height: 150px;
    }
  </style>
  <script type="text/javascript">
    var App = angular.module('App', []);

    App.controller('DemoCtrl', function($scope) {
      $scope.color = 'black';
      $scope.fontSize = '10px';
    });

    App.directive('cover', function() {
      return {
        restrict: 'EA',
        link: function(scope, element, attrs) {
          scope.$watch(attrs.cover, function(newVal, oldVal) {
            element.css(newVal);
          }, true);
        }
      }
    });
  </script>
</head>
<body style="padding: 10px;" ng-controller="DemoCtrl">
```

```

<div class="row">
  <div class="col-xs-3 col-md-2">
    <a class="thumbnail alert-warning">
      <div class="title" cover="{color: color, fontSize: fontSize}">AngularJS Directives in Action</div>
      
    </a>
  </div>
</div>
<button class='btn btn-default' ng-click="color = 'black'">&nbsp;</button>
<button class='btn btn-primary' ng-click="color = 'blue'">&nbsp;</button>
<button class='btn btn-info' ng-click="fontSize = '10px'"><small>10px</small></button>
<button class='btn btn-warning' ng-click="fontSize = '20px'">20px</button>
</body>
</html>

```

In this example, we have two models namely `color` and `fontSize` which we can alter with few buttons underneath the cover layout using `ngClick` directives. Then we pass both properties to the directive in the DOM as an attribute values. The cover directive has a deep watch set up to watch over these properties and apply appropriate styles to the book title whenever any property changes.

With equality check, the watcher will be called if an entire collection is replaced by a new one, an item is added/removed/moved in the collection, and any nested object property is modified within the collection. So it is damn expensive. Make sure to keep the listener callback lightweight by not performing any heavy operation in it as it will be called multiple times than usual. However, both `ngStyle` and `ngClass` directives simply apply CSS styles/classes on an element and do not have deep nested collection so in their case it is not at all expensive and thus perfectly makes sense to use deep watchers in that context by AngularJS.

\$watchCollection for Shallow objects or arrays

By now, we know that `ngRepeat` build-in AngularJS directive takes an object or array to iterate over to generate or destroy repeatable element when model mutates.

If we use a normal watch method (without equality check) on a collection, it will not trigger a listener callback when an item is added in or removed from the collection. Alternatively, If we use deep watch method (with equality check) on the same collection, the listener callback will be called on every updates, including nested objects.

Performance wise this is terrible because `ngRepeat` should only update the DOM if an item added, removed, or moved – it should not touch the repeatable element in case of nested objects/properties are modified that do not affect it. The `$watchCollection` method is to the rescue in such a case!

The `$watchCollection` API watches for changes in arrays or objects. It also watches items in arrays and properties in objects that distinguishes it from the deep watch we saw earlier. To achieve this, it needs to keep a copy of the original array or object, and traverses the old and new collections for each digest, checking for changes using the strict equality operator (i.e. `===` not `==`). This has been used by `ngRepeat` directive since AngularJS version 1.1.4 as it serves it well. You can define it as follows:

```

$scope.$watchCollection('collection', function listener() {});

```

Notice it does not require equality check as a third parameter. It can be used over deep watch for performance gain. So it appears to be a proper solution for `Spinner` directive to update itself when a list changes. Let us replace

`scope.$watch('list')` call in `ch07/watch-spinner.js` with the following:

```

scope.$watchCollection('list', function() {
  ngModelCtrl.$render();
});

```

To make sure that this change does not break anything, go ahead and change the references of *ng-model-spinner.js* in both *ng-model-spinner.html* and *thumbnail-viewer.html* with:

```
<script type="text/javascript" src="../../js/ch07/watch-spinner.js"></script>
```

Then run `npm run test-unit` and `npm run test-e2e` commands in the terminal to verify it. It should be working for you as well.

\$observe HTML Attributes

The watch methods are really for AngularJS models so you can not watch over HTML attributes with it, especially interpolations. HTML attribute is a string or evaluated expression (with double curly braces) such as `data-interpolated="my {{name}}"` but the watch methods expect non interpolated strings. Here is how you can register the \$observe callback,

```
attrs.$observe('interpolatedAttribute', function(newVal) { });
```

Notice that the `$observe` is a method binded on attributes (a.k.a `attrs` object) that only works inside directives. The directive's link method gives access to `attrs` to set up the observer. You can even register it in the compile method of the directive as it does not have a hard dependency on scope.

Also note that the `$watch` expression are evaluated by `$parse` or `scope.$eval` services whereas `$interpolate` service is used to evaluate interpolated strings during the next digest cycle following compilation. The observer is then invoked whenever the interpolated value changes. This is how the `$interpolate` service being used as, `$interpolate("my {{name}}")(scope)`.

When an observer is registered, the expression will be evaluated by the `$interpolate` service to create a compiled function (similar to `$parse` service) which can then be used to set up a watch. However, the `$observe`'s listener callback only returns a new value but not old one. Although it could have returned an old value as well but is not required by the built-in directives that use `$observe` internally such as `ngBindTemplate`, `ngClass`, `ngModel`, etc.

Note that all observers and watchers are dirty checked on every digest cycle so performance-wise there is not much difference between the two.

Speeding up the digest loop

No matter how fast a particular programming language or framework is, it all boils down to how carefully we craft an application to keep it smooth. The same thing applies to AngularJS. Each and every directive is properly benchmarked and scrutinized by AngularJS team but still it's our responsibility to control the total number of bindings we use and when the digest runs, especially how often.

By unregistering unnecessary \$watchers

Each registered watcher via `$watch`, `$watchCollection`, or `$observe` returns `deregistration` method that you can invoke later to unregister the respective watch. You can do following to unregister the registered watch:

```
var killWatcher = $scope.$watch('list', function() {});
killWatcher(); // deregister the above watch
```

However, it is very rare to unregister the watch in AngularJS application but can be handy in few situations. Imagine you are a publisher and allow readers to read few pages of any book they like but also want to notify them to buy the book after

every 4 pages they flip. However, you want to limit the alert to be shown 10 times only not to annoy readers so often. Obviously, you can conditionally show the alert and stop showing it once the counter reaches 10 but the watch is still hanging around in the digest loop which is unnecessary. We can easily get rid of it as shown:

```
var killBuyWatch = $scope.$watch('pageTurnCounter', function(newVal) {
  if ($scope.buyCounter > 10) {
    killBuyWatch();
  } else if (newVal / 4 === 0) {
    $scope.modalOpen = true;
    $scope.buyCounter++;
  }
})
```

Many times, we unintentionally keep watchers alive even though we do not need it. We can even to unregister `$observe` calls the same way.

By using one-time binding syntax

It's very common scenario to have one time watchers that will be evaluated only once. Localization is one example I can think of where you may want to translate form labels in some other language other than English such as:

```
<label>{{"firstName" | translate }}</label>
<label ng-bind='firstName | translate'></label>
```

But bindings used to translate above labels should be dead after evaluation but we can not control its behavior because AngularJS automatically set up a watch for it. Luckily, AngularJS 1.3 has one-time-binding syntax wherein expression stops reevaluating once the DOM is updated which happens after the first digest. AngularJS immediately destroys the watch once it is stable. The expression or binding starts with `::` (double colon) is considered a one-time binding. So the preceding example would change to:

```
<label>{{::"firstName" | translate }}</label>
<label ng-bind= '::firstName | translate'></label>
```

This can even work with custom watcher that takes a string as an expression but not a function. We can easily turn `pageTurnCounter` watcher into a one-time binding as:

```
$scope.$watch('::pageTurnCounter', function(newVal) { });
```

Look into your AngularJS application and who knows you may find such watchers that are of no use but only require once.

By using \$watchCollection over \$watch

If you remember the iscroll directive that we wrote in Chapter 5, we were using a watch method to update the scroll when any movie name is removed. But there was one problem with the watch implementation, here it is:

```
scope.$watch(function() {
  if (myScroll) {
    myScroll.refresh();
  } else {
    myScroll = new IScroll(element[0], {
      scrollbars: angular.isDefined(scope.scrollbars) ? scope.scrollbars : true,
      mousewheel: angular.isDefined(scope.mousewheel) ? scope.mousewheel : true
    });
  }
})
```

```

    });
  }
});

```

You could notice that this watch does not have a listener because we have put the logic into the expression function itself to update the scrollbar in a worse-case scenario replacing `setTimeout` used before. The problem with this approach is that the expression will be evaluated for every digest cycle and as we are doing active DOM manipulation by calling `iscroll`'s constructor method that interacts with the DOM which is what making it vulnerable.

What we can do to fix this vulnerability is to watch over `movies` collection using `$watchCollection` method so that the scrollbar will only be updated when a movie is removed. Lets try that out. Create a copy of `ch05/iscroll-directive.html` and save it in the `ch07/` directory.

First we will add one more attribute named `data-collection` to hold the array/object that we are iterating through as:

```
<div id="wrapper" iscroll data-collection="movies" data-scrollbar="true" data-mousewheel="true">
```

Then we'll update the directive definition to bring the collection into the local scope and replace it with `$watchCollection` so:

```

App.directive('iscroll', function() {
  return {
    restrict: 'EAC',
    scope: {
      scrollbars: '=?',
      mousewheel: '=?',
      collection: '='
    },
    link: function(scope, element, attrs) {
      var myScroll = null;

      scope.$watchCollection('collection', callback);

      function callback() {
        if (myScroll) {
          myScroll.refresh();
        } else {
          myScroll = new IScroll(element[0], {
            scrollbars: angular.isDefined(scope.scrollbars) ? scope.scrollbars : true,
            mousewheel: angular.isDefined(scope.mousewheel) ? scope.mousewheel : true
          });
        }
      }
    }
  }
});

```

That is pretty straight forward. We've just wrapped the previous expression into its own JavaScript method and using collection as an expression instead. We'll get to why we have the `callback()` method separately in a moment.

This should not have affected the `iscroll` implementation and worked properly. However, you'll notice that `iscroll` does not allow you to scroll through the entire list initially but it does if any item is removed. The reason for that is the `iscroll` directive executed before `ngRepeat` finished rendering all the items. So, how can we delay the execution of the `iscroll` directive? Unfortunately, there is no event to track when `ngRepeat` finishes rendering all the items for us to update the scrollbar later, although it may be available in next AngularJS releases.

For now, we will also leverage the previous watch implementation as a one-time watcher by unregistering it immediately after the `iscroll` directive is compiled and linked. So update the link method of the `iscroll` directive as:

```

link: function(scope, element, attrs) {
  var myScroll = null,

```



```

killThisWatcher = scope.$watch(callback);

scope.$watchCollection('collection', callback);

function callback() {
  if (myScroll) {
    myScroll.refresh();
    if (!angular.isDefined(oldVal)) killThisWatcher();
  } else {
    myScroll = new IScroll(element[0], {
      scrollbars: angular.isDefined(scope.scrollbars) ? scope.scrollbars : true,
      mousewheel: angular.isDefined(scope.mousewheel) ? scope.mousewheel : true
    });
  }
}
}
}

```

In here, the watch statement will be registered first to instantiate the iscroll constructor and then watchCollection gets invoked to refresh the scrollbar and freezes itself (will not be called unless collection changes). But the normal watch will be triggered once again by the digest loop to make sure everything is stable, so this time we unregister it. Please note that the order of registration of watchers may affect the behavior in this case.

By preventing DOM manipulation inside watch listener

Earlier we had updated spinner directive to have a nice bouncing effect whenever the spinner value changes. Here is the watch method used to bring up the effect in *ch07/watch-spinner.html*:

```

scope.$watch('opt.default', function(newVal, oldVal) {
  if (newVal && newVal !== oldVal) {
    element.find('button').eq(1)
      .addClass('bounceIn animated')
      .on('webkitAnimationEnd mozAnimationEnd MSAnimationEnd oanimationend animationend', function() {
        angular.element(this).removeClass('bounceIn animated');
      });
  }
});

```

But this watch method has a major issue which is not even noticeable if not looked closely. And that is, we are searching through the descendants of an element in the DOM tree which is not only slow but also fired every time when the listener is called. Selectors causes reflow event wherein web browser recalculates the positions and geometries of elements in the DOM, for the purpose of re-rendering part or all of the document. Obviously it's an user blocking operation that will surely delay the digest to complete the cycle. The best way is to cache such selectors for later use. So update the watch in *js/ch07/watch-spinner.js* with:

```

var $bouncer = element.find('button').eq(1);
$bouncer.on('webkitAnimationEnd mozAnimationEnd MSAnimationEnd oanimationend animationend', function() {
  angular.element(this).removeClass('bounceIn animated');
});
scope.$watch('opt.default', function(newVal, oldVal) {
  if (newVal && newVal !== oldVal) {
    $bouncer.addClass('bounceIn animated');
  }
});

```

Also, the animationEnd event needs to be binded only once and should work as before.

By avoiding digest loop completely

So far we know that the digest loop is the magic happens behind the scenes and makes AngularJS more magical than any other framework. That means AngularJS has to run the digest loop after every user interaction such keypress, click, and so

on to make sure bindings are up to date and the UI is in sync. Having said that the more watchers you have to loop through, the more time the digest will take to complete the cycle. Although there is no harm in it but what if user interactions happen so quick and in a continuous stream that might trigger magnitude of digest cycles to bring an application to a dead end – making it unresponsive and frozen.

If you think that's a myth and will never happen then take a look at few built in directives that will help us to reproduce the problem. These directives allow us to bind special behavior on mouse events such `ngMouseUp`, `ngMouseDown`, `ngMouseOver`, `ngMouseMove` and so on. Though the directives are made available for the benefits but they might do more harm than good, especially `ngMouseMove`. Imagine you want to draw pixels on a HTML5 canvas element, writing a directive for binding native mouse events on the canvas is always the best choice. This is because `mousemove` event fires multiple times and very frequently which can not be even debounced with AngularJS directives. Native mouse event can be delayed using `requestAnimationFrame()` to the point when a browser is ready which prevents a forced call and maintains 60 FPS (frames per seconds) cap to keep the application performant.

So the takeaway is that it's not always necessary to use databinding or AngularJS directives to keep your code looks angularish. Sometimes it's better to go with imperative DOM manipulation approach which is faster than the so-called digest loop. Let us look at a simple example to show how often the digest cycle runs in this particular scenario and how to avoid it altogether. We often see images load progressively wherein an image is being rendered partially as soon as the downloading begins and most of time we are unaware of how soon it will be completely shown. The solution is to write a simple asynchronous image loader directive that will fetch the image in background and show a progress in percentage while it is being downloaded. Later the progress bar will be replaced by the fully loaded image. Sounds cool, let us begin.

First create *image-lazy-loader.html* in *ch07/* directory and put following markup in it:

```
<html ng-app="App">
<head>
  <title>Image Lazy Loader</title>
  <script src="../../bower_components/jquery/jquery.js"></script>
  <script src="../../bower_components/angular/angular.js"></script>
  <script src="../../js/ch07/image-lazy-loader.js"></script>
  <link rel="stylesheet" href="../../bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body style="padding:10px;">
  <div style="width: 200px; height: 200px;" image-lazy-loader="http://upload.wikimedia.org/wikipedia/commons/3/36/Hopet
</body>
</html>
```

You might have already guessed that we are going to call it `imageLazyLoad` directive which takes an image URL to load. You can set the custom size to the image container. Note we are relying on jQuery library because we will use `$.ajax()` over `$http` to fetch an image via XHR call. The `$.ajax` supports XHR parameter to override the default implementation with our own. This allows us to track total vs downloaded image size in order to accurately depict the download progress in percentage.

Now let us write a directive for the same in *js/ch07/image-lazy-loader.js* as:

```
var App = angular.module('App', []);

App.directive('imageLazyLoader', function() {
  return {
    restrict: 'A',
    scope: {},
    template: function(element) {
      element.css({position: 'relative'});

      return '\
      <a class="thumbnail">\
        \
        <div ng-hide="src" class="progress" style="position: absolute;width: 50%;left: 25%;top: 48%;">\
```

```

        <div style="width:{{percentage}}%;" class="progress-bar">{{percentage}}%</div>\
      </div>\
    </a>';
  },
  link: function(scope, element, attrs) {
    $.ajax({
      type: 'GET',
      url: attrs.imageLazyLoader,
      xhr: function() {
        var xhr = new window.XMLHttpRequest();
        xhr.addEventListener("progress", function(evt) {
          scope.$apply(function() {
            scope.percentage = parseInt(evt.loaded / evt.total * 100, false);
          });
        }, false);

        return xhr;
      },
      success: function(data) {
        scope.$apply(function() {
          scope.src = attrs.imageLazyLoader;
        });
      }
    });

    scope.$watch(function() { console.log('digesting'); });
  }
};
});

```

The code is pretty self-explanatory but the watch method at the end may find spooky to you. Of-course it is spooky because it is there to watch over the digest loop. It takes a function as a watch expression that will be run for every digest so that we can keep track of how often digest happens during the download.

Although the code looks perfect but to your surprise, a digest cycle will run for more than 100 times if the image is not in cache, which is totally disturbing. Because if we use multiple instance of this directive in a large application with lots of bindings, the application will surely be crashed. Can we do better?

Yes, as I said before, the purpose is not only to write code that looks angularish but also to keep it responsive and unobtrusive. So let us quickly fix it.

First thing to note that we do not have a hardcore dependency on built-in directives such ngSrc, ngStyle, etc here because we can directly set src to img element instead of doing the same via ngSrc and the same thing is with ngStyle. So update the template as:

```

template: function(element) {
  element.css({position: 'relative'});

  return '\
    <a class="thumbnail">\
      <img style="width: 100%; height: 100%; background: rgba(128, 128, 128, 0.3);">\
      <div class="progress" style="position: absolute;width: 50%;left: 25%;top: 48%;">\
        <div class="progress-bar"></div>\
      </div>\
    </a>';
},

```

Nothing has changed, we have just removed AngularJS directives and bindings. Then update the link method so:

```

link: function(scope, element, attrs) {
  var $img = element.find('img'),
      $progress = element.find('.progress'),
      $progressBar = element.find('.progress-bar');

  $.ajax({

```

```

    type: 'GET',
    url: attrs.imageLazyLoader,
    xhr: function() {
        var xhr = new window.XMLHttpRequest(),
            percentage;

        xhr.addEventListener("progress", function(evt) {
            percentage = parseInt(evt.loaded / evt.total * 100, false) + '%';
            $progressBar.css({width: percentage}).text(percentage);
        }, false);

        return xhr;
    },
    success: function(data) {
        $img.attr('src', attrs.imageLazyLoader);
        $progress.hide();
    }
  });

  scope.$watch(function() { console.log('digesting'); });
}

```

As you can see, we have completely isolated our directive from the digest loop. The real culprit was the progress event that was triggering the digest loop far many times. Moreover, we have achieved the exact same result by compromising neither the functionality nor the performance.

Summary

Alright..!

We have started this chapter with how digest loop plays an important role in two-way data binding in AngularJS. Then we analyzed `$digest` vs `$apply` comparison to get an insight into how AngularJS triggers the digest loop and how often. Afterwards, we learned about various ways to set up watchers and when to use `$watchCollection` over deep `$watch` to get the performance boost. We then used `$observe` over `$watch` with interpolated strings. After that, we found out the way to unregister unnecessary watchers to keep the digest loop short and quick.