AngularJS

Notes for Professionals

100+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
http://GoalKicker.com/AngularJSBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with AngularJS

Version	Release Date
1.6.5	2017-07-03
1.6.4	2017-03-31
1.6.3	2017-03-08
1.6.2	2017-02-07
1.5.11	2017-01-13
1.6.1	2016-12-23
1.5.10	2016-12-15
1.6.0	2016-12-08
1.6.0-rc.2	2016-11-24
1.5.9	2016-11-24
1.6.0-rc.1	2016-11-21
1.6.0-rc.0	2016-10-26
1.2.32	2016-10-11
1.4.13	2016-10-10
1.2.31	2016-10-10
1.5.8	2016-07-22
1.2.30	2016-07-21
1.5.7	2016-06-15
1.4.12	2016-06-15
1.5.6	2016-05-27
1.4.11	2016-05-27
1.5.5	2016-04-18
1.5.4	2016-04-14
1.5.3	2016-03-25
1.5.2	2016-03-19
1.4.10	2016-03-16
1.5.1	2016-03-16
1.5.0	2016-02-05
1.5.0-rc.2	2016-01-28
1.4.9	2016-01-21
1.5.0-rc.1	2016-01-16
1.5.0-rc.0	2015-12-09
1.4.8	2015-11-20
1.5.0-beta.2	2015-11-18
1.4.7	2015-09-30
1.3.20	2015-09-30
1.2.29	2015-09-30
1.5.0-beta.1	2015-09-30
1.5.0-beta.0	2015-09-17
1.4.6	2015-09-17
1.3.19	2015-09-17
1.4.5	2015-08-28
1.3.18	2015-08-19
1.4.4	2015-08-13
1.4.3	2015-07-15
1.3.17	2015-07-07
1.4.2	2015-07-07
1.4.1	2015-06-16
1.3.16	2015-06-06
1.4.0	2015-05-27
1.4.0-rc.2	2015-05-12

1.4.0-rc.1	2015-04-24
1.4.0-rc.0	2015-04-10
1.3.15	2015-03-17
1.4.0-beta.6	2015-03-17
1.4.0-beta.5	2015-02-24
1.3.14	2015-02-24
1.4.0-beta.4	2015-02-09
1.3.13	2015-02-09
1.3.12	2015-02-03
1.4.0-beta.3	2015-02-03
1.3.11	2015-01-27
1.4.0-beta.2	2015-01-27
1.4.0-beta.1	2015-01-20
1.3.10	2015-01-20
1.3.9	2015-01-15
1.4.0-beta.0	2015-01-14
1.3.8	2014-12-19
1.2.28	2014-12-16
1.3.7	2014-12-15
1.3.6	2014-12-09
1.3.5	2014-12-02
1.3.4	2014-11-25
1.2.27	2014-11-21
1.3.3	2014-11-18
1.3.2	2014-11-07
1.3.1	2014-10-31
1.3.0	2014-10-14
1.3.0-rc.5	2014-10-09
1.2.26	2014-10-03
1.3.0-rc.4	2014-10-02
1.3.0-rc.3	2014-09-24
1.2.25	2014-09-17
1.3.0-rc.2	2014-09-17
1.2.24	2014-09-10
1.3.0-rc.1	2014-09-10
1.3.0-rc.0	2014-08-30
1.2.23	2014-08-23
1.3.0-beta.19	2014-08-23
1.2.22	2014-08-12
1.3.0-beta.18	2014-08-12
1.2.21	2014-07-25
1.3.0-beta.17	2014-07-25
1.3.0-beta.16	2014-07-18
1.2.20	2014-07-11
1.3.0-beta.15	2014-07-11
1.2.19	2014-07-01
1.3.0-beta.14	2014-07-01
1.3.0-beta.13	2014-06-16
1.3.0-beta.12	2014-06-14
1.2.18	2014-06-14
1.3.0-beta.11	2014-06-06
1.2.17	2014-06-06
1.3.0-beta.10	2014-05-24
1.3.0-beta.9	2014-05-17
1.3.0-beta.8	2014-05-09

1.3.0-beta.7	2014-04-26
1.3.0-beta.6	2014-04-22
1.2.16	2014-04-04
1.3.0-beta.5	2014-04-04
1.3.0-beta.4	2014-03-28
1.2.15	2014-03-22
1.3.0-beta.3	2014-03-21
1.3.0-beta.2	2014-03-15
1.3.0-beta.1	2014-03-08
1.2.14	2014-03-01
1.2.13	2014-02-15
1.2.12	2014-02-08
1.2.11	2014-02-03
1.2.10	2014-01-25
1.2.9	2014-01-15
1.2.8	2014-01-10
1.2.7	2014-01-03
1.2.6	2013-12-20
1.2.5	2013-12-13
1.2.4	2013-12-06
1.2.3	2013-11-27
1.2.2	2013-11-22
1.2.1	2013-11-15
1.2.0	2013-11-08
1.2.0-rc.3	2013-10-14
1.2.0-rc.2	2013-09-04
1.0.8	2013-08-22
1.2.0rc1	2013-08-13
1.0.7	2013-05-22
1.1.5	2013-05-22
1.0.6	2013-04-04
1.1.4	2013-04-04
1.0.5	2013-02-20
1.1.3	2013-02-20
1.0.4	2013-01-23
1.1.2	2013-01-23
1.1.1	2012-11-27
1.0.3	2012-11-27
1.1.0	2012-09-04
1.0.2	2012-09-04
1.0.1	2012-06-25
1.0.0	2012-06-14
v1.0.0rc12	2012-06-12
v1.0.0rc11	2012-06-11
v1.0.0rc10	2012-05-24
v1.0.0rc9	2012-05-15
v1.0.0rc8	2012-05-07
v1.0.0rc7	2012-05-01
v1.0.0rc6	2012-04-21
v1.0.0rc5	2012-04-12
v1.0.0rc4	2012-04-05
v1.0.0rc3	2012-03-30
v1.0.0rc2	2012-03-21
g3-v1.0.0rc1	2012-03-14
g3-v1.0.0-rc2	2012-03-16

1.0.0rc1	2012-03-14
0.10.6	2012-01-17
0.10.5	2011-11-08
0.10.4	2011-10-23
0.10.3	2011-10-14
0.10.2	2011-10-08
0.10.1	2011-09-09
0.10.0	2011-09-02
0.9.19	2011-08-21
0.9.18	2011-07-30
0.9.17	2011-06-30
0.9.16	2011-06-08
0.9.15	2011-04-12
0.9.14	2011-04-01
0.9.13	2011-03-14
0.9.12	2011-03-04
0.9.11	2011-02-09
0.9.10	2011-01-27
0.9.9	2011-01-14
0.9.7	2010-12-11
0.9.6	2010-12-07
0.9.5	2010-11-25
0.9.4	2010-11-19
0.9.3	2010-11-11
0.9.2	2010-11-03
0.9.1	2010-10-27
0.9.0	2010-10-21

Section 1.1: Getting Started

Create a new HTML file and paste the following content:

```
<!DOCTYPE html>
<html ng-app>
<head>
  <title>Hello, Angular</title>
  <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
</head>
<body ng-init="name='World'">
  <label>Name</label>
  <input ng-model="name" />
  <span>Hello, {{ name }}!</span>
  <p ng-bind="name"></p>
</body>
</html>
```

Live demo

When you open the file with a browser, you will see an input field followed by the text `Hello, World!`. Editing the value in the input will update the text in real-time, without the need to refresh the whole page.

Explanation:

1. Load the Angular framework from a Content Delivery Network.

```
<script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
```

2. Define the HTML document as an Angular application with the `ng-app` directive

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

3. Initialize the `name` variable using `ng-init`

```
<body ng-init=" name = 'World' ">
```

Note that ng-init should be used for demonstrative and testing purposes only. When building an actual application, controllers should initialize the data.

4. Bind data from the model to the view on HTML controls. Bind an `<input>` to the `name` property with `ng-model`

```
<input ng-model="name" />
```

5. Display content from the model using double braces `{{ }}`

```
<span>Hello, {{ name }}</span>
```

6. Another way of binding the `name` property is using `ng-bind` instead of handlebars`"{{ }}"`

```
<span ng-bind="name"></span>
```

The last three steps establish the *two way data-binding*. Changes made to the input update the *model*, which is reflected in the *view*.

There is a difference between using handlebars and `ng-bind`. If you use handlebars, you might see the actual `Hello, {{name}}` as the page loads before the expression is resolved (before the data is loaded) whereas if you use `ng-bind`, it will only show the data when the name is resolved. As an alternative the directive `ng-cloak` can be used to prevent handlebars to display before it is compiled.

Section 1.2: Showcasing all common Angular constructs

The following example shows common AngularJS constructs in one file:

```
<!DOCTYPE html>
<html ng-app="myDemoApp">
  <head>
    <style>.started { background: gold; }</style>
    <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
    <script>
      function MyDataService() {
        return {
          getWorlds: function getWorlds() {
            return ["this world", "another world"];
          }
        };
      }

      function DemoController(worldsService) {
        var vm = this;
        vm.messages = worldsService.getWorlds().map(function(w) {
          return "Hello, " + w + "!";
        });
```

```
      }

      function startup($rootScope, $window) {
        $window.alert("Hello, user! Loading worlds...");
        $rootScope.hasStarted = true;
      }

      angular.module("myDemoApp", [/* module dependencies go here */])
        .service("worldsService", [MyDataService])
        .controller("demoController", ["worldsService", DemoController])
        .config(function() {
          console.log('configuring application');
        })
        .run(["$rootScope", "$window", startup]);
    </script>
  </head>
  <body ng-class="{ 'started': hasStarted }" ng-cloak>
    <div ng-controller="demoController as vm">
      <ul>
        <li ng-repeat="msg in vm.messages">{{ msg }}</li>
      </ul>
    </div>
  </body>
</html>
```

Every line of the file is explained below:

Live Demo

1. `ng-app="myDemoApp"`, the ngApp directive that bootstraps the application and tells angular that a DOM element is controlled by a specific `angular.module` named `"myDemoApp"`;
2. `<script src="angular.min.js">` is the first step in bootstrapping the AngularJS library;

Three functions (`MyDataService`, `DemoController`, and `startup`) are declared, which are used (and explained) below.

3. `angular.module(...)` used with an array as the second argument creates a new module. This array is used to supply a list of module dependencies. In this example we chain calls on the result of the `module(...)` function;
4. `.service(...)` creates an Angular Service and returns the module for chaining;
5. `.controller(...)` creates an Angular Controller and returns the module for chaining;
6. `.config(...)` Use this method to register work which needs to be performed on module loading.
7. `.run(...)` makes sure code is run at startup time and takes an array of items as a parameter. Use this method to register work which should be performed when the injector is done loading all modules.
    - the first item is letting Angular know that the `startup` function requires the built-in `$rootScope` service to be injected as an argument;
    - the second item is letting Angular know that the `startup` function requires the built-in `$window` service to be injected as an argument;
    - the *last* item in the array, `startup`, is the actual function to run on startup;
8. `ng-class` is the ngClass directive to set a dynamic `class`, and in this example utilizes `hasStarted` on the `$rootScope` dynamically
9. `ng-cloak` is a directive to prevent the unrendered Angular html template (e.g. "`{{ msg }}`") to be briefly

shown before Angular has fully loaded the application.

10. `ng-controller` is the directive that asks Angular to instantiate a new controller of specific name to orchestrate that part of the DOM;
11. `ng-repeat` is the directive to make Angular iterate over a collection and clone a DOM template for each item;
12. `{{ msg }}` showcases interpolation: on-the-spot rendering of a part of the scope or controller;

Section 1.3: The importance of scope

As Angular uses HTML to extend a web page and plain Javascript to add logic, it makes it easy to create a web page using **ng-app**, **ng-controller** and some built-in directives such as **ng-if**, **ng-repeat**, etc. With the new **controllerAs** syntax, newcomers to Angular users can attach functions and data to their controller instead of using `$scope`.

However, sooner or later, it is important to understand what exactly this `$scope` thing is. It will keep showing up in examples so it is important to have some understanding.

The good news is that it is a simple yet powerful concept.

When you create the following:

```
<div ng-app="myApp">
 <h1>Hello {{ name }}</h1>
</div>
```

Where does **name** live?

The answer is that Angular creates a `$rootScope` object. This is simply a regular Javascript object and so **name** is a property on the `$rootScope` object:

```
angular.module("myApp", [])
  .run(function($rootScope) {
    $rootScope.name = "World!";
  });
```

And just as with global scope in Javascript, it's usually not such a good idea to add items to the global scope or `$rootScope`.

Of course, most of the time, we create a controller and put our required functionality into that controller. But when we create a controller, Angular does it's magic and creates a `$scope` object for that controller. This is sometimes referred to as the **local scope**.

So, creating the following controller:

```
<div ng-app="myApp">
  <div ng-controller="MyController">
    <h1>Hello {{ name }}</h1>
  </div>
</div>
```

would allow the local scope to be accessible via the `$scope` parameter.

```
angular.module("myApp", [])
  .controller("MyController", function($scope) {
    $scope.name = "Mr Local!";
```

```
});
```

A controller without a `$scope` parameter may simply not need it for some reason. But it is important to realize that, **even with controllerAs syntax**, the local scope exists.

As `$scope` is a JavaScript object, Angular magically sets it up to prototypically inherit from `$rootScope`. And as you can imagine, there can be a chain of scopes. For example, you could create a model in a parent controller and attach to it to the parent controller's scope as `$scope.model`.

Then via the prototype chain, a child controller could access that same model locally with `$scope.model`.

None of this is initially evident, as it's just Angular doing its magic in the background. But understanding `$scope` is an important step in getting to know how Angular works.

Section 1.4: Minification in Angular

What is Minification ?

It is the process of removing all unnecessary characters from source code without changing its functionality.

Normal Syntax

If we use normal angular syntax for writing a controller then after minifiying our files it going to break our functionality.

Controller (Before minification) :

```
var app = angular.module('mainApp', []);
app.controller('FirstController', function($scope) {
    $scope.name= 'Hello World !';
});
```

After using minification tool, It will be minified as like below.

```
var app=angular.module("mainApp",[]);app.controller("FirstController",function(e){e.name= 'Hello
World !'})
```

Here, minification removed unnecessary spaces and the $scope variable from code. So when we use this minified code then its not going to print anything on view. Because $scope is a crucial part between controller and view, which is now replaced by the small 'e' variable. So when you run the application it is going to give Unknown Provider 'e' dependency error.

There are two ways of annotating your code with service name information which are minification safe:

Inline Annotation Syntax

```
var app = angular.module('mainApp', []);
app.controller('FirstController', ['$scope', function($scope) {
    $scope.message = 'Hello World !';
}]);
```

$inject Property Annotation Syntax

```
FirstController.$inject = ['$scope'];
var FirstController = function($scope) {
    $scope.message = 'Hello World !';
```

```
}

var app = angular.module('mainApp', []);
app.controller('FirstController', FirstController);
```

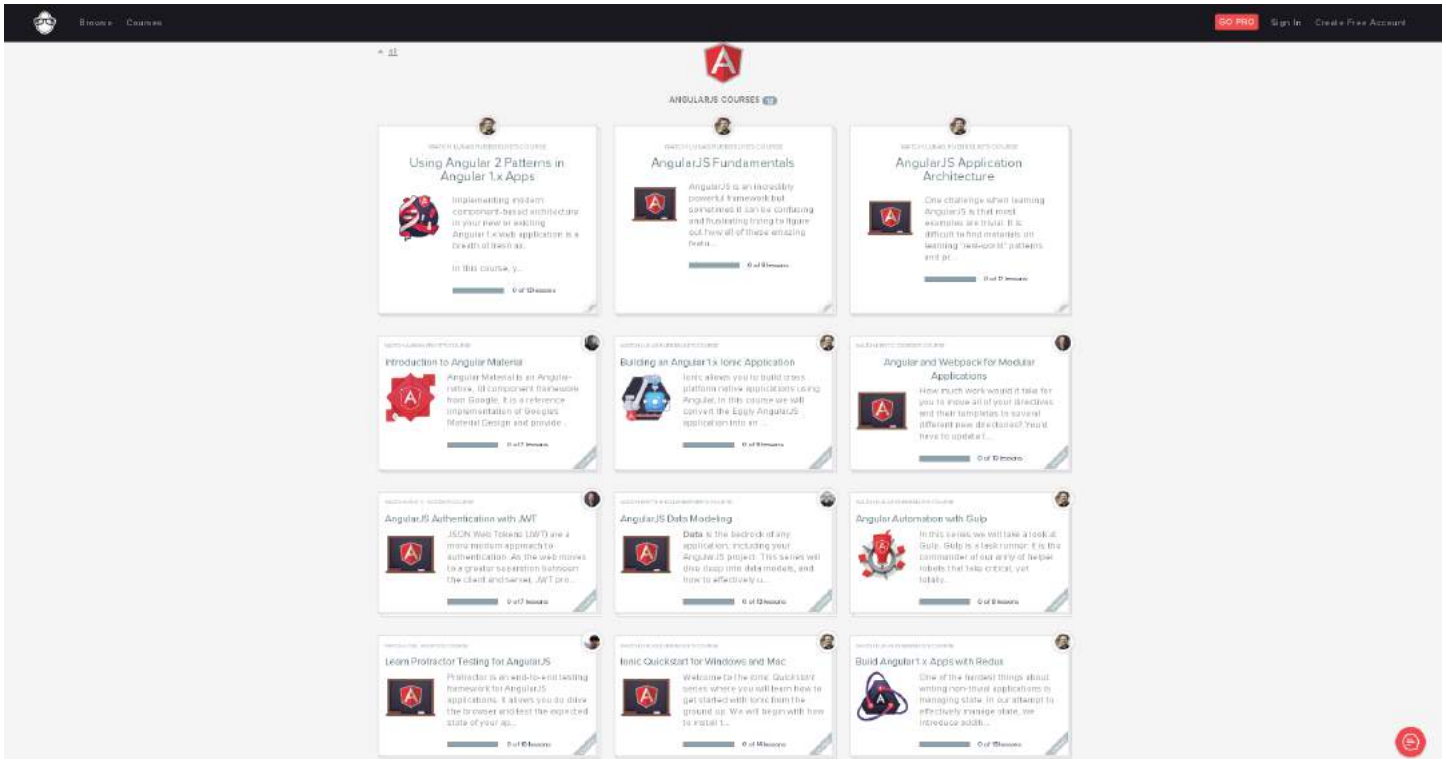
After minification, this code will be

```
var
app=angular.module("mainApp",[]);app.controller("FirstController",["$scope",function(a){a.message="
Hello World !"}]);
```

Here, angular will consider variable 'a' to be treated as $scope, and It will display output as 'Hello World !'.

Section 1.5: AngularJS Getting Started Video Tutorials

There are a lot of good video tutorials for the AngularJS framework on egghead.io



- https://egghead.io/courses/angularjs-app-from-scratch-getting-started
- https://egghead.io/courses/angularjs-application-architecture
- https://egghead.io/courses/angular-material-introduction
- https://egghead.io/courses/building-an-angular-1-x-ionic-application
- https://egghead.io/courses/angular-and-webpack-for-modular-applications
- https://egghead.io/courses/angularjs-authentication-with-jwt
- https://egghead.io/courses/angularjs-data-modeling
- https://egghead.io/courses/angular-automation-with-gulp
- https://egghead.io/courses/learn-protractor-testing-for-angularjs
- https://egghead.io/courses/ionic-quickstart-for-windows
- https://egghead.io/courses/build-angular-1-x-apps-with-redux
- https://egghead.io/courses/using-angular-2-patterns-in-angular-1-x-apps

Section 1.6: The Simplest Possible Angular Hello World

Angular 1 is at heart a DOM compiler. We can pass it HTML, either as a template or just as a regular web page, and

then have it compile an app.

We can tell Angular to treat a region of the page as an *expression* using the `{{ }}` handlebars style syntax. Anything between the curly braces will be compiled, like so:

```
{{ 'Hello' + 'World' }}
```

This will output:

```
HelloWorld
```

ng-app

We tell Angular which portion of our DOM to treat as the master template using the `ng-app` *directive*. A directive is a custom attribute or element that the Angular template compiler knows how to deal with. Let's add an ng-app directive now:

```
<html>
  <head>
    <script src="/angular.js"></script>
  </head>
  <body ng-app>
    {{ 'Hello' + 'World' }}
  </body>
</html>
```

I've now told the body element to be the root template. Anything in it will be compiled.

Directives

Directives are compiler directives. They extend the capabilities of the Angular DOM compiler. This is why **Misko**, the creator of Angular, describes Angular as:

> "What a web browser would have been had it been built for web applications.

We literally create new HTML attributes and elements, and have Angular compile them into an app. `ng-app` is a directive that simply turns on the compiler. Other directives include:

- `ng-click`, which adds a click handler,
- `ng-hide`, which conditionally hides an element, and
- `<form>`, which adds additional behaviour to a standard HTML form element.

Angular comes with around 100 built-in directives which allow you to accomplish most common tasks. We can also write our own, and these will be treated in the same way as the built in directives.

We build an Angular app out of a series of directives, wired together with HTML.

Chapter 2: Controllers

Section 2.1: Your First Controller

A controller is a basic structure used in Angular to preserve scope and handle certain actions within a page. Each controller is coupled with an HTML view.

Below is a basic boilerplate for an Angular app:

```
<!DOCTYPE html>

<html lang="en" ng-app='MyFirstApp'>
    <head>
        <title>My First App</title>

        <!-- angular source -->
        <script src="https://code.angularjs.org/1.5.3/angular.min.js"></script>

        <!-- Your custom controller code -->
        <script src="js/controllers.js"></script>
    </head>
    <body>
        <div ng-controller="MyController as mc">
            <h1>{{ mc.title }}</h1>
            <p>{{ mc.description }}</p>
            <button ng-click="mc.clicked()">
                Click Me!
            </button>
        </div>
    </body>
</html>
```

There are a few things to note here:

```
<html ng-app='MyFirstApp'>
```

Setting the app name with `ng-app` lets you access the application in an external Javascript file, which will be covered below.

```
<script src="js/controllers.js"></script>
```

We'll need a Javascript file where you define your controllers and their actions/data.

```
<div ng-controller="MyController as mc">
```

The `ng-controller` attribute sets the controller for that DOM element and all elements that are children (recursively) below it.

You can have multiple of the same controller (in this case, `MyController`) by saying `... as mc`, we're giving this instance of the controller an alias.

```
<h1>{{ mc.title }}</h1>
```

The `{{ ... }}` notation is an Angular expression. In this case, this will set the inner text of that `<h1>` element to whatever the value of `mc.title` is.

Note: Angular employs dual-way data binding, meaning that regardless of how you update the `mc.title` value, it will be reflected in both the controller and the page.

Also note that Angular expressions do *not* have to reference a controller. An Angular expression can be as simple as `{{ 1 + 2 }}` or `{{ "Hello " + "World" }}`.

```
<button ng-click="mc.clicked()">
```

`ng-click` is an Angular directive, in this case binding the click event for the button to trigger the `clicked()` function of the `MyController` instance.

With those things in mind, let's write an implementation of the `MyController` controller. With the example above, you would write this code in `js/controller.js`.

First, you'll need to instantiate the Angular app in your Javascript.

```
var app = angular.module("MyFirstApp", []);
```

Note that the name we pass here is the same as the name you set in your HTML with the `ng-app` directive.

Now that we have the app object, we can use that to create controllers.

```
app.controller('MyController', function(){
    var ctrl = this;

    ctrl.title = "My First Angular App";
    ctrl.description = "This is my first Angular app!";

    ctrl.clicked = function(){
        alert("MyController.clicked()");
    };
});
```

Note: For anything that we want to be a part of the controller instance, we use the **`this`** keyword.

This is all that is required to build a simple controller.

Section 2.2: Creating Controllers, Minification safe

There are a couple different ways to protect your controller creation from minification.

The first is called inline array annotation. It looks like the following:

```
var app = angular.module('app');
app.controller('sampleController', ['$scope', '$http', function(a, b){
    //logic here
}]);
```

The second parameter of the controller method can accept an array of dependencies. As you can see I've defined `$scope` and `$http` which should correspond to the parameters of the controller function in which `a` will be the `$scope`, and `b` would be `$http`. Take note that the last item in the array should be your controller function.

The second option is using the `$inject` property. It looks like the following:

```
var app = angular.module('app');
app.controller('sampleController', sampleController);
```

```
sampleController.$inject = ['$scope', '$http'];
function sampleController(a, b) {
    //logic here
}
```

This does the same thing as inline array annotation but provides a different styling for those that prefer one option over the other.

The order of injected dependencies is important

When injecting dependencies using the array form, be sure that the list of the dependencies match its corresponding list of arguments passed to the controller function.

Note that in the following example, `$scope` and `$http` are reversed. This will cause a problem in the code.

```
// Intentional Bug: injected dependencies are reversed which will cause a problem
app.controller('sampleController', ['$scope', '$http',function($http, $scope) {
    $http.get('sample.json');
}]);
```

Section 2.3: Using ControllerAs in Angular JS

In Angular `$scope` is the glue between the Controller and the View that helps with all of our data binding needs. Controller As is another way of binding controller and view and is mostly recommended to use. Basically these are the two controller constructs in Angular (i.e $scope and Controller As).

Different ways of using Controller As are -

controllerAs View Syntax

```
<div ng-controller="CustomerController as customer">
    {{ customer.name }}
</div>
```

controllerAs Controller Syntax

```
function CustomerController() {
    this.name = {};
    this.sendMessage = function() { };
}
```

controllerAs with vm

```
function CustomerController() {
    /*jshint validthis: true */
    var vm = this;
    vm.name = {};
    vm.sendMessage = function() { };
}
```

`controllerAs` is syntactic sugar over `$scope`. You can still bind to the View and still access `$scope` methods. Using `controllerAs`, is one of the best practices suggested by the angular core team. There are many reason for this, few of them are -

- `$scope` is exposing the members from the controller to the view via an intermediary object. By setting `this.*`, we can expose just what we want to expose from the controller to the view. It also follow the

standard JavaScript way of using this.

- using `controllerAs` syntax, we have more readable code and the parent property can be accessed using the alias name of the parent controller instead of using the `$parent` syntax.
- It promotes the use of binding to a "dotted" object in the View (e.g. customer.name instead of name), which is more contextual, easier to read, and avoids any reference issues that may occur without "dotting".
- Helps avoid using `$parent` calls in Views with nested controllers.
- Use a capture variable for this when using the `controllerAs` syntax. Choose a consistent variable name such as `vm`, which stands for ViewModel. Because, **`this`** keyword is contextual and when used within a function inside a controller may change its context. Capturing the context of this avoids encountering this problem.

NOTE: using `controllerAs` syntax add to current scope reference to current controller, so it available as field

```
<div ng-controller="Controller as vm>...</div>
```

`vm` is available as `$scope.vm`.

Section 2.4: Creating Minification-Safe Angular Controllers

To create minification-safe angular controllers, you will change the `controller` function parameters.

The second argument in the `module.controller` function should be passed an **array**, where the **last parameter** is the **controller function**, and every parameter before that is the **name** of each injected value.

This is different from the normal paradigm; that takes the **controller function** with the injected arguments.

Given:

```
var app = angular.module('myApp');
```

The controller should look like this:

```
app.controller('ctrlInject',
    [
        /* Injected Parameters */
        '$Injectable1',
        '$Injectable2',
        /* Controller Function */
        function($injectable1Instance, $injectable2Instance) {
            /* Controller Content */
        }
    ]
);
```

Note: *The names of injected parameters are not required to match, but they will be bound in order.*

This will minify to something similar to this:

```
var
a=angular.module('myApp');a.controller('ctrlInject',['$Injectable1','$Injectable2',function(b,c){/*
Controller Content */}]);
```

The minification process will replace every instance of `app` with `a`, every instance of `$Injectable1Instance` with `b`, and every instance of `$Injectable2Instance` with `c`.

Section 2.5: Creating Controllers

```
angular
    .module('app')
    .controller('SampleController', SampleController)

SampleController.$inject = ['$log', '$scope'];
function SampleController($log, $scope){
    $log.debug('*****SampleController******');

    /* Your code below */
}
```

Note: The `.$inject` will make sure your dependencies doesn't get scrambled after minification. Also, make sure it's in order with the named function.

Section 2.6: Nested Controllers

Nesting controllers chains the `$scope` as well. Changing a `$scope` variable in the nested controller changes the same `$scope` variable in the parent controller.

```
.controller('parentController', function ($scope) {
    $scope.parentVariable = "I'm the parent";
});

.controller('childController', function ($scope) {
    $scope.childVariable = "I'm the child";

    $scope.childFunction = function () {
        $scope.parentVariable = "I'm overriding you";
    };
});
```

Now let's try to handle both of them, nested.

```
<body ng-controller="parentController">
    What controller am I? {{parentVariable}}
    <div ng-controller="childController">
        What controller am I? {{childVariable}}
        <button ng-click="childFunction()"> Click me to override! </button>
    </div>
</body>
```

Nesting controllers may have it's benefits, but one thing must be kept in mind when doing so. Calling the `ngController` directive creates a new instance of the controller - which can often create confusion and unexpected results.

Chapter 3: Built-in directives

Section 3.1: Angular expressions - Text vs. Number

This example demonstrates how Angular expressions are evaluated when using `type="text"` and `type="number"` for the input element. Consider the following controller and view:

Controller

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

app.controller('ctrl', function($scope) {
    $scope.textInput = {
        value: '5'
    };
    $scope.numberInput = {
        value: 5
    };
});
```

View

```
<div ng-app="app" ng-controller="ctrl">
    <input type="text" ng-model="textInput.value">
    {{ textInput.value + 5 }}
    <input type="number" ng-model="numberInput.value">
    {{ numberInput.value + 5 }}
</div>
```

- When using `+` in an expression bound to *text* input, the operator will **concatenate** the strings (first example), displaying 55 on the screen`*`.
- When using `+` in an expression bound to *number* input, the operator return the **sum** of the numbers (second example), displaying 10 on the screen`*`.

`*` - That is until the user changes the value in the input field, afterward the display will change accordingly.

Working Example

Section 3.2: ngIf

`ng-if` is a directive similar to `ng-show` but inserts or removes the element from the DOM instead of simply hiding it. Angular 1.1.5 introduced ng-If directive. You can Use ng-if directive above 1.1.5 versions. This is useful because Angular will not process digests for elements inside a removed `ng-if` reducing the workload of Angular especially for complex data bindings.

Unlike `ng-show`, the `ng-if` directive creates a child scope which uses prototypal inheritance. This means that setting a primitive value on the child scope will not apply to the parent. To set a primitive on the parent scope the `$parent` property on the child scope will have to be used.

JavaScript

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

angular.module('MyApp').controller('myController', ['$scope', '$window', function
myController($scope, $window) {
    $scope.currentUser= $window.localStorage.getItem('userName');
```

```
}]);
```

View

```
<div ng-controller="myController">
    <div ng-if="currentUser">
        Hello, {{currentUser}}
    </div>
    <div ng-if="!currentUser">
        <a href="/login">Log In</a>
        <a href="/register">Register</a>
    </div>
</div>
```

DOM If `currentUser` Is Not Undefined

```
<div ng-controller="myController">
    <div ng-if="currentUser">
        Hello, {{currentUser}}
    </div>
    <!-- ng-if: !currentUser -->
</div>
```

DOM If `currentUser` Is Undefined

```
<div ng-controller="myController">
    <!-- ng-if: currentUser -->
    <div ng-if="!currentUser">
        <a href="/login">Log In</a>
        <a href="/register">Register</a>
    </div>
</div>
```

Working Example

Function Promise

The ngIf directive accepts functions as well, which logically require to return true or false.

```
<div ng-if="myFunction()">
    <span>Span text</span>
</div>
```

The span text will only appear if the function returns true.

```
$scope.myFunction = function() {
    var result = false;
    // Code to determine the boolean value of result
    return result;
};
```

As any Angular expression the function accepts any kind of variables.

Section 3.3: ngCloak

> The `ngCloak` directive is used to prevent the Angular html template from being briefly displayed by the browser in its raw (uncompiled) form while your application is loading. - View source

HTML

```
<div ng-cloak>
  <h1>Hello {{ name }}</h1>
</div>
```

`ngCloak` can be applied to the body element, but the preferred usage is to apply multiple ngCloak directives to small portions of the page to permit progressive rendering of the browser view.

The `ngCloak` directive has no parameters.

See also: Preventing flickering

Section 3.4: ngRepeat

`ng-repeat` is a built in directive in Angular which lets you iterate an array or an object and gives you the ability to repeat an element once for each item in the collection.

ng-repeat an array

```
<ul>
    <li ng-repeat="item in itemCollection">
       {{item.Name}}
    </li>
</ul>
```

Where:
item = individual item in the collection
itemCollection = The array you are iterating

ng-repeat an object

```
<ul>
    <li ng-repeat="(key, value) in myObject">
       {{key}} : {{value}}
    </li>
</ul>
```

Where:
key = the property name
value = the value of the property
myObject = the object you are iterating

filter your ng-repeat by user input

```
<input type="text" ng-model="searchText">
<ul>
    <li ng-repeat="string in stringArray | filter:searchText">
       {{string}}
    </li>
</ul>
```

Where:
searchText = the text that the user wants to filter the list by
stringArray = an array of strings, e.g. `['string', 'array']`

You can also display or reference the filtered items elsewhere by assigning the filter output an alias with `as`

`aliasName`, like so:

```
<input type="text" ng-model="searchText">
<ul>
    <li ng-repeat="string in stringArray | filter:searchText as filteredStrings">
       {{string}}
    </li>
</ul>
<p>There are {{filteredStrings.length}} matching results</p>
```

ng-repeat-start and ng-repeat-end

To repeat multiple DOM elements by defining a start and an end point you can use the `ng-repeat-start` and `ng-repeat-end` directives.

```
<ul>
    <li ng-repeat-start="item in [{a: 1, b: 2}, {a: 3, b:4}]">
        {{item.a}}
    </li>
    <li ng-repeat-end>
        {{item.b}}
    </li>
</ul>
```

Output:

- 1
- 2
- 3
- 4

It is important to always close `ng-repeat-start` with `ng-repeat-end`.

Variables

`ng-repeat` also exposes these variables inside the expression

Variable	Type	Details
`$index`	Number	Equals to the index of the current iteration ($index===0 will evaluate to true at the first iterated element; see `$first`)
`$first`	Boolean	Evaluates to true at the first iterated element
`$last`	Boolean	Evaluates to true at the last iterated element
`$middle`	Boolean	Evaluates to true if the element is between the `$first` and `$last`
`$even`	Boolean	Evaluates to true at an even numbered iteration (equivalent to `$index%2===0`)
`$odd`	Boolean	Evaluates to true at an odd numbered iteration (equivalent to `$index%2===1`)

Performance considerations

Rendering `ngRepeat` can become slow, especially when using large collections.

If the objects in the collection have an identifier property, you should always `track by` the identifier instead of the whole object, which is the default functionality. If no identifier is present, you can always use the built-in `$index`.

```
<div ng-repeat="item in itemCollection track by item.id">
<div ng-repeat="item in itemCollection track by $index">
```

Scope of ngRepeat

`ngRepeat` will always create an isolated child scope so care must be taken if the parent scope needs to be accessed inside the repeat.

Here is a simple example showing how you can set a value in your parent scope from a click event inside of `ngRepeat`.

```
scope val:  {{val}}<br/>
ctrlAs val: {{ctrl.val}}
<ul>
    <li ng-repeat="item in itemCollection">
        <a href="#" ng-click="$parent.val=item.value; ctrl.val=item.value;">
            {{item.label}} {{item.value}}
        </a>
    </li>
</ul>

$scope.val = 0;
this.val = 0;

$scope.itemCollection = [{
    id: 0,
    value: 4.99,
    label: 'Football'
},
{
    id: 1,
    value: 6.99,
    label: 'Baseball'
},
{
    id: 2,
    value: 9.99,
    label: 'Basketball'
}];
```

If there was only `val = item.value` at `ng-click` it won't update the `val` in the parent scope because of the isolated scope. That's why the parent scope is accessed with `$parent` reference or with the `controllerAs` syntax (e.g. `ng-controller="mainController as ctrl"`).

Nested ng-repeat

You can also use nested ng-repeat.

```
<div ng-repeat="values in test">
    <div ng-repeat="i in values">
      [{{$parent.$index}},{{$index}}] {{i}}
    </div>
</div>

var app = angular.module("myApp", []);
app.controller("ctrl", function($scope) {
  $scope.test = [
    ['a', 'b', 'c'],
    ['d', 'e', 'f']
  ];
});
```

Here to access the index of parent ng-repeat inside child ng-repeat, you can use `$parent.$index`.

Section 3.5: Built-In Directives Cheat Sheet

`ng-app` Sets the AngularJS section.

`ng-init` Sets a default variable value.

`ng-bind` Alternative to {{ }} template.

`ng-bind-template` Binds multiple expressions to the view.

`ng-non-bindable` States that the data isn't bindable.

`ng-bind-html` Binds inner HTML property of an HTML element.

`ng-change` Evaluates specified expression when the user changes the input.

`ng-checked` Sets the checkbox.

`ng-class` Sets the css class dynamically.

`ng-cloak` Prevents displaying the content until AngularJS has taken control.

`ng-click` Executes a method or expression when element is clicked.

`ng-controller` Attaches a controller class to the view.

`ng-disabled` Controls the form element's disabled property

`ng-form` Sets a form

`ng-href` Dynamically bind AngularJS variables to the href attribute.

`ng-include` Used to fetch, compile and include an external HTML fragment to your page.

`ng-if` Remove or recreates an element in the DOM depending on an expression

`ng-switch` Conditionally switch control based on matching expression.

`ng-model` Binds an input,select, textarea etc elements with model property.

`ng-readonly` Used to set readonly attribute to an element.

`ng-repeat` Used to loop through each item in a collection to create a new template.

`ng-selected` Used to set selected option in element.

`ng-show/ng-hide` Show/Hide elements based on an expression.

`ng-src` Dynamically bind AngularJS variables to the src attribute.

`ng-submit` Bind angular expressions to onsubmit events.

`ng-value` Bind angular expressions to the value of .

`ng-required` Bind angular expressions to onsubmit events.

`ng-style` Sets CSS style on an HTML element.

`ng-pattern` Adds the pattern validator to ngModel.

`ng-maxlength` Adds the maxlength validator to ngModel.

`ng-minlength` Adds the minlength validator to ngModel.

`ng-classeven` Works in conjunction with ngRepeat and take effect only on odd (even) rows.

`ng-classodd` Works in conjunction with ngRepeat and take effect only on odd (even) rows.

`ng-cut` Used to specify custom behavior on cut event.

`ng-copy` Used to specify custom behavior on copy event.

`ng-paste` Used to specify custom behavior on paste event.

`ng-options` Used to dynamically generate a list of elements for the element.

`ng-list` Used to convert string into list based on specified delimiter.

`ng-open` Used to set the open attribute on the element, if the expression inside ngOpen is truthy.

Source (edited a bit)

Section 3.6: ngInclude

ng-include allows you to delegate the control of one part of the page to a specific controller. You may want to do this because the complexity of that component is becoming such that you want to encapsulate all the logic in a dedicated controller.

An example is:

```
<div ng-include
    src="'/gridview'"
    ng-controller='gridController as gc'>
</div>
```

Note that the `/gridview` will need to be served by the web server as a distinct and legitimate url.

Also, note that the `src`-attribute accepts an Angular expression. This could be a variable or a function call for example or, like in this example, a string constant. In this case you need to make sure to **wrap the source URL in single quotes**, so it will be evaluated as a string constant. This is a common source of confusion.

Within the `/gridview` html, you can refer to the `gridController` as if it were wrapped around the page, eg:

```
<div class="row">
  <button type="button" class="btn btn-default" ng-click="gc.doSomething()"></button>
</div>
```

Section 3.7: ng-model-options

`ng-model-options` allows to change the default behavior of `ng-model`, this directive allows to register events that will fire when the ng-model is updated and to attach a debounce effect.

This directive accepts an expression that will evaluate to a definition object or a reference to a scope value.

Example:

```
<input type="text" ng-model="myValue" ng-model-options="{'debounce': 500}">
```

The above example will attach a debounce effect of 500 milliseconds on `myValue`, which will cause the model to update 500 ms after the user finished typing over the `input` (that is, when the `myValue` finished updating).

Available object properties

1. `updateOn`: specifies which event should be bound to the input

   ```
   ng-model-options="{ updateOn: 'blur'}" // will update on blur
   ```

2. `debounce`: specifies a delay of some millisecond towards the model update

   ```
   ng-model-options="{'debounce': 500}" // will update the model after 1/2 second
   ```

3. `allowInvalid`: a boolean flag allowing for an invalid value to the model, circumventing default form validation, by default these values would be treated as **`undefined`**.
4. `getterSetter`: a boolean flag indicating if to treat the `ng-model` as a getter/setter function instead of a plain model value. The function will then run and return the model value.

   Example:

   ```
   <input type="text" ng-model="myFunc" ng-model-options="{'getterSetter': true}">

   $scope.myFunc = function() {return "value";}
   ```

5. `timezone`: defines the timezone for the model if the input is of the `date` or `time`. types

Section 3.8: ngCopy

The `ngCopy` directive specifies behavior to be run on a copy event.

Prevent a user from copying data

```
<p ng-copy="blockCopy($event)">This paragraph cannot be copied</p>
```

In the controller

```
$scope.blockCopy = function(event) {
    event.preventDefault();
    console.log("Copying won't work");
}
```

Section 3.9: ngPaste

The `ngPaste` directive specifies custom behavior to run when a user pastes content

```
<input ng-paste="paste=true" ng-init="paste=false" placeholder='paste here'>
pasted: {{paste}}
```

Section 3.10: ngClick

The `ng-click` directive attaches a click event to a DOM element.

The `ng-click` directive allows you to specify custom behavior when an element of DOM is clicked.

It is useful when you want to attach click events on buttons and handle them at your controller.

This directive accepts an expression with the events object available as `$event`

HTML

```
<input ng-click="onClick($event)">Click me</input>
```

Controller

```
.controller("ctrl", function($scope) {
    $scope.onClick = function(evt) {
        console.debug("Hello click event: %o ",evt);
    }
})
```

HTML

```
<button ng-click="count = count + 1" ng-init="count=0">
  Increment
</button>
<span>
  count: {{count}}
</span>
```

HTML

```
<button ng-click="count()" ng-init="count=0">
  Increment
</button>
<span>
  count: {{count}}
</span>
```

Controller

```
...

$scope.count = function(){
    $scope.count = $scope.count + 1;
}
...
```

When the button is clicked, an invocation of the `onClick` function will print "Hello click event" followed by the event object.

Section 3.11: ngList

The `ng-list` directive is used to convert a delimited string from a text input to an array of strings or vice versa.

The `ng-list` directive uses a default delimiter of `", "` (comma space).

You can set the delimiter manually by assigning `ng-list` a delimeter like this `ng-list="; "`.

In this case the delimiter is set to a semi colon followed by a space.

By default `ng-list` has an attribute `ng-trim` which is set to true. `ng-trim` when false, will respect white space in your delimiter. By default, `ng-list` does not take white space into account unless you set `ng-trim="false"`.

Example:

```
angular.module('test', [])
  .controller('ngListExample', ['$scope', function($scope) {
    $scope.list = ['angular', 'is', 'cool!'];
}]);
```

A customer delimiter is set to be `;`. And the model of the input box is set to the array that was created on the scope.

```
  <body ng-app="test" ng-controller="ngListExample">
    <input ng-model="list" ng-list="; " ng-trim="false">
  </body>
```

The input box will display with the content: `angular; is; cool!`

Section 3.12: ngOptions

`ngOptions` is a directive that simplifies the creation of a html dropdown box for the selection of an item from an array that will be stored in a model. The ngOptions attribute is used to dynamically generate a list of `<option>` elements for the `<select>` element using the array or object obtained by evaluating the ngOptions comprehension expression.

With `ng-options` the markup can be reduced to just a select tag and the directive will create the same select:

```
<select ng-model="selectedFruitNgOptions"
       ng-options="curFruit as curFruit.label for curFruit in fruit">
</select>
```

There is anther way of creating `SELECT` options using `ng-repeat`, but it is not recommended to use `ng-repeat` as it is mostly used for general purpose like, the `forEach` just to loop. Whereas `ng-options` is specifically for creating `SELECT` tag options.

Above example using `ng-repeat` would be

```
<select ng-model="selectedFruit">
    <option ng-repeat="curFruit in fruit" value="{{curFruit}}">
        {{curFruit.label}}
    </option>
</select>
```

FULL EXAMPLE

Lets see the above example in detail also with some variations in it.

Data model for the example:

```
$scope.fruit = [
```

```
    { label: "Apples", value: 4, id: 2 },
    { label: "Oranges", value: 2, id: 1 },
    { label: "Limes", value: 4, id: 4 },
    { label: "Lemons", value: 5, id: 3 }
];
```

```
<!-- label for value in array -->
<select ng-options="f.label for f in fruit" ng-model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option value="{ label: "Apples", value: 4, id: 2 }"> Apples </option>
```

Effects:

`f.label` will be the label of the `<option>` and the value will contain the entire object.

FULL EXAMPLE

```
<!-- select as label for value in array -->
<select ng-options="f.value as f.label for f in fruit" ng-model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option value="4"> Apples </option>
```

Effects:

`f.value` (4) will be the value in this case while the label is still the same.

FULL EXAMPLE

```
<!-- label group by group for value in array -->
<select ng-options="f.label group by f.value for f in fruit" ng-model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option value="{ label: "Apples", value: 4, id: 2 }"> Apples </option>
```

Effects:

Options will be grouped based on there `value`. Options with same `value` will fall under one category

FULL EXAMPLE

```
<!-- label disable when disable for value in array -->
<select ng-options="f.label disable when f.value == 4 for f in fruit" ng-
model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option disabled="" value="{ label: "Apples", value: 4, id: 2 }"> Apples </option>
```

Effects:

"Apples" and "Limes" will be disabled (unable to select) because of the condition `disable when f.value==4`. All

options with `value=4` shall be disabled

FULL EXAMPLE

```
<!-- label group by group for value in array track by trackexpr -->
<select ng-options="f.value as f.label group by f.value for f in fruit track by f.id" ng-
model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option value="4"> Apples </option>
```

Effects:

There is not visual change when using `trackBy`, but Angular will detect changes by the `id` instead of by reference which is most always a better solution.

FULL EXAMPLE

```
<!-- label for value in array | orderBy:orderexpr track by trackexpr -->
<select ng-options="f.label for f in fruit | orderBy:'id' track by f.id" ng-
model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option disabled="" value="{ label: "Apples", value: 4, id: 2 }"> Apples </option>
```

Effects:

`orderBy` is a AngularJS standard filter which arranges options in ascending order(by default) so "Oranges" in this will appear 1st since its `id` = 1.

FULL EXAMPLE

All `<select>` with `ng-options` must have `ng-model` attached.

Section 3.13: ngSrc

Using Angular markup like `{{hash}}` in a src attribute doesn't work right. The browser will fetch from the URL with the literal text `{{hash}}` until Angular replaces the expression inside `{{hash}}`. `ng-src` directive overrides the original `src` attribute for the image tag element and solves the problem

```
<div ng-init="pic = 'pic_angular.jpg'">
    <h1>Angular</h1>
    <img ng-src="{{pic}}">
</div>
```

Section 3.14: ngModel

With ng-model you can bind a variable to any type of input field. You can display the variable using double curly braces, eg `{{myAge}}`.

```
<input type="text" ng-model="myName">
<p>{{myName}}</p>
```

As you type in the input field or change it in any way you will see the value in the paragraph update instantly.

The ng-model variable, in this instance, will be available in your controller as `$scope.myName`. If you are using the `controllerAs` syntax:

```
<div ng-controller="myCtrl as mc">
    <input type="text" ng-model="mc.myName">
    <p>{{mc.myName}}</p>
</div>
```

You will need to refer to the controller's scope by pre-pending the controller's alias defined in the ng-controller attribute to the ng-model variable. This way you won't need to inject `$scope` into your controller to reference your ng-model variable, the variable will be available as `this.myName` inside your controller's function.

Section 3.15: ngClass

Let's assume that you need to show the status of a user and you have several possible CSS classes that could be used. Angular makes it very easy to choose from a list of several possible classes which allow you to specify an object list that include conditionals. Angular is able to use the correct class based on the truthiness of the conditionals.

Your object should contain key/value pairs. The key is a class name that will be applied when the value (conditional) evaluates to true.

```
<style>
    .active { background-color: green; color: white; }
    .inactive { background-color: gray; color: white; }
    .adminUser { font-weight: bold; color: yellow; }
    .regularUser { color: white; }
</style>

<span ng-class="{
   active: user.active,
   inactive: !user.active,
   adminUser: user.level === 1,
   regularUser: user.level === 2
}">John Smith</span>
```

Angular will check the `$scope.user` object to see the `active` status and the `level` number. Depending on the values in those variables, Angular will apply the matching style to the ****.

Section 3.16: ngDblclick

The `ng-dblclick` directive is useful when you want to bind a double-click event into your DOM elements.

This directive accepts an expression

HTML

```
<input type="number" ng-model="num = num + 1" ng-init="num=0">

<button ng-dblclick="num++">Double click me</button>
```

In the above example, the value held at the `input` will be incremented when the button is double clicked.

Section 3.17: ngHref

ngHref is used instead of href attribute, if we have a angular expressions inside href value. The ngHref directive overrides the original href attribute of an html tag using href attribute such as tag, tag etc.

The ngHref directive makes sure the link is not broken even if the user clicks the link before AngularJS has evaluated the code.

Example 1

```
<div ng-init="linkValue = 'http://stackoverflow.com'">
    <p>Go to <a ng-href="{{linkValue}}">{{linkValue}}</a>!</p>
</div>
```

Example 2 This example dynamically gets the href value from input box and load it as href value.

```
<input ng-model="value" />
<a id="link" ng-href="{{value}}">link</a>
```

Example 3

```
<script>
angular.module('angularDoc', [])
.controller('myController', function($scope) {
  // Set some scope value.
  // Here we set bootstrap version.
  $scope.bootstrap_version = '3.3.7';

  // Set the default layout value
  $scope.layout = 'normal';
});
</script>
<!-- Insert it into Angular Code -->
<link rel="stylesheet" ng-href="//maxcdn.bootstrapcdn.com/bootstrap/{{ bootstrap_version
}}/css/bootstrap.min.css">
<link rel="stylesheet" ng-href="layout-{{ layout }}.css">
```

Section 3.18: ngPattern

The `ng-pattern` directive accepts an expression that evaluates to a regular expression pattern and uses that pattern to validate a textual input.

Example:

Lets say we want an `<input>` element to become valid when it's value (ng-model) is a valid IP address.

Template:

```
<input type="text" ng-model="ipAddr" ng-pattern="ipRegex" name="ip" required>
```

Controller:

```
$scope.ipRegex =
/\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b/;
```

Section 3.19: ngShow and ngHide

The `ng-show` directive shows or hides the HTML element based on if the expression passed to it is true or false. If the value of the expression is falsy then it will hide. If it is truthy then it will show.

The `ng-hide` directive is similar. However, if the value is falsy it will show the HTML element. When the expression is truthy it will hide it.

Working JSBin Example

Controller:

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

angular.module('app')
  .controller('ExampleController', ExampleController);

function ExampleController() {

  var vm = this;

  //Binding the username to HTML element
  vm.username = '';

  //A taken username
  vm.taken_username = 'StackOverflow';

}
```

View

```
<section ng-controller="ExampleController as main">

    <p>Enter Password</p>
    <input ng-model="main.username" type="text">

    <hr>

    <!-- Will always show as long as StackOverflow is not typed in -->
    <!-- The expression is always true when it is not StackOverflow -->
    <div style="color:green;" ng-show="main.username != main.taken_username">
      Your username is free to use!
    </div>

    <!-- Will only show when StackOverflow is typed in -->
    <!-- The expression value becomes falsy -->
    <div style="color:red;" ng-hide="main.username != main.taken_username">
      Your username is taken!
    </div>

    <p>Enter 'StackOverflow' in username field to show ngHide directive.</p>

 </section>
```

Section 3.20: ngRequired

The `ng-required` adds or removes the `required` validation attribute on an element, which in turn will enable and disable the `require` validation key for the `input`.

It is used to optionally define if an `input` element is required to have a non-empty value. The directive is helpful when designing validation on complex HTML forms.

HTML

```
<input type="checkbox" ng-model="someBooleanValue">
<input type="text" ng-model="username" ng-required="someBooleanValue">
```

Section 3.21: ngMouseenter and ngMouseleave

The `ng-mouseenter` and `ng-mouseleave` directives are useful to run events and apply CSS styling when you hover into or out of your DOM elements.

The `ng-mouseenter` directive runs an expression one a mouse enter event (when the user enters his mouse pointer over the DOM element this directive resides in)

HTML

```
<div ng-mouseenter="applyStyle = true" ng-class="{'active': applyStyle}">
```

At the above example, when the user points his mouse over the `div`, `applyStyle` turns to **`true`**, which in turn applies the `.active` CSS class at the `ng-class`.

The `ng-mouseleave` directive runs an expression one a mouse exit event (when the user takes his mouse cursor away from the DOM element this directive resides in)

HTML

```
<div ng-mouseenter="applyStyle = true" ng-mouseleaver="applyStyle = false" ng-class="{'active':
applyStyle}">
```

Reusing the first example, now when the user takes him mouse pointer away from the div, the `.active` class is removed.

Section 3.22: ngDisabled

This directive is useful to limit input events based on certain existing conditions.

The `ng-disabled` directive accepts and expression that should evaluate to either a truthy or a falsy values.

`ng-disabled` is used to conditionally apply the `disabled` attribute on an `input` element.

HTML

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

<button ng-disabled="vm.name.length===0" ng-click="vm.submitMe">Submit</button>
```

`vm.name.length===0` is evaluated to true if the `input`'s length is 0, which is turn disables the button, disallowing the user to fire the click event of `ng-click`

Section 3.23: ngValue

Mostly used under `ng-repeat` ngValue is useful when dynamically generating lists of radio buttons using ngRepeat

```
<script>
   angular.module('valueExample', [])
     .controller('ExampleController', ['$scope', function($scope) {
       $scope.names = ['pizza', 'unicorns', 'robots'];
       $scope.my = { favorite: 'unicorns' };
     }]);
</script>
 <form ng-controller="ExampleController">
   <h2>Which is your favorite?</h2>
     <label ng-repeat="name in names" for="{{name}}">
       {{name}}
       <input type="radio"
              ng-model="my.favorite"
              ng-value="name"
              id="{{name}}"
              name="favorite">
     </label>
   <div>You chose {{my.favorite}}</div>
 </form>
```

Working plnkr

Chapter 4: Modules

Section 4.1: Modules

Module serves as a container of different parts of your app such as controllers, services, filters, directives, etc. Modules can be referenced by other modules through Angular's dependency injection mechanism.

Creating a module:

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

Array [ ] passed in above example is the *list of modules* `app` depends on, if there are no dependencies then we pass Empty Array i.e. [ ].

Injecting a module as a dependency of another module:

```
angular.module('app', [
    'app.auth',
    'app.dashboard'
]);
```

Referencing a module:

```
angular
    .module('app');
```

Section 4.2: Modules

Module is a container for various parts of your applications - controller, services, filters, directive, etc.

Why to use Modules
Most applications have a main method that instantiates and wires together the different parts of the application. Angular apps don't have main method.
But in AngularJs the declarative process is easy to understand and one can package code as reusable modules. Modules can be loaded in any order because modules delay execution.

declare a module

```
var app = angular.module('myApp', []);
// Empty array is list of modules myApp is depends on.
// if there are any required dependancies,
// then you can add in module, Like ['ngAnimate']

app.controller('myController', function() {

  // write your business logic here
});
```

Module Loading and Dependencies

1. Configuration Blocks: get executed during provider and configuration phase.

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

```
config(function(injectables) {
  // here you can only inject providers in to config blocks.
});
```

2. Run Blocks: get executed after the injector is created and are used to start the application.

```
angular.module('myModule', []).
run(function(injectables) {
  // here you can only inject instances in to config blocks.
});
```

Chapter 5: Components

Parameter	Details
=	For using two-way data binding. This means that if you update that variable in your component scope, the change will be reflected on the parent scope.
<	One-way bindings when we just want to read a value from a parent scope and not update it.
@	String parameters.
&	For callbacks in case your component needs to output something to its parent scope.
-	-
LifeCycle Hooks	**Details** (requires angular.version >= 1.5.3)
$onInit()	Called on each controller after all the controllers on an element have been constructed and had their bindings initialized. This is a good place to put initialization code for your controller.
$onChanges(changesObj)	Called whenever one-way bindings are updated. The `changesObj` is a hash whose keys are the names of the bound properties that have changed, and the values are an object of the form `{ currentValue, previousValue, isFirstChange() }`.
$onDestroy()	Called on a controller when its containing scope is destroyed. Use this hook for releasing external resources, watches and event handlers.
$postLink()	Called after this controller's element and its children have been linked. This hook can be considered analogous to the ngAfterViewInit and ngAfterContentInit hooks in Angular 2.
$doCheck()	Called on each turn of the digest cycle. Provides an opportunity to detect and act on changes. Any actions that you wish to take in response to the changes that you detect must be invoked from this hook; implementing this has no effect on when $onChanges is called.

Section 5.1: Basic Components and LifeCycle Hooks

What's a component?

- A component is basically a directive that uses a simpler configuration and that is suitable for a component-based architecture, which is what Angular 2 is all about. Think of a component as a widget: A piece of HTML code that you can reuse in several different places in your web application.

Component

```
angular.module('myApp', [])
    .component('helloWorld', {
        template: '<span>Hello World!</span>'
    });
```

Markup

```
<div ng-app="myApp">
  <hello-world> </hello-world>
</div>
```

Live Demo

Using External data in Component:

We could add a parameter to pass a name to our component, which would be used as follows:

```
angular.module("myApp", [])
  .component("helloWorld",{
```

```
    template: '<span>Hello {{$ctrl.name}}!</span>',
    bindings: { name: '@' }
  });
```

Markup

```
<div ng-app="myApp">
  <hello-world name="'John'" > </hello-world>
</div>
```

Live Demo

Using Controllers in Components

Let's take a look at how to add a controller to it.

```
angular.module("myApp", [])
  .component("helloWorld",{
      template: "Hello {{$ctrl.name}}, I'm {{$ctrl.myName}}!",
      bindings: { name: '@' },
      controller: function(){
        this.myName = 'Alain';
      }
  });
```

Markup

```
<div ng-app="myApp">
  <hello-world name="John"> </hello-world>
</div>
```

CodePen Demo

Parameters passed to the component are available in the controller's scope just before its `$onInit` function gets called by Angular. Consider this example:

```
angular.module("myApp", [])
  .component("helloWorld",{
      template: "Hello {{$ctrl.name}}, I'm {{$ctrl.myName}}!",
      bindings: { name: '@' },
      controller: function(){
        this.$onInit = function() {
          this.myName = "Mac" + this.name;
        }
      }
  });
```

In the template from above, this would render "Hello John, I'm MacJohn!".

Note that `$ctrl` is the Angular default value for `controllerAs` if one is not specified.

Live Demo

Using "require" as an Object

In some instances you may need to access data from a parent component inside your component.

This can be achieved by specifying that our component requires that parent component, the require will give us reference to the required component controller, which can then be used in our controller as shown in the example below:

> Notice that required controllers are guaranteed to be ready only after the $onInit hook.

```
angular.module("myApp", [])
  .component("helloWorld",{
      template: "Hello {{$ctrl.name}}, I'm {{$ctrl.myName}}!",
      bindings: { name: '@' },
      require: {
        parent: '^parentComponent'
      },
      controller: function () {
        // here this.parent might not be initiated yet

        this.$onInit = function() {
           // after $onInit, use this.parent to access required controller
           this.parent.foo();
        }

      }
  });
```

Keep in mind, though, that this creates a tight coupling between the child and the parent.

Section 5.2: Components In angular JS

The components in angularJS can be visualised as a custom directive (< html > this in an HTML directive, and something like this will be a custom directive < ANYTHING >). A component contains a view and a controller. Controller contains the business logic which is binded with an view , which the user sees. The component differs from a angular directive because it contains less configuration. An angular component can be defined like this.

```
angular.module("myApp",[]).component("customer", {})
```

Components are defined on the angular modules. They contains two arguments, One is the name of the component and second one is a object which contains key value pair, which defines which view and which controller it is going to use like this .

```
angular.module("myApp",[]).component("customer", {
    templateUrl : "customer.html", // your view here
    controller: customerController, //your controller here
    controllerAs: "cust"        //alternate name for your controller
})
```

"myApp" is the name of the app we are building and customer is the name of our component. Now for calling it in main html file we will just put it like this

```
<customer></customer>
```

Now this directive will be replaced by the view you have specified and the business logic you have written in your controller.

NOTE : Remember component take a object as second argument while directive take a factory function as argument.

Chapter 6: Custom Directives

Parameter	Details
scope	Property to set the scope of the directive. It can be set as false, true or as an isolate scope: { @, =, <, & }.
scope: falsy	Directive uses parent scope. No scope created for directive.
scope: true	Directive inherits parent scope prototypically as a new child scope. If there are multiple directives on the same element requesting a new scope, then they will share one new scope.
scope: { @ }	One way binding of a directive scope property to a DOM attribute value. As the attribute value bound in the parent, it will change in the directive scope.
scope: { = }	Bi-directional attribute binding that changes the attribute in the parent if the directive attribute changes and vice-versa.
scope: { < }	One way binding of a directive scope property and a DOM attribute expression. The expression is evaluated in the parent. This watches the identity of the parent value so changes to an object property in the parent won't be reflected in the directive. Changes to an object property in a directive will be reflected in the parent, since both reference the same object
scope: { & }	Allows the directive to pass data to an expression to be evaluated in the parent.
compile: function	This function is used to perform DOM transformation on the directive template before the link function runs. It accepts `tElement` (the directive template) and `tAttr` (list of attributes declared on the directive). It does not have access to the scope. It may return a function that will be registered as a `post-link` function or it may return an object with `pre` and `post` properties with will be registered as the `pre-link` and `post-link` functions.
link: function/object	The link property can be configured as a function or object. It can receive the following arguments: scope(directive scope), iElement(DOM element where directive is applied), iAttrs(collection of DOM element attributes), controller(array of controllers required by directive), transcludeFn. It is mainly used to for setting up DOM listeners, watching model properties for changes, and updating the DOM. It executes after the template is cloned. It is configured independently if there is no compile function.
pre-link function	Link function that executes before any child link functions. By default, child directive link functions execute before parent directive link functions and the pre-link function enables the parent to link first. One use case is if the child requires data from the parent.
post-link function	Link function that executives after child elements are linked to parent. It is commonly used for attaching event handlers and accessing child directives, but data required by the child directive should not be set here because the child directive will have already been linked.
restrict: string	Defines how to call the directive from within the DOM. Possible values (Assuming our directive name is `demoDirective`): `E` - Element name (**`<demo-directive></demo-directive>`**), `A` - Attribute (**`<div demo-directive></div>`**), `C` - Matching class (**`<div class="demo-directive"></div>`**), `M` - By comment (`<!-- directive: demo-directive -->`). The `restrict` property can also support multiple options, for example - `restrict: "AC"` will restrict the directive to *Attribute* OR *Class*. If omitted, the **default** value is `"EA"` (Element or Attribute).
require: 'demoDirective'	Locate demoDirective's controller on the current element and inject its controller as the fourth argument to the linking function. Throw an error if not found.
require: '?demoDirective'	Attempt to locate the demoDirective's controller or pass null to the link fn if not found.
require: '^demoDirective'	Locate the demoDirective's controller by searching the element and its parents. Throw an error if not found.
require: '^^demoDirective'	Locate the demoDirective's controller by searching the element's parents. Throw an error if not found.
require: '?^demoDirective'	Attempt to locate the demoDirective's controller by searching the element and its parents or pass null to the link fn if not found.
require: '?^^demoDirective'	Attempt to locate the demoDirective's controller by searching the element's parents, or pass null to the link fn if not found.

Here you will learn about the Directives feature of AngularJS. Below you will find information on what Directives are, as well as Basic and Advanced examples of how to use them.

Section 6.1: Creating and consuming custom directives

Directives are one of the most powerful features of angularjs. Custom angularjs directives are used to extend functionality of html by creating new html elements or custom attributes to provide certain behavior to an html tag.

directive.js

```
// Create the App module if you haven't created it yet
var demoApp= angular.module("demoApp", []);

// If you already have the app module created, comment the above line and create a reference of the
app module
var demoApp = angular.module("demoApp");


// Create a directive using the below syntax
// Directives are used to extend the capabilities of html element
// You can either create it as an Element/Attribute/class
// We are creating a directive named demoDirective. Notice it is in CamelCase when we are defining
the directive just like ngModel
// This directive will be activated as soon as any this element is encountered in html

demoApp.directive('demoDirective', function () {

  // This returns a directive definition object
  // A directive definition object is a simple JavaScript object used for configuring the directive's
behaviour,template..etc
  return {
    // restrict: 'AE', signifies that directive is Element/Attribute directive,
    // "E" is for element, "A" is for attribute, "C" is for class, and "M" is for comment.
    // Attributes are going to be the main ones as far as adding behaviors that get used the most.
    // If you don't specify the restrict property it will default to "A"
    restrict :'AE',

    // The values of scope property decides how the actual scope is created and used inside a
directive. These values can be either "false", "true" or "{}". This creates an isolate scope for the
directive.
    // '@' binding is for passing strings. These strings support {{}} expressions for interpolated
values.
    // '=' binding is for two-way model binding. The model in parent scope is linked to the model in
the directive's isolated scope.
    // '&' binding is for passing a method into your directive's scope so that it can be called
within your directive.
    // The method is pre-bound to the directive's parent scope, and supports arguments.
    scope: {
      name: "@",  // Always use small casing here even if it's a mix of 2-3 words
    },

    // template replaces the complete element with its text.
    template: "<div>Hello {{name}}!</div>",

    // compile is called during application initialization. AngularJS calls it once when html page is
loaded.
    compile: function(element, attributes) {
      element.css("border", "1px solid #cccccc");

      // linkFunction is linked with each element with scope to get the element specific data.
      var linkFunction = function($scope, element, attributes) {
        element.html("Name: <b>"+$scope.name +"</b>");
        element.css("background-color", "#ff00ff");
      };
```

```
      return linkFunction;
    }
  };
});
```

This directive can then be used in App as :

```
<html>

   <head>
      <title>Angular JS Directives</title>
   </head>
   <body>
   <script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
   <script src="directive.js"></script>
    <div ng-app = "demoApp">
       <!-- Notice we are using Spinal Casing here -->
       <demo-directive name="World"></demo-directive>

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

Section 6.2: Directive Definition Object Template

```
demoApp.directive('demoDirective', function () {
  var directiveDefinitionObject = {
    multiElement:
    priority:
    terminal:
    scope: {},
    bindToController: {},
    controller:
    controllerAs:
    require:
    restrict:
    templateNamespace:
    template:
    templateUrl:
    transclude:
    compile:
    link: function(){}
  };
  return directiveDefinitionObject;
});
```

1. **`multiElement`** - set to true and any DOM nodes between the start and end of the directive name will be collected and grouped together as directive elements
2. **`priority`** - allows specification of the order to apply directives when multiple directives are defined on a single DOM element. Directives with higher numbers are compiled first.
3. **`terminal`** - set to true and the current priority will be the last set of directives to execute
4. **`scope`** - sets scope of the directive
5. **`bind to controller`** - binds scope properties directly to directive controller
6. **`controller`** - controller constructor function
7. **`require`** - require another directive and inject its controller as the fourth argument to the linking function
8. **`controllerAs`** - name reference to the controller in the directive scope to allow the controller to be referenced from the directive template.
9. **`restrict`** - restrict directive to Element, Attribute, Class, or Comment

10. **templateNameSpace** - sets document type used by directive template: html, svg, or math. html is the default
11. **template** - html markup that defaults to replacing the content of the directive's element, or wraps the contents of the directive element if transclude is true
12. **templateUrl** - url provided asynchronously for the template
13. **transclude** - Extract the contents of the element where the directive appears and make it available to the directive. The contents are compiled and provided to the directive as a transclusion function.
14. **compile** - function to transform the template DOM
15. **link** - only used if the compile property is not defined. The link function is responsible for registering DOM listeners as well as updating the DOM. It is executed after the template has been cloned.

Section 6.3: How to create resuable component using directive

AngularJS directives are what controls the rendering of the HTML inside an AngularJS application. They can be an Html element, attribute, class or a comment. Directives are used to manipulate the DOM, attaching new behavior to HTML elements, data binding and many more. Some of examples of directives which angular provides are ng-model, ng-hide, ng-if.

Similarly one can create his own custom directive and make them resuable. For creating Custom directives Reference. The sense behind creating reusable directives is to make a set of directives/components written by you just like angularjs provides us using angular.js . These reusable directives can be particularly very helpful when you have suite of applications/application which requires a consistent behavior, look and feel. An example of such reusable component can be a simple toolbar which you may want to use across your application or different applications but you want them to behave the same or look the same.

Firstly , Make a folder named resuableComponents in your app Folder and make reusableModuleApp.js

reusableModuleApp.js:

```
(function(){

 var reusableModuleApp = angular.module('resuableModuleApp', ['ngSanitize']);

 //Remember whatever dependencies you have in here should be injected in the app module where it is
intended to be used or it's scripts should be included in your main app
                                //We will be injecting ng-sanitize

 resubaleModuleApp.directive('toolbar', toolbar)

 toolbar.$inject=['$sce'];

 function toolbar($sce){

     return{
     restrict :'AE',
        //Defining below isolate scope actually provides window for the directive to take data from
app that will be using this.
         scope : {
                 value1: '=',
                 value2: '=',
                 },

          }
     template : '<ul>  <li><a ng-click="Add()" href="">{{value1}}</a></li>  <li><a ng-
click="Edit()" href="#">{{value2}}</a></li> </ul> ',
          link : function(scope, element, attrs){
```

```
            //Handle's Add function
            scope.Add = function(){


            };

            //Handle's Edit function
            scope.Edit = function(){

            };
    }
  }

});
```

mainApp.js:

```
(function(){
   var mainApp = angular.module('mainApp', ['reusableModuleApp']); //Inject resuableModuleApp in
your application where you want to use toolbar component

   mainApp.controller('mainAppController', function($scope){
      $scope.value1 = "Add";
      $scope.value2 = "Edit";

   });

 });
```

index.html:

```
<!doctype html>
<html ng-app="mainApp">
<head>
 <title> Demo Making a reusable component
<head>
 <body ng-controller="mainAppController">

    <!-- We are providing data to toolbar directive using mainApp'controller -->
    <toolbar value1="value1" value2="value2"></toolbar>

   <!-- We need to add the dependent js files on both apps here -->
    <script src="js/angular.js"></script>
    <script src="js/angular-sanitize.js"></script>

    <!-- your mainApp.js should be added afterwards --->
     <script src="mainApp.js"></script>

    <!-- Add your reusable component js files here -->
     <script src="resuableComponents/reusableModuleApp.js"></script>

  </body>
</html>
```

Directive are reusable components by default. When you make directives in separate angular module, It actually makes it exportable and reusable across different angularJs applications. New directives can simply be added inside reusableModuleApp.js and reusableModuleApp can have it's own controller, services, DDO object inside directive to define the behavior.

Section 6.4: Basic Directive example

superman-directive.js

```
angular.module('myApp', [])
  .directive('superman', function() {
    return {
      // restricts how the directive can be used
      restrict: 'E',
      templateUrl: 'superman-template.html',
      controller: function() {
        this.message = "I'm superman!"
      },
      controllerAs: 'supermanCtrl',
      // Executed after Angular's initialization. Use commonly
      // for adding event handlers and DOM manipulation
      link: function(scope, element, attributes) {
        element.on('click', function() {
          alert('I am superman!')
        });
      }
    }
  });
```

superman-template.html

```
<h2>{{supermanCtrl.message}}</h2>
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.0/angular.js"></script>
  <script src="superman-directive.js"><script/>
</head>
<body>
<div ng-app="myApp">
  <superman></superman>
</div>
</body>
</html>
```

You can check out more about directive's `restrict` and `link` functions on AngularJS's official documentation on Directives

Section 6.5: Directive decorator

Sometimes you may need additional features from a directive. Instead of rewriting (copy) the directive, you can modify how the directive behaves.

The decorator will be executed during $inject phase.

To do so, provde a .config to your module. The directive is called myDirective, so you have to config myDirectiveDirective. (this in an angular convention [read about providers]).

This example will change the templateUrl of the directive:

```
angular.module('myApp').config(function($provide){
        $provide.decorator('myDirectiveDirective', function($delegate){
             var directive = $delegate[0]; // this is the actual delegated, your directive
             directive.templateUrl = 'newTemplate.html'; // you change the directive template
        return $delegate;
    })
});
```

This example add an onClick event to the directive element when clicked, this happens during compile phase.

```
angular.module('myApp').config(function ($provide) {
        $provide.decorator('myDirectiveTwoDirective', function ($delegate) {
            var directive = $delegate[0];
            var link = directive.link; // this is directive link phase
            directive.compile = function () { // change the compile of that directive
                return function (scope, element, attrs) {
                    link.apply(this, arguments); // apply this at the link phase
                    element.on('click', function(){ // when add an onclick that log hello when the
directive is clicked.
                            console.log('hello!');
                    });
                };
            };
            return $delegate;
        });

    });
```

Similar approach can be used for both Providers and Services.

Section 6.6: Basic directive with template and an isolated scope

Creating a custom directive with *isolated scope* will separate the scope **inside** the directive from the **outside** scope, in order to prevent our directive from accidentally change the data in the parent scope and restricting it from reading private data from the parent scope.

To create an isolated scope and still allow our custom directive to communicate with the outside scope, we can use the `scope` option that describe how to **map** the bindings of the directive's inner scope with the outside scope.

The actual bindings are made with extra **attributes** attached to the directive. The binding settings are defined with the `scope` option and an object with key-value pairs:

- A **key**, which is corresponded to our directive's isolated scope property
- A **value**, which tells Angular how do bind the directive inner scope to a matching **attribute**

Simple example of a directive with an isolated scope:

```
var ProgressBar = function() {
  return {
    scope: { // This is how we define an isolated scope
      current: '=', // Create a REQUIRED bidirectional binding by using the 'current' attribute
      full: '=?maxValue' // Create an OPTIONAL (Note the '?'): bidirectional binding using 'max-
value' attribute to the `full` property in our directive isolated scope
    }
```

```
    template: '<div class="progress-back">' +
              '  <div class="progress-bar"' +
              '       ng-style="{width: getProgress()}">' +
              '  </div>' +
              '</div>',
    link: function(scope, el, attrs) {
      if (scope.full === undefined) {
        scope.full = 100;
      }
      scope.getProgress = function() {
        return (scope.current / scope.size * 100) + '%';
      }
    }
  }
}

ProgressBar.$inject = [];
angular.module('app').directive('progressBar', ProgressBar);
```

Example how to use this directive and bind data from the controller's scope to the directive's inner scope:

Controller:

```
angular.module('app').controller('myCtrl', function($scope) {
    $scope.currentProgressValue = 39;
    $scope.maxProgressBarValue = 50;
});
```

View:

```
<div ng-controller="myCtrl">
    <progress-bar current="currentProgressValue"></progress-bar>
    <progress-bar current="currentProgressValue" max-value="maxProgressBarValue"></progress-bar>
</div>
```

Section 6.7: Building a reusable component

Directives can be used to build reusable components. Here is an example of a "user box" component:

userBox.js

```
angular.module('simpleDirective', []).directive('userBox', function() {
  return {
    scope: {
        username: '=username',
        reputation: '=reputation'
    },
    templateUrl: '/path/to/app/directives/user-box.html'
  };
});
```

Controller.js

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

myApp.controller('Controller', function($scope) {

    $scope.user = "John Doe";
    $scope.rep = 1250;
```

```
    $scope.user2 = "Andrew";
    $scope.rep2 = 2850;

});
```

myPage.js

```
<html lang="en" ng-app="myApp">
  <head>
    <script src="/path/to/app/angular.min.js"></script>
    <script src="/path/to/app/js/controllers/Controller.js"></script>
    <script src="/path/to/app/js/directives/userBox.js"></script>
  </head>

  <body>

    <div ng-controller="Controller">
        <user-box username="user" reputation="rep"></user-box>
        <user-box username="user2" reputation="rep2"></user-box>
    </div>

  </body>
</html>
```

user-box.html

```
<div>{{username}}</div>
<div>{{reputation}} reputation</div>
```

The result will be:

```
John Doe
1250 reputation
Andrew
2850 reputation
```

Section 6.8: Directive inheritance and interoperability

Angular js directives can be nested or be made interoperable.

In this example, directive Adir exposes to directive Bdir it's controller $scope, since Bdir requires Adir.

```
angular.module('myApp',[]).directive('Adir', function () {
        return {
            restrict: 'AE',
            controller: ['$scope', function ($scope) {
                    $scope.logFn = function (val) {
                            console.log(val);
                        }
                }]
            }
    })
```

Make sure to set require: '^Adir' (look at the angular documentation, some versions doesn't require ^ character).

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

```
            require: '^Adir', // Bdir require Adir
            link: function (scope, elem, attr, Parent) {
            // Parent is Adir but can be an array of required directives.
                elem.on('click', function ($event) {
                    Parent.logFn("Hello!"); // will log "Hello! at parent dir scope
                    scope.$apply(); // apply to parent scope.
                });
            }
        }
    }]);
```

You can nest your directive in this way:

```
<div a-dir><span b-dir></span></div>
<a-dir><b-dir></b-dir> </a-dir>
```

Is not required that directives are nested in your HTML.

Chapter 7: Filters

Section 7.1: Accessing a filtered list from outside an ng-repeat

Occasionally you will want to access the result of your filters from outside the `ng-repeat`, perhaps to indicate the number of items that have been filtered out. You can do this using `as [variablename]` syntax on the `ng-repeat`.

```
<ul>
  <li ng-repeat="item in vm.listItems | filter:vm.myFilter as filtered">
    {{item.name}}
  </li>
</ul>
<span>Showing {{filtered.length}} of {{vm.listItems.length}}</span>
```

Section 7.2: Custom filter to remove values

A typical use case for a filter is to remove values from an array. In this example we pass in an array and remove any nulls found in it, returning the array.

```
function removeNulls() {
    return function(list) {
        for (var i = list.length - 1; i >= 0; i--) {
            if (typeof list[i] === 'undefined' ||
                    list[i] === null) {
                list.splice(i, 1);
            }
        }
        return list;
    };
}
```

That would be used in the HTML like

```
{{listOfItems | removeNulls}}
```

or in a controller like

```
listOfItems = removeNullsFilter(listOfItems);
```

Section 7.3: Custom filter to format values

Another use case for filters is to format a single value. In this example, we pass in a value and we are returned an appropriate true boolean value.

```
function convertToBooleanValue() {
    return function(input) {
        if (typeof input !== 'undefined' &&
                input !== null &&
                (input === true || input === 1 || input === '1' || input
                        .toString().toLowerCase() === 'true')) {
            return true;
        }
        return false;
    };
}
```

Which in the HTML would be used like this:

```
{{isAvailable | convertToBooleanValue}}
```

Or in a controller like:

```
var available = convertToBooleanValueFilter(isAvailable);
```

Section 7.4: Using filters in a controller or service

By injecting `$filter`, any defined filter in your Angular module may be used in controllers, services, directives or even other filters.

```
angular.module("app")
  .service("users", usersService)
  .controller("UsersController", UsersController);

function usersService () {
  this.getAll = function () {
    return [{
      id: 1,
      username: "john"
    }, {
      id: 2,
      username: "will"
    }, {
      id: 3,
      username: "jack"
    }];
  };
}

function UsersController ($filter, users) {
  var orderByFilter = $filter("orderBy");

  this.users = orderByFilter(users.getAll(), "username");
  // Now the users are ordered by their usernames: jack, john, will

  this.users = orderByFilter(users.getAll(), "username", true);
  // Now the users are ordered by their usernames, in reverse order: will, john, jack
}
```

Section 7.5: Performing filter in a child array

This example was done in order to demonstrate how you can perform a deep filter in a *child* array without the necessity of a custom filter.

Controller:

```
(function() {
  "use strict";
  angular
    .module('app', [])
    .controller('mainCtrl', mainCtrl);

  function mainCtrl() {
    var vm = this;
```

```
    vm.classifications = ["Saloons", "Sedans", "Commercial vehicle", "Sport car"];
    vm.cars = [
       {
          "name":"car1",
          "classifications":[
             {
                "name":"Saloons"
             },
             {
                "name":"Sedans"
             }
          ]
       },
       {
          "name":"car2",
          "classifications":[
             {
                "name":"Saloons"
             },
             {
                "name":"Commercial vehicle"
             }
          ]
       },
       {
          "name":"car3",
          "classifications":[
             {
                "name":"Sport car"
             },
             {
                "name":"Sedans"
             }
          ]
       }
    ];
  }
})();
```

View:

```
<body ng-app="app" ng-controller="mainCtrl as main">
  Filter car by classification:
  <select ng-model="classificationName"
        ng-options="classification for classification in main.classifications"></select>
  <br>
  <ul>
    <li ng-repeat="car in main.cars |
                 filter: { classifications: { name: classificationName } } track by $index"
                ng-bind-template="{{car.name}} - {{car.classifications | json}}">
    </li>
  </ul>
</body>
```

Check the complete **DEMO**.

Chapter 8: Services

Section 8.1: Creating a service using angular.factory

First define the service (in this case it uses the factory pattern):

```
.factory('dataService', function() {
    var dataObject = {};
    var service = {
        // define the getter method
        get data() {
            return dataObject;
        },
        // define the setter method
        set data(value) {
            dataObject = value || {};
        }
    };
    // return the "service" object to expose the getter/setter
    return service;
})
```

Now you can use the service to share data between controllers:

```
.controller('controllerOne', function(dataService) {
    // create a local reference to the dataService
    this.dataService = dataService;
    // create an object to store
    var someObject = {
        name: 'SomeObject',
        value: 1
    };
    // store the object
    this.dataService.data = someObject;
})

.controller('controllerTwo', function(dataService) {
    // create a local reference to the dataService
    this.dataService = dataService;
    // this will automatically update with any changes to the shared data object
    this.objectFromControllerOne = this.dataService.data;
})
```

Section 8.2: Difference between Service and Factory

1) Services

A service is a `constructor` function that is invoked once at runtime with **`new`**, just like what we would do with plain javascript with only difference that `AngularJs` is calling the **`new`** behind the scenes.

There is one thumb rule to remember in case of services

1. Services are constructors which are called with **`new`**

Lets see a simple example where we would register a service which uses `$http` service to fetch student details, and use it in the controller

```
function StudentDetailsService($http) {
  this.getStudentDetails = function getStudentDetails() {
    return $http.get('/details');
  };
}

angular.module('myapp').service('StudentDetailsService', StudentDetailsService);
```

We just inject this service into the controller

```
function StudentController(StudentDetailsService) {
  StudentDetailsService.getStudentDetails().then(function (response) {
      // handle response
    });
}
angular.module('app').controller('StudentController', StudentController);
```

When to use?

Use `.service()` wherever you want to use a constructor. It is usually used to create public API's just like `getStudentDetails()`. But if you don't want to use a constructor and wish to use a simple API pattern instead, then there isn't much flexibility in `.service()`.

2) Factory

Even though we can achieve all the things using `.factory()` which we would, using `.services()`, it doesn't make `.factory()` "same as" `.service()`. It is much more powerful and flexible than `.service()`

A `.factory()` is a design pattern which is used to return a value.

There are two thumb rules to remember in case of factories

1. Factories return values
2. Factories (can) create objects (Any object)

Lets see some examples on what we can do using `.factory()`

Returning Objects Literals

Lets see an example where factory is used to return an object using a basic Revealing module pattern

```
function StudentDetailsService($http) {
  function getStudentDetails() {
    return $http.get('/details');
  }
  return {
    getStudentDetails: getStudentDetails
  };
}

angular.module('myapp').factory('StudentDetailsService', StudentDetailsService);
```

Usage inside a controller

```
function StudentController(StudentDetailsService) {
  StudentDetailsService.getStudentDetails().then(function (response) {
      // handle response
    });
```

```
}
angular.module('app').controller('StudentController', StudentController);
```

Returning Closures

What is a closure?

Closures are functions that refer to variables that are used locally, BUT defined in an enclosing scope.

Following is an example of a closure

```
function closureFunction(name) {
  function innerClosureFunction(age) { // innerClosureFunction() is the inner function, a closure
    // Here you can manipulate 'age' AND 'name' variables both
  };
};
```

The *"wonderful"* part is that it can access the `name` which is in the parent scope.

Lets use the above closure example inside `.factory()`

```
function StudentDetailsService($http) {
  function closureFunction(name) {
  function innerClosureFunction(age) {
    // Here you can manipulate 'age' AND 'name' variables
    };
  };
};

angular.module('myapp').factory('StudentDetailsService', StudentDetailsService);
```

Usage inside a controller

```
function StudentController(StudentDetailsService) {
  var myClosure = StudentDetailsService('Student Name'); // This now HAS the innerClosureFunction()
  var callMyClosure = myClosure(24); // This calls the innerClosureFunction()
};

angular.module('app').controller('StudentController', StudentController);
```

Creating Constructors/instances

`.service()` creates constructors with a call to **`new`** as seen above. `.factory()` can also create constructors with a call to **`new`**

Lets see an example on how to achieve this

```
function StudentDetailsService($http) {
  function Student() {
    this.age = function () {
        return 'This is my age';
    };
  }
  Student.prototype.address = function () {
        return 'This is my address';
  };
  return Student;
};
```

```
angular.module('myapp').factory('StudentDetailsService', StudentDetailsService);
```

Usage inside a controller

```
function StudentController(StudentDetailsService) {
  var newStudent = new StudentDetailsService();

  //Now the instance has been created. Its properties can be accessed.

 newStudent.age();
 newStudent.address();

};

angular.module('app').controller('StudentController', StudentController);
```

Section 8.3: $sce - sanitize and render content and resources in templates

> $sce ("Strict Contextual Escaping") is a built-in angular service that automatically sanitize content and internal sources in templates.

injecting **external** sources and **raw HTML** into the template requires manual wrapping of`$sce`.

In this example we'll create a simple $sce sanitation filter :`.

Demo

```
.filter('sanitizer', ['$sce', [function($sce) {
     return function(content) {
          return $sce.trustAsResourceUrl(content);
      };
}]);
```

Usage in template

```
<div ng-repeat="item in items">

    // Sanitize external sources
    <ifrmae ng-src="{{item.youtube_url | sanitizer}}">

    // Sanitaize and render HTML
    <div ng-bind-html="{{item.raw_html_content| sanitizer}}"></div>

</div>
```

Section 8.4: How to create a Service

```
angular.module("app")
  .service("counterService", function(){

    var service = {
        number: 0
    };

    return service;
```

```
});
```

Section 8.5: How to use a service

```
    angular.module("app")

        // Custom services are injected just like Angular's built-in services
        .controller("step1Controller", ['counterService', '$scope', function(counterService,
$scope) {
            counterService.number++;
            // bind to object (by reference), not to value, for automatic sync
            $scope.counter = counterService;
        })
```

In the template using this controller you'd then write:

```
// editable
<input ng-model="counter.number" />
```

or

```
// read-only
<span ng-bind="counter.number"></span>
```

Of course, in real code you would interact with the service using methods on the controller, which in turn delegate to the service. The example above simply increments the counter value each time the controller is used in a template.

Services in Angularjs are singletons:

Services are singleton objects that are instantiated only once per app (by the $injector) and lazy loaded (created only when necessary).

> A singleton is a class which only allows one instance of itself to be created - and gives simple, easy access to said instance. As stated here

Section 8.6: How to create a Service with dependencies using 'array syntax'

```
angular.module("app")
  .service("counterService", ["fooService", "barService", function(anotherService, barService){

    var service = {
        number: 0,
        foo: function () {
            return fooService.bazMethod(); // Use of 'fooService'
        },
        bar: function () {
            return barService.bazMethod(); // Use of 'barService'
        }
    };

    return service;
  }]);
```

Section 8.7: Registering a Service

The most common and flexible way to create a service uses the angular.module API factory:

```
angular.module('myApp.services', []).factory('githubService', function() {
   var serviceInstance = {};
   // Our first service
   return serviceInstance;
});
```

The service factory function can be either a function or an array, just like the way we create controllers:

```
// Creating the factory through using the
// bracket notation
angular.module('myApp.services', [])
.factory('githubService', [function($http) {
}]);
```

To expose a method on our service, we can place it as an attribute on the service object.

```
angular.module('myApp.services', [])
    .factory('githubService', function($http) {
        var githubUrl = 'https://api.github.com';
        var runUserRequest = function(username, path) {
        // Return the promise from the $http service
        // that calls the Github API using JSONP
        return $http({
            method: 'JSONP',
            url: githubUrl + '/users/' +
            username + '/' +
            path + '?callback=JSON_CALLBACK'
        });
    }
// Return the service object with a single function
// events
return {
    events: function(username) {
    return runUserRequest(username, 'events');
    }
};
```

Chapter 9: Dependency Injection

Section 9.1: Dynamic Injections

There is also an option to dynamically request components. You can do it using the `$injector` service:

```
myModule.controller('myController', ['$injector', function($injector) {
    var myService = $injector.get('myService');
}]);
```

Note: while this method could be used to prevent the circular dependency issue that might break your app, it is not considered best practice to bypass the problem by using it. Circular dependency usually indicates there is a flaw in your application's architecture, and you should address that instead.

Section 9.2: Dynamically load AngularJS service in vanilla JavaScript

You can load AngularJS services in vanilla JavaScript using AngularJS `injector()` method. Every jqLite element retrieved calling `angular.element()` has a method `injector()` that can be used to retrieve the injector.

```
var service;
var serviceName = 'myService';

var ngAppElement = angular.element(document.querySelector('[ng-app],[data-ng-app]') || document);
var injector = ngAppElement.injector();

if(injector && injector.has(serviceNameToInject)) {
    service = injector.get(serviceNameToInject);
}
```

In the above example we try to retrieve the jqLite element containing the root of the AngularJS application (`ngAppElement`). To do that, we use `angular.element()` method, searching for a DOM element containing `ng-app` or `data-ng-app` attribute or, if it does not exists, we fall back to `document` element. We use `ngAppElement` to retrieve injector instance (with `ngAppElement.injector()`). The injector instance is used to check if the service to inject exists (with `injector.has()`) and then to load the service (with `injector.get()`) inside `service` variable.

Chapter 10: Unit tests

Section 10.1: Unit test a component (1.5+)

Component code:

```
angular.module('myModule', []).component('myComponent', {
  bindings: {
    myValue: '<'
  },
  controller: function(MyService) {
    this.service = MyService;
    this.componentMethod = function() {
      return 2;
    };
  }
});
```

The test:

```
describe('myComponent', function() {
  var component;

  var MyServiceFake = jasmine.createSpyObj(['serviceMethod']);

  beforeEach(function() {
    module('myModule');
    inject(function($componentController) {
      // 1st - component name, 2nd - controller injections, 3rd - bindings
      component = $componentController('myComponent', {
        MyService: MyServiceFake
      }, {
        myValue: 3
      });
    });
  });

  /** Here you test the injector. Useless. */

  it('injects the binding', function() {
    expect(component.myValue).toBe(3);
  });

  it('has some cool behavior', function() {
    expect(component.componentMethod()).toBe(2);
  });
});
```

Run!

Section 10.2: Unit test a filter

Filter code:

```
angular.module('myModule', []).filter('multiplier', function() {
  return function(number, multiplier) {
    if (!angular.isNumber(number)) {
      throw new Error(number + " is not a number!");
```

```
    }
    if (!multiplier) {
      multiplier = 2;
    }
    return number * multiplier;
  }
});
```

The test:

```
describe('multiplierFilter', function() {
  var filter;

  beforeEach(function() {
    module('myModule');
    inject(function(multiplierFilter) {
      filter = multiplierFilter;
    });
  });

  it('multiply by 2 by default', function() {
    expect(filter(2)).toBe(4);
    expect(filter(3)).toBe(6);
  });

  it('allow to specify custom multiplier', function() {
    expect(filter(2, 4)).toBe(8);
  });

  it('throws error on invalid input', function() {
    expect(function() {
      filter(null);
    }).toThrow();
  });
});
```

Run!

Remark: In the `inject` call in the test, your filter needs to be specified by its name + *Filter*. The cause for this is that whenever you register a filter for your module, Angular register it with a `Filter` appended to its name.

Section 10.3: Unit test a service

Service Code

```
angular.module('myModule', [])
  .service('myService', function() {
    this.doSomething = function(someNumber) {
      return someNumber + 2;
    }
  });
```

The test

```
describe('myService', function() {
  var myService;
  beforeEach(function() {
    module('myModule');
    inject(function(_myService_) {
```

```
      myService = _myService_;
    });
  });
  it('should increment `num` by 2', function() {
    var result = myService.doSomething(4);
    expect(result).toEqual(6);
  });
});
```

Run!

Section 10.4: Unit test a controller

Controller code:

```
angular.module('myModule', [])
  .controller('myController', function($scope) {
    $scope.num = 2;
    $scope.doSomething = function() {
      $scope.num += 2;
    }
  });
```

The test:

```
describe('myController', function() {
  var $scope;
  beforeEach(function() {
    module('myModule');
    inject(function($controller, $rootScope) {
      $scope = $rootScope.$new();
      $controller('myController', {
        '$scope': $scope
      })
    });
  });
  it('should increment `num` by 2', function() {
    expect($scope.num).toEqual(2);
    $scope.doSomething();
    expect($scope.num).toEqual(4);
  });
});
```

Run!

Section 10.5: Unit test a directive

Directive code

```
angular.module('myModule', [])
  .directive('myDirective', function() {
    return {
      template: '<div>{{greeting}} {{name}}!</div>',
      scope: {
        name: '=',
        greeting: '@'
      }
    };
```

```
  });
```

The test

```
describe('myDirective', function() {
  var element, scope;
  beforeEach(function() {
    module('myModule');
    inject(function($compile, $rootScope) {
      scope = $rootScope.$new();
      element = angular.element("<my-directive name='name' greeting='Hello'></my-directive>");
      $compile(element)(scope);
      /* PLEASE NEVER USE scope.$digest(). scope.$apply use a protection to avoid to run a digest
loop when there is already one, so, use scope.$apply() instead. */
      scope.$apply();
    })
  });

  it('has the text attribute injected', function() {
    expect(element.html()).toContain('Hello');
  });

  it('should have proper message after scope change', function() {
    scope.name = 'John';
    scope.$apply();
    expect(element.html()).toContain("John");
    scope.name = 'Alice';
    expect(element.html()).toContain("John");
    scope.$apply();
    expect(element.html()).toContain("Alice");
  });
});
```

Run!

Chapter 11: Profiling and Performance

Section 11.1: 7 Simple Performance Improvements

1) Use ng-repeat sparingly

Using `ng-repeat` in views generally results in poor performance, particularly when there are nested `ng-repeat`'s.

This is super slow!

```
<div ng-repeat="user in userCollection">
  <div ng-repeat="details in user">
    {{details}}
  </div>
</div>
```

Try to avoid nested repeats as much as possible. One way to improve the performance of `ng-repeat` is to use `track by $index` (or some other id field). By default, `ng-repeat` tracks the whole object. With `track by`, Angular watches the object only by the `$index` or object id.

```
<div ng-repeat="user in userCollection track by $index">
  {{user.data}}
</div>
```

Use other approaches like pagination, virtual scrolls, infinite scrolls or limitTo: begin whenever possible to avoid iterating over large collections.

2) Bind once

Angular has bidirectional data binding. It comes with a cost of being slow if used too much.

Slower Performance

```
<!-- Default data binding has a performance cost -->
<div>{{ my.data }}</div>
```

Faster Performance (AngularJS >= 1.3)

```
<!-- Bind once is much faster -->
<div>{{ ::my.data }}</div>

<div ng-bind="::my.data"></div>

<!-- Use single binding notation in ng-repeat where only list display is needed  -->
<div ng-repeat="user in ::userCollection">
  {{::user.data}}
</div>
```

Using the "bind once" notation tells Angular to wait for the value to stabilize after the first series of digest cycles. Angular will use that value in the DOM, then remove all watchers so that it becomes a static value and is no longer bound to the model.

The `{{}}` is much slower.

This `ng-bind` is a directive and will place a watcher on the passed variable. So the `ng-bind` will only apply, when the

passed value does actually change.

The brackets on the other hand will be dirty checked and refreshed in every `$digest`, even if it's not necessary.

3) Scope functions and filters take time

AngularJS has a digest loop. All your functions are in a view and filters are executed every time the digest cycle runs. The digest loop will be executed whenever the model is updated and it can slow down your app (filter can be hit multiple times before the page is loaded).

Avoid this:

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.calculateMe()}}</p>
  <p>{{calc.data | heavyFilter}}</p>
</div>
```

Better approach

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.preCalculatedValue}}</p>
  <p>{{calc.data | lightFilter}}</p>
</div>
```

Where the controller can be:

```
app.controller('bigCalulations', function(valueService) {
    // bad, because this is called in every digest loop
    this.calculateMe = function() {
        var t = 0;
        for(i = 0; i < 1000; i++) {
            t += i;
        }
        return t;
    }
    // good, because this is executed just once and logic is separated in service to    keep the
controller light
    this.preCalulatedValue = valueService.valueCalculation(); // returns 499500
});
```

4) Watchers

Watchers tremendously drop performance. With more watchers, the digest loop will take longer and the UI will slow down. If the watcher detects change, it will kick off the digest loop and re-render the view.

There are three ways to do manual watching for variable changes in Angular.

`$watch()` - watches for value changes

`$watchCollection()` - watches for changes in collection (watches more than regular `$watch`)

`$watch(..., true)` - **Avoid this** as much as possible, it will perform "deep watch" and will decline the performance (watches more than `watchCollection`)

Note that if you are binding variables in the view you are creating new watches - use `{{::variable}}` to prevent creating a watch, especially in loops.

As a result you need to track how many watchers you are using. You can count the watchers with this script (credit

to @Words Like Jared Number of watchers)

```
(function() {
    var root = angular.element(document.getElementsByTagName('body')),
        watchers = [],
        f = function(element) {
        angular.forEach(['$scope', '$isolateScope'], function(scopeProperty) {
            if(element.data() && element.data().hasOwnProperty(scopeProperty)) {
                angular.forEach(element.data()[scopeProperty].$$watchers, function(watcher) {
                watchers.push(watcher);
                });
            }
        });

        angular.forEach(element.children(), function(childElement) {
            f(angular.element(childElement));
        });
    };

    f(root);

    // Remove duplicate watchers
    var watchersWithoutDuplicates = [];
    angular.forEach(watchers, function(item) {
        if(watchersWithoutDuplicates.indexOf(item) < 0) {
            watchersWithoutDuplicates.push(item);
        }
    });
    console.log(watchersWithoutDuplicates.length);
})();
```

5) ng-if / ng-show

These functions are very similar in behavior. `ng-if` removes elements from the DOM while `ng-show` only hides the elements but keeps all handlers. If you have parts of the code you do not want to show, use `ng-if`.

It depends on the type of usage, but often one is more suitable than the other.

- If the element is not needed, use `ng-if`
- To quickly toggle on/off, use `ng-show`/`ng-hide`

```
<div ng-repeat="user in userCollection">
  <p ng-if="user.hasTreeLegs">I am special<!-- some complicated DOM --></p>
  <p ng-show="user.hasSubscribed">I am awesome<!-- switch this setting on and off --></p>
</div>
```

If in doubt - use `ng-if` and test!

6) Disable debugging

By default, bind directives and scopes leave extra classes and markup in the code to assist with various debugging tools. Disabling this option means that you no longer render these various elements during the digest cycle.

```
angular.module('exampleApp', []).config(['$compileProvider', function ($compileProvider) {
    $compileProvider.debugInfoEnabled(false);
}]);
```

7) Use dependency injection to expose your resources

Dependency Injection is a software design pattern in which an object is given its dependencies, rather than the object creating them itself. It is about removing the hard-coded dependencies and making it possible to change them whenever needed.

You might wonder about the performance cost associated with such string parsing of all injectable functions. Angular takes care of this by caching the $inject property after the first time. So this doesn't happen every time a function needs to be invoked.

PRO TIP: If you are looking for the approach with the best performance, go with the $inject property annotation approach. This approach entirely avoids the function definition parsing because this logic is wrapped within the following check in the annotate function: if (!($inject = fn.$inject)). If $inject is already available, no parsing required!

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

var DemoController = function (s, h) {
    h.get('https://api.github.com/users/angular/repos').success(function (repos) {
        s.repos = repos;
    });
}
// $inject property annotation
DemoController['$inject'] = ['$scope', '$http'];

app.controller('DemoController', DemoController);
```

PRO TIP 2: You can add an `ng-strict-di` directive on the same element as `ng-app` to opt into strict DI mode which will throw an error whenever a service tries to use implicit annotations. Example:

```
<html ng-app="DemoApp" ng-strict-di>
```

Or if you use manual bootstrapping:

```
angular.bootstrap(document, ['DemoApp'], {
    strictDi: true
});
```

Section 11.2: Bind Once

Angular has reputation for having awesome bidirectional data binding. By default, Angular continuously synchronizes values bound between model and view components any time data changes in either the model or view component.

This comes with a cost of being a bit slow if used too much. This will have a larger performance hit:

Bad performance: `{{my.data}}`

Add two colons `::` before the variable name to use one-time binding. In this case, the value only gets updated once my.data is defined. You are explicitly pointing not to watch for data changes. Angular won't perform any value checks, resulting with fewer expressions being evaluated on each digest cycle.

Good performance examples using one-time binding

```
{{::my.data}}
<span ng-bind="::my.data"></span>
<span ng-if="::my.data"></span>
<span ng-repeat="item in ::my.data">{{item}}</span>
```

```
<span ng-class="::{ 'my-class': my.data }"></div>
```

Note: This however removes the bi-directional data binding for `my.data`, so whenever this field changes in your application, the same won't be reflected in the view automatically. So **use it only for values that won't change throughout the lifespan of your application**.

Section 11.3: ng-if vs ng-show

These functions are very similar in behaviour. The difference is that `ng-if` removes elements from the DOM. If there are large parts of the code that will not be shown, then `ng-if` is the way to go. `ng-show` will only hide the elements but will keep all the handlers.

ng-if

The ngIf directive removes or recreates a portion of the DOM tree based on an expression. If the expression assigned to ngIf evaluates to a false value then the element is removed from the DOM, otherwise a clone of the element is reinserted into the DOM.

ng-show

The ngShow directive shows or hides the given HTML element based on the expression provided to the ngShow attribute. The element is shown or hidden by removing or adding the ng-hide CSS class onto the element.

Example

```
<div ng-repeat="user in userCollection">
  <p ng-if="user.hasTreeLegs">I am special
    <!-- some complicated DOM -->
  </p>
  <p ng-show="user.hasSubscribed">I am aweosme
    <!-- switch this setting on and off -->
  </p>
</div>
```

Conclusion

It depends from the type of usage, but often one is more suitable than the other (e.g., if 95% of the time the element is not needed, use `ng-if`; if you need to toggle the DOM element's visibility, use `ng-show`).

When in doubt, use `ng-if` and test!

Note: `ng-if` creates a new isolated scope, whereas `ng-show` and `ng-hide` don't. Use `$parent.property` if parent scope property is not directly accessible in it.

Section 11.4: Watchers

Watchers needed for watch some value and detect that this value is changed.

After call `$watch()` or `$watchCollection` new watcher add to internal watcher collection in current scope.

So, what is watcher?

Watcher is a simple function, which is called on every digest cycle, and returns some value. Angular checks the returned value, if it is not the same as it was on the previous call - a callback that was passed in second parameter to function `$watch()` or `$watchCollection` will be executed.

```
(function() {
```

```
  angular.module("app", []).controller("ctrl", function($scope) {
    $scope.value = 10;
    $scope.$watch(
      function() { return $scope.value; },
      function() { console.log("value changed"); }
    );
  }
})();
```

Watchers are performance killers. The more watchers you have, the longer they take to make a digest loop, the slower UI. If a watcher detects changes, it will kick off the digest loop (recalculation on all screen)

There are three ways to do manual watch for variable changes in Angular.

`$watch()` - just watches for value changes

`$watchCollection()` - watches for changes in collection (watches more than regular $watch)

`$watch(..., true)` - **Avoid this** as much as possible, it will perform "deep watch" and will kill the performance (watches more than watchCollection)

Note that if you are binding variables in the view, you are creating new watchers - use `{{::variable}}` not to create watcher, especially in loops

As a result you need to track how many watchers are you using. You can count the watchers with this script (credit to @Words Like Jared - How to count total number of watches on a page?

```
(function() {
  var root = angular.element(document.getElementsByTagName("body")),
      watchers = [];

  var f = function(element) {

    angular.forEach(["$scope", "$isolateScope"], function(scopeProperty) {
      if(element.data() && element.data().hasOwnProperty(scopeProperty)) {
        angular.forEach(element.data()[scopeProperty].$$watchers, function(watcher) {
          watchers.push(watcher);
        });
      }
    });

    angular.forEach(element.children(), function(childElement) {
      f(angular.element(childElement));
    });

  };

  f(root);

  // Remove duplicate watchers
  var watchersWithoutDuplicates = [];
  angular.forEach(watchers, function(item) {
    if(watchersWithoutDuplicates.indexOf(item) < 0) {
      watchersWithoutDuplicates.push(item);
    }
  });

  console.log(watchersWithoutDuplicates.length);
```

```
})();
```

If you don't want to create your own script, there is an open source utility called ng-stats that uses a real-time chart embedded into the page to give you insight into the number of watches Angular is managing, as well as the frequency and duration of digest cycles over time. The utility exposes a global function named `showAngularStats` that you can call to configure how you want the chart to work.

```
showAngularStats({
  "position": "topleft",
  "digestTimeThreshold": 16,
  "autoload": true,
  "logDigest": true,
  "logWatches": true
});
```

The example code above displays the following chart on the page automatically (interactive demo).

Section 11.5: Always deregister listeners registered on other scopes other than the current scope

You must always unregister scopes other then your current scope as shown below:

```
//always deregister these
$rootScope.$on(...);
$scope.$parent.$on(...);
```

You don't have to deregister listners on current scope as angular would take care of it:

```
//no need to deregister this
$scope.$on(...);
```

`$rootScope.$on` listeners will remain in memory if you navigate to another controller. This will create a memory leak if the controller falls out of scope.

Don't

```
angular.module('app').controller('badExampleController', badExample);
badExample.$inject = ['$scope', '$rootScope'];

function badExample($scope, $rootScope) {
    $rootScope.$on('post:created', function postCreated(event, data) {});
}
```

Do

```
angular.module('app').controller('goodExampleController', goodExample);
goodExample.$inject = ['$scope', '$rootScope'];

function goodExample($scope, $rootScope) {
    var deregister = $rootScope.$on('post:created', function postCreated(event, data) {});

    $scope.$on('$destroy', function destroyScope() {
```

```
        deregister();
    });
}
```

Section 11.6: Scope functions and filters

AngularJS has digest loop and all your functions in a view and filters are executed every time the digest cycle is run. The digest loop will be executed whenever the model is updated and it can slow down your app (filter can be hit multiple times, before the page is loaded).

You should avoid this:

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.calculateMe()}}</p>
  <p>{{calc.data | heavyFilter}}</p>
</div>
```

Better approach

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.preCalculatedValue}}</p>
  <p>{{calc.data | lightFilter}}</p>
</div>
```

Where controller sample is:

```
.controller("bigCalulations", function(valueService) {
  // bad, because this is called in every digest loop
  this.calculateMe = function() {
    var t = 0;
    for(i = 0; i < 1000; i++) {
      t = t + i;
    }
    return t;
  }
  //good, because it is executed just once and logic is separated in service to    keep the
controller light
  this.preCalulatedValue = valueService.caluclateSumm(); // returns 499500
});
```

Section 11.7: Debounce Your Model

```
<div ng-controller="ExampleController">
    <form name="userForm">
        Name:
        <input type="text" name="userName"
          ng-model="user.name"
          ng-model-options="{ debounce: 1000 }" />
            <button ng-click="userForm.userName.$rollbackViewValue();
user.name=''">Clear</button><br />
    </form>
    <pre>user.name = </pre>
</div>
```

The above example we are setting a debounce value of 1000 milliseconds which is 1 second. This is a considerable delay, but will prevent the input from repeatedly thrashing `ng-model` with many `$digest` cycles.

By using debounce on your input fields and anywhere else where an instant update is not required, you can increase the performance of your Angular apps quite substantially. Not only can you delay by time, but you can also delay when the action gets triggered. If you don't want to update your ng-model on every keystroke, you can also update on blur as well.

Chapter 12: Events

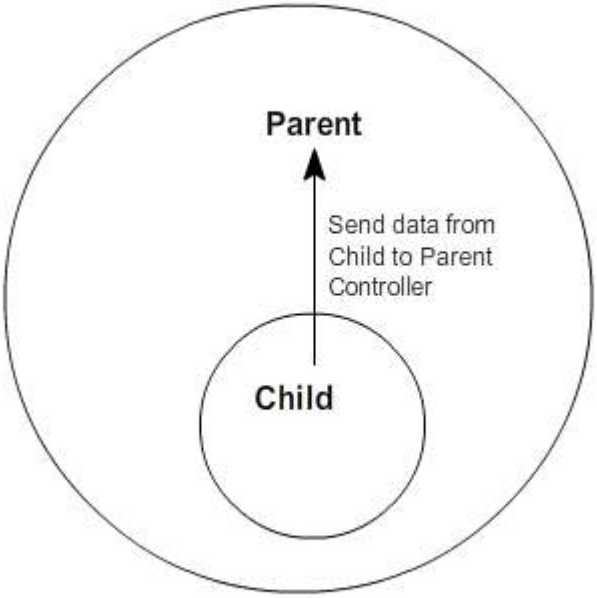
Parameters	Values types
event	Object {name: "eventName", targetScope: Scope, defaultPrevented: false, currentScope: ChildScope}
args	data that has been passed along with event execution

Section 12.1: Using angular event system

$scope.$emit

Using `$scope.$emit` will fire an event name upwards through the scope hierarchy and notify to the `$scope`.The event life cycle starts at the scope on which `$emit` was called.

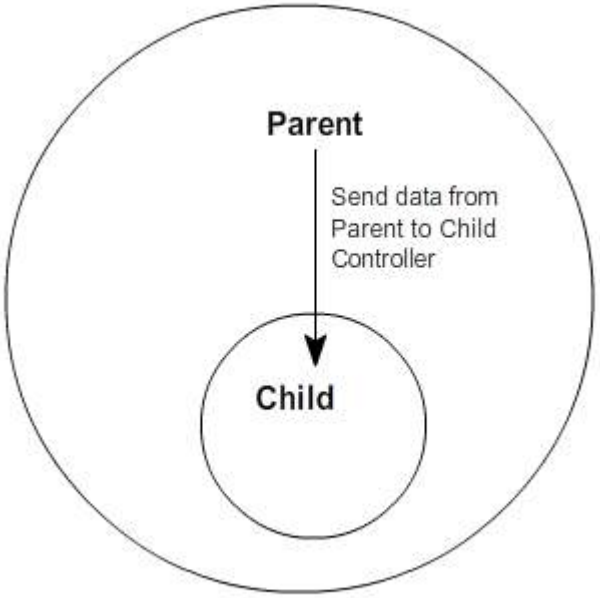
Working wireframe :



$scope.$broadcast

Using `$scope.$broadcast` will fire an event down the `$scope`. We can listen of these events using `$scope.$on`

Working wireframe :



Syntax :

```
// firing an event upwards
$scope.$emit('myCustomEvent', 'Data to send');

// firing an event downwards
$scope.$broadcast('myCustomEvent', {
  someProp: 'some value'
});

// listen for the event in the relevant $scope
$scope.$on('myCustomEvent', function (event, data) {
  console.log(data); // 'Data from the event'
});
```

Instead of `$scope` you can use `$rootScope`, in that case your event will be available in all the controllers regardless of that controllers scope

Clean registered event in AngularJS

The reason to clean the registered events because even the controller has been destroyed the handling of registered event are still alive. So the code will run as unexpected for sure.

```
// firing an event upwards
$rootScope.$emit('myEvent', 'Data to send');

// listening an event
var listenerEventHandler = $rootScope.$on('myEvent', function(){
    //handle code
});

$scope.$on('$destroy', function() {
    listenerEventHandler();
});
```

Section 12.2: Always deregister $rootScope.$on listeners on the scope $destory event

$rootScope.$on listeners will remain in memory if you navigate to another controller. This will create a memory leak if the controller falls out of scope.

Don't

```
angular.module('app').controller('badExampleController', badExample);

badExample.$inject = ['$scope', '$rootScope'];
function badExample($scope, $rootScope) {

    $rootScope.$on('post:created', function postCreated(event, data) {});

}
```

Do

```
angular.module('app').controller('goodExampleController', goodExample);

goodExample.$inject = ['$scope', '$rootScope'];
function goodExample($scope, $rootScope) {

    var deregister = $rootScope.$on('post:created', function postCreated(event, data) {});

    $scope.$on('$destroy', function destroyScope() {
        deregister();
    });

}
```

Section 12.3: Uses and significance

These events can be used to communicate between 2 or more controllers.

`$emit` dispatches an event upwards through the scope hierarchy, while `$broadcast` dispatches an event downwards to all child scopes.This has been beautifully explained here.

There can be basically two types of scenario while communicating among controllers:

1. When controllers have Parent-Child relationship. (we can mostly use `$scope` in such scenarios)
2. When controllers are not independent to each other and are needed to be informed about each others activity. (we can use `$rootScope` in such scenarios)

eg: For any ecommerce website, suppose we have `ProductListController`(which controls the product listing page when any product brand is clicked) and `CartController` (to manage cart items) . Now, when we click on **Add to Cart** button , it has to be informed to `CartController` as well, so that it can reflect new cart item count/details in the navigation bar of the website. This can be achieved using `$rootScope`.

With `$scope.$emit`

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.4/angular.js"></script>
    <script>
```

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

        app.controller("FirstController", function ($scope) {
            $scope.$on('eventName', function (event, args) {
                $scope.message = args.message;
            });
        });

        app.controller("SecondController", function ($scope) {
            $scope.handleClick = function (msg) {
                $scope.$emit('eventName', {message: msg});
            };
        });

    </script>
  </head>
  <body ng-app="app">
    <div ng-controller="FirstController" style="border:2px ;padding:5px;">
        <h1>Parent Controller</h1>
        <p>Emit Message : {{message}}</p>
        <br />
        <div ng-controller="SecondController" style="border:2px;padding:5px;">
            <h1>Child Controller</h1>
            <input ng-model="msg">
            <button ng-click="handleClick(msg);">Emit</button>
        </div>
    </div>
  </body>
</html>
```

With `$scope.$broadcast`:

```
<html>
  <head>
    <title>Broadcasting</title>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.4/angular.js"></script>
    <script>
        var app = angular.module('app', []);

        app.controller("FirstController", function ($scope) {
            $scope.handleClick = function (msg) {
                $scope.$broadcast('eventName', {message: msg});
            };

        });

        app.controller("SecondController", function ($scope) {
            $scope.$on('eventName', function (event, args) {
                $scope.message = args.message;
            });
        });

    </script>
  </head>
  <body ng-app="app">
    <div ng-controller="FirstController" style="border:2px solid ; padding:5px;">
        <h1>Parent Controller</h1>
        <input ng-model="msg">
        <button ng-click="handleClick(msg);">Broadcast</button>
        <br /><br />
        <div ng-controller="SecondController" style="border:2px solid ;padding:5px;">
```

```
            <h1>Child Controller</h1>
            <p>Broadcast Message : {{message}}</p>
        </div>
    </div>
  </body>
</html>
```

Chapter 13: Sharing Data

Section 13.1: Using ngStorage to share data

Firstly, include the **ngStorage** source in your index.html.

An example injecting `ngStorage` src would be:

```
<head>
    <title>Angular JS ngStorage</title>
    <script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script src="https://rawgithub.com/gsklee/ngStorage/master/ngStorage.js"></script>
</head>
```

`ngStorage` gives you 2 storage namely: `$localStorage` and `$sessionStorage`. You need to require ngStorage and Inject the services.

Suppose if `ng-app="myApp"`, then you would be injecting `ngStorage` as following:

```
var app = angular.module('myApp', ['ngStorage']);
        app.controller('controllerOne', function($localStorage,$sessionStorage) {
        // an object to share
        var sampleObject = {
            name: 'angularjs',
            value: 1
        };
       $localStorage.valueToShare = sampleObject;
       $sessionStorage.valueToShare = sampleObject;
    })
.controller('controllerTwo', function($localStorage,$sessionStorage) {
    console.log('localStorage: '+ $localStorage +'sessionStorage: '+$sessionStorage);
})
```

`$localStorage` and `$sessionStorage` is globally accessible through any controllers as long as you inject those services in the controllers.

You can also use the `localStorage` and `sessionStorage` of `HTML5`. However, using `HTML5` `localStorage` would require you to serialize and deserialize your objects before using or saving them.

For example:

```
var myObj = {
    firstname: "Nic",
    lastname: "Raboy",
    website: "https://www.google.com"
}
//if you wanted to save into localStorage, serialize it
window.localStorage.set("saved", JSON.stringify(myObj));

//unserialize to get object
var myObj = JSON.parse(window.localStorage.get("saved"));
```

Section 13.2: Sharing data from one controller to another using service

We can create a `service` to **`set`** and **`get`** the data between the `controllers` and then inject that service in the

controller function where we want to use it.

Service :

```
app.service('setGetData', function() {
  var data = '';
    getData: function() { return data; },
    setData: function(requestData) { data = requestData; }
});
```

Controllers :

```
app.controller('myCtrl1', ['setGetData',function(setGetData) {

  // To set the data from the one controller
  var data = 'Hello World !!';
  setGetData.setData(data);

}]);

app.controller('myCtrl2', ['setGetData',function(setGetData) {

  // To get the data from the another controller
  var res = setGetData.getData();
  console.log(res); // Hello World !!

}]);
```

Here, we can see that `myCtrl1` is used for `setting` the data and `myCtrl2` is used for `getting` the data. So, we can share the data from one controller to another contrller like this.

Chapter 14: How data binding works

Section 14.1: Data Binding Example

```
<p ng-bind="message"></p>
```

This 'message' has to be attached to the current elements controller's scope.

```
$scope.message = "Hello World";
```

At a later point of time , even if the message model is updated , that updated value is reflected in the HTML element. When angular compiles the template "Hello World" will be attached to the innerHTML of the current world. Angular maintains a Watching mechanism of all the directives atttached to the view. It has a Digest Cycle mechanism where it iterates through the Watchers array, it will update the DOM element if there is a change in the previous value of the model.

There is no periodic checking of Scope whether there is any change in the Objects attached to it. Not all the objects attached to scope are watched . Scope prototypically maintains a **$$WatchersArray** . Scope only iterates through this WatchersArray when $digest is called .

Angular adds a watcher to the WatchersArray for each of these

1. {{expression}}?—?In your templates (and anywhere else where there's an expression) or when we define ng-model.
2. $scope.$watch('expression/function')?—?In your JavaScript we can just attach a scope object for angular to watch.

$watch function takes in three parameters:

1. First one is a watcher function which just returns the object or we can just add an expression.

2. Second one is a listener function which will be called when there is a change in the object. All the things like DOM changes will be implemented in this function.

3. The third being an optional parameter which takes in a boolean . If its true , angular deep watches the object & if its false Angular just does a reference watching on the object. Rough Implementation of $watch looks like this

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
   var watcher = {
       watchFn: watchFn,
       listenerFn: listenerFn || function() { },
       last: initWatchVal  // initWatchVal is typically undefined
   };
   this.$$watchers.push(watcher); // pushing the Watcher Object to Watchers
};
```

There is an interesting thing in Angular called Digest Cycle. The $digest cycle starts as a result of a call to $scope.$digest(). Assume that you change a $scope model in a handler function through the ng-click directive. In

that case AngularJS automatically triggers a $digest cycle by calling $digest().In addition to ng-click, there are several other built-in directives/services that let you change models (e.g. ng-model, $timeout, etc) and automatically trigger a $digest cycle. The rough implementation of $digest looks like this.

```
Scope.prototype.$digest = function() {
      var dirty;
      do {
          dirty = this.$$digestOnce();
      } while (dirty);
}
Scope.prototype.$$digestOnce = function() {
   var self = this;
   var newValue, oldValue, dirty;
   _.forEach(this.$$watchers, function(watcher) {
          newValue = watcher.watchFn(self);
          oldValue = watcher.last;   // It just remembers the last value for dirty checking
          if (newValue !== oldValue) { //Dirty checking of References
   // For Deep checking the object , code of Value
   // based checking of Object should be implemented here
             watcher.last = newValue;
             watcher.listenerFn(newValue,
                  (oldValue === initWatchVal ? newValue : oldValue),
                   self);
          dirty = true;
          }
     });
   return dirty;
 };
```

If we use JavaScript's **setTimeout()** function to update a scope model, Angular has no way of knowing what you might change. In this case it's our responsibility to call $apply() manually, which triggers a $digest cycle. Similarly, if you have a directive that sets up a DOM event listener and changes some models inside the handler function, you need to call $apply() to ensure the changes take effect. The big idea of $apply is that we can execute some code that isn't aware of Angular, that code may still change things on the scope. If we wrap that code in $apply , it will take care of calling $digest(). Rough implementation of $apply().

```
Scope.prototype.$apply = function(expr) {
       try {
         return this.$eval(expr); //Evaluating code in the context of Scope
       } finally {
         this.$digest();
       }
};
```

Chapter 15: Routing using ngRoute

Section 15.1: Basic example

This example shows setting up a small application with 3 routes, each with it's own view and controller, using the `controllerAs` syntax.

We configure our router at the angular `.config` function

1. We inject `$routeProvider` into `.config`
2. We define our route names at the `.when` method with a route definition object.
3. We supply the `.when` method with an object specifying our `template` or `templateUrl`, `controller` and `controllerAs`

app.js

```
angular.module('myApp', ['ngRoute'])
  .controller('controllerOne', function() {
    this.message = 'Hello world from Controller One!';
  })
  .controller('controllerTwo', function() {
    this.message = 'Hello world from Controller Two!';
  })
  .controller('controllerThree', function() {
    this.message = 'Hello world from Controller Three!';
  })
  .config(function($routeProvider) {
    $routeProvider
    .when('/one', {
      templateUrl: 'view-one.html',
      controller: 'controllerOne',
      controllerAs: 'ctrlOne'
    })
    .when('/two', {
      templateUrl: 'view-two.html',
      controller: 'controllerTwo',
      controllerAs: 'ctrlTwo'
    })
    .when('/three', {
      templateUrl: 'view-three.html',
      controller: 'controllerThree',
      controllerAs: 'ctrlThree'
    })
    // redirect to here if no other routes match
    .otherwise({
      redirectTo: '/one'
    });
  });
```

Then in our HTML we define our navigation using `<a>` elements with `href`, for a route name of `helloRoute` we will route as `<a href="#/helloRoute">My route</a>`

We also provide our view with a container and the directive `ng-view` to inject our routes.

index.html

```
<div ng-app="myApp">
  <nav>
```

```
    <!-- links to switch routes -->
    <a href="#/one">View One</a>
    <a href="#/two">View Two</a>
    <a href="#/three">View Three</a>
  </nav>
  <!-- views will be injected here -->
  <div ng-view></div>
  <!-- templates can live in normal html files -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>
</div>
```

Section 15.2: Defining custom behavior for individual routes

The simplest manner of defining custom behavior for individual routes would be fairly easy.

In this example we use it to authenticate a user :

1) `routes.js`: create a new property (like `requireAuth`) for any desired route

```
angular.module('yourApp').config(['$routeProvider', function($routeProvider) {
    $routeProvider
        .when('/home', {
            templateUrl: 'templates/home.html',
            requireAuth: true
        })
        .when('/login', {
            templateUrl: 'templates/login.html',
        })
        .otherwise({
            redirectTo: '/home'
        });
}])
```

2) In a top-tier controller that isn't bound to an element inside the `ng-view` (to avoid conflict with angular `$routeProvider`), check if the `newUrl` has the `requireAuth` property and act accordingly

```
angular.module('YourApp').controller('YourController', ['$scope', 'session', '$location',
    function($scope, session, $location) {

        $scope.$on('$routeChangeStart', function(angularEvent, newUrl) {

            if (newUrl.requireAuth && !session.user) {
                // User isn't authenticated
                $location.path("/login");
            }

        });
    }
]);
```

Section 15.3: Route parameters example

This example extends the basic example passing parameters in the route in order to use them in the controller

To do so we need to:

1. Configure the parameter position and name in the route name
2. Inject `$routeParams` service in our Controller

app.js

```
angular.module('myApp', ['ngRoute'])
  .controller('controllerOne', function() {
    this.message = 'Hello world from Controller One!';
  })
  .controller('controllerTwo', function() {
    this.message = 'Hello world from Controller Two!';
  })
  .controller('controllerThree', ['$routeParams', function($routeParams) {
    var routeParam = $routeParams.paramName

    if ($routeParams.message) {
        // If a param called 'message' exists, we show it's value as the message
        this.message = $routeParams.message;
    } else {
        // If it doesn't exist, we show a default message
        this.message = 'Hello world from Controller Three!';
    }
  }])
  .config(function($routeProvider) {
    $routeProvider
    .when('/one', {
      templateUrl: 'view-one.html',
      controller: 'controllerOne',
      controllerAs: 'ctrlOne'
    })
    .when('/two', {
      templateUrl: 'view-two.html',
      controller: 'controllerTwo',
      controllerAs: 'ctrlTwo'
    })
    .when('/three', {
      templateUrl: 'view-three.html',
      controller: 'controllerThree',
      controllerAs: 'ctrlThree'
    })
    .when('/three/:message', { // We will pass a param called 'message' with this route
      templateUrl: 'view-three.html',
      controller: 'controllerThree',
      controllerAs: 'ctrlThree'
    })
    // redirect to here if no other routes match
    .otherwise({
      redirectTo: '/one'
    });
  });
```

Then, withoud making any changes in our templates, only adding a new link with custom message, we can see the new custom message in our view.

index.html

```
<div ng-app="myApp">
  <nav>
    <!-- links to switch routes -->
    <a href="#/one">View One</a>
    <a href="#/two">View Two</a>
    <a href="#/three">View Three</a>
    <!-- New link with custom message -->
    <a href="#/three/This-is-a-message">View Three with "This-is-a-message" custom message</a>
  </nav>
  <!-- views will be injected here -->
  <div ng-view></div>
  <!-- templates can live in normal html files -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>
</div>
```

Chapter 16: ng-class directive

Section 16.1: Three types of ng-class expressions

Angular supports three types of expressions in the `ng-class` directive.

1. String

```
<span ng-class="MyClass">Sample Text</span>
```

Specifying an expression that evaluates to a string tells Angular to treat it as a $scope variable. Angular will check the $scope and look for a variable called "MyClass". Whatever text is contained in "MyClass" will become the actual class name that is applied to this ****. You can specify multiple classes by separating each class with a space.

In your controller, you may have a definition that looks like this:

```
$scope.MyClass = "bold-red deleted error";
```

Angular will evaluate the expression `MyClass` and find the $scope definition. It will apply the three classes "bold-red", "deleted", and "error" to the **** element.

Specifying classes this way lets you easily change the class definitions in your controller. For example, you may need to change the class based on other user interactions or new data that is loaded from the server. Also, if you have a lot of expressions to evaluate, you can do so in a function that defines the final list of classes in a `$scope` variable. This can be easier than trying to squeeze many evaluations into the `ng-class` attribute in your HTML template.

2. Object

This is the most commonly-used way of defining classes using `ng-class` because it easily lets you specify evaluations that determine which class to use.

Specify an object containing key-value pairs. The key is the class name that will be applied if the value (a conditional) evaluates as true.

```
<style>
    .red { color: red; font-weight: bold; }
    .blue { color: blue; }
    .green { color: green; }
    .highlighted { background-color: yellow; color: black; }
</style>

<span ng-class="{ red: ShowRed, blue: ShowBlue, green: ShowGreen, highlighted: IsHighlighted
}">Sample Text</span>

<div>Red: <input type="checkbox" ng-model="ShowRed"></div>
<div>Green: <input type="checkbox" ng-model="ShowGreen"></div>
<div>Blue: <input type="checkbox" ng-model="ShowBlue"></div>
<div>Highlight: <input type="checkbox" ng-model="IsHighlighted"></div>
```

3. Array

An expression that evaluates to an array lets you use a combination of **strings** (see #1 above) and **conditional objects** (#2 above).

```
<style>
    .bold { font-weight: bold; }
```

```
    .strike { text-decoration: line-through; }
    .orange { color: orange; }
</style>

<p ng-class="[ UserStyle, {orange: warning} ]">Array of Both Expression Types</p>
<input ng-model="UserStyle" placeholder="Type 'bold' and/or 'strike'"><br>
<label><input type="checkbox" ng-model="warning"> warning (apply "orange" class)</label>
```

This creates a text input field bound to the scope variable `UserStyle` which lets the user type in any class name(s). These will be dynamically applied to the **<p>** element as the user types. Also, the user can click on the checkbox that is data-bound to the `warning` scope variable. This will also be dynamically applied to the **<p>** element.

Chapter 17: Directives using ngModelController

Section 17.1: A simple control: rating

Let us build a simple control, a rating widget, intended to be used as:

```
<rating min="0" max="5" nullifier="true" ng-model="data.rating"></rating>
```

No fancy CSS for now; this would render as:

```
0 1 2 3 4 5 x
```

Clicking on a number selects that rating; and clicking the "x" sets the rating to null.

```
app.directive('rating', function() {

    function RatingController() {
        this._ngModel = null;
        this.rating = null;
        this.options = null;
        this.min = typeof this.min === 'number' ? this.min : 1;
        this.max = typeof this.max === 'number' ? this.max : 5;
    }

    RatingController.prototype.setNgModel = function(ngModel) {
        this._ngModel = ngModel;

        if( ngModel ) {
            // KEY POINT 1
            ngModel.$render = this._render.bind(this);
        }
    };

    RatingController.prototype._render = function() {
        this.rating = this._ngModel.$viewValue != null ? this._ngModel.$viewValue : -
Number.MAX_VALUE;
    };

    RatingController.prototype._calculateOptions = function() {
        if( this.min == null || this.max == null ) {
            this.options = [];
        }
        else {
            this.options = new Array(this.max - this.min + 1);
            for( var i=0; i < this.options.length; i++ ) {
                this.options[i] = this.min + i;
            }
        }
    };

    RatingController.prototype.setValue = function(val) {
        this.rating = val;
        // KEY POINT 2
        this._ngModel.$setViewValue(val);
    };

    // KEY POINT 3
```

```
    Object.defineProperty(RatingController.prototype, 'min', {
        get: function() {
            return this._min;
        },
        set: function(val) {
            this._min = val;
            this._calculateOptions();
        }
    });

    Object.defineProperty(RatingController.prototype, 'max', {
        get: function() {
            return this._max;
        },
        set: function(val) {
            this._max = val;
            this._calculateOptions();
        }
    });

    return {
        restrict: 'E',
        scope: {
            // KEY POINT 3
            min: '<?',
            max: '<?',
            nullifier: '<?'
        },
        bindToController: true,
        controllerAs: 'ctrl',
        controller: RatingController,
        require: ['rating', 'ngModel'],
        link: function(scope, elem, attrs, ctrls) {
            ctrls[0].setNgModel(ctrls[1]);
        },
        template:
            '<span ng-repeat="o in ctrl.options" href="#" class="rating-option" ng-
class="{\'rating-option-active\': o <= ctrl.rating}" ng-click="ctrl.setValue(o)">{{ o }}</span>' +
            '<span ng-if="ctrl.nullifier" ng-click="ctrl.setValue(null)" class="rating-
nullifier">&#10006;</span>'
    };
});
```

Key points:

1. Implement `ngModel.$render` to transfer the model's *view value* to your view.
2. Call `ngModel.$setViewValue()` whenever you feel the view value should be updated.
3. The control can of course be parameterized; use `'<'` scope bindings for parameters, if in Angular >= 1.5 to clearly indicate input - one way binding. If you have to take action whenever a parameter changes, you can use a JavaScript property (see `Object.defineProperty()`) to save a few watches.

Note 1: In order not to overcomplicate the implementation, the rating values are inserted in an array - the `ctrl.options`. This is not needed; a more efficient, but also more complex, implementation could use DOM manipulation to insert/remove ratings when `min`/`max` change.

Note 2: With the exception of the `'<'` scope bindings, this example can be used in Angular < 1.5. If you are on Angular >= 1.5, it would be a good idea to tranform this to a component and use the `$onInit()` lifecycle hook to initialize `min` and `max`, instead of doing so in the controller's constructor.

And a necessary fiddle: https://jsfiddle.net/h81mgxma/

Section 17.2: A couple of complex controls: edit a full object

A custom control does not have to limit itself to trivial things like primitives; it can edit more interesting things. Here we present two types of custom controls, one for editing persons and one for editing addresses. The address control is used to edit the person's address. An example of usage would be:

```
<input-person ng-model="data.thePerson"></input-person>
<input-address ng-model="data.thePerson.address"></input-address>
```

The model for this example is deliberately simplistic:

```
function Person(data) {
  data = data || {};
  this.name = data.name;
  this.address = data.address ? new Address(data.address) : null;
}

function Address(data) {
  data = data || {};
  this.street = data.street;
  this.number = data.number;
}
```

The address editor:

```
app.directive('inputAddress', function() {

    InputAddressController.$inject = ['$scope'];
    function InputAddressController($scope) {
        this.$scope = $scope;
        this._ngModel = null;
        this.value = null;
        this._unwatch = angular.noop;
    }

    InputAddressController.prototype.setNgModel = function(ngModel) {
        this._ngModel = ngModel;

        if( ngModel ) {
            // KEY POINT 3
            ngModel.$render = this._render.bind(this);
        }
    };

    InputAddressController.prototype._makeWatch = function() {
        // KEY POINT 1
        this._unwatch = this.$scope.$watchCollection(
            (function() {
                return this.value;
            }).bind(this),
            (function(newval, oldval) {
                if( newval !== oldval ) { // skip the initial trigger
                    this._ngModel.$setViewValue(newval !== null ? new Address(newval) : null);
                }
            }).bind(this)
        );
    };
```

```
    InputAddressController.prototype._render = function() {
        // KEY POINT 2
        this._unwatch();
        this.value = this._ngModel.$viewValue ? new Address(this._ngModel.$viewValue) : null;
        this._makeWatch();
    };

    return {
        restrict: 'E',
        scope: {},
        bindToController: true,
        controllerAs: 'ctrl',
        controller: InputAddressController,
        require: ['inputAddress', 'ngModel'],
        link: function(scope, elem, attrs, ctrls) {
            ctrls[0].setNgModel(ctrls[1]);
        },
        template:
            '<div>' +
                '<label><span>Street:</span><input type="text" ng-model="ctrl.value.street"
/></label>' +
                '<label><span>Number:</span><input type="text" ng-model="ctrl.value.number"
/></label>' +
            '</div>'
    };
});
```

Key points:

1. We are editing an object; we do not want to change directly the object given to us from our parent (we want our model to be compatible with the immutability principle). So we create a shallow watch on the object being edited and update the model with `$setViewValue()` whenever a property changes. We pass a *copy* to our parent.
2. Whenever the model changes from the outside, we copy it and save the copy to our scope. Immutability principles again, though the internal copy is not immutable, the external could very well be. Additionally we rebuild the watch (`this._unwatch();this._makeWatch();`), to avoid triggering the watcher for changes pushed to us by the model. (We only want the watch to trigger for changes made in the UI.)
3. Other that the points above, we implement `ngModel.$render()` and call `ngModel.$setViewValue()` as we would for a simple control (see the rating example).

The code for the person custom control is almost identical. The template is using the **`<input-address>`**. In a more advanced implementation we could extract the controllers in a reusable module.

```
app.directive('inputPerson', function() {

    InputPersonController.$inject = ['$scope'];
    function InputPersonController($scope) {
        this.$scope = $scope;
        this._ngModel = null;
        this.value = null;
        this._unwatch = angular.noop;
    }

    InputPersonController.prototype.setNgModel = function(ngModel) {
        this._ngModel = ngModel;

        if( ngModel ) {
            ngModel.$render = this._render.bind(this);
```

```
        }
    };

    InputPersonController.prototype._makeWatch = function() {
        this._unwatch = this.$scope.$watchCollection(
            (function() {
                return this.value;
            }).bind(this),
            (function(newval, oldval) {
                if( newval !== oldval ) { // skip the initial trigger
                    this._ngModel.$setViewValue(newval !== null ? new Person(newval) : null);
                }
            }).bind(this)
        );
    };

    InputPersonController.prototype._render = function() {
        this._unwatch();
        this.value = this._ngModel.$viewValue ? new Person(this._ngModel.$viewValue) : null;
        this._makeWatch();
    };

    return {
        restrict: 'E',
        scope: {},
        bindToController: true,
        controllerAs: 'ctrl',
        controller: InputPersonController,
        require: ['inputPerson', 'ngModel'],
        link: function(scope, elem, attrs, ctrls) {
            ctrls[0].setNgModel(ctrls[1]);
        },
        template:
            '<div>' +
                '<label><span>Name:</span><input type="text" ng-model="ctrl.value.name" /></label>'
+
                '<input-address ng-model="ctrl.value.address"></input-address>' +
            '</div>'
    };
});
```

Note: Here the objects are typed, i.e. they have proper constructors. This is not obligatory; the model can be plain JSON objects. In this case just use `angular.copy()` instead of the constructors. An added advantage is that the controller becomes identical for the two controls and can easily be extracted into some common module.

The fiddle: https://jsfiddle.net/3tzyqfko/2/

Two versions of the fiddle having extracted the common code of the controllers: https://jsfiddle.net/agj4cp0e/ and https://jsfiddle.net/ugb6Lw8b/

Chapter 18: ui-router

Section 18.1: Basic Example

app.js

```
angular.module('myApp', ['ui.router'])
  .controller('controllerOne', function() {
    this.message = 'Hello world from Controller One!';
  })
  .controller('controllerTwo', function() {
    this.message = 'Hello world from Controller Two!';
  })
  .controller('controllerThree', function() {
    this.message = 'Hello world from Controller Three!';
  })
  .config(function($stateProvider, $urlRouterProvider) {
    $stateProvider
      .state('one', {
        url: "/one",
        templateUrl: "view-one.html",
        controller: 'controllerOne',
        controllerAs: 'ctrlOne'
      })
      .state('two', {
        url: "/two",
        templateUrl: "view-two.html",
        controller: 'controllerTwo',
        controllerAs: 'ctrlTwo'
      })
      .state('three', {
        url: "/three",
        templateUrl: "view-three.html",
        controller: 'controllerThree',
        controllerAs: 'ctrlThree'
      });

      $urlRouterProvider.otherwise('/one');
  });
```

index.html

```
<div ng-app="myApp">
  <nav>
    <!-- links to switch routes -->
    <a ui-sref="one">View One</a>
    <a ui-sref="two">View Two</a>
    <a ui-sref="three">View Three</a>
  </nav>
  <!-- views will be injected here -->
  <div ui-view></div>
  <!-- templates can live in normal html files -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>
```

```
  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>
</div>
```

Section 18.2: Multiple Views

app.js

```
angular.module('myApp', ['ui.router'])
  .controller('controllerOne', function() {
    this.message = 'Hello world from Controller One!';
  })
  .controller('controllerTwo', function() {
    this.message = 'Hello world from Controller Two!';
  })
  .controller('controllerThree', function() {
    this.message = 'Hello world from Controller Three!';
  })
  .controller('controllerFour', function() {
    this.message = 'Hello world from Controller Four!';
  })
  .config(function($stateProvider, $urlRouterProvider) {
    $stateProvider
      .state('one', {
        url: "/one",
        views: {
          "viewA": {
            templateUrl: "view-one.html",
            controller: 'controllerOne',
            controllerAs: 'ctrlOne'
          },
          "viewB": {
            templateUrl: "view-two.html",
            controller: 'controllerTwo',
            controllerAs: 'ctrlTwo'
          }
        }
      })
      .state('two', {
        url: "/two",
        views: {
          "viewA": {
            templateUrl: "view-three.html",
            controller: 'controllerThree',
            controllerAs: 'ctrlThree'
          },
          "viewB": {
            templateUrl: "view-four.html",
            controller: 'controllerFour',
            controllerAs: 'ctrlFour'
          }
        }
      });

    $urlRouterProvider.otherwise('/one');
  });
```

index.html

```
<div ng-app="myApp">
  <nav>
    <!-- links to switch routes -->
    <a ui-sref="one">Route One</a>
    <a ui-sref="two">Route Two</a>
  </nav>
  <!-- views will be injected here -->
  <div ui-view="viewA"></div>
  <div ui-view="viewB"></div>
  <!-- templates can live in normal html files -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-four.html">
    <h1>{{ctrlFour.message}}</h1>
  </script>
</div>
```

Section 18.3: Using resolve functions to load data

app.js

```
angular.module('myApp', ['ui.router'])
  .service('User', ['$http', function User ($http) {
    this.getProfile = function (id) {
      return $http.get(...) // method to load data from API
    };
  }])
  .controller('profileCtrl', ['profile', function profileCtrl (profile) {
    // inject resolved data under the name of the resolve function
    // data will already be returned and processed
    this.profile = profile;
  }])
  .config(['$stateProvider', '$urlRouterProvider', function ($stateProvider, $urlRouterProvider) {
    $stateProvider
      .state('profile', {
        url: "/profile/:userId",
        templateUrl: "profile.html",
        controller: 'profileCtrl',
        controllerAs: 'vm',
        resolve: {
          profile: ['$stateParams', 'User', function ($stateParams, User) {
            // $stateParams will contain any parameter defined in your url
            return User.getProfile($stateParams.userId)
              // .then is only necessary if you need to process returned data
              .then(function (data) {
                return doSomeProcessing(data);
              });
          }]
        }
      }]);
```

```
    $urlRouterProvider.otherwise('/');
});
```

profile.html

```
<ul>
  <li>Name: {{vm.profile.name}}</li>
  <li>Age: {{vm.profile.age}}</li>
  <li>Sex: {{vm.profile.sex}}</li>
</ul>
```

View UI-Router Wiki entry on resolves here.

Resolve functions must be resolved before the `$stateChangeSuccess` event is fired, which means that the UI will not load until *all* resolve functions on the state have finished. This is a great way to ensure that data will be available to your controller and UI. However, you can see that a resolve function should be fast in order to avoid hanging the UI.

Section 18.4: Nested Views / States

app.js

```
var app = angular.module('myApp',['ui.router']);

app.config(function($stateProvider,$urlRouterProvider) {

    $stateProvider

    .state('home', {
        url: '/home',
        templateUrl: 'home.html',
        controller: function($scope){
            $scope.text = 'This is the Home'
        }
    })

    .state('home.nested1',{
        url: '/nested1',
        templateUrl:'nested1.html',
        controller: function($scope){
            $scope.text1 = 'This is the nested view 1'
        }
    })

    .state('home.nested2',{
        url: '/nested2',
        templateUrl:'nested2.html',
        controller: function($scope){
            $scope.fruits = ['apple','mango','oranges'];
        }
    });

    $urlRouterProvider.otherwise('/home');

});
```

index.html

```
  <div ui-view></div>
```

```
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.min.js"></script>
    <script src="angular-ui-router.min.js"></script>
    <script src="app.js"></script>
```

home.html

```
<div>
<h1> {{text}} </h1>
<br>
    <a ui-sref="home.nested1">Show nested1</a>
    <br>
    <a ui-sref="home.nested2">Show nested2</a>
    <br>

    <div ui-view></div>
</div>
```

nested1.html

```
<div>
<h1> {{text1}} </h1>
</div>
```

nested2.html

```
<div>
    <ul>
      <li ng-repeat="fruit in fruits">{{ fruit }}</li>
    </ul>
</div>
```

Chapter 19: Custom filters

Section 19.1: Use a filter in a controller, a service or a filter

You will have to inject `$filter`:

```
angular
  .module('filters', [])
  .filter('percentage', function($filter) {
    return function (input) {
      return $filter('number')(input * 100) + ' %';
    };
  });
```

Section 19.2: Create a filter with parameters

By default, a filter has a single parameter: the variable it is applied on. But you can pass more parameter to the function:

```
angular
  .module('app', [])
  .controller('MyController', function($scope) {
    $scope.example = 0.098152;
  })
  .filter('percentage', function($filter) {
    return function (input, decimals) {
      return $filter('number')(input * 100, decimals) + ' %';
    };
  });
```

Now, you can give a precision to the `percentage` filter:

```
<span ng-controller="MyController">{{ example | percentage: 2 }}</span>
=> "9.81 %"
```

... but other parameters are optional, you can still use the default filter:

```
<span ng-controller="MyController">{{ example | percentage }}</span>
=> "9.8152 %"
```

Section 19.3: Simple filter example

Filters format the value of an expression for display to the user. They can be used in view templates, controllers or services. This example creates a filter (`addZ`) then uses it in a view. All this filter does is add a capital 'Z' to the end of the string.

example.js

```
angular.module('main', [])
    .filter('addZ', function() {
            return function(value) {
                return value + "Z";
            }
     })
    .controller('MyController', ['$scope', function($scope) {
        $scope.sample = "hello";
```

```
    }])
```

example.html

Inside the view, the filter is applied with the following syntax: `{ variable | filter}`. In this case, the variable we defined in the controller, `sample`, is being filtered by the filter we created, `addZ`.

```
<div ng-controller="MyController">
   <span>{{sample | addZ}}</span>
</div>
```

Expected output

```
helloZ
```

Chapter 20: Built-in helper Functions

Section 20.1: angular.equals

The `angular.equals` function compares and determines if 2 objects or values are equal, `angular.equals` performs a deep comparison and returns true if and only if at least 1 of the following conditions is met.

> angular.equals(value1, value2)

1. If the objects or values pass the

   `===`

   comparison
2. If both objects or values are of the same type, and all of their properties are also equal by using `angular.equals`
3. Both values are equal to **`NaN`**
4. Both values represent the same regular expression's result.

This function is helpful when you need to deep compare objects or arrays by their values or results rather than just references.

Examples

```
angular.equals(1, 1) // true
angular.equals(1, 2) // false
angular.equals({}, {}) // true, note that {}==={} is false
angular.equals({a: 1}, {a: 1}) // true
angular.equals({a: 1}, {a: 2}) // false
angular.equals(NaN, NaN) // true
```

Section 20.2: angular.toJson

The function `angular.toJson` will take an object and serialize it into a JSON formatted string.

Unlike the native function `JSON.stringify`, This function will remove all properties beginning with `$$` (as angular usually prefixes internal properties with `$$`)

```
angular.toJson(object)
```

As data needs to be serialized before passing through a network, this function is useful to turn any data you wish to transmit into JSON.

This function is also useful for debugging as it works similarly to a `.toString` method would act.

Examples:

```
angular.toJson({name: "barf", occupation: "mog", $$somebizzareproperty: 42})
// "{"name":"barf","occupation":"mog"}"
angular.toJson(42)
// "42"
angular.toJson([1, "2", 3, "4"])
// "[1,"2",3,"4"]"
var fn = function(value) {return value}
```

```
angular.toJson(fn)
// undefined, functions have no representation in JSON
```

Section 20.3: angular.copy

The `angular.copy` function takes an object, array or a value and creates a deep copy of it.

> angular.copy()

Example:

Objects:

```
let obj = {name: "vespa", occupation: "princess"};
let cpy = angular.copy(obj);
cpy.name = "yogurt"
// obj = {name: "vespa", occupation: "princess"}
// cpy = {name: "yogurt", occupation: "princess"}
```

Arrays:

```
var w = [a, [b, [c, [d]]]];
var q = angular.copy(w);
// q = [a, [b, [c, [d]]]]
```

At the above example `angular.equals(w, q)` will evaluate to true because `.equals` tests equality by value. however `w === q` will evaluate to false because strict comparison between objects and arrays is done by reference.

Section 20.4: angular.isString

The function `angular.isString` returns true if the object or value given to it is of the type `string`

> angular.isString(value1)

Examples

```
angular.isString("hello") // true
angular.isString([1, 2]) // false
angular.isString(42) // false
```

This is the equivalent of performing

```
typeof someValue === "string"
```

Section 20.5: angular.isArray

The `angular.isArray` function returns true if and only if the object or value passed to it is of the type `Array`.

> angular.isArray(value)

Examples

```
angular.isArray([]) // true
angular.isArray([2, 3]) // true
angular.isArray({}) // false
angular.isArray(17) // false
```

It is the equivalent of

```
Array.isArray(someValue)
```

Section 20.6: angular.merge

The function angular.merge takes all the enumerable properties from the source object to deeply extend the destination object.

The function returns a reference to the now extended destination object

> angular.merge(destination, source)

Examples

```
angular.merge({}, {}) // {}
angular.merge({name: "king roland"}, {password: "12345"})
// {name: "king roland", password: "12345"}
angular.merge({a: 1}, [4, 5, 6]) // {0: 4, 1: 5, 2: 6, a: 1}
angular.merge({a: 1}, {b: {c: {d: 2}}}) // {"a":1,"b":{"c":{"d":2}}}
```

Section 20.7: angular.isDefined and angular.isUndefined

The function `angular.isDefined` tests a value if it is defined

> angular.isDefined(someValue)

This is the equivalent of performing

```
value !== undefined; // will evaluate to true is value is defined
```

Examples

```
angular.isDefined(42) // true
angular.isDefined([1, 2]) // true
angular.isDefined(undefined) // false
angular.isDefined(null) // true
```

The function `angular.isUndefined` tests if a value is undefined (it is effectively the opposite of `angular.isDefined`)

> angular.isUndefined(someValue)

This is the equivalent of performing

```
value === undefined; // will evaluate to true is value is undefined
```

Or just

```
!angular.isDefined(value)
```

Examples

```
angular.isUndefined(42) // false
angular.isUndefined(undefined) // true
```

Section 20.8: angular.isDate

The `angular.isDate` function returns true if and only if the object passed to it is of the type Date.

> angular.isDate(value)

Examples

```
angular.isDate("lone star") // false
angular.isDate(new Date()) // true
```

Section 20.9: angular.noop

The `angular.noop` is a function that performs no operations, you pass `angular.noop` when you need to provide a function argument that will do nothing.

> angular.noop()

A common use for `angular.noop` can be to provide an empty callback to a function that will otherwise throw an error when something else than a function is passed to it.

Example:

```
$scope.onSomeChange = function(model, callback) {
    updateTheModel(model);
    if (angular.isFunction(callback)) {
        callback();
    } else {
        throw new Error("error: callback is not a function!");
    }
};

$scope.onSomeChange(42, function() {console.log("hello callback")});
// will update the model and print 'hello callback'
$scope.onSomeChange(42, angular.noop);
// will update the model
```

Additional examples:

```
angular.noop() // undefined
angular.isFunction(angular.noop) // true
```

Section 20.10: angular.isElement

The `angular.isElement` returns true if the argument passed to it is a DOM Element or a jQuery wrapped Element.

> angular.isElement(elem)

This function is useful to type check if a passed argument is an element before being processed as such.

Examples:

```
angular.isElement(document.querySelector("body"))
// true
angular.isElement(document.querySelector("#some_id"))
// false if "some_id" is not using as an id inside the selected DOM
angular.isElement("<div></div>")
// false
```

Section 20.11: angular.isFunction

The function `angular.isFunction` determines and returns true if and only if the value passed to is a reference to a function.

The function returns a reference to the now extended destination object

> angular.isFunction(fn)

Examples

```
var onClick = function(e) {return e};
angular.isFunction(onClick); // true

var someArray = ["pizza", "the", "hut"];
angular.isFunction(someArray ); // false
```

Section 20.12: angular.identity

The `angular.identity` function returns the first argument passed to it.

> angular.identity(argument)

This function is useful for functional programming, you can provide this function as a default in case an expected function was not passed.

Examples:

```
angular.identity(42) // 42
```

```
var mutate = function(fn, num) {
    return angular.isFunction(fn) ? fn(num) : angular.identity(num)
}

mutate(function(value) {return value-7}, 42) // 35
```

```
mutate(null, 42) // 42
mutate("mount. rushmore", 42) // 42
```

Section 20.13: angular.forEach

The `angular.forEach` accepts an object and an iterator function. It then runs the iterator function over each enumerable property/value of the object. This function also works on arrays.

Like the JS version of `Array.prototype.forEach` The function does not iterate over inherited properties (prototype properties), however the function will not attempt to process a `null` or an `undefined` value and will just return it.

> angular.forEach(object, function(value, key) { // function});

Examples:

```
angular.forEach({"a": 12, "b": 34}, (value, key) => console.log("key: " + key + ", value: " +
value))
// key: a, value: 12
// key: b, value: 34
angular.forEach([2, 4, 6, 8, 10], (value, key) => console.log(key))
// will print the array indices: 1, 2, 3, 4, 5
angular.forEach([2, 4, 6, 8, 10], (value, key) => console.log(value))
// will print the array values: 2, 4, 6, 7, 10
angular.forEach(undefined, (value, key) => console.log("key: " + key + ", value: " + value))
// undefined
```

Section 20.14: angular.isNumber

The `angular.isNumber` function returns true if and only if the object or value passed to it is of the type Number, this includes +Infinity, -Infinity and NaN

> angular.isNumber(value)

This function will not cause a type coercion such as

```
"23" == 23 // true
```

Examples

```
angular.isNumber("23") // false
angular.isNumber(23) // true
angular.isNumber(NaN) // true
angular.isNumber(Infinity) // true
```

This function will not cause a type coercion such as

```
"23" == 23 // true
```

Section 20.15: angular.isObject

The `angular.isObject` return true if and only if the argument passed to it is an object, this function will also return true for an Array and will return false for `null` even though `typeof null` is `object` .

> angular.isObject(value)

This function is useful for type checking when you need a defined object to process.

Examples:

```
angular.isObject({name: "skroob", job: "president"})
// true
angular.isObject(null)
// false
angular.isObject([null])
// true
angular.isObject(new Date())
// true
angular.isObject(undefined)
// false
```

Section 20.16: angular.fromJson

The function `angular.fromJson` will deserialize a valid JSON string and return an Object or an Array.

> angular.fromJson(string|object)

Note that this function is not limited to only strings, it will output a representation of any object passed to it.

Examples:

```
angular.fromJson("{\"yogurt\": \"strawberries\"}")
// Object {yogurt: "strawberries"}
angular.fromJson('{jam: "raspberries"}')
// will throw an exception as the string is not a valid JSON
angular.fromJson(this)
// Window {external: Object, chrome: Object, _gaq: Y, angular: Object, ng339: 3…}
angular.fromJson([1, 2])
// [1, 2]
typeof angular.fromJson(new Date())
// "object"
```

Chapter 21: digest loop walkthrough

Section 21.1: $digest and $watch

Implementing two-way-data-binding, to achieve the result from the previous example, could be done with two core functions:

- **$digest** is called after a user interaction (binding DOM=>variable)
- **$watch** sets a callback to be called after variable changes (binding variable=>DOM)

note: this is example is a demonstration, not the actual angular code

```
<input id="input"/>
<span id="span"></span>
```

The two functions we need:

```
var $watches = [];
function $digest(){
    $watches.forEach(function($w){
        var val = $w.val();
        if($w.prevVal !== val){
            $w.callback(val, $w.prevVal);
            $w.prevVal = val;
        }
    })
}
function $watch(val, callback){
    $watches.push({val:val, callback:callback, prevVal: val() })
}
```

Now we could now use these functions to hook up a variable to the DOM (angular comes with built-in directives which will do this for you):

```
var realVar;
//this is usually done by ng-model directive
input1.addEventListener('keyup',function(e){
    realVar=e.target.value;
    $digest()
}, true);

//this is usually done with {{expressions}} or ng-bind directive
$watch(function(){return realVar},function(val){
    span1.innerHTML = val;
});
```

Off-course, the real implementations are more complex, and support parameters such as **which element** to bind to, and **what variable** to use

A running example could be found here: https://jsfiddle.net/azofxd4j/

Section 21.2: the $scope tree

The previous example is good enough when we need to bind a single html element, to a single variable.

In reality - we need to bind many elements to many variables:

```
<span ng-repeat="number in [1,2,3,4,5]">{{number}}</span>
```

This `ng-repeat` binds 5 elements to 5 variables called `number`, with a different value for each of them!

The way angular achieves this behavior is using a separate context for each element which needs separate variables. This context is called a scope.

Each scope contains properties, which are the variables bound to the DOM, and the `$digest` and `$watch` functions are implemented as methods of the scope.

The DOM is a tree, and variables need to be used in different levels of the tree:

```
<div>
    <input ng-model="person.name" />
    <span ng-repeat="number in [1,2,3,4,5]">{{number}} {{person.name}}</span>
</div>
```

But as we saw, the context(or scope) of variables inside `ng-repeat` is different to the context above it. To solve this - angular implements scopes as a tree.

Each scope has an array of children, and calling its `$digest` method will run all of its children's `$digest` method.

This way - after changing the input - `$digest` is called for the div's scope, which then runs the `$digest` for its 5 children - which will update its content.

A simple implementation for a scope, could look like this:

```
function $scope(){
    this.$children = [];
    this.$watches = [];
}

$scope.prototype.$digest = function(){
    this.$watches.forEach(function($w){
        var val = $w.val();
        if($w.prevVal !== val){
            $w.callback(val, $w.prevVal);
          $w.prevVal = val;
        }
    });
    this.$children.forEach(function(c){
        c.$digest();
    });
}

$scope.prototype.$watch = function(val, callback){
    this.$watches.push({val:val, callback:callback, prevVal: val() })
}
```

note: this is example is a demonstration, not the actual angular code

Section 21.3: two way data binding

Angular has some magic under its hood. it enables binding DOM to real js variables.

Angular uses a loop, named the "*digest loop*", which is called after any change of a variable - calling callbacks which update the DOM.

For example, the `ng-model` directive attaches a `keyup` eventListener to this input:

```
<input ng-model="variable" />
```

Every time the `keyup` event fires, the *digest loop* starts.

At some point, the *digest loop* iterates over a callback which updates the contents of this span:

```
<span>{{variable}}</span>
```

The basic life-cycle of this example, summarizes (very Schematically) how angular works::

1. Angular scans html
    - `ng-model` directive creates a `keyup` listener on input
    - `expression` inside span adds a callback to *digest cycle*
2. User interacts with input
    - `keyup` listener starts *digest cycle*
    - *digest cycle* calles the callback
    - Callback updates span's contents

Chapter 22: Angular $scopes

Section 22.1: A function available in the entire app

Be careful, this approach might be considered as a bad design for angular apps, since it requires programmers to remember both where functions are placed in the scope tree, and to be aware of scope inheritance. In many cases it would be preferred to inject a service (Angular practice - using scope inheritance vs injection.

This example only show how scope inheritance could be used for our needs, and the how you could take advantage of it, and not the best practices of designing an entire app.

In some cases, we could take advantage of scope inheritance, and set a function as a property of the rootScope. This way - all of the scopes in the app (except for isolated scopes) will inherit this function, and it could be called from anywhere in the app.

```
angular.module('app', [])
.run(['$rootScope', function($rootScope){
    var messages = []
    $rootScope.addMessage = function(msg){
        messages.push(msg);
    }
}]);

<div ng-app="app">
    <a ng-click="addMessage('hello world!')">it could be accsessed from here</a>
    <div ng-include="inner.html"></div>
</div>
```

inner.html:

```
<div>
    <button ng-click="addMessage('page')">and from here to!</button>
</div>
```

Section 22.2: Avoid inheriting primitive values

In javascript, assigning a non-primitive value (Such as Object, Array, Function, and many more), keeps a reference (an address in the memory) to the assigned value.

Assigning a primitive value (String, Number, Boolean, or Symbol) to two different variables, and changing one, won't change both:

```
var x = 5;
var y = x;
y = 6;
console.log(y === x, x, y); //false, 5, 6
```

But with a non-primitive value, since both variables are simply keeping references to the same object, changing one variable **will** change the other:

```
var x = { name : 'John Doe' };
var y = x;
y.name = 'Jhon';
console.log(x.name === y.name, x.name, y.name); //true, John, John
```

In angular, when a scope is created, it is assigned all of its parent's properties However, changing properties afterwards will only affect the parent scope if it is a non-primitive value:

```
angular.module('app', [])
.controller('myController', ['$scope', function($scope){
    $scope.person = { name: 'John Doe' }; //non-primitive
    $scope.name = 'Jhon Doe'; //primitive
}])
.controller('myController1', ['$scope', function($scope){}]);

<div ng-app="app" ng-controller="myController">
    binding to input works: {{person.name}}<br/>
    binding to input does not work: {{name}}<br/>
    <div ng-controller="myController1">
        <input ng-model="person.name" />
        <input ng-model="name" />
    </div>
</div>
```

Remember: in Angular scopes can be created in many ways (such as built-in or custom directives, or the `$scope.$new()` function), and keeping track of the scope tree is probably impossible.

Using only non-primitive values as scope properties will keep you on the safe side (unless you need a property to not inherit, or other cases where you are aware of scope inheritance).

Section 22.3: Basic Example of $scope inheritance

```
angular.module('app', [])
.controller('myController', ['$scope', function($scope){
    $scope.person = { name: 'John Doe' };
}]);

<div ng-app="app" ng-conroller="myController">
    <input ng-model="person.name" />
    <div ng-repeat="number in [0,1,2,3]">
        {{person.name}} {{number}}
    </div>
</div>
```

In this example, the ng-repeat directive creates a new scope for each of its newly created children.

These created scopes are children of their parent scope (in this case the scope created by myController), and therfore, they inherit all of its proporties, such as person.

Section 22.4: How can you limit the scope on a directive and why would you do this?

Scope is used as the "glue" that we use to communicate between the parent controller, the directive, and the directive template. Whenever the AngularJS application is bootstrapped, a rootScope object is created. Each scope created by controllers, directives and services are prototypically inherited from rootScope.

Yes, we can limit the scope on a directive . We can do so by creating an isolated scope for directive.

> There are 3 types of directive scopes:
>
> 1. Scope : False (Directive uses its parent scope)

2. Scope : True (Directive gets a new scope)
3. Scope : { } (Directive gets a new isolated scope)

Directives with the new isolated scope: When we create a new isolated scope then it will not be inherited from the parent scope. This new scope is called Isolated scope because it is completely detached from its parent scope. Why? should we use isolated scope: We should use isolated scope when we want to create a custom directive because it will make sure that our directive is generic, and placed anywhere inside the application. Parent scope is not going to interfere with the directive scope.

Example of isolated scope:

```
var app = angular.module("test",[]);

app.controller("Ctrl1",function($scope){
    $scope.name = "Prateek";
    $scope.reverseName = function(){
        $scope.name = $scope.name.split('').reverse().join('');
    };
});
app.directive("myDirective", function(){
    return {
        restrict: "EA",
        scope: {},
        template: "<div>Your name is : {{name}}</div>"+
        "Change your name : <input type='text' ng-model='name'/>"
    };
});
```

There're 3 types of prefixes AngularJS provides for isolated scope these are :

1. "@" (Text binding / one-way binding)
2. "=" (Direct model binding / two-way binding)
3. "&" (Behaviour binding / Method binding)

All these prefixes receives data from the attributes of the directive element like :

```
<div my-directive
 class="directive"
 name="{{name}}"
 reverse="reverseName()"
 color="color" >
</div>
```

Section 22.5: Using $scope functions

While declaring a function in the $rootscope has it's advantages, we can also declare a $scope function any part of the code that is injected by the $scope service. Controller, for instance.

Controller

```
myApp.controller('myController', ['$scope', function($scope){
    $scope.myFunction = function () {
        alert("You are in myFunction!");
    };
}]);
```

Now you can call your function from the controller using:

```
$scope.myfunction();
```

Or via HTML that is under that specific controller:

```
<div ng-controller="myController">
    <button ng-click="myFunction()"> Click me! </button>
</div>
```

Directive

An angular directive is another place you can use your scope:

```
myApp.directive('triggerFunction', function() {
    return {
        scope: {
            triggerFunction: '&'
        },
        link: function(scope, element) {
            element.bind('mouseover', function() {
                scope.triggerFunction();
            });
        }
    };
});
```

And in your HTML code under the same controller:

```
<div ng-controller="myController">
    <button trigger-function="myFunction()"> Hover over me! </button>
</div>
```

Of course, you can use ngMouseover for the same thing, but what's special about directives is that you can customize them the way you want. And now you know how to use your $scope functions inside them, be creative!

Section 22.6: Creating custom $scope events

Like normal HTML elements, it is possible for $scopes to have their own events. $scope events can be subscribed to using the following manner:

```
 $scope.$on('my-event', function(event, args) {
    console.log(args); // { custom: 'data' }
});
```

If you need unregister an event listener, the **$on** function will return an unbinding function. To continue with the above example:

```
var unregisterMyEvent = $scope.$on('my-event', function(event, args) {
    console.log(args); // { custom: 'data' }
    unregisterMyEvent();
});
```

There are two ways of triggering your own custom $scope event **$broadcast** and **$emit**. To notify the parent(s) of a scope of a specific event, use **$emit**

```
$scope.$emit('my-event', { custom: 'data' });
```

The above example will trigger any event listeners for `my-event` on the parent scope and will continue up the scope tree to **$rootScope** unless a listener calls `stopPropagation` on the event. Only events triggered with **$emit** may call `stopPropagation`

The reverse of **$emit** is **$broadcast**, which will trigger any event listeners on all child scopes in the scope tree that are children of the scope that called **$broadcast**.

```
$scope.$broadcast('my-event', { custom: 'data' });
```

Events triggered with **$broadcast** cannot be canceled.

Chapter 23: AngularJS gotchas and traps

Section 23.1: Things to do when using html5Mode

When using `html5Mode([mode])` it is necessary that:

1. You specify the base URL for the application with a `<base href="">` in the head of your `index.html`.
2. It is important that the `base` tag comes before any tags with url requests. Otherwise, this might result in this error - `"Resource interpreted as stylesheet but transferred with MIME type text/html"`. For example:

```
<head>
    <meta charset="utf-8">
    <title>Job Seeker</title>

    <base href="/">

    <link rel="stylesheet" href="bower_components/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="/styles/main.css">
</head>
```

3. If you do no want to specify a `base` tag, configure `$locationProvider` to not require a `base` tag by passing a definition object with `requireBase:false` to `$locationProvider.html5Mode()` like this:

```
$locationProvider.html5Mode({
    enabled: true,
    requireBase: false
});
```

4. In order to support direct loading of HTML5 URLs, you need to enabler server-side URL rewriting. From AngularJS / Developer Guide / Using $location

> Using this mode requires URL rewriting on server side, basically you have to rewrite all your links to entry point of your application (e.g. `index.html`). Requiring a `<base>` tag is also important for this case, as it allows Angular to differentiate between the part of the url that is the application base and the path that should be handled by the application.

An excellent resource for request rewriting examples for various HTTP server implementations can be found in the ui-router FAQ - How to: Configure your server to work with html5Mode. For example, Apache

```
RewriteEngine on

# Don't rewrite files or directories
RewriteCond %{REQUEST_FILENAME} -f [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^ - [L]

# Rewrite everything else to index.html to allow html5 state links
RewriteRule ^ index.html [L]
```

nginx

```
server {
    server_name my-app;

    root /path/to/app;

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

Express

```
var express = require('express');
var app = express();

app.use('/js', express.static(__dirname + '/js'));
app.use('/dist', express.static(__dirname + '/../dist'));
app.use('/css', express.static(__dirname + '/css'));
app.use('/partials', express.static(__dirname + '/partials'));

app.all('/*', function(req, res, next) {
    // Just send the index.html for other files to support HTML5Mode
    res.sendFile('index.html', { root: __dirname });
});

app.listen(3006); //the port you want to use
```

Section 23.2: Two-way data binding stops working

One should have in mind that:

1. Angular's data binding relies on JavaScript's prototypal inheritance, thus it's subject to variable shadowing.
2. A child scope normally prototypically inherits from its parent scope. One exception to this rule is a directive which has an isolated scope as it doesn't prototypically inherit.
3. There are some directives which create a new child scope: `ng-repeat`, `ng-switch`, `ng-view`, `ng-if`, `ng-controller`, `ng-include`, etc.

This means that when you try to two-way bind some data to a primitive which is inside of a child scope (or vice-versa), things may not work as expected. Here's an example of how easily is to *"break"* AngularJS.

This issue can easily be avoided following these steps:

1. Have a "." inside your HTML template whenever you bind some data
2. Use `controllerAs` syntax as it promotes the use of binding to a "dotted" object
3. $parent can be used to access parent `scope` variables rather than child scope. like inside `ng-if` we can use `ng-model="$parent.foo"`..

An alternative for the above is to bind `ngModel` to a getter/setter function that will update the cached version of the model when called with arguments, or return it when called without arguments. In order to use a getter/setter function, you need to add `ng-model-options="{ getterSetter: true }"` to the element with the `ngModal` attribute, and to call the getter function if you want to display its value in expression (Working example).

Example

View:

```
<div ng-app="myApp" ng-controller="MainCtrl">
    <input type="text" ng-model="foo" ng-model-options="{ getterSetter: true }">
    <div ng-if="truthyValue">
        <!-- I'm a child scope (inside ng-if), but i'm synced with changes from the outside scope -->
        <input type="text" ng-model="foo">
    </div>
    <div>$scope.foo: {{ foo() }}</div>
</div>
```

Controller:

```
angular.module('myApp', []).controller('MainCtrl', ['$scope', function($scope) {
    $scope.truthyValue = true;

    var _foo = 'hello'; // this will be used to cache/represent the value of the 'foo' model

    $scope.foo = function(val) {
        // the function return the the internal '_foo' varibale when called with zero arguments,
        // and update the internal `_foo` when called with an argument
        return arguments.length ? (_foo = val) : _foo;
    };
}]);
```

Best Practice: It's best to keep getters fast because Angular is likely to call them more frequently than other parts of your code (reference).

Section 23.3: 7 Deadly Sins of AngularJS

Below is the list of some mistakes that developers often make during the use of AngularJS functionalities, some learned lessons and solutions to them.

1. Manipulating DOM through the controller

It's legal, but must be avoided. Controllers are the places where you define your dependencies, bind your data to the view and make further business logic. You can technically manipulate the DOM in a controller, but whenever you need same or similar manipulation in another part of your app, another controller will be needed. So the best practice of this approach is creating a directive that includes all manipulations and use the directive throughout your app. Hence, the controller leaves the view intact and does it's job. In a directive, linking function is the best place to manipulate the DOM. It has full access to the scope and element, so using a directive, you can also take the advantage of reusability.

```
link: function($scope, element, attrs) {
    //The best place to manipulate DOM
}
```

You can access DOM elements in linking function through several ways, such as the `element` parameter, `angular.element()` method, or pure Javascript.

2. Data binding in transclusion

AngularJS is famous with its two-way data binding. However you may encounter sometimes that your data is only one-way bound inside directives. Stop there, AngularJS is not wrong but probably you. Directives are a little dangerous places since child scopes and isolated scopes are involved. Assume you have the following directive with one transclusion

```
<my-dir>
```

```
    <my-transclusion>
    </my-transclusion>
</my-dir>
```

And inside my-transclusion, you have some elements which are bound to the data in the outer scope.

```
<my-dir>
    <my-transclusion>
        <input ng-model="name">
    </my-transclusion>
</my-dir>
```

The above code will not work correctly. Here, transclusion creates a child scope and you can get the name variable, right, but whatever change you make to this variable will stay there. So, you can truly acces this variable as **$parent.name**. However, this use might not be the best practice. A better approach would be wrapping the variables inside an object. For example, in the controller you can create:

```
$scope.data = {
    name: 'someName'
}
```

Then in the transclusion, you can access this variable via 'data' object and see that two-way binding works perfectly!

```
<input ng-model="data.name">
```

Not only in transclusions, but throughout the app, it's a good idea to use the dotted notation.

3. Multiple directives together

It is actually legal to use two directives together within the same element, as long as you obey by the rule: two isolated scopes cannot exist on the same element. Generally speaking, when creating a new custom directive, you allocate an isolated scope for easy parameter passing. Assuming that the directives myDirA and myDirB have isoleted scopes and myDirC has not, following element will be valid:

```
<input my-dir-a my-dirc>
```

whereas the following element will cause console error:

```
<input my-dir-a my-dir-b>
```

Therefore, directives must be used wisely, taking the scopes into consideration.

4. Misuse of $emit

$emit, $broadcast and $on, these work in a sender-receiver principle. In others words, they are a means of communication between controllers. For example, the following line emits the 'someEvent' from controller A, to be catched by the concerned controller B.

```
$scope.$emit('someEvent', args);
```

And the following line catches the 'someEvent'

```
$scope.$on('someEvent', function(){});
```

So far everything seems perfect. But remember that, if the controller B is not invoked yet, the event will not be

caught, which means both emitter and receiver controllers have to be invoked to get this working. So again, if you are not sure that you definitely have to use $emit, building a service seems a better way.

5. Misuse of $scope.$watch

$scope.$watch is used for watching a variable change. Whenever a variable has changed, this method is invoked. However, one common mistake done is changing the variable inside $scope.$watch. This will cause inconsistency and infinite $digest loop at some point.

```
$scope.$watch('myCtrl.myVariable', function(newVal) {
    this.myVariable++;
});
```

So in the above function, make sure you have no operations on myVariable and newVal.

6. Binding methods to views

This is one of the deadlisest sins. AngularJS has two-way binding, and whenever something changes, the views are updated many many times. So, if you bind a method to an attribute of a view, that method might potentially be called a hundred times, which also drives you crazy during debugging. However, there are only some attributes that are built for method binding, such as ng-click, ng-blur, ng-on-change, etc, that expect methods as paremeter. For instance, assume you have the following view in your markup:

```
<input ng-disabled="myCtrl.isDisabled()" ng-model="myCtrl.name">
```

Here you check the disabled status of the view via the method isDisabled. In the controller myCtrl, you have:

```
vm.isDisabled = function(){
    if(someCondition)
        return true;
    else
        return false;
}
```

In theory, it may seem correct but technically this will cause an overload, since the method will run countless times. In order to resolve this, you should bind a variable. In your controller, the following variable must exist:

```
vm.isDisabled
```

You can initiate this variable again in the activation of the controller

```
if(someCondition)
    vm.isDisabled = true
else
    vm.isDisabled = false
```

If the condition is not stable, you may bind this to another event. Then you should bind this variable to the view:

```
<input ng-disabled="myCtrl.isDisabled" ng-model="myCtrl.name">
```

Now, all the attributes of the view have what they expect and the methods will run only whenever needed.

7. Not using Angular's functionalities

AngularJS provides great convenience with some of its functionalities, not only simplifying your code but also

making it more efficient. Some of these features are listed below:

1. **angular.forEach** for the loops (Caution, you can't "break;" it, you can only prevent getting into the body, so consider performance here.)
2. **angular.element** for DOM selectors
3. **angular.copy**: Use this when you should not modify the main object
4. **Form validations** are already awesome. Use dirty, pristine, touched, valid, required and so on.
5. Besides Chrome debugger, use **remote debugging** for mobile development too.
6. And make sure you use **Batarang**. It's a free Chrome extension where you can easily inspect scopes
   .

Chapter 24: Using AngularJS with TypeScript

Section 24.1: Using Bundling / Minification

The way the $scope is injected in the controller's constructor functions is a way to demonstrate and use the basic option of angular dependency injection but is not production ready as it cannot be minified. Thats because the minification system changes the variable names and anguar's dependency injection uses the parameter names to know what has to be injected. So for an example the ExampleController's constructor function is minified to the following code.

```
function n(n){this.setUpWatches(n)
```

and `$scope` is changed to `n`!
to overcome this we can add an $inject array(`string[]`). So that angular's DI knows what to inject at what position is the controllers constructor function.
So the above typescript changes to

```
module App.Controllers {
    class Address {
        line1: string;
        line2: string;
        city: string;
        state: string;
    }
    export class SampleController {
        firstName: string;
        lastName: string;
        age: number;
        address: Address;
        setUpWatches($scope: ng.IScope): void {
            $scope.$watch(() => this.firstName, (n, o) => {
                //n is string and so is o
            });
        };
        static $inject : string[] = ['$scope'];
        constructor($scope: ng.IScope) {
            this.setUpWatches($scope);
        }
    }
}
```

Section 24.2: Angular Controllers in Typescript

As defined in the AngularJS Documentation

> When a Controller is attached to the DOM via the ng-controller directive, Angular will instantiate a new Controller object, using the specified Controller's constructor function. A new child scope will be created and made available as an injectable parameter to the Controller's constructor function as $scope.

Controllers can be very easily made using the typescript classes.

```
module App.Controllers {
```

```
    class Address {
        line1: string;
        line2: string;
        city: string;
        state: string;
    }
    export class SampleController {
        firstName: string;
        lastName: string;
        age: number;
        address: Address;
        setUpWatches($scope: ng.IScope): void {
            $scope.$watch(() => this.firstName, (n, o) => {
                //n is string and so is o
            });
        };
        constructor($scope: ng.IScope) {
            this.setUpWatches($scope);
        }
    }
}
```

The Resulting Javascript is

```
var App;
(function (App) {
    var Controllers;
    (function (Controllers) {
        var Address = (function () {
            function Address() {
            }
            return Address;
        }());
        var SampleController = (function () {
            function SampleController($scope) {
                this.setUpWatches($scope);
            }
            SampleController.prototype.setUpWatches = function ($scope) {
                var _this = this;
                $scope.$watch(function () { return _this.firstName; }, function (n, o) {
                    //n is string and so is o
                });
            };
            ;
            return SampleController;
        }());
        Controllers.SampleController = SampleController;
    })(Controllers = App.Controllers || (App.Controllers = {}));
})(App || (App = {}));
//# sourceMappingURL=ExampleController.js.map
```

After making the controller class let the angular js module about the controller can be done simple by using the class

```
app
 .module('app')
 .controller('exampleController', App.Controller.SampleController)
```

Section 24.3: Using the Controller with ControllerAs Syntax

The Controller we have made can be instantiated and used using `controller as` Syntax. That's because we have put variable directly on the controller class and not on the `$scope`.

Using `controller as someName` is to seperate the controller from `$scope` itself.So, there is no need of injecting $scope as the dependency in the controller.

Traditional way :

```
// we are using $scope object.
app.controller('MyCtrl', function ($scope) {
  $scope.name = 'John';
});

<div ng-controller="MyCtrl">
  {{name}}
</div>
```

Now, with `controller as` Syntax :

```
// we are using the "this" Object instead of "$scope"
app.controller('MyCtrl', function() {
  this.name = 'John';
});

<div ng-controller="MyCtrl as info">
  {{info.name}}
</div>
```

If you instantiate a "class" in JavaScript, you might do this :

```
var jsClass = function () {
  this.name = 'John';
}
var jsObj = new jsClass();
```

So, now we can use **`jsObj`** instance to access any method or property of **`jsClass`**.

In angular, we do same type of thing.we use controller as syntax for instantiation.

Section 24.4: Why ControllerAs Syntax?

Controller Function

Controller function is nothing but just a JavaScript constructor function. Hence, when a view loads the **`function`** `context(`**`this`**`)` is set to the controller object.

Case 1 :

```
this.constFunction = function() { ... }
```

It is created in the `controller object`, not on `$scope`. views can not access the functions defined on controller object.

Example :

```
<a href="#123" ng-click="constFunction()"></a> // It will not work
```

Case 2 :

```
$scope.scopeFunction = function() { ... }
```

It is created in the `$scope object`, not on `controller object`. views can only access the functions defined on `$scope` object.

Example :

```
<a href="#123" ng-click="scopeFunction()"></a> // It will work
```

Why ControllerAs ?

- **`ControllerAs`** syntax makes it much clearer where objects are being manipulated.Having `oneCtrl.name` and `anotherCtrl.name` makes it much easier to identify that you have an `name` assigned by multiple different controllers for different purposes but if both used same `$scope.name` and having two different HTML elements on a page which both are bound to **`{{name}}`** then it is difficult to identify which one is from which controller.
- Hiding the `$scope` and exposing the members from the controller to the view via an `intermediary object`. By setting **`this`**`.*`, we can expose just what we want to expose from the controller to the view.

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

Here, in above case `{{ name }}` will be very confusing to use and We also don't know which one related to which controller.

```
<div ng-controller="FirstCtrl as first">
    {{ first.name }}
    <div ng-controller="SecondCtrl as second">
        {{ second.name }}
        <div ng-controller="ThirdCtrl as third">
            {{ third.name }}
        </div>
    </div>
</div>
```

Why $scope ?

- Use `$scope` when you need to access one or more methods of $scope such as `$watch`, `$digest`, `$emit`, `$http` etc.
- limit which properties and/or methods are exposed to `$scope`, then explicitly passing them to `$scope` as needed.

Chapter 25: $http request

Section 25.1: Timing of an $http request

The $http requests require time which varies depending on the server, some may take a few milliseconds, and some may take up to a few seconds. Often the time required to retrieve the data from a request is critical. Assuming the response value is an array of names, consider the following example:

Incorrect

```
$scope.names = [];

$http({
    method: 'GET',
    url: '/someURL'
}).then(function successCallback(response) {
            $scope.names = response.data;
        },
        function errorCallback(response) {
            alert(response.status);
        });

alert("The first name is: " + $scope.names[0]);
```

Accessing `$scope.names[0]` right below the $http request will often throw an error - this line of code executes before the response is received from the server.

Correct

```
$scope.names = [];

$scope.$watch('names', function(newVal, oldVal) {
    if(!(newVal.length == 0)) {
        alert("The first name is: " + $scope.names[0]);
    }
});

$http({
    method: 'GET',
    url: '/someURL'
}).then(function successCallback(response) {
            $scope.names = response.data;
        },
        function errorCallback(response) {
            alert(response.status);
        });
```

Using the $watch service we access the `$scope.names` array only when the response is received. During initialization, the function is called even though `$scope.names` was initialized before, therefore checking if the `newVal.length` is different than 0 is necessary. Be aware - any changes made to `$scope.names` will trigger the watch function.

Section 25.2: Using $http inside a controller

The `$http` service is a function which generates an HTTP request and returns a promise.

General Usage

```
// Simple GET request example:
$http({
  method: 'GET',
  url: '/someUrl'
}).then(function successCallback(response) {
    // this callback will be called asynchronously
    // when the response is available
  }, function errorCallback(response) {
    // called asynchronously if an error occurs
    // or server returns response with an error status.
  });
```

Usage inside controller

```
appName.controller('controllerName',
    ['$http', function($http){

    // Simple GET request example:
    $http({
        method: 'GET',
        url: '/someUrl'
    }).then(function successCallback(response) {
        // this callback will be called asynchronously
        // when the response is available
    }, function errorCallback(response) {
        // called asynchronously if an error occurs
        // or server returns response with an error status.
    });
}])
```

Shortcut Methods

`$http` service also has shortcut methods. Read about http methods here

Syntax

```
$http.get('/someUrl', config).then(successCallback, errorCallback);
$http.post('/someUrl', data, config).then(successCallback, errorCallback);
```

Shortcut Methods

- $http.get
- $http.head
- $http.post
- $http.put
- $http.delete
- $http.jsonp
- $http.patch

Section 25.3: Using $http request in a service

HTTP requests are widely used repeatedly across every web app, so it is wise to write a method for each common request, and then use it in multiple places throughout the app.

Create a `httpRequestsService.js`

httpRequestsService.js

```
appName.service('httpRequestsService', function($q, $http){

    return {
        // function that performs a basic get request
        getName: function(){
            // make sure $http is injected
            return $http.get("/someAPI/names")
                .then(function(response) {
                    // return the result as a promise
                    return response;
                }, function(response) {
                    // defer the promise
                    return $q.reject(response.data);
                });
        },

        // add functions for other requests made by your app
        addName: function(){
            // some code...
        }
    }
})
```

The service above will perform a get request inside the service. This will be available to any controller where the service has been injected.

Sample usage

```
appName.controller('controllerName',
    ['httpRequestsService', function(httpRequestsService){

        // we injected httpRequestsService service on this controller
        // that made the getName() function available to use.
        httpRequestsService.getName()
            .then(function(response){
                // success
            }, function(error){
                // do something with the error
            })
    }])
```

Using this approach we can now use **httpRequestsService.js** anytime and in any controller.

Chapter 26: Constants

Section 26.1: Create your first constant

```
angular
  .module('MyApp', [])
  .constant('VERSION', 1.0);
```

Your constant is now declared and can be injected in a controller, a service, a factory, a provider, and even in a config method:

```
angular
  .module('MyApp')
  .controller('FooterController', function(VERSION) {
    this.version = VERSION;
  });
```

```
<footer ng-controller="FooterController as Footer">{{ Footer.version }}</footer>
```

Section 26.2: Use cases

There is no revolution here, but angular constant can be useful specially when your application and/or team starts to grow ... or if you simply love writing beautiful code!

- **Refactor code.** Example with event's names. If you use a lot of events in your application, you have event's names a little every where. A when a new developper join your team, he names his events with a different syntax, ... You can easily prevent this by grouping your event's names in a constant:

```
angular
  .module('MyApp')
  .constant('EVENTS', {
    LOGIN_VALIDATE_FORM: 'login::click-validate',
    LOGIN_FORGOT_PASSWORD: 'login::click-forgot',
    LOGIN_ERROR: 'login::notify-error',
    ...
  });
```

```
angular
  .module('MyApp')
  .controller('LoginController', function($scope, EVENT) {
    $scope.$on(EVENT.LOGIN_VALIDATE_FORM, function() {
      ...
    });
  })
```

... and now, your event's names can take benefits from autocompletion !

- **Define configuration.** Locate all your configuration in a same place:

```
angular
  .module('MyApp')
  .constant('CONFIG', {
    BASE_URL: {
      APP: 'http://localhost:3000',
      API: 'http://localhost:3001'
    },
```

```
    STORAGE: 'S3',
    ...
  });
```

- **Isolate parts.** Sometimes, there are some things you are not very proud of ... like hardcoded value for example. Instead of let them in your main code, you can create an angular constant

```
angular
  .module('MyApp')
  .constant('HARDCODED', {
    KEY: 'KEY',
    RELATION: 'has_many',
    VAT: 19.6
  });
```

... and refactor something like

```
$scope.settings = {
  username: Profile.username,
  relation: 'has_many',
  vat: 19.6
}
```

to

```
$scope.settings = {
  username: Profile.username,
  relation: HARDCODED.RELATION,
  vat: HARDCODED.VAT
}
```

Chapter 27: Form Validation

Section 27.1: Form and Input States

Angular Forms and Inputs have various states that are useful when validating content

Input States

State	Description
`$touched`	Field has been touched
`$untouched`	Field has not been touched
`$pristine`	Field has not been modified
`$dirty`	Field has been modified
`$valid`	Field content is valid
`$invalid`	Field content is invalid

All of the above states are boolean properties and can be either true or false.

With these, it is very easy to display messages to a user.

```
<form name="myForm" novalidate>
    <input name="myName" ng-model="myName" required>
    <span ng-show="myForm.myName.$touched && myForm.myName.$invalid">This name is invalid</span>
</form>
```

Here, we are using the `ng-show` directive to display a message to a user if they've modified a form but it's invalid.

Section 27.2: CSS Classes

Angular also provides some CSS classes for forms and inputs depending on their state

Class	Description
`ng-touched`	Field has been touched
`ng-untouched`	Field has not been touched
`ng-pristine`	Field has not been modified
`ng-dirty`	Field has been modified
`ng-valid`	Field is valid
`ng-invalid`	Field is invalid

You can use these classes to add styles to your forms

```
input.ng-invalid {
  background-color: crimson;
}
input.ng-valid {
  background-color: green;
}
```

Section 27.3: Basic Form Validation

One of Angular's strength's is client-side form validation.

Dealing with traditional form inputs and having to use interrogative jQuery-style processing can be time-consuming and finicky. Angular allows you to produce professional *interactive* forms relatively easily.

The **ng-model** directive provides two-way binding with input fields and usually the **novalidate** attribute is also placed on the form element to prevent the browser from doing native validation.

Thus, a simple form would look like:

```
<form name="form" novalidate>
  <label name="email"> Your email </label>
  <input type="email" name="email" ng-model="email" />
</form>
```

For Angular to validate inputs, use exactly the same syntax as a regular *input* element, except for the addition of the **ng-model** attribute to specify which variable to bind to on the scope. Email is shown in the prior example. To validate a number, the syntax would be:

```
<input type="number" name="postalcode" ng-model="zipcode" />
```

The final steps to basic form validation are connecting to a form submit function on the controller using **ng-submit**, rather than allowing the default form submit to occur. This is not mandatory but it is usually used, as the input variables are already available on the scope and so available to your submit function. It is also usually good practice to give the form a name. These changes would result in the following syntax:

```
<form name="signup_form" ng-submit="submitFunc()" novalidate>
  <label name="email"> Your email </label>
  <input type="email" name="email" ng-model="email" />
  <button type="submit">Signup</button>
</form>
```

This above code is functional but there is other functionality that Angular provides.

The next step is to understand that Angular attaches class attributes using **ng-pristine**, **ng-dirty**, **ng-valid** and **ng-invalid** for form processing. Using these classes in your css will allow you to style **valid/invalid** and **pristine/dirty** input fields and so alter the presentation as the user is entering data into the form.

Section 27.4: Custom Form Validation

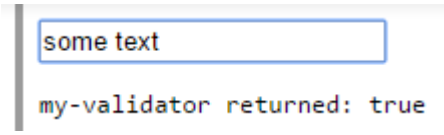
In some cases basic validation is not enough. Angular support custom validation adding validator functions to the `$validators` object on the `ngModelController`:

```
angular.module('app', [])
  .directive('myValidator', function() {
    return {
      // element must have ng-model attribute
      // or $validators does not work
      require: 'ngModel',
      link: function(scope, elm, attrs, ctrl) {
        ctrl.$validators.myValidator = function(modelValue, viewValue) {
            // validate viewValue with your custom logic
            var valid = (viewValue && viewValue.length > 0) || false;
            return valid;
        };
      }
    };
```

The validator is defined as a directive that require `ngModel`, so to apply the validator just add the custom directive to the input form control.

```
<form name="form">
  <input type="text"
         ng-model="model"
         name="model"
         my-validator>
  <pre ng-bind="'my-validator returned: ' + form.model.$valid"></pre>
</form>
```

And `my-validator` doesn't have to be applied on native form control. It can be any elements, as long as it as `ng-model` in its attributes. This is useful when you have some custom build ui component.



Section 27.5: Async validators

Asynchronous validators allows you to validate form information against your backend (using $http).

These kind of validators are needed when you need to access server stored information you can't have on your client for various reasons, such as the users table and other database information.

To use async validators, you access the `ng-model` of your `input` and define callback functions for the `$asyncValidators` property.

Example:

The following example checks if a provided name already exists, the backend will return a status that will reject the promise if the name already exists or if it wasn't provided. If the name doesn't exist it will return a resolved promise.

```
ngModel.$asyncValidators.usernameValidate = function (name) {
     if (name) {
          return AuthenticationService.checkIfNameExists(name); // returns a promise
     } else {
          return $q.reject("This username is already taken!"); // rejected promise
     }
};
```

Now every time the `ng-model` of the input is changed, this function will run and return a promise with the result.

Section 27.6: ngMessages

`ngMessages` is used to enhanced the style for displaying validation messages in the view.

Traditional approach

Before `ngMessages`, we normally display the validation messages using Angular pre-defined directives `ng-class`.This approach was litter and a `repetitive` task.

Now, by using `ngMessages` we can create our own custom messages.

Example

Html :

```
<form name="ngMessagesDemo">
  <input name="firstname" type="text" ng-model="firstname" required>
  <div ng-messages="ngMessagesDemo.firstname.$error">
    <div ng-message="required">Firstname is required.</div>
  </div>
</form>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.3.16/angular.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.3.16/angular-
messages.min.js"></script>
```

JS :

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

app.controller('mainCtrl', function ($scope) {
    $scope.firstname = "Rohit";
});
```

Section 27.7: Nested Forms

Sometimes it is desirable to nest forms for the purpose of grouping controls and inputs logically on the page. However, HTML5 forms should not be nested. Angular supplies `ng-form` instead.

```
<form name="myForm" noValidate>
  <!-- nested form can be referenced via 'myForm.myNestedForm' -->
  <ng-form name="myNestedForm" noValidate>
    <input name="myInput1" ng-minlength="1" ng-model="input1" required />
    <input name="myInput2" ng-minlength="1" ng-model="input2" required />
  </ng-form>

  <!-- show errors for the nested subform here -->
  <div ng-messages="myForm.myNestedForm.$error">
    <!-- note that this will show if either input does not meet the minimum -->
    <div ng-message="minlength">Length is not at least 1</div>
  </div>
</form>

<!-- status of the form -->
<p>Has any field on my form been edited? {{myForm.$dirty}}</p>
<p>Is my nested form valid? {{myForm.myNestedForm.$valid}}</p>
<p>Is myInput1 valid? {{myForm.myNestedForm.myInput1.$valid}}</p>
```

Each part of the form contributes to the overall form's state. Therefore, if one of the inputs `myInput1` has been edited and is `$dirty`, its containing form will also be `$dirty`. This cascades to each containing form, so both `myNestedForm` and `myForm` will be `$dirty`.

Chapter 28: Angular promises with $q service

Section 28.1: Wrap simple value into a promise using $q.when()

If all you need is to wrap the value into a promise, you don't need to use the long syntax like here:

```
//OVERLY VERBOSE
var defer;
defer = $q.defer();
defer.resolve(['one', 'two']);
return defer.promise;
```

In this case you can just write:

```
//BETTER
return $q.when(['one', 'two']);
```

> **$q.when and its alias $q.resolve**
>
> Wraps an object that might be a value or a (3rd party) then-able promise into a $q promise. This is useful when you are dealing with an object that might or might not be a promise, or if the promise comes from a source that can't be trusted.
>
> — AngularJS $q Service API Reference - $q.when

With the release of AngularJS v1.4.1

You can also use an ES6-consistent alias `resolve`

```
//ABSOLUTELY THE SAME AS when
return $q.resolve(['one', 'two'])
```

Section 28.2: Using angular promises with $q service

`$q` is a built-in service which helps in executing asynchronous functions and using their return values(or exception) when they are finished with processing.

`$q` is integrated with the `$rootScope.Scope` model observation mechanism, which means faster propagation of resolution or rejection into your models and avoiding unnecessary browser repaints, which would result in flickering UI.

In our example, we call our factory `getMyData`, which return a promise object. If the object is `resolved`, it returns a random number. If it is `rejected`, it return a rejection with an error message after 2 seconds.

In Angular factory

```
function getMyData($timeout, $q) {
  return function() {
    // simulated async function
    var promise = $timeout(function() {
      if(Math.round(Math.random())) {
```

```
        return 'data received!'
      } else {
        return $q.reject('oh no an error! try again')
      }
    }, 2000);
    return promise;
  }
}
```

Using Promises on call

```
angular.module('app', [])
.factory('getMyData', getMyData)
.run(function(getData) {
  var promise = getData()
    .then(function(string) {
      console.log(string)
    }, function(error) {
      console.error(error)
    })
    .finally(function() {
      console.log('Finished at:', new Date())
    })
})
```

To use promises, inject `$q` as dependency. Here we injected `$q` in `getMyData` factory.

```
var defer = $q.defer();
```

A new instance of deferred is constructed by calling `$q.defer()`

A deferred object is simply an object that exposes a promise as well as the associated methods for resolving that promise. It is constructed using the `$q.deferred()` function and exposes three main methods: `resolve()`, `reject()`, and `notify()`.

- `resolve(value)` – resolves the derived promise with the value.
- `reject(reason)` – rejects the derived promise with the reason.
- `notify(value)` - provides updates on the status of the promise's execution. This may be called multiple times before the promise is either resolved or rejected.

Properties

The associated promise object is accessed via the promise property. `promise` – {Promise} – promise object associated with this deferred.

A new promise instance is created when a deferred instance is created and can be retrieved by calling `deferred.promise`.

The purpose of the `promise` object is to allow for interested parties to get access to the result of the deferred task when it completes.

Promise Methods -

- `then(successCallback, [errorCallback], [notifyCallback])` – Regardless of when the promise was or will be resolved or rejected, then calls one of the success or error callbacks asynchronously as soon as the result is available. The callbacks are called with a single argument: the result or rejection reason. Additionally, the notify callback may be called zero or more times to provide a progress indication, before the promise is resolved or rejected.

- `catch(errorCallback)` – shorthand for promise.then(null, errorCallback)
- `finally(callback, notifyCallback)` – allows you to observe either the fulfillment or rejection of a promise, but to do so without modifying the final value.

One of the most powerful features of promises is the ability to chain them together. This allows the data to flow through the chain and be manipulated and mutated at each step. This is demonstrated with the following example:

Example 1:

```
// Creates a promise that when resolved, returns 4.
function getNumbers() {

  var promise = $timeout(function() {
    return 4;
  }, 1000);

  return promise;
}

// Resolve getNumbers() and chain subsequent then() calls to decrement
// initial number from 4 to 0 and then output a string.
getNumbers()
  .then(function(num) {
      // 4
      console.log(num);
      return --num;
  })
  .then(function (num) {
      // 3
      console.log(num);
      return --num;
  })
   .then(function (num) {
      // 2
      console.log(num);
      return --num;
  })
  .then(function (num) {
      // 1
      console.log(num);
      return --num;
  })
  .then(function (num) {
      // 0
      console.log(num);
      return 'And we are done!';
  })
  .then(function (text) {
      // "And we are done!"
      console.log(text);
  });
```

Section 28.3: Using the $q constructor to create promises

The `$q` constructor function is used to create promises from asynchronous APIs that use callbacks to return results.

> $q(function(resolve, reject) {...})

The constructor function receives a function that is invoked with two arguments, `resolve` and `reject` that are functions which are used to either resolve or reject the promise.

Example 1:

```
function $timeout(fn, delay) {
    return = $q(function(resolve, reject) {
         setTimeout(function() {
              try {
                 let r = fn();
                 resolve(r);
              }
              catch (e) {
                 reject(e);
              }
         }, delay);
     };
}
```

The above example creates a promise from the WindowTimers.setTimeout API. The AngularJS framework provides a more elaborate version of this function. For usage, see the AngularJS $timeout Service API Reference.

Example 2:

```
$scope.divide = function(a, b) {
    return $q(function(resolve, reject) {
      if (b===0) {
        return reject("Cannot devide by 0")
      } else {
        return resolve(a/b);
      }
    });
}
```

The above code showing a promisified division function, it will return a promise with the result or reject with a reason if the calculation is impossible.

You can then call and use `.then`

```
$scope.divide(7, 2).then(function(result) {
    // will return 3.5
}, function(err) {
    // will not run
})

$scope.divide(2, 0).then(function(result) {
    // will not run as the calculation will fail on a divide by 0
}, function(err) {
    // will return the error string.
})
```

Section 28.4: Avoid the $q Deferred Anti-Pattern

> **Avoid this Anti-Pattern**
>
> ```
> var myDeferred = $q.defer();
>
> $http(config).then(function(res) {
> ```

```
    myDeferred.resolve(res);
  }, function(error) {
    myDeferred.reject(error);
  });

  return myDeferred.promise;
```

There is no need to manufacture a promise with `$q.defer` as the $http service already returns a promise.

```
//INSTEAD
return $http(config);
```

Simply return the promise created by the $http service.

Section 28.5: Using $q.all to handle multiple promises

You can use the `$q.all` function to call a `.then` method after an array of promises has been successfully resolved and fetch the data they resolved with.

Example:

JS:

```
$scope.data = []

$q.all([
   $http.get("data.json"),
   $http.get("more-data.json"),
   ]).then(function(responses) {
     $scope.data = responses.map((resp) => resp.data);
   });
```

The above code runs `$http.get` 2 times for data in local json files, when both `get` method complete they resolve their associated promises, when all the promises in the array are resolved, the `.then` method starts with both promises data inside the `responses` array argument.

The data is then mapped so it could be shown on the template, we can then show

HTML:

```
<ul>
   <li ng-repeat="d in data">
      <ul>
         <li ng-repeat="item in d">{{item.name}}: {{item.occupation}}</li>
      </ul>
   </li>
</ul>
```

JSON:

```
[{
  "name": "alice",
  "occupation": "manager"
}, {
  "name": "bob",
  "occupation": "developer"
}]
```

Section 28.6: Deferring operations using $q.defer

We can use `$q` to defer operations to the future while having a pending promise object at the present, by using `$q.defer` we create a promise that will either resolve or reject in the future.

This method is not equivalent of using the `$q` constructor, as we use `$q.defer` to promisify an existing routine that may or may not return (or had ever returned) a promise at all.

Example:

```
var runAnimation = function(animation, duration) {
    var deferred = $q.defer();
    try {
        ...
        // run some animation for a given duration
        deferred.resolve("done");
    } catch (err) {
        // in case of error we would want to run the error hander of .then
        deferred.reject(err);
    }
    return deferred.promise;
}

// and then
runAnimation.then(function(status) {}, function(error) {})
```

1. Be sure you always return a the `deferred.promise` object or risk an error when invoking `.then`
2. Make sure you always resolve or reject your deferred object or `.then` may not run and you risk a memory leak

Chapter 29: Prepare for Production - Grunt

Section 29.1: View preloading

When the first time view is requested, normally Angular makes `XHR` request to get that view. For mid-size projects, the view count can be significant and it can slow down the application responsiveness.

The **good practice is to pre-load** all the views at once for small and mid size projects. For larger projects it is good to aggregate them in some meaningful bulks as well, but some other methods can be handy to split the load. To automate this task it is handy to use Grunt or Gulp tasks.

To pre-load the views, we can use `$templateCache` object. That is an object, where angular stores every received view from the server.

It is possible to use `html2js` module, that will convert all our views to one module - js file. Then we will need to inject that module into our application and that's it.

To create concatenated file of all the views we can use this task

```
module.exports = function (grunt) {
 //set up the location of your views here
 var viewLocation = ['app/views/**.html'];

 grunt.initConfig({
        pkg: require('./package.json'),
            //section that sets up the settings for concatenation of the html files into one file
            html2js: {
                        options: {
                            base: '',
                            module: 'app.templates', //new module name
                            singleModule: true,
                            useStrict: true,
                            htmlmin: {
                                collapseBooleanAttributes: true,
                                collapseWhitespace: true
                            }
                        },
                        main: {
                            src: viewLocation,
                            dest: 'build/app.templates.js'
                        }
                    },
            //this section is watching for changes in view files, and if there was a change, it will
regenerate the production file. This task can be handy during development.
            watch: {
                views:{
                    files: viewLocation,
                    tasks: ['buildHTML']
                },
            }
        });

        //to automatically generate one view file
        grunt.loadNpmTasks('grunt-html2js');

        //to watch for changes and if the file has been changed, regenerate the file
        grunt.loadNpmTasks('grunt-contrib-watch');
```

```
        //just a task with friendly name to reference in watch
        grunt.registerTask('buildHTML', ['html2js']);
};
```

To use this way of concatination, you need to make 2 changes: In your `index.html` file you need to reference the concatenated view file

```
<script src="build/app.templates.js"></script>
```

In the file, where you are declaring your app, you need to inject the dependency

```
angular.module('app', ['app.templates'])
```

If you are using popular routers like `ui-router`, there are no changes in the way, how you are referencing templates

```
    .state('home', {
        url: '/home',
        views: {
            "@": {
                controller: 'homeController',
                //this will be picked up from $templateCache
                templateUrl: 'app/views/home.html'
            },
        }

    })
```

Section 29.2: Script optimisation

It is **good practice to combine JS files together** and minify them. For larger project there could be hundreds of JS files and it adds unnecessary latency to load each file separately from the server.

For angular minification it is required to to have all functions annotated. That in necessary for Angular dependency injection proper minificaiton. (During minification, function names and variables will be renamed and it will break dependency injection if no extra actions will be taken.)

During minificaiton `$scope` and `myService` variables will be replaced by some other values. Angular dependency injection works based on the name, as a result, these names shouldn't change

```
.controller('myController', function($scope, myService){
})
```

Angular will understand the array notation, because minification won't replace string literals.

```
.controller('myController', ['$scope','myService', function($scope, myService){
}])
```

- Firstly we will concatinate all files end to end.
- Secondly we will use `ng-annotate` module, that will prepare code for minification
- Finally we will apply `uglify` module.

module.exports = function (grunt) { //set up the location of your scripts here for reusing it in code var scriptLocation = ['app/scripts/*.js'];

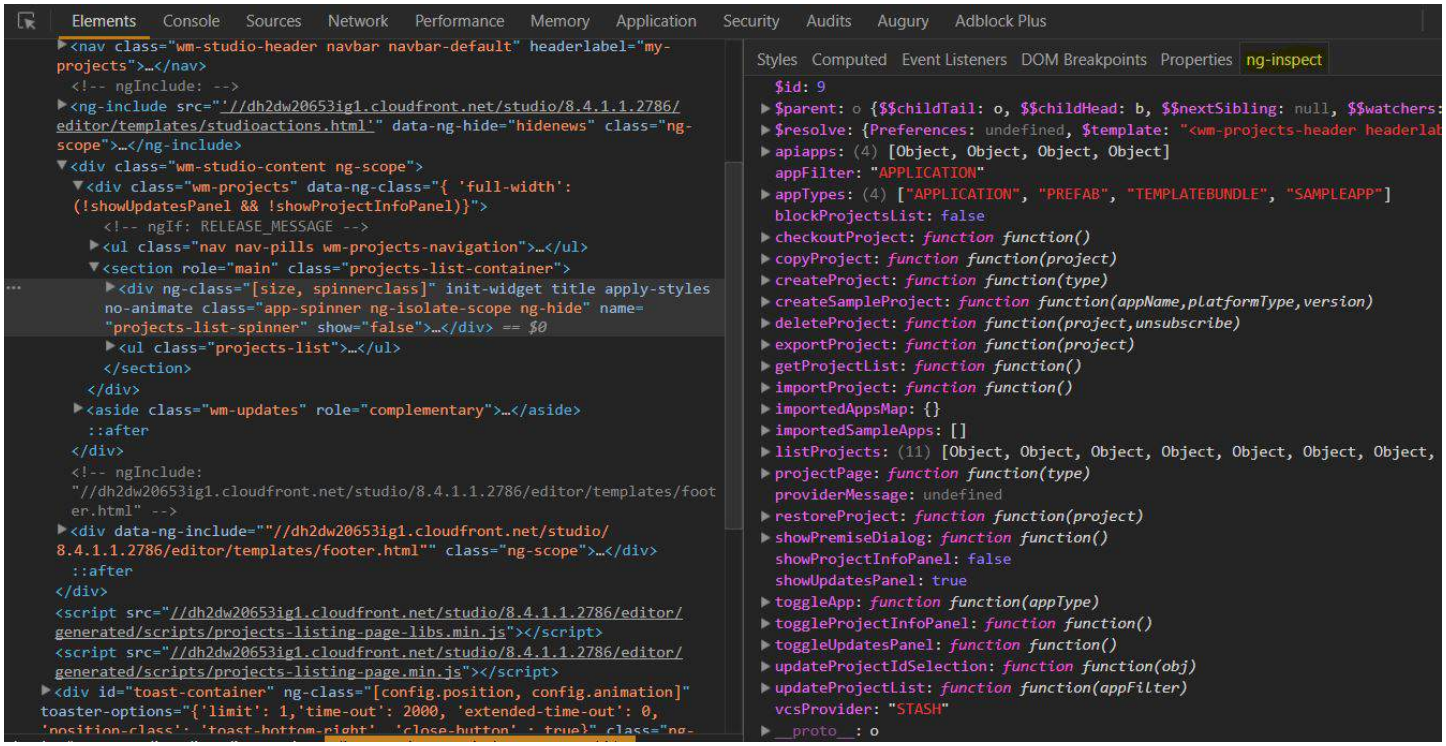
```
 grunt.initConfig({
        pkg: require('./package.json'),
            //add necessary annotations for safe minification
         ngAnnotate: {
            angular: {
                src: ['staging/concatenated.js'],
                dest: 'staging/anotated.js'
            }
        },
        //combines all the files into one file
        concat: {
                js: {
                    src: scriptLocation,
                    dest: 'staging/concatenated.js'
                }
            },
        //final uglifying
        uglify: {
            options: {
                report: 'min',
                mangle: false,
                sourceMap:true
            },
            my_target: {
                files: {
                    'build/app.min.js': ['staging/anotated.js']
                }
            }
        },

        //this section is watching for changes in JS files, and if there was a change, it will
regenerate the production file. You can choose not to do it, but I like to keep concatenated version
up to date
        watch: {
            scripts: {
                files: scriptLocation,
                tasks: ['buildJS']
            }
        }

});

    //module to make files less readable
    grunt.loadNpmTasks('grunt-contrib-uglify');

    //mdule to concatenate files together
    grunt.loadNpmTasks('grunt-contrib-concat');

    //module to make angularJS files ready for minification
    grunt.loadNpmTasks('grunt-ng-annotate');

    //to watch for changes and if the file has been changed, regenerate the file
    grunt.loadNpmTasks('grunt-contrib-watch');

    //task that sequentially executes all steps to prepare JS file for production
    //concatinate all JS files
    //annotate JS file (prepare for minification
    //uglify file
     grunt.registerTask('buildJS', ['concat:js', 'ngAnnotate', 'uglify']);
};
```

Chapter 30: Debugging

Section 30.1: Using ng-inspect chrome extension

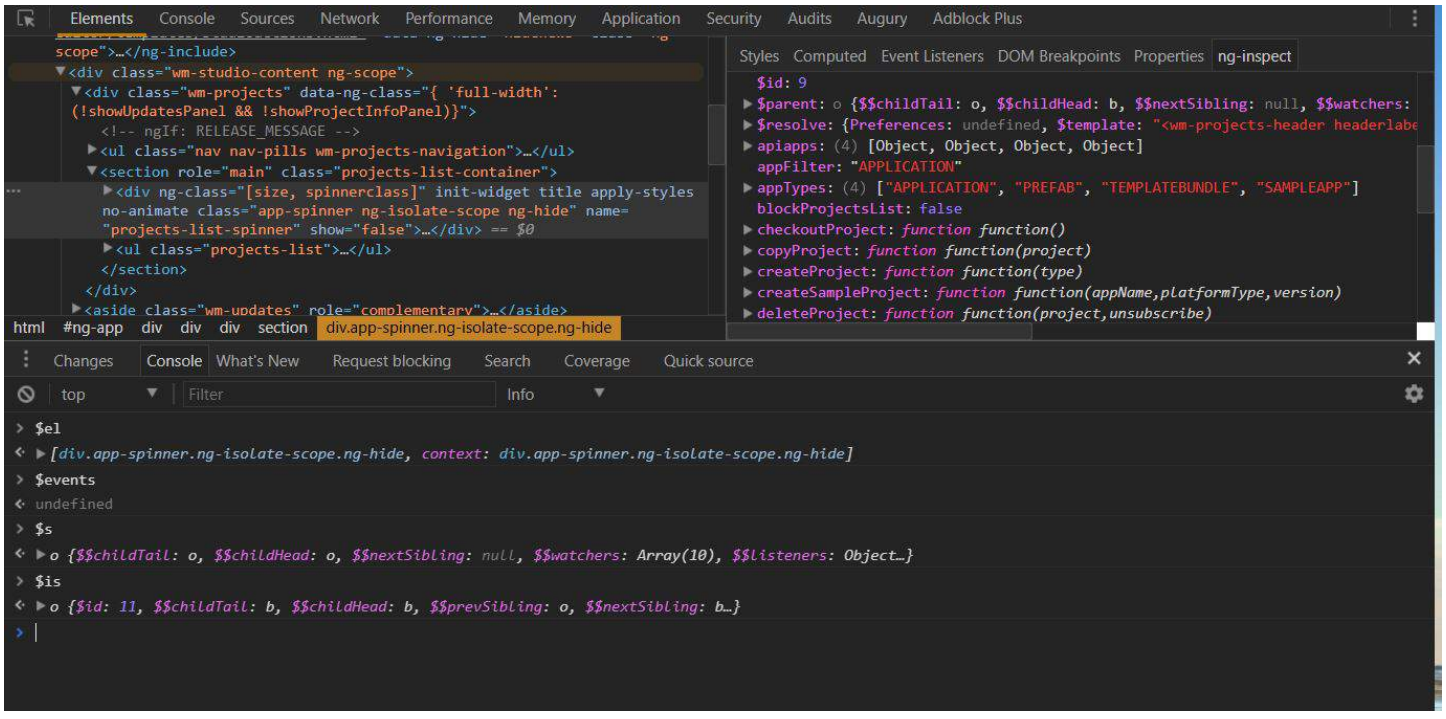
ng-inspect is a light weight Chrome extension for debugging AngularJS applications.

When a node is selected from the elements panel, the scope related info is displayed in the ng-inspect panel.



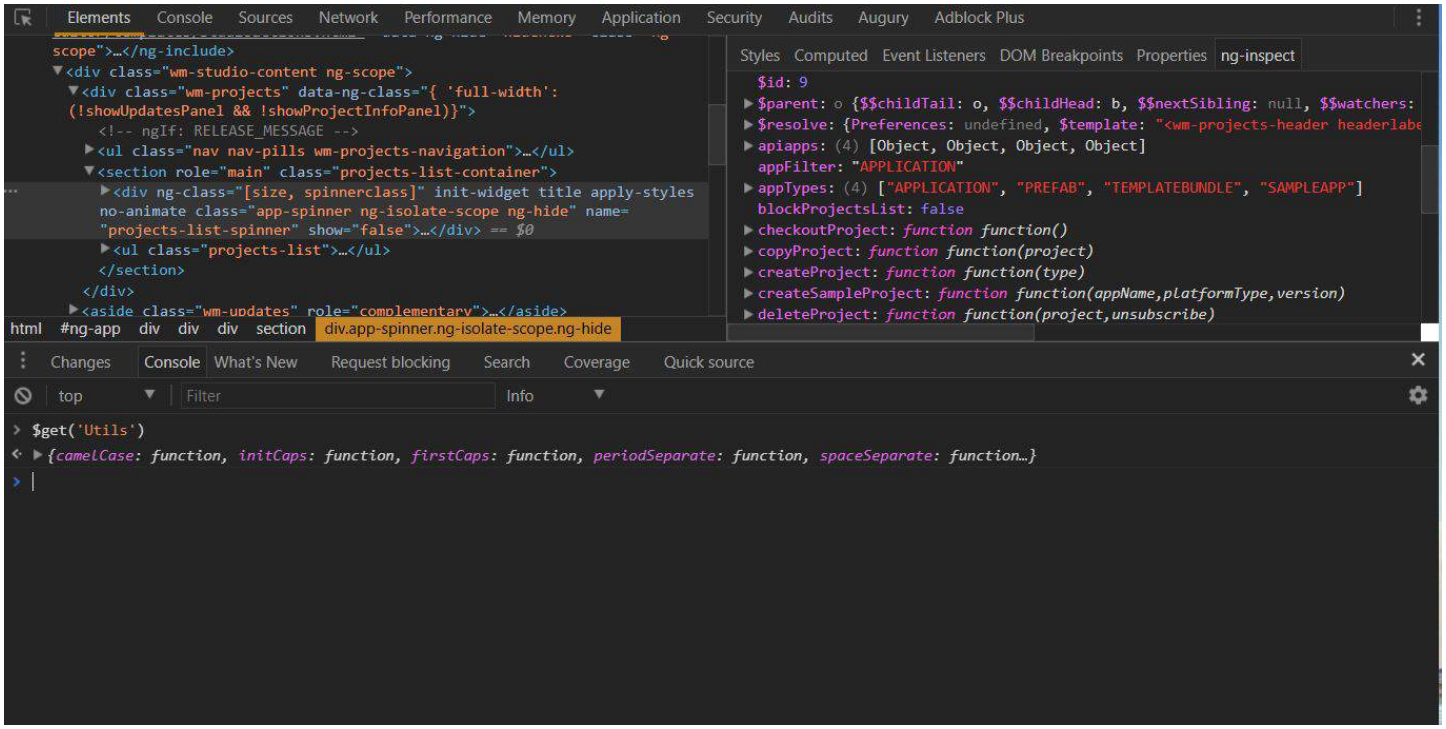
Exposes few global variables for quick access of scope/isolateScope.

```
$s      -- scope of the selected node
$is     -- isolateScope of the selected node
$el     -- jQuery element reference of the selected node (requiers jQuery)
$events -- events present on the selected node (requires jQuery)
```



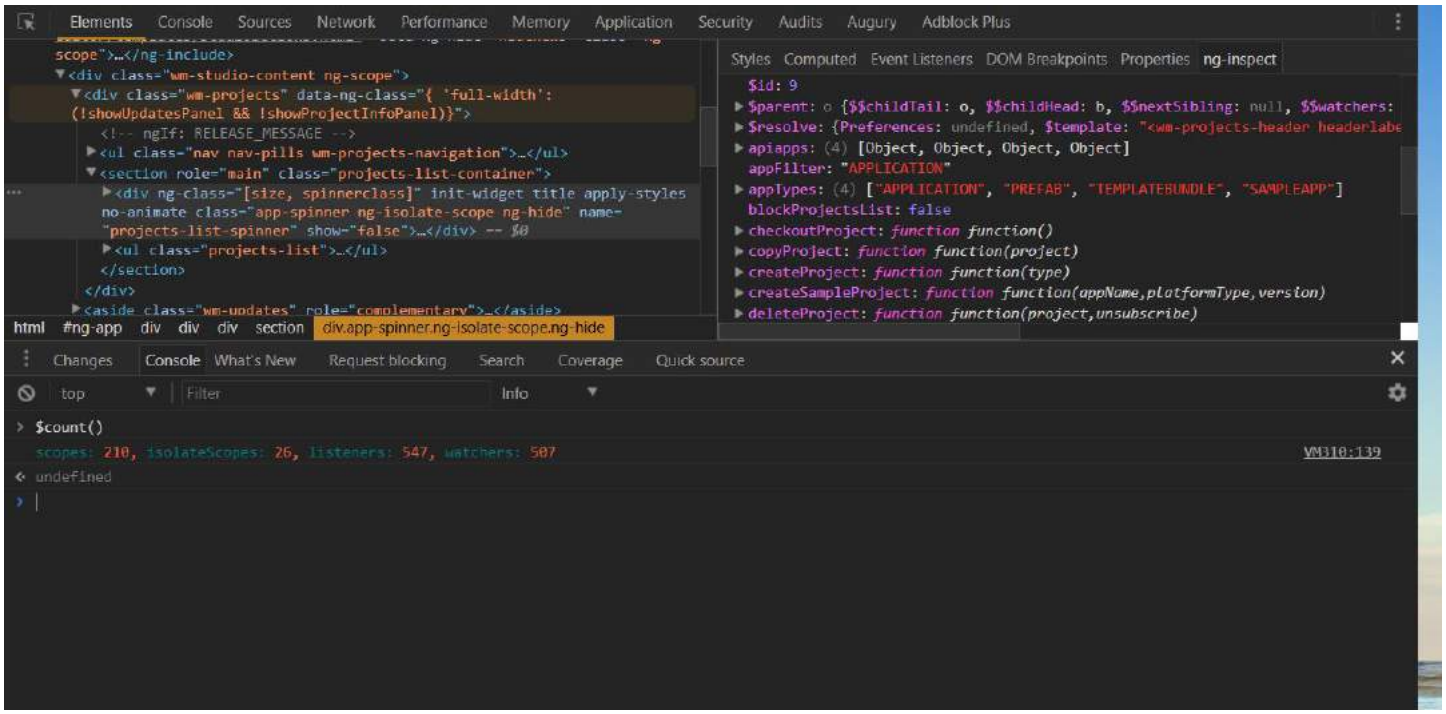
Provides easy access to Services/Factories.

Use `$get()` to retrieve the instance of a service/factory by name.



Performance of the application can be monitored by counting the no.of scopes,isolateScopes, watchers and listeners on the application.

Use `$count()` to get the count of scopes, isolateScopes, watchers and listeners.



> Note: This extension will work only when the debugInfo is enabled.

Download ng-inspect here

Section 30.2: Getting the Scope of element

In an angular app everything goes around scope, if we could get an elements scope then it is easy to debug the angular app. How to access the scope of element:

```
angular.element(myDomElement).scope();
e.g.
angular.element(document.getElementById('yourElementId')).scope()  //accessing by ID
```

Getting the scope of the controller:

```
 angular.element('[ng-controller=ctrl]').scope()
```

Another easy way to access a DOM element from the console (as jm mentioned) is to click on it in the 'elements' tab, and it automatically gets stored as $0.

```
angular.element($0).scope();
```

Section 30.3: Basic debugging in markup

Scope testing & output of model

```
<div ng-app="demoApp" ng-controller="mainController as ctrl">
    {{$id}}
    <ul>
        <li ng-repeat="item in ctrl.items">
            {{$id}}<br/>
            {{item.text}}
        </li>
    </ul>
    {{$id}}
    <pre>
       {{ctrl.items | json : 2}}
    </pre>
</div>


angular.module('demoApp', [])
.controller('mainController', MainController);

function MainController() {
    var vm = this;
    vm.items = [{
        id: 0,
        text: 'first'
    },
    {
        id: 1,
        text: 'second'
    },
    {
        id: 2,
        text: 'third'
    }];
}
```

Sometimes it can help to see if there is a new scope to fix scoping issues. `$scope.$id` can be used in an expression everywhere in your markup to see if there is a new $scope.

In the example you can see that outside of the ul-tag is the same scope ($id=2) and inside the `ng-repeat` there are new child scopes for each iteration.

An output of the model in a pre-tag is useful to see the current data of your model. The `json` filter creates a nice looking formatted output. The pre-tag is used because inside that tag any new-line character `\n` will be correctly displayed.

demo

Chapter 31: Providers

Section 31.1: Provider

`Provider` is available both in configuration and run phases.

> The Provider recipe is syntactically defined as a custom type that implements a `$get` method.
>
> You should use the Provider recipe only when you want to expose an API for application-wide configuration that must be made before the application starts. This is usually interesting only for reusable services whose behavior might need to vary slightly between applications.

```
angular.module('app',[])
  .provider('endpointProvider', function() {
    var uri = 'n/a';

    this.set = function(value) {
      uri = value;
    };

    this.$get = function() {
      return {
        get: function() {
          return uri;
        }
      };
    };
  })
  .config(function(endpointProviderProvider) {
    endpointProviderProvider.set('http://some.rest.endpoint');
  })
  .controller('MainCtrl', function(endpointProvider) {
    var vm = this;
    vm.endpoint = endpointProvider.get();
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

Without `config` phase result would be

> endpoint = n/a

Section 31.2: Factory

`Factory` is available in run phase.

> The Factory recipe constructs a new service using a function with zero or more arguments (these are

> dependencies on other services). The return value of this function is the service instance created by this recipe.
>
> Factory can create a service of any type, whether it be a primitive, object literal, function, or even an instance of a custom type.

```
angular.module('app',[])
  .factory('endpointFactory', function() {
    return {
      get: function() {
        return 'http://some.rest.endpoint';
      }
    };
  })
  .controller('MainCtrl', function(endpointFactory) {
    var vm = this;
    vm.endpoint = endpointFactory.get();
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

Section 31.3: Constant

`Constant` is available both in configuration and run phases.

```
angular.module('app',[])
  .constant('endpoint', 'http://some.rest.endpoint') // define
  .config(function(endpoint) {
    // do something with endpoint
    // available in both config- and run phases
  })
  .controller('MainCtrl', function(endpoint) {       // inject
    var vm = this;
    vm.endpoint = endpoint;                          // usage
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{ ::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

Section 31.4: Service

`Service` is available in run phase.

> The Service recipe produces a service just like the Value or Factory recipes, but it does so by *invoking a constructor with the new operator*. The constructor can take zero or more arguments, which represent dependencies needed by the instance of this type.

```
angular.module('app',[])
  .service('endpointService', function() {
    this.get = function() {
      return 'http://some.rest.endpoint';
    };
  })
  .controller('MainCtrl', function(endpointService) {
    var vm = this;
    vm.endpoint = endpointService.get();
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

Section 31.5: Value

`Value` is available both in configuration and run phases.

```
angular.module('app',[])
  .value('endpoint', 'http://some.rest.endpoint') // define
  .run(function(endpoint) {
    // do something with endpoint
    // only available in run phase
  })
  .controller('MainCtrl', function(endpoint) {    // inject
    var vm = this;
    vm.endpoint = endpoint;                       // usage
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{ ::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

Chapter 32: Decorators

Section 32.1: Decorate service, factory

Below is example of service decorator, overriding `null` date returned by service.

```
angular.module('app', [])
  .config(function($provide) {
    $provide.decorator('myService', function($delegate) {
      $delegate.getDate = function() { // override with actual date object
        return new Date();
      };
      return $delegate;
    });
  })
  .service('myService', function() {
    this.getDate = function() {
      return null; // w/o decoration we'll be returning null
    };
  })
  .controller('myController', function(myService) {
    var vm = this;
    vm.date = myService.getDate();
  });
```

```
<body ng-controller="myController as vm">
  <div ng-bind="vm.date | date:'fullDate'"></div>
</body>
```

> Saturday, August 6, 2016

Section 32.2: Decorate directive

Directives can be decorated just like services and we can modify or replace any of it's functionality. Note that directive itself is accessed at position 0 in $delegate array and name parameter in decorator must include `Directive` suffix (case sensitive).

So, if directive is called `myDate`, it can be accessed using `myDateDirective` using `$delegate[0]`.

Below is simple example where directive shows current time. We'll decorate it to update current time in one second intervals. Without decoration it will always show same time.

```
<body>
  <my-date></my-date>
</body>
```

```
angular.module('app', [])
  .config(function($provide) {
    $provide.decorator('myDateDirective', function($delegate, $interval) {
      var directive = $delegate[0]; // access directive

      directive.compile = function() { // modify compile fn
        return function(scope) {
          directive.link.apply(this, arguments);
          $interval(function() {
            scope.date = new Date(); // update date every second
          }, 1000);
        };
```

```
    };

    return $delegate;
  });
})
.directive('myDate', function() {
  return {
    restrict: 'E',
    template: '<span>Current time is {{ date | date:\'MM:ss\' }}</span>',
    link: function(scope) {
      scope.date = new Date(); // get current date
    }
  };
});
```

> Current time is 08:33

Section 32.3: Decorate filter

When decorating filters, name parameter must include `Filter` suffix (case sensitive). If filter is called `repeat`, decorator parameter is `repeatFilter`. Below we'll decorate custom filter that repeats any given string *n* times so that result is reversed. You can also decorate angular's build-in filters the same way, although not recommended as it can affect the functionality of the framework.

```
<body>
  <div ng-bind="'i can haz cheeseburger ' | repeat:2"></div>
</body>

angular.module('app', [])
  .config(function($provide) {
    $provide.decorator('repeatFilter', function($delegate) {
      return function reverse(input, count) {
        // reverse repeated string
        return ($delegate(input, count)).split('').reverse().join('');
      };
    });
  })
  .filter('repeat', function() {
    return function(input, count) {
      // repeat string n times
      return (input || '').repeat(count || 1);
    };
  });
```

> i can haz cheeseburger i can haz cheeseburger

> regrubeseehc zah nac i regrubeseehc zah nac i

Chapter 33: Grunt tasks

Section 33.1: Run application locally

> Following example requires that node.js is installed and npm is available.
> Full working code can be forked from GitHub @ https://github.com/mikkoviitala/angular-grunt-run-local

Usually one of the first things you want to do when developing new web application is to make it run locally.

Below you'll find complete example achieving just that, using grunt (javascript task runner), npm (node package manager) and bower (yet another package manager).

Beside your actual application files you'll need to install few 3rd party dependencies using tools mentioned above. In your project directory, **preferably root**, you'll need three (3) files.

- package.json (dependencies managed by npm)
- bower.json (dependencies managed by bower)
- gruntfile.js (grunt tasks)

So your project directory looks like so:

package.json

We'll be installing **grunt** itself, **matchdep** to make our life easier allowing us to filter dependencies by name, **grunt-express** used to start express web server via grunt and **grunt-open** to open urls/files from a grunt task.

So these packages are all about "infrastructure" and helpers we'll be building our application on.

```
{
  "name": "app",
  "version": "1.0.0",
  "dependencies": {},
  "devDependencies": {
    "grunt": "~0.4.1",
    "matchdep": "~0.1.2",
    "grunt-express": "~1.0.0-beta2",
    "grunt-open": "~0.2.1"
  },
  "scripts": {
    "postinstall": "bower install"
  }
}
```

bower.json

Bower is (or at least should be) all about front-end and we'll be using it to install **angular**.

```
{
```

```
  "name": "app",
  "version": "1.0.0",
  "dependencies": {
    "angular": "~1.3.x"
  },
  "devDependencies": {}
}
```

gruntfile.js

Inside gruntfile.js we'll have the actual "running application locally" magic, which opens our application in new browser window, running on http://localhost:9000/

```
'use strict';

// see http://rhumaric.com/2013/07/renewing-the-grunt-livereload-magic/

module.exports = function(grunt) {
  require('matchdep').filterDev('grunt-*').forEach(grunt.loadNpmTasks);

  grunt.initConfig({
    express: {
      all: {
        options: {
          port: 9000,
          hostname: 'localhost',
          bases: [__dirname]
        }
      }
    },

    open: {
      all: {
        path: 'http://localhost:<%= express.all.options.port%>'
      }
    }
  });

  grunt.registerTask('app', [
    'express',
    'open',
    'express-keepalive'
  ]);
};
```

Usage

To get your application up & running from scratch, save above files to your project's root directory (any empty folder will do). Then fire up console/command line and type in the following to install all required dependencies.

```
npm install -g grunt-cli bower
npm install
```

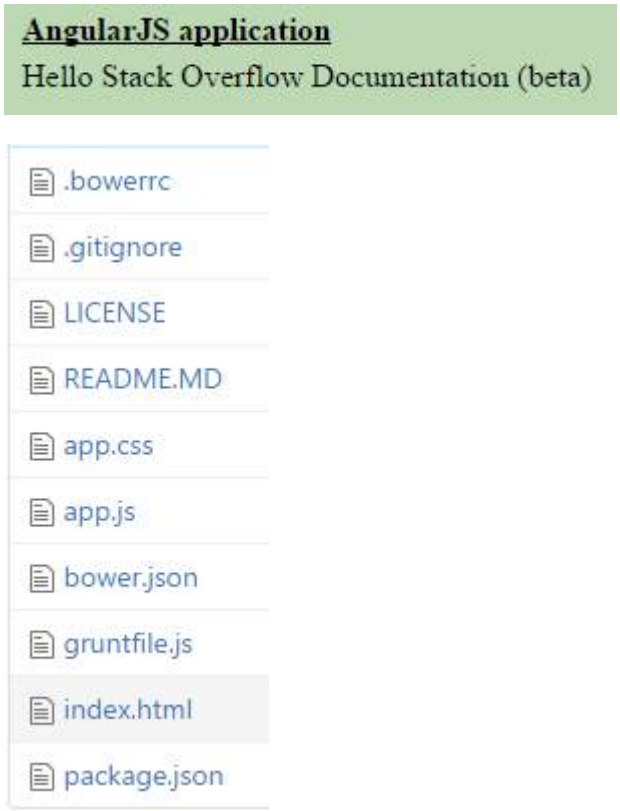
And then run your application using

```
grunt app
```

Note that yes, you'll be needing your actual application files, too.

For almost-minimal example browse GitHub repository mentioned in beginning of this example.

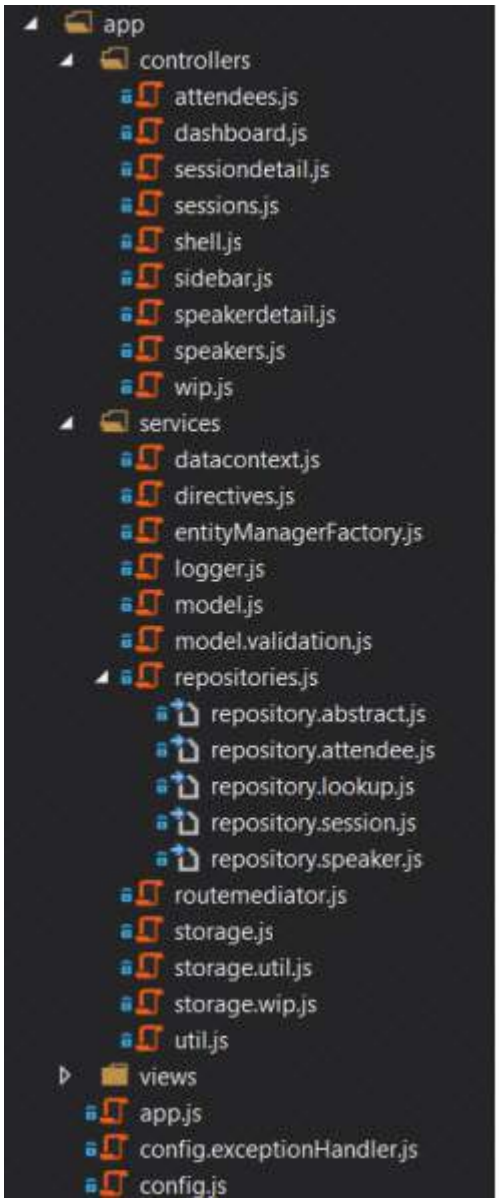
There structure ain't that different. There's just `index.html` template, angular code in `app.js` and few styles in `app.css`. Other files are for Git and editor configuration and some generic stuff. Give it a try!

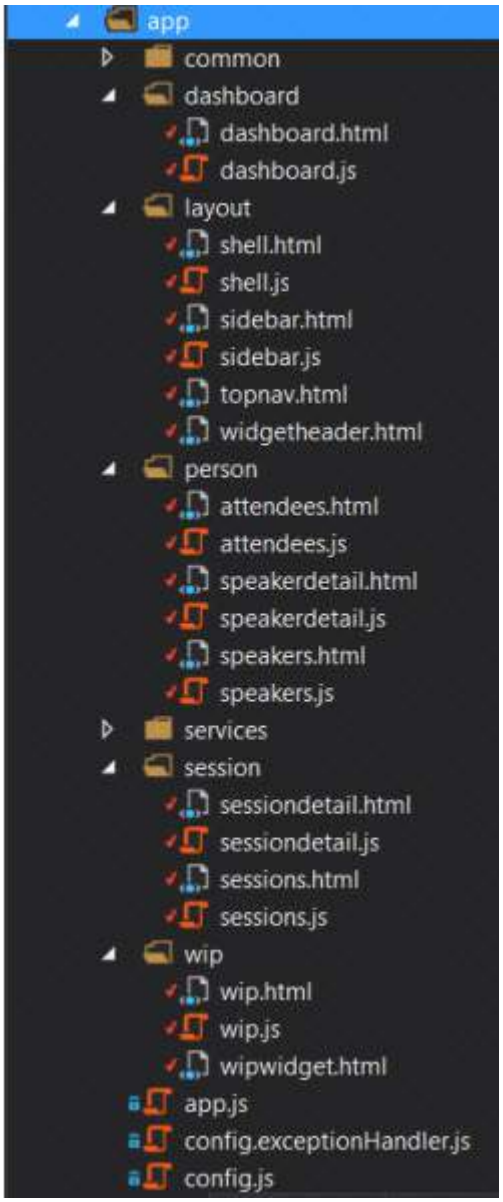


Chapter 34: Angular Project - Directory Structure

Section 34.1: Directory Structure

A common question among new Angular programmers - "What should be the structure of the project?". A good structure helps toward a scalable application development. When we start a project we have two choices, **Sort By Type** (left) and **Sort By Feature** (right). The second is better, especially in large applications, the project becomes a lot easier to manage.





Sort By Type (left)

The application is organized by the files' type.

- **Advantage** - Good for small apps, for programmers only starting to use Angular, and is easy to convert to the second method.
- **Disadvantage** - Even for small apps it starts to get more difficult to find a specific file. For instance, a view and it's controller are in two seperate folders.

Sort By Feature (right)

The suggested organizing method where the filed are sorted by features' type.

All of the layout views and controllers go in the layout folder, the admin content goes in the admin folder, and so on.

- **Advantage** - When looking for a section of code determining a certain feature it's all located in one folder.
- **Disadvantage** - Services are a bit different as they “service” many features.

You can read more about it on Angular Structure: Refactoring for Growth

The suggested file structure combining both of the aforementioned methods:

```
├── app
│   ├── app.js
│   ├── controllers
│   │   ├── home
│   │   │   ├── FirstCtrl.js
│   │   │   └── FirstCtrl.spec.js
│   │   │   └── SecondCtrl.js
│   │   │   └── SecondCtrl.spec.js
│   │   └── about
│   │       └── ThirdCtrl.js
│   │       └── ThirdCtrl.spec.js
│   ├── directives
│   │   ├── home
│   │   │   └── directive1.js
│   │   │   └── directive1.spec.js
│   │   └── about
│   │       ├── directive2.js
│   │       ├── directive2.spec.js
│   │       └── directive3.js
│   │       └── directive3.spec.js
│   ├── filters
│   │   ├── home
│   │   └── about
│   └── services
│       ├── CommonService.js
│       ├── CommonService.spec.js
│       ├── cache
│       │   ├── Cache1.js
│       │   ├── Cache1.spec.js
│       │   └── Cache2.js
│       │   └── Cache2.spec.js
│       └── models
│           ├── Model1.spec.js
│           ├── Model1.js
│           └── Model2.spec.js
```

Credit to: Angular Style Guide

Chapter 35: AngularJS bindings options (`=`, `@`, `&` etc.)

Section 35.1: Bind optional attribute

```
bindings: {
   mandatory: '='
   optional: '=?',
   foo: '=?bar'
}
```

Optional attributes should be marked with question mark: `=?` or `=?bar`. It is protection for `($compile:nonassign)` exception.

Section 35.2: @ one-way binding, attribute binding

Pass in a literal value (not an object), such as a string or number.

Child scope gets his own value, if it updates the value, parent scope has his own old value (child scope can't modify the parens scope value). When parent scope value is changed, child scope value will be changed as well. All interpolations appears every time on digest call, not only on directive creation.

```
<one-way text="Simple text." <!-- 'Simple text.' -->
         simple-value="123" <!-- '123' Note, is actually a string object. -->
         interpolated-value="{{parentScopeValue}}" <!-- Some value from parent scope. You can't
change parent scope value, only child scope value. Note, is actually a string object. -->
         interpolated-function-value="{{parentScopeFunction()}}" <!-- Executes parent scope
function and takes a value. -->

         <!-- Unexpected usage. -->
         object-item="{{objectItem}}" <!-- Converts object|date to string. Result might be:
'{"a":5,"b":"text"}'. -->
         function-item="{{parentScopeFunction}}"> <!-- Will be an empty string. -->
</one-way>
```

Section 35.3: = two-way binding

Passing in a value by reference, you want to share the value between both scopes and manipulate them from both scopes. You should not use {{...}} for interpolation.

```
<two-way text="'Simple text.'" <!-- 'Simple text.' -->
         simple-value="123" <!-- 123 Note, is actually a number now. -->
         interpolated-value="parentScopeValue" <!-- Some value from parent scope. You may change it
in one scope and have updated value in another. -->
         object-item="objectItem" <!-- Some object from parent scope. You may change object
properties in one scope and have updated properties in another. -->

         <!-- Unexpected usage. -->
         interpolated-function-value="parentScopeFunction()" <!-- Will raise an error. -->
         function-item="incrementInterpolated"> <!-- Pass the function by reference and you may use
it in child scope. -->
</two-way>
```

Passing function by reference is a bad idea: to allow scope to change the definition of a function, and two unnecessary watcher will be created, you need to minimize watchers count.

Section 35.4: & function binding, expression binding

Pass a method into a directive. It provides a way to execute an expression in the context of the parent scope. Method will be executed in the scope of the parent, you may pass some parameters from the child scope there. You should not use {{...}} for interpolation. When you use & in a directive, it generates a function that returns the value of the expression evaluated against the parent scope (not the same as = where you just pass a reference).

```
<expression-binding interpolated-function-value="incrementInterpolated(param)" <!--
interpolatedFunctionValue({param: 'Hey'}) will call passed function with an argument. -->
                    function-item="incrementInterpolated" <!-- functionItem({param: 'Hey'})() will
call passed function, but with no possibility set up a parameter. -->
                    text="'Simple text.'"  <!-- text() == 'Simple text.'-->
                    simple-value="123" <!-- simpleValue() == 123 -->
                    interpolated-value="parentScopeValue" <!-- interpolatedValue() == Some value
from parent scope. -->
                    object-item="objectItem"> <!-- objectItem() == Object item from parent scope. -
->
</expression-binding>
```

All parameters will be wrapped into functions.

Section 35.5: Available binding through a simple sample

```
angular.component("SampleComponent", {
  bindings: {
     title: '@',
     movies: '<',
     reservation: "=",
     processReservation: "&"
  }
});
```

Here we have all binding elements.

@ indicates that we need a very **basic binding**, from the parent scope to the children scope, without any watcher, in any way. Every update in the parent scope would stay in the parent scope, and any update on the child scope would not be communicated to the parent scope.

< indicates a **one way binding**. Updates in the parent scope would be propagated to the children scope, but any update in the children scope would not be applied to the parent scope.

= is already known as a two-way binding. Every update on the parent scope would be applied on the children ones, and every child update would be applied to the parent scope.

& is now used for an output binding. According to the component documentation, it should be used to reference the parent scope method. Instead of manipulating the children scope, just call the parent method with the updated data!

Chapter 36: Lazy loading

Section 36.1: Preparing your project for lazy loading

After including `oclazyload.js` in your index file, declare `ocLazyLoad` as a dependency in `app.js`

```
//Make sure you put the correct dependency! it is spelled different than the service!
angular.module('app', [
 'oc.lazyLoad',
 'ui-router'
])
```

Section 36.2: Usage

In order to lazily load files inject the `$ocLazyLoad` service into a controller or another service

```
.controller('someCtrl', function($ocLazyLoad) {
 $ocLazyLoad.load('path/to/file.js').then(...);
});
```

Angular modules will be automatically loaded into angular.

Other variation:

```
$ocLazyLoad.load([
  'bower_components/bootstrap/dist/js/bootstrap.js',
  'bower_components/bootstrap/dist/css/bootstrap.css',
  'partials/template1.html'
]);
```

For a complete list of variations visit the official documentation

Section 36.3: Usage with router

UI-Router:

```
.state('profile', {
 url: '/profile',
 controller: 'profileCtrl as vm'
 resolve: {
  module: function($ocLazyLoad) {
   return $ocLazyLoad.load([
    'path/to/profile/module.js',
    'path/to/profile/style.css'
   ]);
  }
 }
});
```

ngRoute:

```
.when('/profile', {
     controller: 'profileCtrl as vm'
     resolve: {
      module: function($ocLazyLoad) {
       return $ocLazyLoad.load([
        'path/to/profile/module.js',
        'path/to/profile/style.css'
       ]);
```

```
      }
     }
    });
```

Section 36.4: Using dependency injection

The following syntax allows you to specify dependencies in your `module.js` instead of explicit specification when using the service

```
//lazy_module.js
angular.module('lazy', [
 'alreadyLoadedDependency1',
 'alreadyLoadedDependency2',
 ...
 [
  'path/to/lazily/loaded/dependency.js',
  'path/to/lazily/loaded/dependency.css'
 ]
]);
```

Note: this syntax will only work for lazily loaded modules!

Section 36.5: Using the directive

```
<div oc-lazy-load="['path/to/lazy/loaded/directive.js', 'path/to/lazy/loaded/directive.html']">

<!-- myDirective available here -->
<my-directive></my-directive>

</div>
```

Chapter 37: HTTP Interceptor

The $http service of AngularJS allows us to communicate with a backend and make HTTP requests. There are cases where we want to capture every request and manipulate it before sending it to the server. Other times we would like to capture the response and process it before completing the call. Global http error handling can be also a good example of such need. Interceptors are created exactly for such cases.

Section 37.1: Generic httpInterceptor step by step

Create an HTML file with the following content:

```
<!DOCTYPE html>
<html>
<head>
  <title>Angular Interceptor Sample</title>
  <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
  <script src="app.js"></script>
  <script src="appController.js"></script>
  <script src="genericInterceptor.js"></script>
</head>
<body ng-app="interceptorApp">
    <div ng-controller="appController as vm">
        <button ng-click="vm.sendRequest()">Send a request</button>
    </div>
</body>
</html>
```

Add a JavaScript file called 'app.js':

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

interceptorApp.config(function($httpProvider) {
        $httpProvider.interceptors.push('genericInterceptor');
});
```

Add another one called 'appController.js':

```
(function() {
    'use strict';

    function appController($http) {
        var vm = this;

        vm.sendRequest = function(){
            $http.get('http://google.com').then(function(response){
                console.log(response);
            });
        };
    }

    angular.module('interceptorApp').controller('appController',['$http', appController]);
})();
```

And finally the file containing the interceptor itself 'genericInterceptor.js':

```
(function() {
    "use strict";
```

```
    function genericInterceptor($q) {
        this.responseError = function (response) {
            return $q.reject(response);
        };

        this.requestError = function(request){
            if (canRecover(rejection)) {
                return responseOrNewPromise
            }
            return $q.reject(rejection);
        };

        this.response = function(response){
            return response;
        };

        this.request = function(config){
            return config;
        }
    }

    angular.module('interceptorApp').service('genericInterceptor', genericInterceptor);
})();
```

The 'genericInterceptor' cover the possible functions which we can override adding extra behavior to our application.

Section 37.2: Getting Started

Angular's builtin `$http` service allows us to send HTTP requests. Oftentime, the need arise to do things before or after a request, for example adding to each request an authentication token or creating a generic error handling logic.

Section 37.3: Flash message on response using http interceptor

In the view file

In the base html (index.html) where we usually include the angular scripts or the html that is shared across the app, leave an empty div element, the flash messages will be appearing inside this div element

```
<div class="flashmessage" ng-if="isVisible">
    {{flashMessage}}
</div>
```

Script File

In the config method of angular module, inject the httpProvider, the httpProvider has an interceptor array property, push the custom interceptor, In the current example the custom interceptor intercepts only the response and calls a method attached to rootScope.

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

    interceptorTest.config(['$httpProvider',function ($httpProvider) {

        $httpProvider.interceptors.push(["$rootScope",function ($rootScope) {
            return {     //intercept only the response
                        'response': function (response)
```

```
                                    {

$rootScope.showFeedBack(response.status,response.data.message);

                                      return response;
                                    }
                    };
        }]);

    }])
```

Since only providers can be injected into the config method of an angular module (that is httpProvider and not the rootscope), declare the method attached to rootscope inside the run method of angular module.

Also display the message inside $timeout so that the message will have the flash property, that is disappearing after a threshold time. In our example its 3000 ms.

```
interceptorTest.run(["$rootScope","$timeout",function($rootScope,$timeout){
     $rootScope.showFeedBack = function(status,message){

        $rootScope.isVisible = true;
        $rootScope.flashMessage = message;
        $timeout(function(){$rootScope.isVisible = false },3000)
    }
}]);
```

Common pitfalls

Trying to inject **$rootScope or any other services** inside **config** method of angular module, the lifecycle of angular app doesnt allow that and unknown provider error will be thrown. Only **providers** can be injected in **config** method of the angular module

Chapter 38: Print

Section 38.1: Print Service

Service:

```
angular.module('core').factory('print_service', ['$rootScope', '$compile', '$http',
'$timeout','$q',
    function($rootScope, $compile, $http, $timeout,$q) {

        var printHtml = function (html) {
            var deferred = $q.defer();
            var hiddenFrame = $('<iframe style="display: none"></iframe>').appendTo('body')[0];

            hiddenFrame.contentWindow.printAndRemove = function() {
                hiddenFrame.contentWindow.print();
                $(hiddenFrame).remove();
                deferred.resolve();
            };

            var htmlContent =   "<!doctype html>"+
                                "<html>"+
                                    '<head><link rel="stylesheet" type="text/css"
href="/style/css/print.css"/></head>'+
                                    '<body onload="printAndRemove();">' +
                                        html +
                                    '</body>'+
                                "</html>";

            var doc = hiddenFrame.contentWindow.document.open("text/html", "replace");
            doc.write(htmlContent);
            doc.close();
            return deferred.promise;
        };

        var openNewWindow = function (html) {
            var newWindow = window.open("debugPrint.html");
            newWindow.addEventListener('load', function(){
                $(newWindow.document.body).html(html);
            }, false);
        };

        var print = function (templateUrl, data) {

            $rootScope.isBeingPrinted = true;

            $http.get(templateUrl).success(function(template){
                var printScope = $rootScope.$new()
                angular.extend(printScope, data);
                var element = $compile($('<div>' + template + '</div>'))(printScope);
                var waitForRenderAndPrint = function() {
                    if(printScope.$$phase || $http.pendingRequests.length) {
                        $timeout(waitForRenderAndPrint, 1000);
                    } else {
                        // Replace printHtml with openNewWindow for debugging
                        printHtml(element.html());
                        printScope.$destroy();
                    }
                };
                waitForRenderAndPrint();
```

```
            });
        };

        var printFromScope = function (templateUrl, scope, afterPrint) {
            $rootScope.isBeingPrinted = true;
            $http.get(templateUrl).then(function(response){
                var template = response.data;
                var printScope = scope;
                var element = $compile($('<div>' + template + '</div>'))(printScope);
                var waitForRenderAndPrint = function() {
                    if (printScope.$$phase || $http.pendingRequests.length) {
                        $timeout(waitForRenderAndPrint);
                    } else {
                        // Replace printHtml with openNewWindow for debugging
                        printHtml(element.html()).then(function() {
                            $rootScope.isBeingPrinted = false;
                            if (afterPrint) {
                                afterPrint();
                            }
                        });
                    }
                };
                waitForRenderAndPrint();
            });
        };

        return {
            print : print,
            printFromScope : printFromScope
        }
    }
]);
```

Controller :

```
var template_url = '/views/print.client.view.html';
print_service.printFromScope(template_url,$scope,function(){
      // Print Completed
});
```

Chapter 39: Performance Profiling

Section 39.1: All About Profiling

What is Profiling?

By definition Profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.

Why is it necessary?

Profiling is important because you can't optimise effectively until you know what your program is spending most of its time doing. Without measuring your program execution time (profiling), you won't know if you've actually improved it.

Tools and Techniques :

1. Chrome's in-built dev tools

   This includes a comprehensive set of tools to be used for profiling.You can go deep to find out bottlenecks in your javascript file, css files, animations, cpu consumption, memory leaks, network, security etc.

   Make a Timeline recording and look for suspiciously long Evaluate Script events. If you find any, you can enable the JS Profiler and re-do your recording to get more detailed information about exactly which JS functions were called and how long each took. Read more...

2. FireBug (use with Firefox)
3. Dynatrace (use with IE)
4. Batarang (use with Chrome)

   It's an outdated add-on for chrome browser though it's stable and can be used to monitor models, performance, dependencies for an angular application. It works fine for small scale application and can give you an insight of what does scope variable holds at various levels. It tells you about active watchers, watch expressions, watch collections in the app.

5. Watcher (use with Chrome)

   Nice and simplistic UI to count the number of watchers in a Angular app.

6. Use the following code to manually find out the number of watchers in your angular app (credit to @Words Like Jared Number of watchers)

```
(function() {
    var root = angular.element(document.getElementsByTagName('body')),
        watchers = [],
        f = function(element) {
        angular.forEach(['$scope', '$isolateScope'], function(scopeProperty) {
            if(element.data() && element.data().hasOwnProperty(scopeProperty)) {
                angular.forEach(element.data()[scopeProperty].$$watchers, function(watcher) {
```

```
                watchers.push(watcher);
                });
            }
        });

        angular.forEach(element.children(), function(childElement) {
            f(angular.element(childElement));
        });
    };

    f(root);

    // Remove duplicate watchers
    var watchersWithoutDuplicates = [];
    angular.forEach(watchers, function(item) {
        if(watchersWithoutDuplicates.indexOf(item) < 0) {
            watchersWithoutDuplicates.push(item);
        }
    });
    console.log(watchersWithoutDuplicates.length);
})();
```

7. There are several online tools/websites available which facilitates wide range of functionalities to create a profile of your application.

   One such site is : https://www.webpagetest.org/

   With this you can run a free website speed test from multiple locations around the globe using real browsers (IE and Chrome) and at real consumer connection speeds. You can run simple tests or perform advanced testing including multi-step transactions, video capture, content blocking and much more.

Next Steps:

Done with Profiling. It only brings you half way down the road. The very next task is to actually turn your findings into action items to optimise your application. See this documentation on how you can improve the performance of your angular app with simple tricks.

Happy Coding :)

Chapter 40: Distinguishing Service vs Factory

Section 40.1: Factory VS Service once-and-for-all

By definition:

Services are basically constructor functions. They use 'this' keyword.

Factories are simple functions hence return an object.

Under the hood:

Factories internally calls provider function.

Services internally calls Factory function.

Debate:

Factories can run code before we return our object literal.

But at the same time, Services can also be written to return an object literal and to run code before returning. Though that is contra productive as services are designed to act as constructor function.

In fact, constructor functions in JavaScript can return whatever they want.

So which one is better?

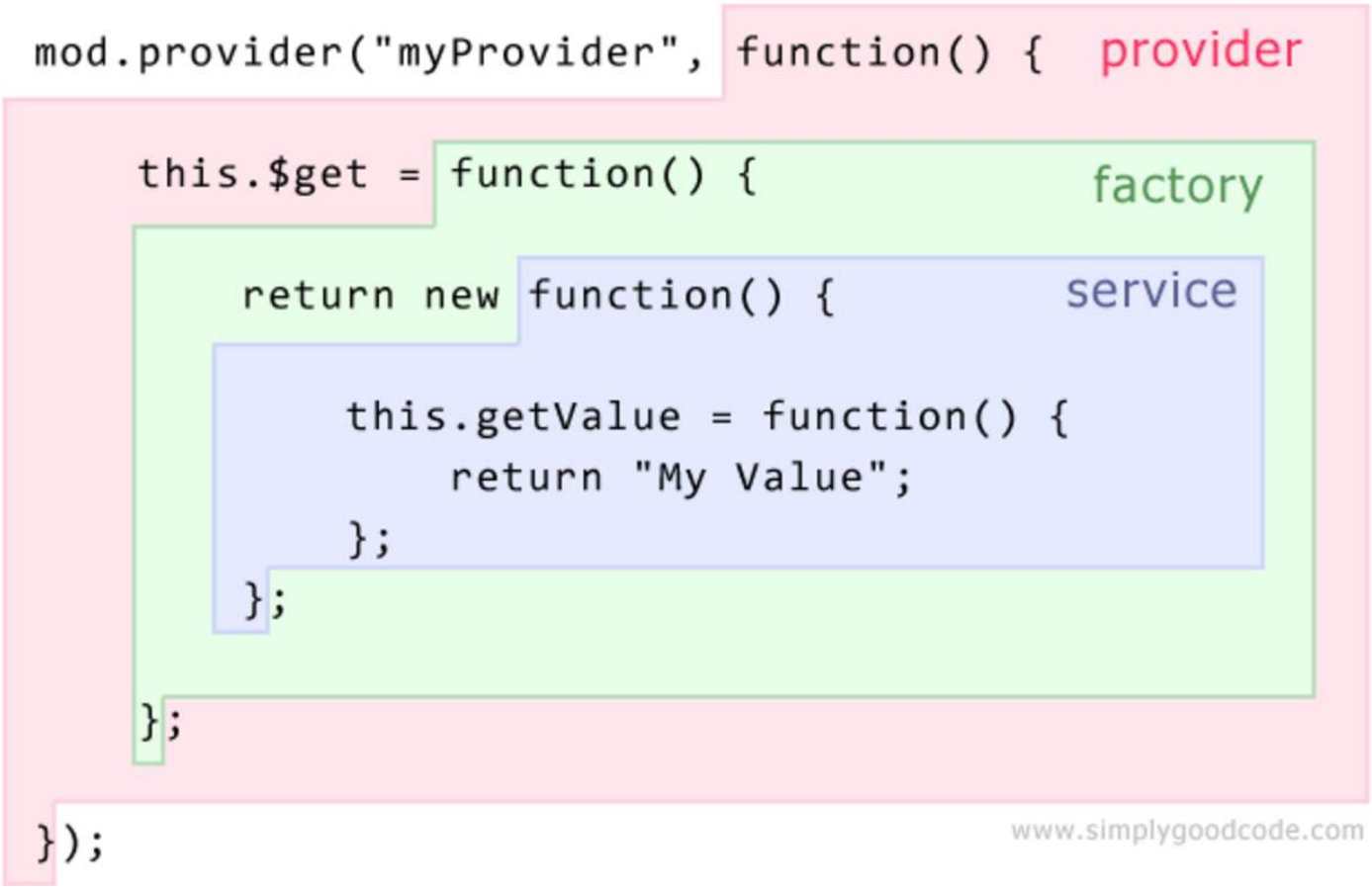
The constructor syntax of services is more close to class syntax of ES6. So migration will be easy.

Summary

So in summary, provider, factory, and service are all providers.

A factory is a special case of a provider when all you need in your provider is a $get() function. It allows you to write it with less code.

A service is a special case of a factory when you want to return an instance of a new object, with the same benefit of writing less code.


mod.provider("myProvider", function() {
provider
this.$get = function() {
factory
return new function() {
service
this.getValue = function() {
return "My Value";
};
};
};
});
www.simplygoodcode.com

Chapter 41: Use of in-built directives

Section 41.1: Hide/Show HTML Elements

This example hide show html elements.

```
<!DOCTYPE html>
<html ng-app="myDemoApp">
  <head>
    <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
    <script>

      function HideShowController() {
        var vm = this;
        vm.show=false;
        vm.toggle= function() {
          vm.show=!vm.show;
        }
      }

      angular.module("myDemoApp", [/* module dependencies go here */])
        .controller("hideShowController", [HideShowController]);
    </script>
  </head>
  <body ng-cloak>
    <div ng-controller="hideShowController as vm">
      <a style="cursor: pointer;" ng-show="vm.show" ng-click="vm.toggle()">Show Me!</a>
      <a style="cursor: pointer;" ng-hide="vm.show" ng-click="vm.toggle()">Hide Me!</a>
    </div>
  </body>
</html>
```

Live Demo

Step by step explanation:

1. `ng-app="myDemoApp"`, the ngApp directive tells angular that a DOM element is controlled by a specific angular.module named "myDemoApp".
2. `<script src="[//angular include]">` include angular js.
3. `HideShowController` function is defined containing another function named `toggle` which help to hide show the element.
4. `angular.module(...)` creates a new module.
5. `.controller(...)` Angular Controller and returns the module for chaining;
6. `ng-controller` directive is key aspect of how angular supports the principles behind the Model-View-Controller design pattern.
7. `ng-show` directive shows the given HTML element if expression provided is true.
8. `ng-hide` directive hides the given HTML element if expression provided is true.
9. `ng-click` directive fires a toggle function inside controller

Chapter 42: ng-repeat

Variable	Details
`$index`	*number* iterator offset of the repeated element (0..length-1)
`$first`	*boolean* true if the repeated element is first in the iterator.
`$middle`	*boolean* true if the repeated element is between the first and last in the iterator.
`$last`	*boolean* true if the repeated element is last in the iterator.
`$even`	*boolean* true if the iterator position `$index` is even (otherwise false).
`$odd`	*boolean* true if the iterator position `$index` is odd (otherwise false).

The `ngRepeat` directive instantiates a template once per item from a collection. The collection must be an array or an object. Each template instance gets its own scope, where the given loop variable is set to the current collection item, and `$index` is set to the item index or key.

Section 42.1: ng-repeat-start + ng-repeat-end

AngularJS 1.2 `ng-repeat` handle multiple elements with `ng-repeat-start` and `ng-repeat-end`:

```
// table items
$scope.tableItems = [
    {
        row1: 'Item 1: Row 1',
        row2: 'Item 1: Row 2'
    },
    {
        row1: 'Item 2: Row 1',
        row2: 'Item 2: Row 2'
    }
];

// template
<table>
    <th>
        <td>Items</td>
    </th>
    <tr ng-repeat-start="item in tableItems">
        <td ng-bind="item.row1"></td>
    </tr>
    <tr ng-repeat-end>
        <td ng-bind="item.row2"></td>
    </tr>
</table>
```

Output:

Items
Item 1: Row 1
Item 1: Row 2
Item 2: Row 1
Item 2: Row 2

Section 42.2: Iterating over object properties

```
<div ng-repeat="(key, value) in myObj"> ... </div>
```

For example

```
<div ng-repeat="n in [42, 42, 43, 43]">
  {{n}}
</div>
```

Section 42.3: Tracking and Duplicates

`ngRepeat` uses $watchCollection to detect changes in the collection. When a change happens, `ngRepeat` then makes the corresponding changes to the DOM:

- When an item is added, a new instance of the template is added to the DOM.
- When an item is removed, its template instance is removed from the DOM.
- When items are reordered, their respective templates are reordered in the DOM.

Duplicates

- `track by` for any list that may include duplicate values.
- `track by` also speeds up list changes significantly.
- If you don't use `track by` in this case, you get the error: `[ngRepeat:dupes]`

```
$scope.numbers = ['1','1','2','3','4'];

<ul>
  <li ng-repeat="n in numbers track by $index">
    {{n}}
  </li>
</ul>
```

Chapter 43: Session storage

Section 43.1: Handling session storage through service using angularjs

Session storage service :

Common factory service that will save and return the saved session data based on the key.

```
 'use strict';

/**
 * @ngdoc factory
 * @name app.factory:storageService
 * @description This function will communicate with HTML5 sessionStorage via Factory Service.
 */

app.factory('storageService', ['$rootScope', function($rootScope) {

    return {
        get: function(key) {
            return sessionStorage.getItem(key);
        },
        save: function(key, data) {
            sessionStorage.setItem(key, data);
        }
    };
}]);
```

In controller :

Inject the storageService dependency in the controller to set and get the data from the session storage.

```
app.controller('myCtrl',['storageService',function(storageService) {

  // Save session data to storageService
  storageService.save('key', 'value');

  // Get saved session data from storageService
  var sessionData = storageService.get('key');

});
```

Chapter 44: Angular MVC

In **AngularJS** the **MVC** pattern is implemented in JavaScript and HTML. The view is defined in HTML, while the model and controller are implemented in JavaScript. There are several ways that these components can be put together in AngularJS but the simplest form starts with the view.

Section 44.1: The Static View with controller

mvc demo
Hello World

Section 44.2: Controller Function Definition

```
var indexController = myApp.controller("indexController", function ($scope) {
    // Application logic goes here
});
```

Section 44.3: Adding information to the model

```
var indexController = myApp.controller("indexController", function ($scope) {
    // controller logic goes here
    $scope.message = "Hello Hacking World"
});
```

Chapter 45: ng-style

The 'ngStyle' directive allows you to set CSS style on an HTML element conditionally. Much like how we could use *style* attribute on HTML element in non-AngularJS projects, we can use `ng-style` in angularjs do apply styles based on some boolean condition.

Section 45.1: Use of ng-style

Below example changes the opacity of the image based on the "status" parameter.

```
<img class="img-responsive"  ng-src="{{imagesrc}}"
   ng-style="{'opacity' : (status == 2) ? 1 : 0.5}">
```

Chapter 46: ng-view

ng-view is one of in-build directive that angular uses as a container to switch between views. {info} ngRoute is no longer a part of the base angular.js file, so you'll need to include the angular-route.js file after your the base angular javascript file. We can configure a route by using the “when” function of the $routeProvider. We need to first specify the route, then in a second parameter provide an object with a templateUrl property and a controller property.

Section 46.1: Registration navigation

1. We injecting the module in the application

```
var Registration=angular.module("myApp",["ngRoute"]);
```

2. now we use $routeProvider from "ngRoute"

```
    Registration.config(function($routeProvider) {

});
```

3. finally we integrating the route, we define "/add" routing to the application in case application get "/add" it divert to regi.htm

```
Registration.config(function($routeProvider) {
    $routeProvider
    .when("/add", {
        templateUrl : "regi.htm"
    })
});
```

Section 46.2: ng-view

`ng-view` is a directive used with `$route` to render a partial view in the main page layout. Here in this example, Index.html is our main file and when user lands on "/" route the templateURL home.html will be rendered in Index.html where `ng-view` is mentioned.

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

.config(function($routeProvider){
  $routeProvider.when("/",
    {
      templateUrl: "home.html",
      controller: "homeCtrl"
    }
  );
});

angular.module('ngApp').controller('homeCtrl',['$scope', function($scope) {
  $scope.welcome= "Welcome to stackoverflow!";
}]);

//Index.html
<body ng-app="ngApp">
    <div ng-view></div>
</body>

//Home Template URL or home.html
```

```
<div><h2>{{welcome}}</h2></div>
```

Chapter 47: The Self Or This Variable In A Controller

This is an explanation of a common pattern and generally considered best practice that you may see in AngularJS code.

Section 47.1: Understanding The Purpose Of The Self Variable

When using "controller as syntax" you would give your controller an alias in the html when using the ng-controller directive.

```
<div ng-controller="MainCtrl as main">
</div>
```

You can then access properties and methods from the ***main*** variable that represents our controller instance. For example, let's access the ***greeting*** property of our controller and display it on the screen:

```
<div ng-controller="MainCtrl as main">
    {{ main.greeting }}
</div>
```

Now, in our controller, we need to set a value to the greeting property of our controller instance (as opposed to $scope or something else):

```
angular
.module('ngNjOrg')
.controller('ForgotPasswordController',function ($log) {
  var self = this;

  self.greeting = "Hello World";
})
```

In order to have the HTML display correctly we needed to set the greeting property on ***this*** inside of our controller body. I am creating an intermediate variable named ***self*** that holds a reference to this. Why? Consider this code:

```
angular
.module('ngNjOrg')
.controller('ForgotPasswordController',function ($log) {
  var self = this;

  self.greeting = "Hello World";

  function itsLate () {
    this.greeting = "Goodnight";
  }

})
```

In this above code you may expect the text on the screen to update when the method *itsLate* is called, but in fact it does not. JavaScript uses function level scoping rules so the "this" inside of itsLate refers to something different that "this" outside of the method body. However, we can get the desired result if we use the ***self*** variable:

```
 angular
.module('ngNjOrg')
```

```
.controller('ForgotPasswordController',function ($log) {
  var self = this;

  self.greeting = "Hello World";

  function itsLate () {
    self.greeting = "Goodnight";
  }

})
```

This is the beauty of using a "self" variable in your controllers- you can access this anywhere in your controller and can always be sure that it is referencing your controller instance.

Chapter 48: Controllers with ES6

Section 48.1: Controller

it is very easy to write an angularJS controller with ES6 if your are familiarized with the **Object Oriented Programming** :

```
class exampleContoller{

       constructor(service1,service2,...serviceN){
          let ctrl=this;
              ctrl.service1=service1;
              ctrl.service2=service2;
               .
               .
               .
              ctrl.service1=service1;
              ctrl.controllerName = 'Example Controller';
              ctrl.method1(controllerName)

       }

       method1(param){
          let ctrl=this;
               ctrl.service1.serviceFunction();
              .
              .
              ctrl.scopeName=param;
       }
       .
       .
       .
       methodN(param){
          let ctrl=this;
               ctrl.service1.serviceFunction();
              .
              .
       }


   }
   exampleContoller.$inject = ['service1','service2',...,'serviceN'];
   export default exampleContoller;
```

Chapter 49: Custom filters with ES6

Section 49.1: FileSize Filter using ES6

We have here a file Size filter to describe how to add costum filter to an existing module :

```
let fileSize=function (size,unit,fixedDigit) {
return size.toFixed(fixedDigit) + ' '+unit;
};

  let fileSizeFilter=function () {
        return function (size) {
            if (isNaN(size))
                size = 0;

            if (size < 1024)
                return size + ' octets';

            size /= 1024;

            if (size < 1024)
               return fileSize(size,'Ko',2);

            size /= 1024;

            if (size < 1024)
               return  fileSize(size,'Mo',2);

            size /= 1024;

            if (size < 1024)
               return  fileSize(size,'Go',2);

            size /= 1024;
            return  fileSize(size,'To',2);
        };
    };
export default fileSizeFilter;
```

The filter call into the module :

```
import fileSizeFilter from 'path...';
let myMainModule =
    angular.module('mainApp', [])
   .filter('fileSize', fileSizeFilter);
```

The html code where we call the filter :

```
<div ng-app="mainApp">

    <div>
        <input type="text" ng-model="size" />
    </div>
    <div>
        <h3>Output:</h3>
        <p>{{size| Filesize}}</p>
    </div>
</div>
```

Chapter 50: Migration to Angular 2+

AngularJS has been totally rewritten using the TypeScript language and renamed to just Angular.

There is a lot that can be done to an AngularJS app to ease the migration process. As the official upgrade guide says, several "preparation steps" can be performed to refactor your app, making it better and closer to the new Angular style.

Section 50.1: Converting your AngularJS app into a componend-oriented structure

In the new Angular framework, **Components** are the main building blocks that compose the user interface. So one of the first steps that helps an AngularJS app to be migrated to the new Angular is to refactor it into a more component-oriented structure.

Components were also introduced in the old AngularJS starting from version **1.5+**. Using Components in an AngularJS app will not only make its structure closer to the new Angular 2+, but it will also make it more modular and easier to maintain.

Before going further I recommend to look at the official AngularJS documentation page about Components, where their advantages and usage are well explained.

I would rather mention some tips about how to convert the old `ng-controller` oriented code to the new `component` oriented style.

Start breaking your your app into components

All the component-oriented apps have typically one or few components that include other sub-components. So why not creating the first component which simply will contain your app (or a big piece of it).

Assume that we have a piece of code assigned to a controller, named `UserListController`, and we want to make a component of it, which we'll name `UserListComponent`.

current HTML:

```
<div ng-controller="UserListController as listctrl">
    <ul>
        <li ng-repeat="user in myUserList">
            {{ user }}
        </li>
    </ul>
</div>
```

current JavaScript:

```
app.controller("UserListController", function($scope, SomeService) {

    $scope.myUserList = ['Shin', 'Helias', 'Kalhac'];

    this.someFunction = function() {
        // ...
    }

    // ...
}
```

new HTML:

```
<user-list></user-list>
```

new JavaScript:

```
app.component("UserList", {
    templateUrl: 'user-list.html',
    controller: UserListController
});

function UserListController(SomeService) {

    this.myUserList = ['Shin', 'Helias', 'Kalhac'];

    this.someFunction = function() {
        // ...
    }

    // ...
}
```

Note how we are no longer injecting `$scope` into the controller function and we are now declaring `this.myUserList` instead of `$scope.myUserList`;

new template file `user-list.component.html`:

```
<ul>
    <li ng-repeat="user in $ctrl.myUserList">
        {{ user }}
    </li>
</ul>
```

Note how we are now referring to the variable `myUserList`, which belongs to the controller, using `$ctrl.myUserList` from the html instead of `$scope.myUserList`.

That is because, as you probably figured out after reading the documentation, `$ctrl` in the template now refers to the controller function.

What about controllers and routes?

In case your controller was bound to the template using the routing system instead of `ng-controller`, so if you have something like this:

```
$stateProvider
  .state('users', {
    url: '/users',
    templateUrl: 'user-list.html',
    controller: 'UserListController'
  })
  // ..
```

you can just change your state declaration to:

```
$stateProvider
    .state('users', {
      url: '/',
      template: '<user-list></user-list>'
```

```
    })
    // ..
```

What's next?

Now that you have a component containing your app (whether it contains the entire application or a part of it, like a view), you should now start to break your component into multiple nested components, by wrapping parts of it into new sub-components, and so on.

You should start using the Component features like

- **Inputs** and **Outputs** bindings
- **lifecycle hooks** such as `$onInit()`, `$onChanges()`, etc...

After reading the Component documentation you should already know how to use all those component features, but if you need a concrete example of a real simple app, you can check this.

Also, if inside your component's controller you have some functions that hold a lot of logic code, a good idea can be considering to move that logic into services.

Conclusion

Adopting a component-based approach pushes your AngularJS one step closer to migrate it to the new Angular framework, but it also makes it better and much more modular.

Of course there are a lot of other steps you can do to go further into the new Angular 2+ direction, which I will list in the following examples.

Section 50.2: Introducing Webpack and ES6 modules

By using a ***module loader*** like Webpack we can benefit the built-in module system available in **ES6** (as well as in **TypeScript**). We can then use the import and export features that allow us to specify what pieces of code can we are going to share between different parts of the application.

When we then take our applications into production, module loaders also make it easier to package them all up into production bundles with batteries included.

Chapter 51: SignalR with AngularJs

In this Article we focus on "How to create a simple project using AngularJs And SignalR", in this training you need to know about "how create app with angularjs", "how to create/use service on angular" And basic knowledge about SignalR" for this we recommend https://www.codeproject.com/Tips/590660/Introduction-to-SignalR).

Section 51.1: SignalR And AngularJs [ChatProject]

step 1: Create Project

```
- Application
    - app.js
    - Controllers
        - appController.js
    - Factories
        - SignalR-factory.js
- index.html
- Scripts
    - angular.js
    - jquery.js
    - jquery.signalR.min.js
- Hubs
```

> SignalR version use: signalR-2.2.1

Step 2: Startup.cs And ChatHub.cs

Go to your *"/Hubs"* directory and Add 2 files *[Startup.cs, ChatHub.cs]*

Startup.cs

```
using Microsoft.Owin;
using Owin;
[assembly: OwinStartup(typeof(SignalR.Hubs.Startup))]

namespace SignalR.Hubs
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.MapSignalR();
        }
    }
}
```

ChatHub.cs

```
using Microsoft.AspNet.SignalR;

namespace SignalR.Hubs
{
    public class ChatHub : Hub
    {
        public void Send(string name, string message, string time)
        {
```

```
            Clients.All.broadcastMessage(name, message, time);
        }
    }
}
```

step 3: create angular app

Go to your *"/Application"* directory and Add *[app.js]* file

app.js

```
var app = angular.module("app", []);
```

step 4: create SignalR Factory

Go to your *"/Application/Factories"* directory and Add *[SignalR-factory.js]* file

SignalR-factory.js

```
app.factory("signalR", function () {
    var factory = {};

    factory.url = function (url) {
        $.connection.hub.url = url;
    }

    factory.setHubName = function (hubName) {
        factory.hub = hubName;
    }

    factory.connectToHub = function () {
        return $.connection[factory.hub];
    }

    factory.client = function () {
        var hub = factory.connectToHub();
        return hub.client;
    }

    factory.server = function () {
        var hub = factory.connectToHub();
        return hub.server;
    }

    factory.start = function (fn) {
        return $.connection.hub.start().done(fn);
    }

    return factory;
});
```

step 5: update app.js

```
var app = angular.module("app", []);

app.run(function(signalR) {
    signalR.url("http://localhost:21991/signalr");
});
```

> localhost:21991/signalr | **this is your SignalR Hubs Urls**

step 6: add controller

Go to your *"/Application/Controllers"* directory and Add *[appController.js]* file

```
app.controller("ctrl", function ($scope, signalR) {
    $scope.messages = [];
    $scope.user = {};

    signalR.setHubName("chatHub");

    signalR.client().broadcastMessage = function (name, message, time) {
        var newChat = { name: name, message: message, time: time };

        $scope.$apply(function() {
            $scope.messages.push(newChat);
        });
    };

    signalR.start(function () {
        $scope.send = function () {
            var dt = new Date();
            var time = dt.getHours() + ":" + dt.getMinutes() + ":" + dt.getSeconds();

            signalR.server().send($scope.user.name, $scope.user.message, time);
        }
    });
});
```

> signalR.setHubName("chatHub") | **[ChatHub] (public class) >** *ChatHub.cs*

> **Note:** do not insert *HubName* with upper Case, **first letter** is lower Case.

> **signalR.client()** | this method try to connect to your hubs and get all functions in the Hubs, in this sample we have "chatHub", to get "broadcastMessage()" function;

step 7: add index.html in route of directory

index.html

```
<!DOCTYPE html>
<html ng-app="app" ng-controller="ctrl">
<head>
    <meta charset="utf-8" />
    <title>SignalR Simple Chat</title>
</head>
<body>
    <form>
        <input type="text" placeholder="name" ng-model="user.name" />
        <input type="text" placeholder="message" ng-model="user.message" />
        <button ng-click="send()">send</button>
```

```
        <ul>
            <li ng-repeat="item in messages">
                <b ng-bind="item.name"></b> <small ng-bind="item.time"></small> : {{item.message}}
            </li>
        </ul>
    </form>

    <script src="Scripts/angular.min.js"></script>
    <script src="Scripts/jquery-1.6.4.min.js"></script>
    <script src="Scripts/jquery.signalR-2.2.1.min.js"></script>
    <script src="signalr/hubs"></script>
    <script src="app.js"></script>
    <script src="SignalR-factory.js"></script>
</body>
</html
```

> Result with Image

> User 1 (send and receive)
>
> User 2 (send and receive)

Chapter 52: angularjs with data filter, pagination etc

Provider example and query about display data with filter, pagination etc in Angularjs.

Section 52.1: Angularjs display data with filter, pagination

```
<div ng-app="MainApp" ng-controller="SampleController">
    <input ng-model="dishName" id="search" class="form-control" placeholder="Filter text">
    <ul>
        <li dir-paginate="dish in dishes |  filter : dishName | itemsPerPage: pageSize" pagination-
id="flights">{{dish}}</li>
    </ul>
    <dir-pagination-controls boundary-links="true" on-page-change="changeHandler(newPageNumber)"
pagination-id="flights"></dir-pagination-controls>
</div>
<script type="text/javascript" src="angular.min.js"></script>
<script type="text/javascript" src="pagination.js"></script>
<script type="text/javascript">

var MainApp = angular.module('MainApp', ['angularUtils.directives.dirPagination'])
MainApp.controller('SampleController', ['$scope', '$filter', function ($scope, $filter) {

    $scope.pageSize = 5;

    $scope.dishes = [
    'noodles',
    'sausage',
    'beans on toast',
    'cheeseburger',
    'battered mars bar',
    'crisp butty',
    'yorkshire pudding',
    'wiener schnitzel',
    'sauerkraut mit ei',
    'salad',
    'onion soup',
    'bak choi',
    'avacado maki'
    ];

    $scope.changeHandler = function (newPage) { };
}]);
</script>
```

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Aayushi Jain	Chapter 46
Abdellah Alaoui	Chapter 8
Adam Harrison	Chapters 2 and 3
Aeolingamenfel	Chapters 2 and 7
Ajeet Lakhani	Chapter 11
Alon Eitan	Chapters 3, 5, 6, 11, 15, 23 and 27
Alvaro Vazquez	Chapters 8 and 15
Aman	Chapter 30
Andrea	Chapter 9
Anfelipe	Chapter 11
Anirudha	Chapter 11
AnonDCX	Chapter 8
Aron	Chapter 3
Ashok choudhary	Chapter 44
Ashwin Ramaswami	Chapter 11
atul mishra	Chapter 11
AWolf	Chapters 3 and 30
Ayan	Chapter 3
BarakD	Chapter 5
Bon Macalindong	Chapters 2, 3 and 4
Bouraoui KACEM	Chapters 48 and 49
casraf	Chapter 6
CENT1PEDE	Chapters 3 and 25
chatuur	Chapter 2
Cosmin Ababei	Chapter 23
Dania	Chapter 11
Daniel	Chapter 6
Daniel Molin	Chapter 11
daniellmb	Chapter 11
David G.	Chapter 1
Deepak Bansal	Chapters 39 and 40
developer033	Chapter 7
DillonChanis	Chapter 3
Divya Jain	Chapters 3, 42 and 45
doctorsherlock	Chapter 23
DotBot	Chapter 15
Dr. Cool	Chapters 2, 3, 11 and 16
Durgpal Singh	Chapter 11
Ed Hinchliffe	Chapter 7
elliot	Chapters 10 and 22
Eric Siebeneich	Chapter 3
fantarama	Chapter 27
Faruk Yazıcı	Chapter 23
Filipe Amaral	Chapter 6
Flash	Chapter 8
fracz	Chapter 10
Gaara	Chapter 6
Gabriel Pires	Chapter 10
ganqqwerty	Chapter 28

garyx	Chapter 27
Gavishiddappa Gadagi	Chapter 8
georgeawg	Chapter 28
Gourav Garg	Chapter 41
Grundy	Chapters 2 and 11
gustavohenke	Chapter 7
H. Pauwelyn	Chapter 1
H.T	Chapter 18
Hadiiiiiiiiiiiiiiiiiii	Chapters 2 and 5
Hubert Grzeskowiak	Chapters 5 and 8
Igor Raush	Chapter 1
IncrediApp	Chapter 3
Istvan Reiter	Chapter 37
JanisP	Chapters 11 and 29
Jared Hooper	Chapter 2
jaredsk	Chapters 22 and 25
Jeroen	Chapter 1
jhampton	Chapter 11
Jim	Chapters 42 and 47
Jinw	Chapter 6
jitender	Chapter 34
jkris	Chapter 6
John F.	Chapters 5 and 28
kelvinelove	Chapter 3
Krupesh Kotecha	Chapter 3
Lex	Chapters 8 and 13
Liron Ilayev	Chapters 2, 3, 22 and 25
Lucas L	Chapters 14 and 35
M. Junaid Salaat	Chapter 5
m.e.conroy	Chapter 3
M22an	Chapter 11
Maaz.Musa	Chapter 1
Maher	Chapter 51
Makarov Sergey	Chapter 35
Manikandan Velayutham	Chapter 46
Mansouri	Chapter 5
Mark Cidade	Chapter 2
Matthew Green	Chapters 2, 7 and 11
MeanMan	Chapter 37
Mikko Viitala	Chapters 1, 3, 13, 27, 31, 32 and 33
Mitul	Chapter 3
MoLow	Chapters 9, 21 and 22
Muli Yulzary	Chapter 36
Nad Flores	Chapter 2
Naga2Raja	Chapter 3
Nemanja Trifunovic	Chapter 1
ngLover	Chapters 3 and 23
Nguyen Tran	Chapter 12
Nhan	Chapter 11
Nico	Chapters 7, 10 and 35
Nikos Paraskevopoulos	Chapter 17
Nishant123	Chapter 8
ojus kulkarni	Chapter 4
Omri Aharon	Chapter 9

Paresh Maghodiya	Chapter 52
Parv Sharma	Chapter 24
Pat	Chapter 19
pathe.kiran	Chapter 1
Patrick	Chapter 1
Phil	Chapter 23
Piet	Chapter 3
Prateek Gupta	Chapter 22
Praveen Poonia	Chapters 2 and 28
Pushpendra	Chapter 6
Ravi Singh	Chapter 5
redunderthebed	Chapter 3
Richard Hamilton	Chapters 1, 3 and 27
Rohit Jindal	Chapters 12, 13, 24, 27, 28 and 43
ronapelbaum	Chapter 10
Ryan Hamley	Chapter 18
RyanDawkins	Chapters 11 and 12
Sasank Sunkavalli	Chapter 14
Sender	Chapter 42
sgarcia.dev	Chapters 6, 11, 15 and 18
shaN	Chapter 12
shane	Chapter 27
Shashank Vivek	Chapter 12
ShinDarth	Chapter 50
Sunil Lama	Chapters 1 and 13
superluminary	Chapter 1
svarog	Chapters 3, 15, 20, 27 and 28
Syed Priom	Chapter 1
Sylvain	Chapters 19 and 26
Teqchiqe	Chapter 4
theblindprophet	Chapters 3 and 11
thegreenpizza	Chapter 2
timbo	Chapters 1, 3 and 27
Tomislav Stankovic	Chapter 3
Umesh Shende	Chapter 46
user3632710	Chapter 11
Ven	Chapter 1
Vinay K	Chapter 30
vincentvanjoe	Chapter 11
Vishal Singh	Chapter 3
Yasin Patel	Chapters 1 and 11
Yuri Blanc	Chapter 6
Ze Rubeus	Chapter 11
ziaulain	Chapter 38
zucker	Chapters 35 and 42

You may also like