AngularJS Handbook

Easy Web App Development

Rick L.

AngularJS Handbook

Easy Web App Development

By Rick L.

Table of Contents

Disclaimer

While all attempts have been made to verify the information provided in this book, the author does assume any responsibility for errors, omissions, or contrary interpretations of the subject matter contained within. The information provided in this book is for educational and entertainment purposes only. The reader is responsible for his or her own actions and the author does not accept any responsibilities for any liabilities or damages, real or perceived, resulting from the use of this information.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document.

Chapter 1- Overview of AngularJS

This framework works by performing an extension of what is provided in HTML. If HTML had been created for the purpose of the development of web-apps, then it would have been the angularJS. HTML is the best when it comes to declaration of static documents. However, it is not good for the declaration of dynamic views in our web applications. With Angular, the HTML vocabulary used in your application can be greatly extended. After doing that, you will get an environment which is easy to read, expressive, and faster to perform your development.

In this book, we will explore the various features of AngularJS which you as a developer can take advantage of so as to improve the look and feel of your application.

Chapter 2- AngularJS and HTML

As you are aware, AngularJS is used to extend the features of HTML. Most of you are aware of how the standard HTML can be used. However, AngularJS is used to extend the features of this. In this chapter, we want to create an HTML page, and then turn the code written in HTML into a template which will be used by the AngularJS so as to display the same data dynamically. Only basic information will be added to the HTML page.

Here is the first code for the web page:

```
<ul>
<li>
    <span> Motorola </span>
<p>
    I really like Motorola phones.
```

** **

**
**

<li>

Nokia are the best in Wi-Fi connection

** **

I really like to make use of devices.

** **

**
**

**
**

You can continue with the addition of static HTML to the above file. An example of this is given below:

** The number of available phones is: 2 **

In the example, we have just used the static HTML so as to render the content of the web page which we have just created. With AngularJS, the same content can be rendered dynamically.

Our aim is to make the page we have created dynamic. A test case should also be implemented so that the controller that we create is tested. In AngularJS, the MV model (Model-View-Controller) is used for the purpose of organizing the structure of the application. With this, the code is decoupled, and the concept of separation of concerns is implemented. The model, view, and the controller will be added in this app. What happens in Angular is that the model is projected into a template via the view. This is an indication that in case something is changed, all of the binding points will also be updated so that the view is updated. The view is constructed from the model in Angular.

Consider the example given below:

```
<html ng-app=“myApp”>
<head>
…
<script src=“myapps/angular/angular.js”></script>
<script src=“js/controllers.js”></script>
</head>
  <body ng-controller=“mply “Repeat”?PnListCtrl”>
<ul>
<li ng-repeat=“model in phones”>
<span>{{model.name}}</span>
<p>{{model.snippet}}</p>
    </li>
  </ul>
</body>
</html>
```

What you should have noticed is that the hard coded phone list has been replaced by AngularJS expressions and an ngRepeat directive.

The controller for the app should just be a constructor function which will take the parameter “$scope.” This is shown below:

```
var myApp = angular.module(‘myApp’, []);
   myApp.controller(‘PhoneListCtrl’, function ($scope) {
$scope.phones = [
   {‘name’: ‘Motorola,
‘snippet’: ‘ I really like Motorola phones.
.’},
  {‘name’: ‘ Nokia are the best in Wi-Fi connection ‘,
    ‘snippet’: ‘Their devices are really liked by many.’},
   {‘name’: ‘MOTOROLA XOOM™’,
  ‘snippet’: ‘ I really like to make use of devices.
.’}
];
});
```

What we have done is that we have declared the controller, and then we have finally linked it to the module.

Tests

In AngularJS, the controller and the view are separated. With this, the ode becomes testable even during the time of development. In case the controller is available in the global space, then we can mock the "*scope*" object so as to instantiate it. This is shown in the example given below:

```
describe('PnListCtrl', function(){
   it(' a "phones" model with 2 phones should be created', function() {
var scope = {},
   ctrl = new PnListCtrl(scop);
   expect(scop.phones.length).toBe(2);
});
});
```

The non-global controllers should then be created. No one will want to have their controller function to be in their global namespace. Instead, most people will want it to be registered via the anonymous constructor function of the module which we have just created.

In this case, we will be proved by the service "$controller" from AngularJS. This will be tasked with retrieving the controller by its name. Consider the example given below, which shows how this can be done:

```
describe('PnListCtrl', function(){

   beforeEach(module('pncatApp'));

   it('you should create "phones" model with the 2 phones',
inject(function($controller) {

var scope = {},

   ctrl = $controller('PnListCtrl', {$scope:scope});

   expect(scope.phones.length).toBe(2);

}));

});
```

What we have done is that we have instructed the Angular to load each module before any of the tests is done. The service "*$controller*" should then be injected into our service. The service has also been used for creation of a new "*PnListCtrl.*" The instance is then used for the purpose of verification.

The Experiments

Another binding can be added to the file. This is shown below:

** The available number of phones: {{phones.length}} **

A new model property can be created in the controller and then bound from the template. An example of this is shown below:

$scope.name = “John”;

The new binding can be added to the file as follows:

** Hi, {{name}}! **

You can now refresh your browser and then observe what happens. The unit test for the controller should then be updated in the “*./test/unit/controllersSpec.js*” so that the new change is reflected.

A repeater for the creation of a simple table should then be created in the file. This can be

implemented as follows:

```
<table>
    <tr><th>the row number</th></tr>
    <tr ng-repeat="j in [0, 1, 2, 3, 4, 5, 6, 7]"><td>{{j}}</td></tr>
</table>
```

The list should then be modified to the following:

```
<table>
  <tr><th>the row number</th></tr>
  <tr ng-repeat="j in [0, 1, 2, 3, 4, 5, 6, 7]"><td>{{j+1}}</td></tr>
</table>
```

The *"ng-repeat"* can be used for the creation of another table with some other specifications. That is also very powerful in Angular. At this point, you will have an app working dynamically and it has the model, the view, and the controller located separately.

Chapter 3- XHRs

You should not always handle small and hard-coded data in your application. We should be able to fetch large data sets from our AngularJS servers by the use of the pre-built services named "*$http.*"

We will have the file "*app/phones/devices.json,*" which has the list of phones that we have. These will be stored in a JSON format. A sample part of the file looks as follows:

[

{

"age": 15,

"id": "motorola ",

"name": "Motorola DEFY\u2122 with MOTOBLUR\u2122",

"snippet": "Do you enjoy using Motorola devices and phones?"

…

},

…

]

Controller

The "*$http"* service for Angular will be used in the controller for making an HTTP request to the web server so as to fetch data from the file "*app/phones/devices.json.*"This is just one of the ways how this can be handled or the processing of operations related to web apps. With AngularJS, these services are injected where they are needed.

The code for the controller should be as shown below:

```
var myApp = angular.module('myApp', []);
    myApp.controller('PnListCtrl', function ($scope, $http) {
    $http.get('phones/devices.json').success(function(data) {
    $scope.phones = data;
});
    $scope.orderProp = 'age';
});
```

For a service to be used in Angular, the names of the dependencies which are needed have to be defined as arguments to the controller constructor function. This is shown below:

myApp.controller(‘PnListCtrl’, function ($scope, $http) {…}

The dependency constructor in AngularJS will provide the services to our controller once it has been launched. The process of creating any of the transitive dependencies which the service may have is also taken care of. Consider the example given below, showing how this can be done:

```
var myApp = angular.module(‘myApp’, []);

myApp.controller(‘PnListCtrl’, [‘$scope’, ‘$http’,

function ($scope, $http) {

    $http.get(‘phones/devices.json’).success(function(data) {

    $scope.phones = data;

});


  $scope.orderProp = ‘age’;

}]);
```

The test can be done as follows:

```
describe(‘myApp controllers’, function() {
```

```
describe('PnListCtrl', function(){

    var scope, ctrl, $httpBackend;

// Loading our app module definition before the test.

    beforeEach(module('myApp'));

// The injector will ignore the leading and the trailing underscores here (i.e. _$httpBackend_).


// This will allow us to inject the service but also attach it to the variable

// with a name which is same as the service so as to avoid a name conflict.

beforeEach(inject(function(_$httpBackend_, $rootScope, $controller) {


    $httpBackend = _$httpBackend_;

    $httpBackend.expectGET('phones/devices.json').

respond([{name: 'Motorola'}, {name: 'Motorola Phone'}]);

scope = $rootScope.$new();

    ctrl = $controller('PnListCtrl', {$scope: scope});

}));
```

Notice the two helper functions in the above test case. These can be used for configuration and accessing the injector.

Consider the code given below:

```
it('You should create "phones" model having 2 phones which have been fetched from the xhr', function() {

    expect(scope.phones).toBeUndefined();

    $httpBackend.flush();

    expect(scope.phones).toEqual([{name: 'Motorola'},

    {name: 'Motorola Phone'}]);

});
```

We want to perform a verification on our model which does not exist on the scope before we have received the response. After calling the function "*$httpBackend.flush(),*" the request queue will be flushed into the browser. The promise which is returned by the service "$http" will be resolved with our resolved response. We will make the assertions, and then verify whether the model belongs to the scope. We can then verify whether the "*orderProp"* has been set correctly. This is shown below:

```
it('the default value should be set for the orderProp model', function() {

    expect(scope.orderProp).toBe('age');

});
```

In the karma tab, the following output should be observed:

Chrome 22.0: Executed 2 of 2 SUCCESS (0.028 secs / 0.007 secs)

“*<pre>{{phones | filter:query | orderBy:orderProp | json}}</pre>” c*an be added at the bottom of the file so that the available phones can be listed or shown in a JSON format. The following can be added to the callback:

$scope.devices = data.splice(0, 5);

Chapter 4- Links & Images

You should know how to add thumbnail images to the devices that you have in your list. Links should also be added to these.

Note that in the JSON file, the links for the devices and their URLs are available, as well as unique IDs. The URLs should be pointing to the URL "*app/img/devices/.*"

Consider the code snippet given below:

[

{

…

"id": "motorola ",

"imageUrl": "img/phones/motorola.0.jpg",

"name": "Motorola PHONE ",

…

},

…

]

The template should be as follows:

…

<ul class=“phones”>

<li ng-repeat=“device in phones | filter:query | orderBy:orderProp” class=“thumbnail”>

<img ng-src=”{{device.imageUrl}}”>

{{device.name}}

** {{device.snippet}} **

**
**

**
**

…

The double-binding curly brae has been used in the attribute values for “*href*” so as to allow for the links to be dynamically generated. In the second step, the binding *“{{device.name}}”* was added as an element of content.

The directive “*ngRs”* has been used so as to add an image next to each device. The test

can be implemented as follows:

```
…
it(‘ device specific links should be rendered ‘, function() {
  var myquery = element(by.model(‘query’));
    myquery.sendKeys(‘nexus’);
element.all(by.css(‘.phones li a’)).first().click();
browser.getLocationAbsUrl().then(function(url) {
    expect(url).toBe(‘/phones/motorola’);
});
});
…
```

We need to perform a verification of whether the app is generating the right links for the device views which you may need to implement. The directive “*ng-rsc”* can be replaced with the plain “*src.”*

Chapter 5- Working with Animations

Sometimes, you might need to enhance your application by adding some animations to it. This can be done by use of both CSS and JavaScript. For the animations to be enabled throughout the application, we have to use the "*ngAnimate"* module. The common "*ng*" directives have to be used for the purpose of triggering the hooks for the animations we are tapping into. This should be done automatically. Once the animation has been found, it is executed in between the standard DOM element which has been issued to the element at that given time. The "*ngRepeat"* and "*ngClass"* can be used for doing this, just by removing or adding elements to them.

The AngularJS module "*ngAnimate"* will provide us with the functionality for performing the animation. This module has been distributed separately from our core Angular framework. For the purpose of performing the extra animations, we will use the jQuery.

We now need to install the dependencies to be used on the client-side of the application. Bower can be used for this purpose. We need to update the configuration file "*bower.json*" so that it includes the new dependency. This can be done as follows:

{

"name": "angular-seed",

"description": "A project for AngularJS starting",

"version": "0.0.0",

“homepage”: “https://mysite.com/angular/angular-project”,

“license”: “MIT”,

“private”: true,

“dependencies”: {

“angular”: “1.4.x”,

“angular-mocks”: “1.4.x”,

“jquery”: “~2.1.1”,

“bootstrap”: “~3.1.1”,

“angular-route”: “1.4.x”,

“angular-resource”: “1.4.x”,

“angular-animate”: “1.4.x”

}

}

All the elements installed in the above step should be compatible, otherwise, there will be trouble. The bower has also been instructed to install the jQuery. However, this cannot be said to be an Angular library, but it is a standard jQuery library.

Instruct the bower to download and install the dependencies by executing the following command:

npm install

Note that incompatibility might occur in case there is a new version of Angular which has been released after the last time you executed the above command. To solve this problem, you just have to delete the folder for "*app/bower_components*" before having to execute the above command.

The Template

We have to make some changes to the HTML template so as to link the asset files defining our animations and the file "*angular-animate.js.*" The module for animation, that is, "*ngAnimate,"* is defined in this file, and this has the code which is responsible for making the application sense the animation effect.

The following changes have to be made to the file:

…

<!—this is for the CSS Transitions and/or the Keyframe Animations —>

<link rel="stylesheet" href="css/animations.css">

…

<!—the jQuery will be used for JavaScript animations (this should included before angular.js) —>

<script src="bower_components/jquery/dist/jquery.js"></script>

…

<!—this is the required module for enabling animation support in the AngularJS —>

<script src="bower_components/angular-animate/angular-animate.js"></script>

<!—this is for JavaScript Animations —>

<script src=“js/animations.js”></script>

…

For those who are using Angular version 1.4, make sure that you are using jQuery 2.1 and above. Any version below this will not be compatible with Angular 1.4. This will mean that the animation will not be detected, or it may work in a manner which is not expected.

The animations can be implemented in either the CSS code (“*animations.css*”) or in the JavaScript code (“*animations.js”*). We need to begin by creating a new module which will use the “*ngAnimate”* module as a dependency in the same way we did it in the “*ngResource.”*

The “*animations.js”* file should be as follows:

angular.module(‘myAnimations’, [‘ngAnimate’]);

// …

// this module will be used for definition of the animations later

// …

The module should then be attached to the application module. This is shown below:

```
// …
angular.module('myApp', [
'ngRoute',
'myAnimations',
'myControllers',
'myFilters',
'myServices',
]);
// …
```

At this moment, our module will be aware.

Animation of “ngRepeat” with the CSS Transition Animations

The CSS animations should be added to the “*ngRepeat*” directive which is present in the web page with our devices. We should begin by adding an extra CSS class to the element which has been repeated so that it can be hooked with the CSS animation code. This is shown below:

<!—

Let us change the repeater HTML so as to include a new CSS class

That we will use later for the animations:

—>

<ul class=“phones”>

<li ng-repeat=“device in phones | filter:query | orderBy:orderProp”

class=“thumbnail device-listing”>

<img ng-src=”{{device.imageUrl}}”>

{{device.name}}

** {{device.snippet}} **

**
**

**
**

Note how the CSS class has been added. This is needed in the HTML file so that the animation may work.

The CSS code for animation should be as follows:

```
.device-listing.ng-enter,
.device-listing.ng-leave,
.device-listing.ng-move {
    -webkit-transition: 0.5s linear all;
    -moz-transition: 0.5s linear all;
    -o-transition: 0.5s linear all;
    transition: 0.5s linear all;
}
.device-listing.ng-enter,
.device-listing.ng-move {
    opacity: 0;
    height: 0;
    overflow: hidden;
}
```

```
.device-listing.ng-move.ng-move-active,
.device-listing.ng-enter.ng-enter-active {
    opacity: 1;
    height: 120px;
}
.device-listing.ng-leave {
    opacity: 1;
    overflow: hidden;
}
.device-listing.ng-leave.ng-leave-active {
    opacity: 0;
    height: 0;
    padding-top: 0;
    padding-bottom: 0;
}
```

The CSS class for “*device-listing”* has been combined together with the animation hooks which will occur when the items have been inserted into and then removed from the list.

The class “*ng-enter”* will be applied to the element after a new device has been added to the list and then displayed on the page. The class “*ng-move”* will be applied after the elements have been moved around the list. The class “*ng-leave”* will be applied once the

elements have been removed from the list.

Animation of “ngView”

We now need to add an animation for transitions between our route changes in the “*ngView.”* A new CSS class has to be added to the HTML as we did in our previous example. Other than adding it to the “*ng-Repeat*” element, we will add it to the “*ngView.”* This means that the HTML code has to be changed a bit. This will mean that the animation will be more controlled between the changes in the view.

The “*index.html”* file should be as follows:

```
<div class=“view-container”>
    <div ng-view class=“view-frame”></div>
</div>
```

The “*view-container”* CSS class has been used to nest the “*ng-view”* directive inside the parent element. The class will add the style “*position: relative*” so that the positioning of the directive “*ng-view*” is relative to the parent as the transitions are animated.

The CSS for the transition should be added to the file “*animation.css.”* This is shown below:

```
.view-container {
    position: relative;
}
.view-frame.ng-enter, .view-frame.ng-leave {
    background: white;
    position: absolute;
    top: 0;
    left: 0;
                                        right: 0;
                                                }
.view-frame.ng-enter {
    -webkit-animation: 0.5s fade-in;
    -moz-animation: 0.5s fade-in;
    -o-animation: 0.5s fade-in;
    animation: 0.5s fade-in;
    z-index: 100;
}
.view-frame.ng-leave {
    -webkit-animation: 0.5s fade-out;
    -moz-animation: 0.5s fade-out;
    -o-animation: 0.5s fade-out;
```

```
    animation: 0.5s fade-out;
    z-index:99;
}
@keyframes fade-in {
    from { opacity: 0; }
    to { opacity: 1; }
}
@-moz-keyframes fade-in {
    from { opacity: 0; }
    to { opacity: 1; }
}
@-webkit-keyframes fade-in {
    from { opacity: 0; }
    to { opacity: 1; }
}
@keyframes fade-out {
    from { opacity: 1; }
    to { opacity: 0; }
}
@-moz-keyframes fade-out {
    from { opacity: 1; }
```

```
    to { opacity: 0; }
}
@-webkit-keyframes fade-out {
    from { opacity: 1; }
    to { opacity: 0; }
}
/* the vendor-prefixes should not be forgotten */
```

Animation of the “ngClass” with Javascript

We need to add another animation to our application. A nice thumbnail will be observed in the “*device-list.html”* page. Once the thumbnail has been clicked, the image will change. The CSS class has to be changed so that we can ensure that the displaying of the images is accurate. This is shown below:

```
.phone-images {
    background-color: white;
    width: 450px;
    height: 450px;
    overflow: hidden;
    position: relative;
    float: left;
}
…

img.device {
    float: left;
    margin-right: 3em;
    margin-bottom: 2em;
```

```
    background-color: white;
    padding: 2em;
    height: 400px;
    width: 400px;
    display: none;
}
img.device:first-child {
    display: block;
}
```

You are thinking that we will create an animation which is CSS enabled. However, we can do this by use of the "*animation()*" module of JavaScript. This is shown in the example given below:

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

    myAnimations.animation('.device', function() {
    var animUp = function(element, cName, done) {
if(cName != 'active') {
    return;
}
```

```
element.css({
    position: ‘absolute’,
    top: 500,
    left: 0,
    display: ‘block’
});
jQuery(element).animate({
    top: 0
}, done);
return function(cancel) {
if(cancel) {
    element.stop();
}
};
}
var animDown = function(element, cName, done) {
if(cName != ‘active’) {
    return;
}
element.css({
    position: ‘absolute’,
```

```
    left: 0,
    top: 0
});
jQuery(element).animate({
    top: -500
}, done);
return function(cancel) {
if(cancel) {
    element.stop();
}
};
}
return {
    addClass: animUp,
    removeClass: animDown
};
});
```

As you have noticed, the animation has been implemented by use of jQuery. With both Angular and JavaScript, we do not require the jQuery so as to perform the animation. Writing your own animation library in JavaScript will be difficult.

Chapter 6- $watch() , $digest(), and $apply()

These methods are widely used in AngularJS. You should learn how to use these in Angular.

After the creation of data binding somewhere in one's view to a variable on the object "*$scope,*" a "*watch*" will be internally created by the Angular. With the watch, it will mean that it will watch what is contained in the "*$scope*" object. This will be an indication that the framework will be watching the variable. To create watches, we use the function *"$scope.$watch()."*

In most cases, the functions *"$scope.$digest()"* and *"$scope.$watch()"* can be called on your behalf, but in some other cases, you might have to call them yourself. This is why you should know and understand how these work.

The function "*$scope.$apply*" is used for execution of a particular code, and then the function "*$scope.$digest()"* is called after that. This will have all of the watches checked and all of the corresponding functions for the watch listener called. The function *"$apply()"* is good when it comes to integration of the AngularJS code with any other code.

$watch()

The function *"$scope.watch()"* is used for creation of a watch of a particular variable. Once a watch has been registered, two parameters have to be passed to the "*$watch()*" function. These two parameters include the following:

- A value function
- A listener function

Consider the example given below showing how this can be done:

```
$scope.$watch(function() {},
   function() {}
);
```

The first function represents the value function, while the second function represents the listener function. With the value function, the value which is being watched will be returned. It is then the work of the AngularJS to check the value which has been returned against the one that was returned by the watch function last time. AngularJS has changed the way in which checking of whether a value was returned can be done. An example of this is shown below:

```
$scope.$watch(function(scope) { return scope.data.myVariable },
```

```
    function() {}
);
```

With the above function, the value "*scope.data.myVariable*" will be returned by the value function. In case the value for this variable is changed, then a different value will have to be returned. The listener function will then be called by the AngularJS.

If the value has been changed, the listener function will try to do whatever that it can do. By use of this value, the value function will be in a position to access the "$scope" together with its variables. The global variables can also be watched by the value function in case it is needed, but most often, the variable "*$scope*" will be watched.

If the value is changed, the listener function can do whatever that it needs to do. For some, they might need to change the content of the variable while with others, they might need to change the content of an HTML element. Consider the example given below:

```
$scope.$watch(function(scope) { return scope.data.myVariable },

function(nValue, oValue) {
    document.getElementById("").innerHTML =
    "" + newValue + "";
}
);
```

The above example will set the inner HTML of the HTML element to the new value of the variable, and this has been embedded in the *“b”* element for the value of making the content bold.

$digest()

The function *"$scope.$digest()"* will iterate through the watches in the object "*$scope*" and the child "*$scope*" object. When the function" *$digest()"* has iterated through the watches, the value function for each watch will be called. If the value returned by our value function is found to be different to the value which it had returned last time that it was called, the listener function for the watch will then be called.

Whenever AngularJS finds it necessary, it just calls the *"$digest()"* function. Situations may be found in which the AngularJS does not call the *"$digest()"* function. To solve this problem, just call the function *"$scope.$digest()."*

$apply()

The function *"$scope.$apply()"* will take a function as a parameter which has been executed, and then the function "$scope.$apply()" will internally be called. This is a way for you to ensure that all of the watches have been checked. All of the data bindings will then be refreshed.

Consider the example given below:

```
$scope.$apply(function() {

$scope.data.myVariable = "A new value";

});
```

Once a function has been passed to this function as a parameter, the value of *"$scope.data.myVar*" will be changed. Once the function has exited, the function *"$scope.$digest()"* will be called by the AngularJS, meaning that the watches will be checked for the changes in the watched values.

Consider the example given below, which shows how these functions can be used in AngularJS:

```
<div ng-controller="appController">

{{data.time}}

<br/>

<button ng-click="updateTime()">update time - ng-click</button>


<button id="upTimeButton"  >change time</button>

</div>

<script>

var module       = angular.module("myapplication", []);

var appController1 = module.controller("appController", function($scope) {


$scope.data = { time : new Date() };

$scope.updateTime = function() {

$scope.data.time = new Date();

}

document.getElementById("upTimeButton")

.addEventListener('click', function() {

console.log("change time clicked");

$scope.data.time = new Date();

});

});

</script>
```

With the above example, the variable "*$scope.data.time*" will be bound to an interpolation directive which will merge the variable value to a HTML page. With this binding, a watch will be internally created in the variable *"$scope.data.time.*"

We have created two buttons for our application. The first button has a listener of type "*ng-click"* attached to it. Clicking of this button will call the function *"$scope.updateTime(),"* and then the AngularJS updates the bindings by calling the function *"$scope.$digest()."*

A JavaScript event listener has been attached to the second button. This listener will be executed once the second button has been clicked. The only difference between the two listeners is that with the one for the second button, the data bindings are not updated once the second button has been clicked. The reason behind this is that the function *"$scope.$digest()"* will not be called after the execution of the event listener for the second button. This means that the clicking of the second button will have the effect of updating the time in the variable *"$scope.data.time,"* but this new time will not be displayed for visibility.

For us to solve this problem, the call for "$scope.$digest()" can be added to the last line of the event listener. This is shown in the example given below:

document.getElementById("upTimeButton")

.addEventListener('click', function() {

console.log("change time clicked");

$scope.data.time = new Date();

$scope.$digest();

});

The addition of the above call has been done in the last line of the program. That is how it should be done.

Instead of doing the above, that is, calling the *"$digest()"* function inside the event listener, the "*apply()"* function can also be used as shown below:

document.getElementById("upTimeButton")

.addEventListener('click', function() {

$scope.$apply(function() {

console.log("change time clicked");

$scope.data.time = new Date();

});

});

The function "*$scope.$apply()"* has been called from within the event listener for the button, and the update of the variable *"$scope.data.time*" has been passed as a parameter to the function *"$apply()."* Once all of the function "$apply()" has finished, AngularJS

will call the function *“$digest()”* internally so that all of the data bindings are updated.

Chapter 7- The Hierarchy of “$scope”

The object “$scope” for AngularJS, which is used by views, is organized into an hierarchy. We begin with the root scope, which is then made up of one or more children. Each of the available views is made up of its own scope, which is just the child of the root scope. This means that whatever variable that the view controller has set on the variable “*$scope*” will be invisible to the controllers.

Consider the AngularJS code given below as an example:

```
<!DOCTYPE html>

<html lang=“en”>

<head>

<script src=“https://ajax.googleapis.com/ajax/libs/angularjs/1.2.5/angular.min.js”>
</script>


</head>

<body ng-app=“myapplication”>

<div ng-controller=“appController1”>

{{data.theVariable}}

</div>

<div ng-controller=“appController2”>
```

```
{{data.theVariable}}

</div>

<script>

var module      = angular.module(“myapplication”, []);

var appController1 = module.controller(“appController1”, function($scope) {

$scope.data = { theVariable : “first value”};

});

var appController2 = module.controller(“appController2”, function($scope) {

$scope.data = { theVariable : “second value”};

});

</script>

</body>

</html>
```

In the above example, we have used two views, and each of the views has its own controller. Each controller will set the “*data.theVariable*” variable to a different value.

Once the program has been executed, you will get the following hierarchy:

Root $scope

- $scope for appController 1
- $scope for appController 2

As shown above, the two views have used different scope objects. With the example, two different values will also be written for the *“{{data.theVar}}”* data bindings inside our two views. Different values will also be set for the variable “*data.theVariable*” in the “*$scope*” object.

Chapter 8- Refactoring AngularJS Apps

Directives which are component-based are increasingly becoming popular in AngularJS. The reason for this is the ease of use associated with components. However, most apps for Angular which are in use today are not based on components.

For Angular2 users, components can be used as follows:

< confirmation

[message]=“‘Start firing?’”

(ok)=“startFiring()”>

</ confirmation>

As shown in the above example, a component is just an element in custom HTML, and it matches a component name which has been defined somewhere else. Each component has its inputs and outputs, and these are supplied as properties to the element.

The above component can also be defined as shown below:

```
@Component({
selector: ‘confirmation’,
properties: [‘message’],
events: [‘ok’]
})
@View({
template: `
<div>
{{message}}
    <button (click)=“ok()”>OK</button>
    </div>
`})
class Confirmation {
    okEvents = new EventEmitter();
    ok() {
    this.okEvents.next();
}
}
```

An interface boundary definition is used for specification of the inputs and outputs which the component can work with. The UI template has been used for definition of the user

interface for our users. The component logic has been used behind the UI template to define the behavior of the component and what it does when the user is interacting with it. The definition of a component as a directive can be done as shown below:

```
@Component({
    selector: ‘confirmation’,
    properties: [‘message’],
    events: [‘ok’]
})
@View({
template: `
    <div>{{message}}
    <button (click)=“ok()”>OK</button>
</div>
`
})
class Confirmation {
    okEvents = new EventEmitter();
    ok() {
    this.okEvents.next();
}
}
```

The above code is completely different from what we had before.

Refactoring can easily be done or used in AngularJS. We want to give an example of how this can be done:

Consider the code given below for the file "*index.html"*:

** **

** **

For the above to be replaced, we will have to create a directive for "*bar."* This will use the inherited scope and the template "*bar.htm."*

The code for "*bar_directive.js"* should be as follows:

module.directive('bar', function() {

return {

scope: true,

templateUrl: ‘bar.html’

};

});

For this to work effectively, both the directive and the template have to be co-located and their names should also match. This is shown below:

components/foobar/

- **bar_directive.js**
- **bar.html**

The original “*ng-inlude”* should then be replaced with the new directive. This is shown below:

** **

** **

<footbar></footbar>

After the addition of the above component directive, that is, “*footbar,”* the template itself

will look as shown below:

<div ng-controller=“FoobarCtrl as c”>

** {{c.getText()}} **

** **

We then want to move and then rename the file which has the FootCtrl so that it becomes co-located with our component directive. This is shown below:

components/foobar/

- **bar_directive.js**
- **bar_controller.js**
- **bar.html**

The controller should then be used in the directive, and then the same ctrl alias that our “*ng-controller”* will be used. The file “*bar_diretive.js”* should be as follows:

module.directive(‘foobar’, function() {

return {

scope: true,

controller: ‘FoobarCtrl’,

controllerAs: ‘ctrl’,

templateUrl: ‘bar.html’

};

});

The wrapper “ng-controller” can now be removed from the template. This is shown below:

<div ng-controller=“FooBarCtrl as c”>

** {{c.getText()}} **

** **

One can also choose to keep this element for the purpose of styling, and then the attribute “*ng-controller*” is removed.

Consider the example given below:

<div ng-controller=“FoobarCtrl as c”

class=” foo-bar”>

** {{c.getText()}} **

** **

In the above example representing a template, we have an "*ng-controller.*" It has been used to instantiate a "FoobarCtrl" and then alias it as "c."

We now need to create a new directive. This should then create an inherited scope, and then use the "FoobarCtrl" and then alias it as "c." This is shown below:

```
module.directive('bar', function() {
return {
    scope: true,
    controller: 'FoobarCtrl',
    controllerAs: 'c'
};
});
```

Our new directive and the controller in existence will be moved to the same location. The two should also have the same prefix. This is shown below:

components/foobar/

- **bar_directive.js**
- **bar_controller.js**

Finally, in the original template, the "*ng-controller"* can be replaced with an instance of the component "*foobar.*" This is shown below:

<div ng-controller="FoobarCtrl as c"

class="foo-bar">

<foobar class="foo-bar">

** {{c.getText()}} **

</foobar>

** **

This is what we have for now:

<div ng-controller="FoobarCtrl as c"

class="foo-bar">

<foobar class="foo-bar">

** {{c.getText()}} **

</foobar>

** **

A template file should then be created for the component directive. This is shown below:

components/foobar/

- **bar_directive.js**
- **bar_controller.js**
- **bar.html**

The new template should then be referenced from the directive. This can be done as follows:

```
module.directive('foobar', function() {
return {
scope: true,
controller: 'FoobarCtrl',
controllerAs: 'c',
templateUrl: 'bar.html'
};
```

});

We will then use what was contained in the directive element so as to populate the new template. This is shown below:

** {{c.getText()} **

The directive element will itself become empty. The "*index.html*" file will then be as follows:

<foobar>

** {{c.getText()}} **

</foobar>

After the "*ng-controller*" has been replaced with the directive, the original template will be left with the following:

<foobar>

<h2>{{c.getTitle()}}</h2>

** {{mainC.getText()}} **

</foobar>

Transclusion can be used for the purpose of supporting both of the cases. We will begin by creating a template file for our component directive. This is shown below:

components/foobar/

- **bar_directive.js**
- **bar_controller.js**
- **bar.html**

The new template can then be referenced from our directive. At the same time, transclusion will also be enabled, as shown below:

module.directive(‘foobar’, function() {

return {

transclude: true,

scope: true,

controller: ‘FoobarCtrl’,

controllerAs: ‘c’,

templateUrl: ‘bar.html’

};

});

The population of the template file will be done by use of a part of the original template which is needed to be put inside the component. The directive “ng-transclude” will be added for the other part. The “*bar.html”* file should have the following ode:

<h2>{{c.getTitle()}}</h2>

**<p ng-transclude> **

In our directive element, we will leave the part which is to be transcluded. This is shown below:

<foobar>

<h2>{{c.getTitle()}}</h2>

** {{mainC.getText()}} **

</foobar>

Consider the next example.

We have a “<ul>” element for listing some data. This is shown below:

** **

<h1>Foobars</h1>

<ul>

<li ng-repeat=“foo in c.bars”>

****

{{file.text}}

**
**

**
**

** **

We need to perform the listing into its own component. A directive and a template file should be created for the component. This should be called in the directive as shown below:

components/foobar_list/

- **foobar_list_directive.js**
- **foobar_list.html**

The directive will create an inherited scope, and then reference the template which has been created. This is shown below:

```
module.directive('foobarList', function() {
return {
    scope: true,
    templateUrl: 'list.html'
};
});
```

The template will be populated by copying the HTML file which has been populated from our original template. This is shown below:

```
<ul>
<li ng-repeat="foo in c.bars">
<img ng-src="foo.imageUrl"/>
{{file.text}}
</li>
</ul>
```

A new instance of this component can then be used in the original template. It will replace the markup which has been created. This is shown below:

```
<div>

<h1>Foobars</h1>

<ul>

<li ng-repeat="foo in c.bars">

<img ng-src="foo.imageUrl"/>

{{file.text}}

</li>

</ul>

<foobar-list></foobar-list>

</div>
```

An example can also be created which makes use of transclusion. An example of this is given below:

```
<div>

<h1>Foobars</h1>

<ul>

<li ng-repeat="foo in c.bars">
```

<h3>

{{mainC.itemHeader}}

</h3>

****

{{file.text}}

**
**

**
**

** **

The "*<ul>*" element from our previous example has some content coming from the outside, and transclusion can be used for the purpose of both of these. In the above example, we have used an item header which has been provided by "*mainC*"

When the listing is extracted into its own component, a directive and a template should then be created for that file. This is shown below:

components/foobar_list/

- **bar_list_directive.js**
- **bar_list.html**

The directive will create an inherited scope, and the newly created template file will then

be referenced. Transclusion will also be enabled.

The code for the file "*bar_list_directive.js*" should be as follows:

```
module.directive('foobarList', function() {
return {
    transclude: true,
    scope: true,
    templateUrl: 'bar_list.html'
};
});
```

The population of the template will be done by copying the HTML which has been extracted from our original template. The part for the item header will be replaced with an "*ng-transclude*" element.

The code for "*bar_list.htm"* should be as follows:

```
<ul>
<li ng-repeat="foo in c.bars">
```

<ng-transclude></ng-transclude>

****

{{file.text}}

**
**

**
**

A new instance of this component can then be used in our new template. It will replace the markup which has been extracted, but an item header will be left in place and then transclusion will be done to each item. This is shown below:

** **

<h1>My Foobars</h1>

<ul>

<li ng-repeat=“foo in c.bars”>

<h3>

{{mainC.itemHeader}}

</h3>

****

{{file.text}}

**
**

**
**

<foobar-list>

<h3>

{{mainC.itemHeader}}

</h3>

</foobar-list>

** **

We now have a component template. Two external references can be seen in it, and these reference the controller "*mainC.*" Note that this is not the controller for the component, but it is just another controller which has been defined on the upper or higher level of our hierarchy. This is shown below:

** **

** **

{{c.getText(mainC.getLanguage())}}

** **

** **

The external references should be removed one by one, and the code should be kept working all the time.

The expression "*mainC.isFooBarVisible()*" has to be chosen first. It refers to calling a method which is located on the parent controller, and we have used the result to control

whether or not the element is visible. For the external reference to be replaced, a binding to the component directive has to be introduced named "*visible.*"

The code for the "*bar_diretive.js*" should be as follows:

```
module.directive('foobar', function() {
return {
scope: true,
controller: 'FoobarCtrl',
controllerAs: 'c',
bindToController: {
visible: '='
},
templateUrl: 'bar.html'
};
});
```

The above code shows that the component has an input named "*visible,*" and the user should supply the component to it. You should also have noticed that we have introduced "*bindToController,*" but the attribute "*scope: true*" has been kept unchanged. However, it isn't an isolated scope directive. The reason we do not want to isolate it is that there are other external references inside it, and we don't need to break these references. In Angular

1.4, it is possible for one to combine the “*scope: true*” and the “*bindToController*” so as to introduce binding to scopes which are not isolated.

For the “*visible”* binding which is new, we have to plug in a value wherever the component “*foobar*” is to be used. We have the capability of accessing the “*mainC*,” and the template originally contained in the template can be used. This is shown below:

<foobar

visible=“mainC.isFooBarVisible()”>

</foobar>

The scope binding can now be referred to inside the component, and the dependency will be broken to the external world. This is shown below for the “*bar.html”*:

** **

** **

{{c.getText(mainC.getLanguage())}}

** **

** **

Another binding to the definition for the directive named “*language”* can now be

introduced. This is shown below:

```
module.directive('foobar', function() {
return {
scope: true,
controller: 'FoobarCtrl',
controllerAs: 'c',
bindToController: {
visible: '=',
language: '='
},
templateUrl: 'bar.html'
};
});
```

A value for it should then be plugged in where the component is being used. This is shown below:

```
<foobar
visible="mainC.isFooBarVisible()"
```

language=“mainC.getLanguage()”>

</foobar>

The binding should also be done inside the component template. This is shown below:

** **

** {{c.getText(c.language)}} **

** **

Our template will then be free of the external references. All of the data which is needed will be specified explicitly at the interface boundary.

An Example of the “$watch” Expression

External references cannot only be found in the component template. In the component controller, one may also have the expressions *“$watch,” “$watchCollection,”* or *“$watchGroup.”* The link functions can also be used if available. Consider the example given below, which shows how this can be replaced with a bound input:

```
function FoobarCtrl($scope) {

var c = this;

$scope.$watch(‘mainC.foo’, function(nFoo) {

if (nFoo.active) {

ctrl.actFoo = nFoo;

} else {

c.actFoo = null;

}

});

}
```

Our watch expression has referenced the “*foo”* attribute of the “*mainC.”* A bound input should be used instead of this. Consider the code which is given below:

```
module.directive('foobar', function() {

return {

scope: true,

controller: 'FoobarCtrl',

controllerAs: 'c',

bindToController: {

foo: '='

},

templateUrl: 'bar.html'

};

});
```

Once the component is being used, the actual value should be plugged in. This is shown below:

```
<foobar foo="mainC.foo">

</foobar>
```

Inside the watch expression for the controller, the bound input can then be used. This is shown below:

```
function FoobarCtrl($scope) {

var c = this;

$scope.$watch('c.foo', function(nFoo) {

if (nFoo.active) {

ctrl.actFoo = nFoo;

} else {

ctrl.actFoo = null;

}

});

}
```

An external reference may not be used in certain situations. We want to create a code for the controller which will reference a code which will just reference a scope attribute which is inherited just in plain JavaScript. Here is the code for the example:

```
function FoobarCtrl($scope) {

var c = this;

c.getActiveFoo = function() {

if ($scope.mainC.foo.active) {
```

return $scope.mainC.foo;

} else {

return null;

}

};

}

The above example is just similar to the previous one, with the only exception being that a bound input will be needed. This is shown below:

module.directive('foobar', function() {

return {

scope: true,

controller: 'FoobarCtrl',

controllerAs: 'c',

bindToController: {

foo: '='

},

templateUrl: 'bar.html'

};

});

The foo value will be provided during the use of the component. This is shown below:

<foobar foo=“mainC.foo”>

</foobar>

Our controller code will then be in a position to use the bound input. This is shown below:

```
function FoobarCtrl($scope) {
var c = this;
ctrl.getActiveFoo = function() {
if (c.foo.active) {
return c.foo;
} else {
return null;
}
};
}
```

Event handler and Bound output

All of the external components used in components are not logical inputs. Others act as logical outputs. Consider the example given below:

** **

** {{c.getText(c.language)}} **

<button ng-click=“mainC.deleteFoobar()”>

Delete

</button>

** **

In the above example, and inside our component template, we have the “*ng-click”* directive. This has been used to invoke a method on a “*mainC”* reference which has been inherited. That is what we have done.

For the dependency to be broken, an expression binding named “*onDelete”* has to be introduced on the definition of the directive. This is shown below:

module.directive(‘foobar’, function() {

return {

scope: true,

controller: ‘FoobarCtrl’,

controllerAs: ‘c’,

bindToController: {

visible: ‘=’,

language: ‘=’,

onDelete: ‘&’

},

templateUrl: ‘bar.html’

};

});

For the inputs and outputs to be distinguished in AngularJS, the syntax [] is used for the input and the syntax () used for the output. However, in the earlier versions of AngularJS, such as 1.x, there was no mechanism as to how these can be distinguished. This is why most people prefer to use the prefix “*on”* for outputs.

Once the component has been used, an expression for the new binding will then be plugged in. The user of the component will then decide on what the delete means to him or her. In our case, we have used it so as to invoke the function “*deleteFoobar*” of our main controller. This is shown below:

<foobar

visible=“mainC.isFooBarVisible()”

language=“mainC.getLanguage()”

onDelete=“mainC.deleteFoobar()”>

</foobar>

The expression binding will then be invoked from inside the component template. This is shown below:

** **

** {{c.getText(c.language)}} **

<button ng-click=“c.onDelete()”>

Delete

</button>

** **

Consider the sample code given below:

function FoobarCtrl($scope) {

```
    var c = this;
c.onFooClick = function() {
    mainC.setActiveFoo(c.foo);
};
}
```

In the above example, we have used a controller function which will call a function in the parent scope. The external function call must not be an event handler in a particular template. It can be contained in a controller. This is what we did with the inputs.

What we should do to the definition of the directive is that we should add an expression binding. This is shown below:

```
module.directive('foobar', function() {
return {
scope: true,
controller: 'FoobarCtrl',
controllerAs: 'c',
bindToController: {
    onActivateFoo: '&'
},
```

templateUrl: ‘bar.html’

};

});

Once the template has been used, there is an invocation of “*mainCtrl.setActiveFoo*” which we set in the component controller. This is shown below:

<foobar

on-activate-foo=“mainC.setActiveFoo(foo)”>

</foobar>

The expression binding should then be invoked inside the controller other than the external inherited controller. An object argument should be provided to the Angular so as to know the value of the “*foo”* which is to be used inside the bound expression. This is shown below:

function FoobarCtrl($scope) {

var c = this;

c.onFooClick = function() {

c.onActivateFoo({foo: c.foo});

};

}

Consider the example given below:

```
module.directive('foobar', function() {
return {
    scope: true,
    controller: 'FoobarCtrl',
    controllerAs: 'c',
bindToController: {
    visible: '=',
    language: '=',
    onDelete: '&'
},
templateUrl: 'bar.html'
};
});
```

We have defined all the inputs and the outputs in the above example.

Conclusion

It can be concluded that AngularJS is a very useful framework for the development of web applications. The usefulness of AngularJS is seen on what it does to HTML. Generally, it was developed for the purpose of extending what is contained in HTML. Note that HTML was not developed for the purpose of development of web-apps.

However, AngularJS comes in to fill this gap. It works by extending the vocabulary which is contained in HTML so as to get extra and amazing functionalities. With HTML, it is very easy for one to declare and display dynamic content. However, when it comes to dealing with dynamic content, then HTML is not the best. This is where AngularJS comes in, and it serves to fill this gap.

As we have said, AngularJS is used to extend the functionalities of HTML. A good example is when we create an HTML page and then change it by use of the AngularJS. This page can then be changed to display the content dynamically by the use of AngularJS. That is how the two work together. Applications always use data. This is very important for any application.

However, other than dealing with only small amounts of data, you should learn to work with large data sets in AngularJS. This is possible in AngularJS, and your resultant application will be very useful. AngularJS can also be used for the purpose of adding animations to your application. These make your web application very attractive to its users. There is a module named *"ngAnimate"* in AngularJS which can be used for the purpose of performing these animations. Those are some of the features which have been discussed in this book.