



SECUREAUTH

AngularJS Best Practices Guide

Copyright Information

2017. SecureAuth® is a copyright of SecureAuth Corporation. SecureAuth's IdP software, appliances, and other products and solutions, are copyrighted products of SecureAuth Corporation.

June, 2017

For information on supporting this product, contact your SecureAuth sales representative:

Email: support@secureauth.com

Phone: +1.949.777.6959 or +1-866- 859-1526

Website: <https://www.secureauth.com/Support.aspx>

Contents

Introduction	1
Benefits	1
Software Requirements	2
SecureAuth-Specific Workflow	3
New 'Data-View' Attribute	3
Bootstrap and New Themes	4
GulpJS	4
Sass	4
Themes-Master Scaffold Overview and Structure	4
Directives	6
Services	7
Views	7
App.js	8
Creating a New Theme	8
AngularJS Overview	9
Structure	9
Objects	9
Properties at Runtime	10
Class	10
Functions	11

Directives	11
Directive Behavior	13
Directives and CSS	13
Filters	14
Expressions	17
Templates	18
Data Binding Expression	18
View, Controller, and Scope	20
\$scope	20
Modules	22
Config Phase Components	25
Routing	25
Working with Routing in a Local Environment	26
Views	27
Services	27
Factories	29
Providers	30
AJAX	31
RESTful	31
AJAX + RESTful	32
(resource)	34
AngularJS Animations	37
Animation Classes	38
Compiling/Deploying	38
Troubleshooting Compiling and Deploying	39
Test Driving a Design	40
QUnit	40
Common Changes	42
Styling Changes	42
Text Changes	44
Function Changes	45
Adding a New Element or Directive	46
GitHub Tool	50
Reproducing a Project	50
GULP tools	53
Route Object: Resolving Property	54
View Controller	54
Load Dependencies	54
Conclusions	54
Questions and Answers	55
Best Practices	57
MVC Flow	57
Large Applications	58

Factory Methods.....	58
Module Methods.....	59
Controllers.....	59
Dependency Annotation.....	60
\$Inject Property Annotation.....	60
Implicit Annotation.....	60
Strict Dependency Injection.....	61
'willBreak' Service.....	61
Declarative Programming.....	61
Large AngularJS Application.....	62
References	63

Introduction

AngularJS is a structural framework for dynamic web applications that extends the HTML vocabulary resulting in an expressive and readable markup. It offers a toolset for building web pages that goes beyond the static structures allowed by white-bread versions of HTML. While HTML enables the designer to create web pages replete with text, pictures, and links, it has little flexibility when it comes to dynamic logic and workflow.

AngularJS extends the HTML vocabulary to accommodate many more techniques and strategies, including the very techniques SecureAuth uses for authenticating users. AngularJS is fully extensible and works well with other libraries. Every feature can be modified or replaced to suit your unique development workflow and feature needs.

Benefits

- + Easy customization of HTML markup, CSS, and JS
- + HTML5 syntax and API feature sets
- + AngularJS to extend HTML vocabulary resulting in an extraordinarily expressive, readable, and quick to develop workflow
- + One framework for every device with Bootstrap
- + Automate and enhance workflow with Gulp
- + SASS for CSS preprocessor
- + Available documentation for common HTML elements, dozens of custom HTML and CSS components, and helpful JS plug-ins
- + Out-of-band patch and fix
- + Use of modular page components
- + Allow us to move some of our view logic from the back-end to the front-end
- + Not difficult to learn
- + Create powerful front-end themes quickly
- + Useful for single-page applications

Single-page applications (SPAs) are web applications that fit on a single web page. They are essentially desktop apps, like Google maps, designed to be used on all devices. Each HTML page contains mini-views (HTML fragments) that can be loaded in the background. There is no reloading of the page and requires handling of browser history, navigation, and bookmarks.

The purpose of this guide is not to instruct you in the use of AngularJS, since many tutorials are available online to provide that service, but to show you how SecureAuth uses this framework and has enhanced its basic HTML vocabulary for designing web pages within the SecureAuth IdP environment.

This guide is divided into sections that discuss these major topics:

- + Structure
- + Compiling and Deploying
- + Troubleshooting
- + Common Changes
- + GULP tools
- + SecureAuth-specific Directives
- + Advanced Strategies

Software Requirements

In addition to your normal programming applications, you should make sure the following programs are installed:

- + AngularJS
- + GULP
- + GitHub

Bootstrap and New Themes

Bootstrap makes front-end web development faster and easier. One code base is used for every device with CSS media queries. The reasons we are using Bootstrap are:

- + Increase speed of development by utilizing ready-made blocks of code with cross-browser compatibility and CSS functionality.
- + Ensures consistency regardless of who's working on the project.
- + Framework takes into account the future of design and development with HTML5 and CSS3
- + A great aspect of bootstrap is that you can make it your own. You can sit down and go through the whole framework and keep what you need and ditch what you don't.

GulpJS

We recommend the use of GulpJS for your streaming build system. By using a node's stream file, manipulation is all done in memory and a file is not written until you tell it to do so. It is the streaming nature that enables GulpJS to pipe and pass around the data being manipulated or used by plugins. These plugins are only intended to do one job at a time, so it is not uncommon to pass a singular file through multiple plugins.

For more on GulpJS, refer to <http://gulpjs.com/>.

Sass

Sass (syntactically awesome stylesheets) is a 'pre-processing' scripting language that extends CSS by allowing developers to write code in one language and then compile it into CSS.

Sass is an extension of CSS3, adding nested rules, variables, mix-ins, selector inheritance, and more. We recommend the use of this extension in order to provide greater latitude to style choices for SecureAuth presentation pages.

Themes-Master Scaffold Overview and Structure

After installation of the AngularJS applications, the essential theme files and folders structure under 2016 Light resemble the example in Figure 1.

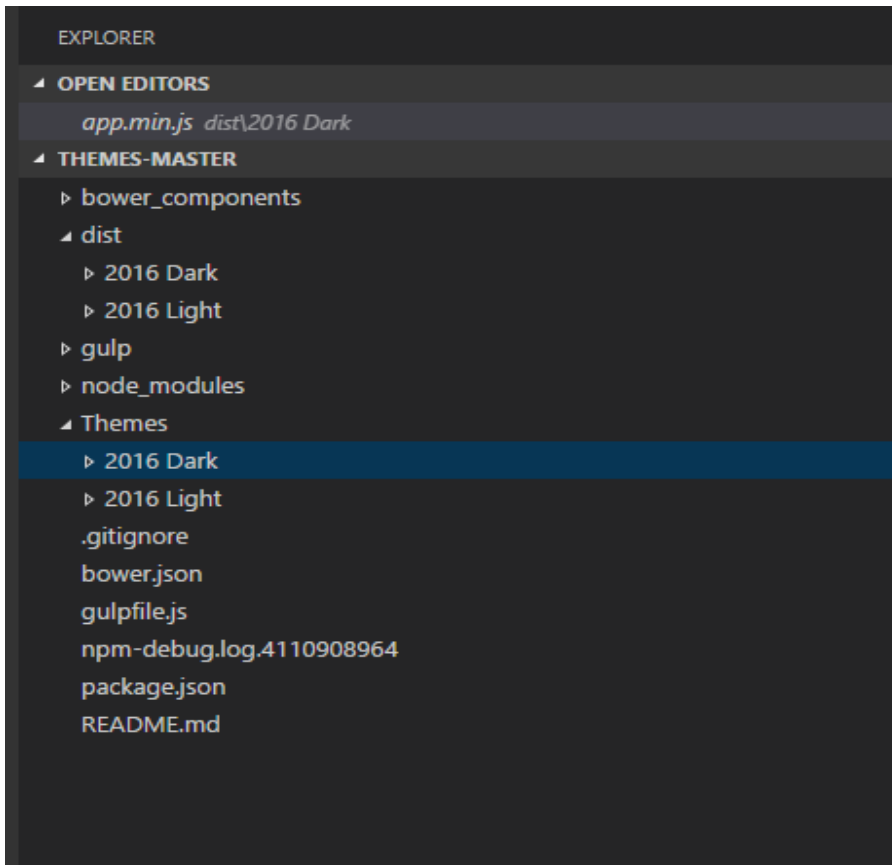


FIGURE 1. 2016 Light Theme Files and Folders Structure

To program a new theme, specify the theme in the 'Themes' folder. If needed, copy or clone a new theme from the available themes. (The 'dist' folder includes the compiled version of the theme.)

For example, when the 2016 Light element is expanded, a screen like Figure 2 appears.

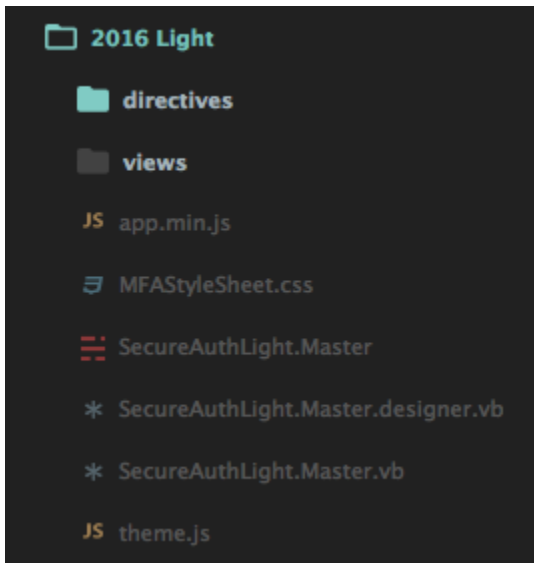


FIGURE 2. 2016 Light Element Expansion Example

The elements in the Themes-Master folder include:

- + directives – modular components
- + views – layout base on the state view
- + app.min.js – main application JavaScript (see page 38)
- + MFAShyleSheet.css – main application CSS
- + theme.js – helper/custom JavaScript

Directives

Directives are organized by states and provide the HTML within each directive tags. Directives are annotated in the following manner:

```
{{directive}}.html
```

In HTML, you will see native angular directives such as ‘ng-if’ or ‘ng-repeat’. These are documented in <https://docs.angularjs.org/api/ng/directive> and provide basic display logic. (Some of the most important directives are briefly described later in this guide.)

Directives can also contain the logic localized to all directive tags. That is,

```
{{directive}}.directive.js
```

Directive logic follows this syntax:

```
'use strict';  
angular.module('secureauth')
```

```
.directive('{directive}', function (config, {service}) {

    var {directive}Controller = function () {
        var vm = this;
        angular.extend(vm, {
            // Objects
        });
    };
    return {
        restrict: 'EA',
        controller: {directive}Controller,
        controllerAs: '{directive}',
        templateUrl: config.theme + '/directives/{currentState}/{directive}/
{directive}.html',
        bindToController: true
    };
});
```

Common directives that are used in many states are contained in the `/common` folder.

For more on directives, refer to “Directives” starting on page 11.

Services

Services are organized based on state. These services are used by directives to get the asp-generated objects in order to regenerate it in the design that the theme will specify. Services include:

- + `domModel.js` – this service contains the logic that parses html dom objects and builds new objects used by this angular theme engine to rebuild those same objects within the theme design. These functions are called throughout the services.
- + `App.scss` – this service contains the main stylesheet. It is compiled during “gulp theme:build” to `MFAStyleSheet.css`. See <http://sass-lang.com/guide> for documentation on SCSS syntax and usage.

For more on this topic, refer to “Services” starting on page 27.

Views

Views are organized by states and provide the HTML shell for each state. Each state HTML contains directive tags or custom angular generated by HTML tags. For more on this, refer to “Views” starting on page 27.

App.js

The app.js file in the 'dist' folder contains the aggregated results of combining the JS files you compiled in your gulp file. It defines and initializes the module. Among other features, the app.js file determines what state the current application is in. This includes

- + ServerState - get the state from the 'data-view' attribute in the head
- + RenderState - parse MLALoginControl data to get current state
- + PathState - get state from URL

Creating a New Theme

New themes can be created by right-clicking on an existing theme element and selecting the **clone** option. Once cloned, rename the cloned theme as needed. The file structure and its required elements are populated for the new theme.

AngularJS Overview

This section includes a more exhaustive run-down on the language itself: the objects and arguments most critical to using AngularJS.

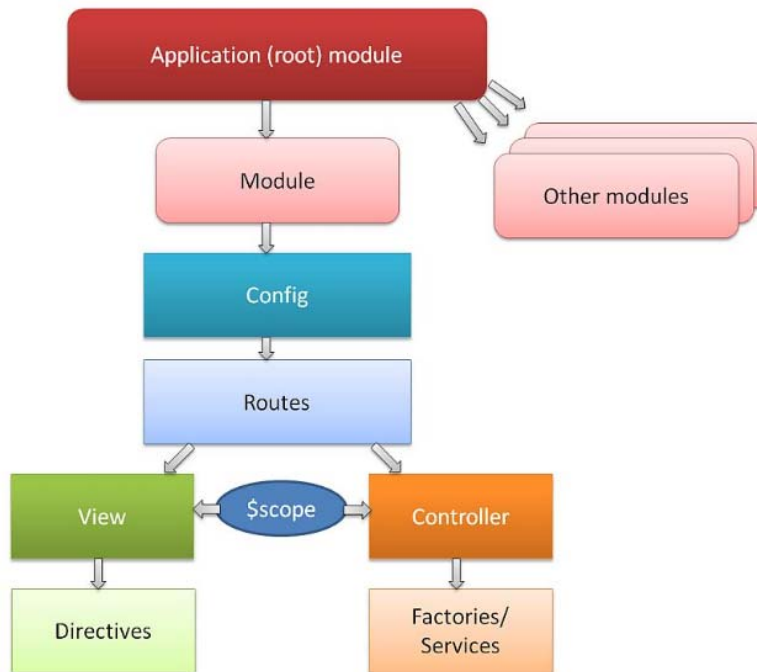


FIGURE 3. AngularJS Basic Flow

Structure

AngularJS as used in SecureAuth webpage theme creation includes the following structural elements:

- + App.js
- + Views
- + Directives

Each of these essential elements are discussed on the following pages.

Objects

Every component within AngularJS is an object including functions and arrays. Objects include both properties and methods. These are presented as:

- + A collection of name-value pairs

- + Names are strings
- + Values can be anything
- + Properties and methods can be added at runtime

Objects can inherit other objects as in this example:

```
var customer = {name: "Jack", gender: "male"};
var circle1 = {radius: 9, getArea: someFunction};
var circle2 = {
  radius: 9,
  getRadius: function() {
    return this.radius;
  }
}
```

Properties at Runtime

One of the simplest ways to create objects is to use an argument like this example:

- + One of the simplest ways to create an object is:

```
var obj = new Object();
obj.x = 10;
obj.y = 12;
obj.method = function() {...}
```

- + This adds at runtime two properties to the obj:
 - object!
- + The object is built-in data type

This adds two properties at runtime to the `obj = object`. The object is built as a data type.

Class

You can create constructor-function in JavaScript in this manner:

```
function Point() {
  this.x = 1;
  this.y = 1;
}
var p = new Point();
```

An example of the correct use of properties and classes is shown in the following example:

```
function Circle(radius)
{
```

```
    this.radius = radius;
    this.getArea = function();
    {
        return (this.radius * this.radius) * Math.PI;
    };
}
var myobj = new Circle(5);
document.write(myobj.getArea() );
```

Functions

Every function in JS is a **Function object** and perform these operations:

- + Can be passed as arguments
- + Can store name / value pairs
- + Can be anonymous or named

The usage of certain functions may prove ineffective or inefficient. For example:

```
var myfunction = new Function("a","b", "return a+b;");
print(myfunction(3,3) );
```

Functions can take other functions as arguments. For example:

```
-fetchUrl ( onSuccess, onError )
```

You can use such anonymous functions in this manner:

```
-fetchUrl( function() (...), function() (...) );
```

Directives

Directives apply special behavior to attributes or elements in HTML. AngularJS directives allow the developer to specify custom and reusable HTML-like elements and attributes that define data bindings and the behavior of presentation components. Some of the most common SecureAuth-derived directives are shown in Table 1.

TABLE 1. SecureAuth-Derived Directives

Directive	Meaning
ng-app	Declares the root element of an AngularJS application under which directives can be used to declare bindings and define behavior.

TABLE 1. SecureAuth-Derived Directives

Directive	Meaning
ng-bind	Replaces the text content of the specified HML with the value of given expression Sets the text of a DOM element to the value of an expression. For example, <code></code> displays the value of 'name' inside the span element. Any change to the variable 'name' in the application's scope reflect instantly in the DOM.
ng-model	Stores/updates the value of the input field into a variable Similar to ng-bind, but establishes a two-way data binding between the view and the scope.
ng-model-options	Provides tuning for how model updates are done.
ng-class	Enables class attributes to be dynamically loaded.
ng-controller	Specifies a JavaScript controller class that evaluates HTML expressions.
ng-repeat	Instantiate an element once per item from a collection.
ng-show ng-hide	Conditionally shows or hides an element, depending on the value of a Boolean expression. Show and hide is achieved by setting the CSS display style.
ng-switch	Conditionally instantiates one template from a set of choices, depending on the value of a selection expression.
ng-view	Handles routes that resolve JSON before rendering templates driven by specified controllers.
ng-if	Allows the display of the element that follows, if the conditions are true. When the condition is false, the element is removed from the DOM. When true, a clone of the compiled element is re-inserted.
ng-aria	Enables accessibility support for common ARIA attributes .
ng-animate	Provides support for JavaScript, CSS3 transition and CSS3 keyframe animation hooks within existing core and custom directives.

Since `ng-*` attributes are not valid in HTML specifications, `data-ng-*` can also be used as a prefix. For example, both `ng-app` and `data-ng-app` are valid in AngularJS.

For example, the correct use of both the `Ng-init` and `Ng-repeat` directives is shown below:

```
<html data-ng-app="" >
<head>
  <title>Title</title>
  <meta charset="UTF-8" />
  <script src=../angular.min.js" type="text/javascript">
</script>
</head>

<body>
```

```
<div data-ng-init="names = ( 'Jack', 'John', 'Tina' )">
  <h1>Cool loop!</h1>
  <ul>
    <li data-ng-repeat="name in names">{{name}}</li>
  </ul>
</div>
</body>
</html>
```

Directive Behavior

Prevailing behavior with directives should include limiting DOM manipulation. This involves aggressively restricting all DOM access (which usually translates to all jQuery usage) to a thin “segregation layer”. Everything outside of the segregated DOM layer can be tested rapidly in isolation from the browser using a lean JavaScript engine such as node.js.

Directives and CSS

Cascading Style sheets (CSS) work seamlessly with directives, following this simple rule:

TABLE 2. Directives and CSS Interrelationship

Event	Starting CSS	Ending CSS	Directives
Enter	.ng-enter	.ng-enter-active	ngRepeat, ngInclude, ngif, ngView
Leave	.ng-leave	.ng-leave-active	ngRepeat, ngInclude, ngif, ngView
Move	.ng-move	.ng-move-active	ngRepeat

An example of this usage is shown below:

```
/* starting animation */
.ng-enter {
  -webkit-transition: 1s;
  transition: 1s;
  margin-left: 100%;
}
/* ending animation */
.ng-enter-active {
  margin-left: 0;
}
/* starting animation */
.ng-leave {
```

```
-webkit-transition: 1s;
transition: 1s;
margin-left: 0;
}
/* ending animation */
.ng-leave-active {
  margin-left: 100%;
}
```

Another example of the use of directive as a function is shown below:

```
angular.module('contactWorks.directives');
directive('focus', [function () {
  return {
    restrict: 'A';
    link: function (scope, iElement, iAttrs) {
      iElement.focus();
    }
  }
}]);
```

Filters

Filters are used to filter the output:

```
<html ng-app>
  <head>
    <title></title>
  </head>

  <body>
```

For instance, you might choose to filter the output by order (`orderBy`), as in this example:

```
<div name="container"
  ng-init="contacts=[{name:'Renan Martins'}, {name:'Tania Gonzales'}, {name:'Anabela de Malhadas'}]">
  <input type="text" ng-model="newContact" />
  <button ng-click="contacts.push({name:newContact})">ADD CONTACT</button>

  <ul ng-repeat="contact in contacts | orderBy: 'name'">
    <li>{{contact.name}}</li>
  </ul>
  <hr>
  <ul ng-repeat="contact in contacts | orderBy: '-name' | limitTo:6 ">
    <li>{{contact.name | uppercase }}</li>
  </ul>
</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.min.js"></script>
</body>
</html>
```



Filters format the output such as formatting for currency, numbers, date, and lowercase.

An example of the correct use of a filter is shown in the following example:

```
<!DOCTYPE html>
<html data-ng-app="">
<head>
  <title>Title</title>
  <meta charset="UTF-8">
  <script src="../angular.min.js" type="text/javascript">
</script>
</head>
<body>
  <div data-ng-init="customers = [{name:'jack'}, {name:'tina'}]">
    <h1>Cool loop!</h1>
    <ul>
      <li data-ng-repeat="customer in customers | orderBy:'name'">
        {{ customer.name | uppercase }}</li>
    </ul>
  </div>
</body>
</html>
```



Another example shows the use of a filter when using user input filters:

```
<!DOCTYPE html>

<html data-ng-app="">
<head>
  <title>Title</title>
  <meta charset="UTF-8">
  <script src="../angular.min.js" type="text/javascript">
</script>
</head>
<body>
  <div data-ng-init=
    "customers = [{name:'jack'}, {name:'tina'}, {name:'john'}, {name:'donald'}]">
    <h1>Customers</h1>

    <input type="text" data-ng-model="userInput" />
    <ul>
      <li data-ng-repeat="customer in customers | orderBy:'name' | filter:userInput">{{
        customer.name | uppercase }}</li>
    </ul>
  </div>
</body>
</html>
```



Another example of HTML using a filter is shown here:

```
<div ng-app="myApp">
  <div>
    {{ 'World' | greet }}
  </div>
</div>
```

This uses the `ng-app` directive to define an app and filters it.

Yet another example of a filter is shown in this example:

```
<div name="container"
  ng-init="contacts=[{name:'Renan Martins'}, {name:'Tania Gonzales'}, {name:'Anabela de Malhadas'}]">
  <input type="text" ng-model="newContact" />
  <button ng-click="contacts.push({name:newContact})">ADD CONTACT</button>

  <ul ng-repeat="contact in contacts | orderBy: 'name'">
    <li>{{contact.name}}</li>
  </ul>

  <hr>

  <ul ng-repeat="contact in contacts | orderBy: '-name' | limitTo:6 ">
    <li>{{contact.name | uppercase }}</li>
  </ul>
</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.min.js"></script>
</body>
/html>
```



Expressions

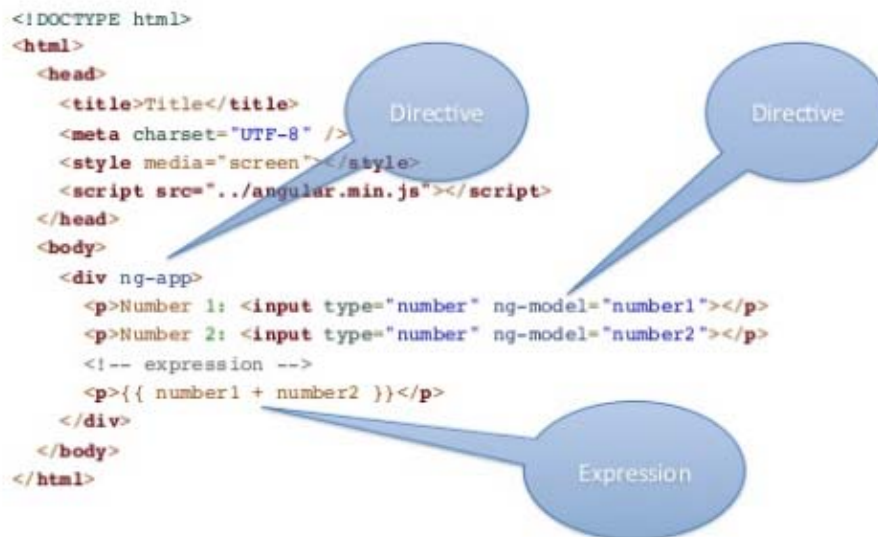
Angular expressions are JavaScript-like code snippets usually placed in bindings like this:

```
-{{ expression }}
```

Valid Expressions in AngularJS include:

```
-{{ 1+2 }}  
{{ a+ b }}  
-{{ items[index] }}
```

Filters are used to format or filter data in this manner:




A valid HTML 5 Example is:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Title</title>
    <meta charset="UTF-8" />
    <style media="screen"></style>
    <script src="angular.min.js"></script>
  </head>
  <body>
    <div data-ng-app="">
      <p>Number 1: <input type="number" data-ng-model="number1"></p>
      <p>Number 2: <input type="number" data-ng-model="number2"></p>
      <!-- expression -->
      <p>{{ number1 + number2 }}</p>
    </div>
  </body>
</html>

```



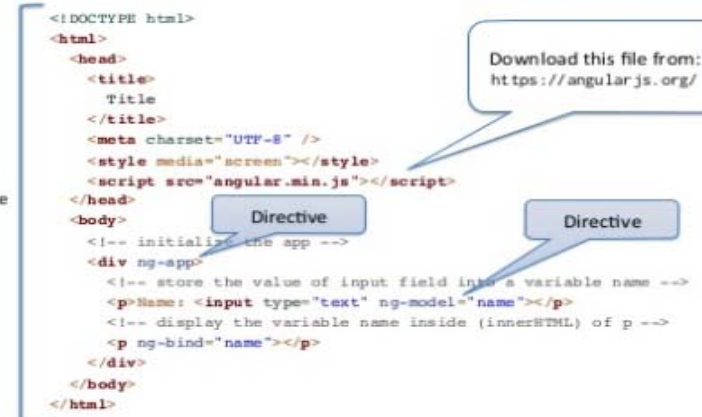
Templates

Templates are HTML code with additional markup, directives, expressions, filters, and other components as shown in this example:

```

<!DOCTYPE html>
<html>
  <head>
    <title>
      Title
    </title>
    <meta charset="UTF-8" />
    <style media="screen"></style>
    <script src="angular.min.js"></script>
  </head>
  <body>
    <!-- initialize the app -->
    <div ng-app>
      <!-- store the value of input field into a variable name -->
      <p>Name: <input type="text" ng-model="name"></p>
      <!-- display the variable name inside (innerHTML) of p -->
      <p ng-bind="name"></p>
    </div>
  </body>
</html>

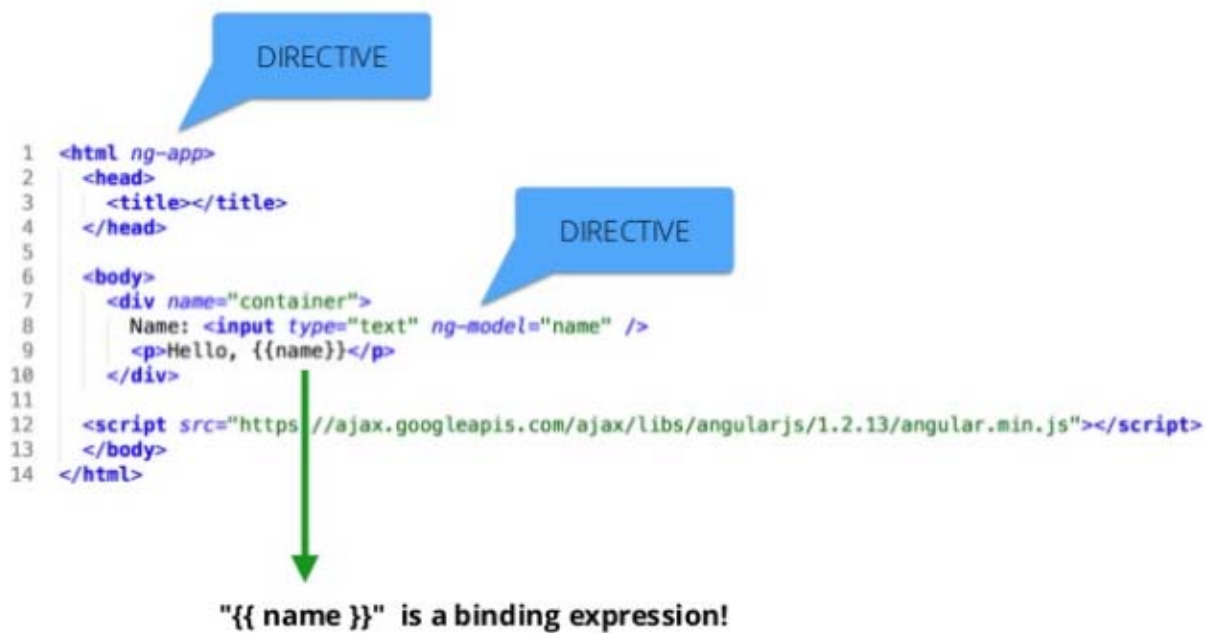
```



Data Binding Expression

This concept binds a model to a View using the expression wrapper {{ }}.

When used with directives, data binding can prove powerful, as in this example:



Two-way data binding is an effective tool for linking a view to a model.

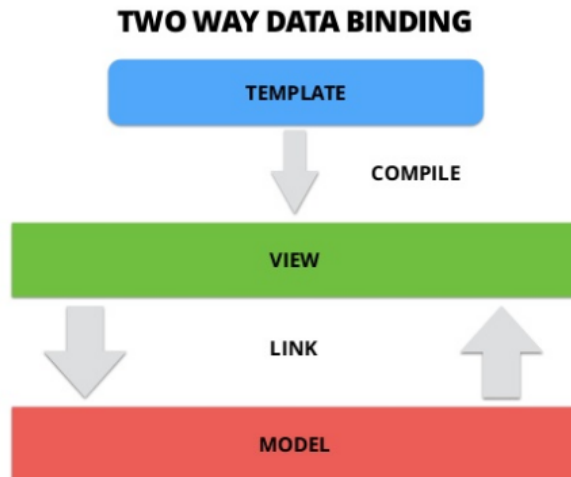


FIGURE 4. Two-Way Data Binding

View, Controller, and Scope

As indicated in Figure 5, there is an implicit linkage between view, controller, and scope in AngularJS.



`$scope` is an object that can be used to communicate between View and Controller

FIGURE 5. View, Controller, and Scope Linkage

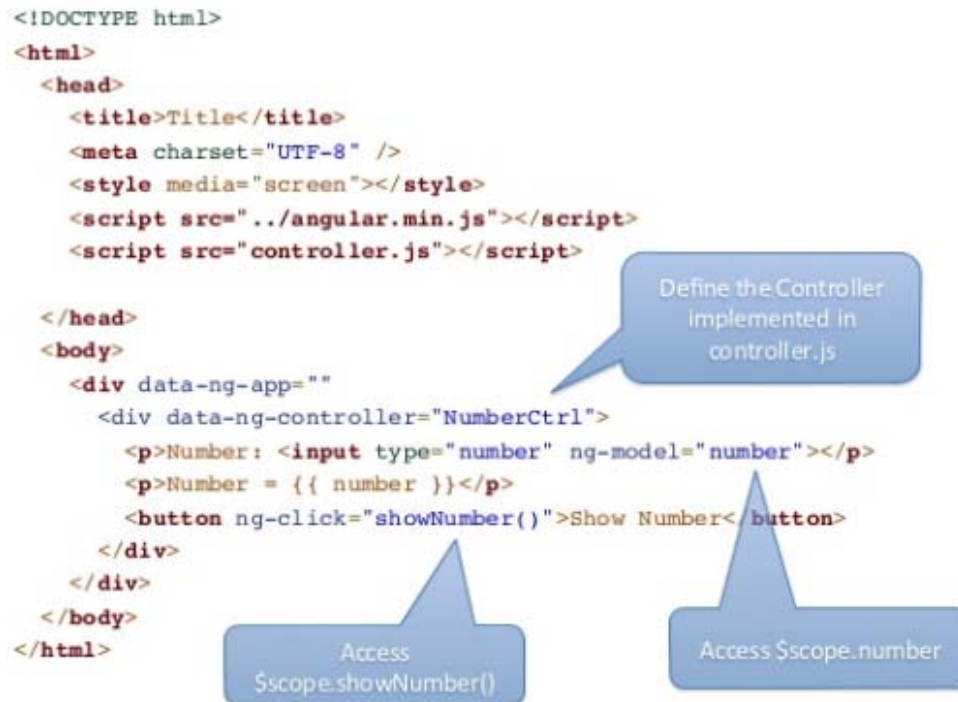
`$scope`

`$scope` is an object that links the controller to the view. AngularJS invokes the constructor with a `$scope` object and runs inside `ng-app (div)`. Put in the context of code, the `$scope` intermediary can be used to link view with controller as in the following example:

```
// Angular will inject the $scope object, you don't have to
// worry about it
function NumberCtrl ($scope) {
  // $scope is bound to view, so the communication
  // to view is done using the $scope
  $scope.number = 1;
  $scope.showNumber = function showNumber() {
    window.alert ( "your number = " + $scope.number );
  };
}
```

WARNING: `$scope` will not work for AngularJS 1.3 or earlier. Only versions later than AngularJS 1.3 use `$scope` in this manner.

An example of the use of the `ng-app` wrapper to define the controller implemented in `controller.js` is shown in the following example:



When `$scope` is used to link the controller to a view, one of these conditions must occur:

- + Setup the initial state of `$scope` object
- + Add behavior to the `$scope` object

Do not use a controller when one of these conditions occurs:

- + Manipulating the DOM (use data binding and/or directives)
- + Formatting output (using form controls)
- + Filtering the output (using filters)
- + Sharing code or state (using services)

A template for a controller is shown in the following example:

```

// Create new module 'myApp' using the angular.module method
// The module is not dependent on any other module
var myModule = angular.module('myModule', []);
myModule.controller( 'MyCtrl', function ($scope) {
  // Your controller code here:
});

```

The definition of a controller can then be used to construct a module as discussed in the following topic.

Modules

A module is a reusable container for different features of your app such as controllers, services, filters, and directives as shown in Figure 6.

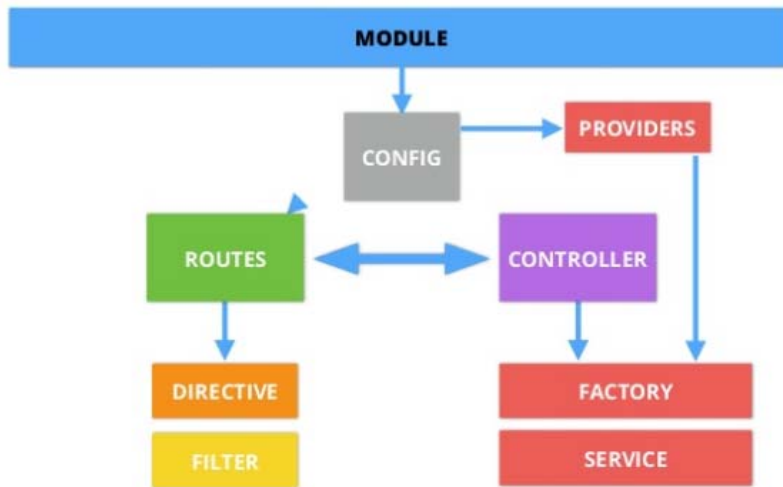


FIGURE 6. Module Container

Yet another arrangement of a module object is shown in Figure 7.

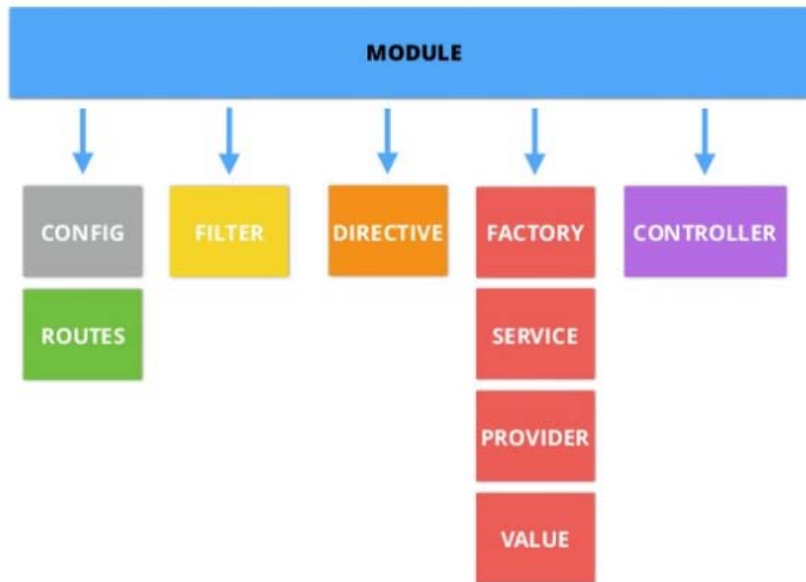


FIGURE 7. Module Object Flow Chart

Modules can be loaded in any order. You can build our own filters and directives.

NOTE: If you have defined many controllers, you might be polluting JS namespace. Be judicious in your use of these objects.

An example of a module and its use is shown in the following example:

```
// declare a module
var myAppModule = angular.module('myApp', []);
// configure the module
// in this example we will create a greeting filter

myAppModule.filter('greet', function() {
  return function(name) {
    return 'Hello,' + name + '!';
  };
});
```

To create a controller in a module, use a structure like the following example:

```
var myModule = angular.module('myModule', []);
myModule.controller('MyCtrl', function($scope) {
  var model = { "firstname": "Jack",
```

```

        "lastname"; "Smith"};
    $scope.model = model;
    $scope.click = function() {
        alert($scope.model.firstname);
    };
});

```

This then contributes to creating a module object as shown in the following example:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Title</title>
    <meta charset="UTF-8" />
    <style media="screen"></style>
    <script src="../angular.min.js"></script>
    <script src="myModule.js"></script>
  </head>
  <body>
    <div ng-app="myModule">
      <div ng-controller="MyCtrl">
        <p>Firstname: <input type="text" ng-model="model.firstname"></p>
        <p>Lastname: <input type="text" ng-model="model.lastname"></p>
        <p>{{model.firstname + " " + model.lastname}}</p>
        <button ng-click="click()">Show Number</button>
      </div>
    </div>
  </body>
</html>

```



Modules can also be used as dependencies as in the following example:

```

angular.module('myApp', [
    'ngRoute',
    'myApp.filters',
    'myApp.services',
    'myApp.directives',
    'myApp.controllers',
    'ngRoute',
    'ngResource',
    'ui.bootstrap',
])

```

Config Phase Components

Components register against the module in the config phase using providers. For example, to register a controller manually use `$controllerProvider` in this manner:

```
angular.module('moduleName', [])
  .config(function($controllerProvider){
    $controllerProvider.register('Ctrl',function() {
      //controller code
    })
  });
```

For more on providers, refer to “Providers” on page 30.

Routing

Routing is useful particularly when building a SPA app because everything can be handled on a single page. The use of routing can address several questions:

- + How should back-buttons work?
- + How do I link between “pages”?
- + How about URLs?

For example, you can link several operations in this manner:

```
<html data-ng-app="myApp">
<head>
  <title>Demonstration of Routing - index</title>
  <meta charset="UTF-8" />
  <script src="../../angular.min.js" type="text/javascript"></script>
  <script src="angular-route.min.js" type="text/javascript"></script>
  <script src="myapp.js" type="text/javascript">
</script>
</head>

<body>
  <div data-ng-view=""></div>
</body>
</html>
```



A specific directive is used for this purpose: `ngRoute`.

```
//This module is dependent on ngRoute. Load ngRoute before this
var myApp = angular.module('myApp', ['ngRoute']);

//Configure routing
myApp.config(function($routeProvider) {
  // Usually we have different controllers for different views
  // In this demonstration, the controller does nothing
  $routeProvider.when('/', {
    templateUrl: 'view1.html',
    controller: 'MySimpleCtrl' });
  $routeProvider.when('/view2', {
    templateUrl: 'view2.html',
    controller: 'MySimpleCtrl' });
  $routeProvider.otherwise({ redirectTo: '/' });
});

//Now add a new controller to MyApp
myApp.controller('MySimpleCtrl', function ($scope) {
});
```

However, as shown in the previous example, `ngRoute` must be loaded before an argument like this can be made.

Working with Routing in a Local Environment

If you are returned the message “cross origin requests are only supported for HTTP”, this means either

- + You should disable web security in your browser
- + Or you should use a web server and access files `http://...`

To disable web security in Chrome from the run prompt, enter:

```
Taskkill /F /IM chrome.exe
```

where `chrome.exe` is located at:

```
C:\Program Files (x86)\Google\Chrome\Application\chrome.exe
```

and possible switches are:

```
--disable-web-security
```

and

```
--allow-file-access-from-files
```

Views

A view is an HTML fragment and can be expressed as shown in the following example:

- + view1.html:

```
<h1>View 1 </h1>
<p><a href="#/view1">To View 1</a></p>
```
- + view2.html

```
<h2>View 2 </h2>
<p><a href="#/view2">To View 1</a></p>
```

Services

View-independent business logic should not be in the controller; it should be relegated to a service component.



Controllers are view-specific whereas services are application-specific.

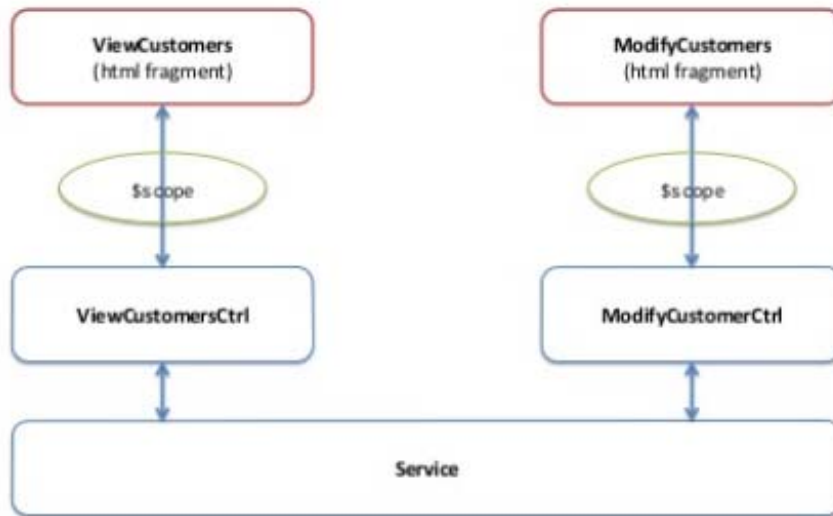


FIGURE 8. Controllers vs. Services

Move from view to view and service is still alive. The controller's responsibility is to bind the model to a view. The model can be fetched from service but the controller is not responsible for manipulating (create, destroy, update) the data. In many cases, rather than using `$scope` to link views to controllers, it may prove more useful to specify controller using a service instead.

AngularJS has many built in services; for example, `$http` (for a full discussion of services, see the online discussion at <http://docs.angularjs.org/api/service>).

An example of adding a new controller using a service, is shown in the following example:

```
// Add a new controller to MyApp. This controller uses Service.
myApp.controller('ViewCtrl', function($scope, CustomerService) {
    $scope.contacts = CustomerService.contacts;
});

// Add a new controller to MyApp. This controller uses Service.
myApp.controller('ModifyCtrl', function($scope, CustomerService) {
    $scope.contacts = CustomerService.contacts;
});

// Create a factory object that contains services for the controllers
myApp.factory('CustomerService', function() {
    var factory = {};
    factory.contact5s = ( {name: "Jack", salary: 3000}, {name: "Tina", salary
5000}, {name: "John", salary: 4000});
    return factory;
});
```

Another use of a service is shown in this example:

```
// Service is instantiated with new keyword
// Service function can use "this" and the return value is this.
myApp.service('CustomerService', function() {
  this.contacts =
    [ {name: "Jack", salary: 3000},
      [ {name: "Tina", salary: 5000},
        [ {name: "John", salary: 4000},
          ]];
});
```

As an alternative to a service, you might choose to select and define one or more factories as discussed in the next topic.

Factories

Factories are a reasonable alternative to services.



FIGURE 9. Factories Component

The differences between services and factories can be summed up in the following example:

```
angular.module('contactWorks.factories', []).factory('contacts',
function () {
  var contacts = [];

  var save = function (contact) {
    contact.id = Math.random().toString(36).substring(7);
    contacts.push(contact);
    return contact.id;
  };

  var get = function (id) {
    return _.find(contacts, function(contact) { return contact.id
    === id; });
  };

  var remove = function (id) {
    for (var i = 0; i < contacts.length; i++) {
      if (contacts[i].id === id) {
        contacts.splice(i, 1);
      }
    }
  };
});
```

In general, if you want your function to be defined as a **normal** function, use factory. If you want your function to be instantiated with a **new** operator, use service.

Providers

A Provider is yet another type of function to consider when constructing an AngularJS argument.



FIGURE 10. Providers Component

In general, the features of providers, services, and factories are compared in Table 3.

TABLE 3. Providers, Services, and Factories Comparison

Features	Factory	Service	Value	Constant	Provider
Can have dependencies	yes	yes	no	no	yes
Object available in config phase	no	no	no	yes	yes
Can create functions/primitives	yes	no	yes	yes	yes

AJAX

Asynchronous JavaScript + XML (AJAX) is used only occasionally, since XML is rarely needed; more often the code of choice is JSON. However, when used, AJAX can be implemented to send and retrieve data asynchronously from server in background.

There are a group of technologies that use this asynchronous version, including HTML, CSS DOM, XML/JSON, XMLHttpRequest, and JavaScript.

An example of using AJAX is:

```
<script type="text/javascript">
  var myapp = angular.module("myapp", []);
  myapp.controller("MyController", function($scope, $http) {
    $scope.myData = { };
    $scope.myData.doClick = function(item, event) {
      var responsePromise = $http.get("text.txt");
      responsePromise.success(function(data, status, headers, config){
        $scope.myData.fromServer = data;
      });
      responsePromise.error(function(data, status, headers, config){
        alert("AJAX failed!");
      });
    }
  });
</script>
```

RESTful

Web Service APIs that adhere to REST architecture constraints are known as RESTful APIs. These constraints are:

- + Base URI, such as <https://www.example/resources>

- + Internet media type for data such as JSON or XML
- + Standard HTTP methods: GET, POST, PUT, DELETE
- + Links to reference (such as Reference state and Related resources)

RESTful API HTTP methods are shown in Table 4.

TABLE 4. RESTful API HTTP Methods

Resource	GET	PUT	POST	DELETE
Collection URI such as http://example.com/resources	List the URIs and other details of the collections members	Replace the entire collection with another collection	Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by operation	Delete the entire collection
Element URI, such as http://example.com/resources/item1?	Retrieve a representation of the collection's addressed member, expressed in an appropriate internal media type	Replace the collection's addressed member; if it doesn't exist, create it	Not generally used. Treat a collection's addressed member in its own right, and create a new entry of it	Delete the collection's addressed member

AJAX + RESTful

A hybrid form of AJAX and RESTful creates a syntax that can prove useful in specific situations.

A web app can fetch using RESTful data from the server. By using AJAX, this fetching is done asynchronously and in the background. AJAX makes a HTTP GET request using URL in this manner:

<http://example.com/resources/item17>

and receives data of item 17 in JSON which can be displayed in view (web page).

The following example of this use shows a weather example from wunderground.com. In this app, the user must make an account and receive a key in order to get Helsinki weather in JSON (<http://api.wunderground.com/api/your-key/conditions/q/Helsinki.json>).

```
{
  "response": {
    "version": "0.1",
```

```
    "termsofService": "http://www.wunderground.com/weather/api/d/terms.html",
    "features": {
      "conditions": 1
    }
  }.
  "current_observation": {
    "image": {
      "url": "http://icons.wxug.com/graphics/wu2/logo_130x80.png",
      "title": "Weather Underground",
      "link": "http://www.wunderground.com"
    },
    "display_location": {
      "full": "Helsinki, Finland",
      "city": "Helsinki",
      "state": " ",
      "state_name": "Finland",
      "country": "FI",
      "country_iso3166": "FI",
      "zip": "00000",
      "magic": "1",
      "wmo": "02974",
      "latitude": "60.31999969",
      "longitude": "24.96999931",
      "elevation": "56.0000000"
    }
  }
```

```

<!DOCTYPE html>
<html>
<head>
  <script src="../../angular.min.js" type="text/javascript"></script>
  <title></title>
</head>
<body data-ng-app="myapp">
  <div data-ng-controller="MyController">
    <button data-ng-click="myData.doClick(item, $event)">Get Helsinki Weather</button><br />
    Data from server: {{myData.fromServer}}
  </div>

  <script type="text/javascript">
    var myapp = angular.module("myapp", []);

    myapp.controller("MyController", function($scope, $http) {
      $scope.myData = {};
      $scope.myData.doClick = function(item, event) {
        var responsePromise = $http.get("http://api.wunderground.com/api/key/conditions/
q/Helsinki.json");

        responsePromise.success(function(data, status, headers, config) {
          $scope.myData.fromServer = "" + data.current_observation.weather +
            " " + data.current_observation.temp_c + " c";
        });
        responsePromise.error(function(data, status, headers, config) {
          alert("AJAX failed!");
        });
      });
    });
  </script>
</body>
</html>

```

This is JSON object!

This code example creates a **Get Helsinki Weather** button shown below the banner. After pressing the **Get Helsinki Weather** button, the current temperature and other details appears as shown in this example:



FIGURE 11. The Get Helsinki Weather Button Example

\$resource

Built on top of the \$http service, \$resource is a factory function that lets you interact with RESTful back-ends easily. \$resource does not come bundled with the main Angular script and must be separately downloaded from the Angular-resource.min.js file.

Your main app should declare dependency on the `ngResource` module in order to use `$resource`.

`$resource` expects a classic RESTful backend such as:

```
http://en.Wikipedia.org/wiki/Representational_state_transfer#Applied_to_web_services
```

You can create the backend by whatever technology you require - even JavaScript (for example, Node.js).

An example of the use of `$resource` in a script is shown in the following example:

```
// Load ngResource before this
var restApp = angular.module('restApp', ['ngResource']);

restApp.controller("RestCtrl", function($scope, $resource) {
  $scope.doClick = function() {
    var title = $scope.movietitle;
    var searchString = 'http://api.rottentomatoes.com/api/public/v1.0/movies.json?apikey=key&q=' + title + '&page_limit=5';

    var result = $resource(searchString);

    var root = result.get(function() { // {method: 'GET'}
      $scope.movies = root.movies;
    });
  }
});
```

Tuntematon

- Tuntematon sotilas (The Unknown Soldier) - 1955
- Tuntematon emanta (The Unknown Woman) - 2011
- The Unknown Soldier (Tuntematon sotilas) - 1985

The result of this code is to create a text field and fetch button; when text is entered and the button clicked, relevant text appears as shown in the example above.

`$resource` contains convenient methods for most operations including

- + Get (GET)
- + Save (POST)
- + Query (GET isarray:true)
- + Remove (DELETE)

Calling any of these will invoke `$https` (an AJAX call) with the specified HTTP method (GET, POST, DELETE), destination, and parameters.

Passing Parameters

```
// Load ngResource before this
var restApp = angular.module('restApp', ['ngResource']);

restApp.controller("RestCtrl", function($scope, $resource) {
  $scope.doClick = function() {
    var searchString = 'http://api.rottentomatoes.com/api/public/
v1.0/movies.json?apikey=key&q=:title&page_limit=5';
    var result = $resource(searchString);
    var root = result.get({title: $scope.movies.title}, function() {
      $scope.movies = root.movies;
    });
  }
});
```

`:title -> parametrized URL template`

Giving the parameter from \$scope

You can use services with `$resource` in code as shown below:

```
// Load ngResource before this
var restApp = angular.module('restApp', ['ngResource']);

restApp.controller("RestCtrl", function($scope, MovieService) {
  $scope.doClick = function() {
    var root = MovieService.resource.get({title: $scope.movies.title},
    function() {
      $scope.movies = root.movies;
    });
  }
});

restApp.factory('MovieService', function($resource) {
  factory = {};
  factory.resource = $resource('http://api.rottentomatoes...&q=:titles&page_limit=5');
  return factory;
});
```

Controller responsible for binding

Service responsible for the resource

Or as in this example:

```
// Load ngResource before this
var restApp = angular.module('restApp', ['ngResource']);

restApp.controller('RestCtrl', function($scope, MovieService) {
  $scope.doClick = function() {
    var root = MovieService.get({title: $scope.movietitle},
    function() {
      $scope.movies = root.movies;
    });
  }
});

restApp.factory('MovieService', function($resource) {
  return $resource('http://api.rottentomatoes...&q=:title&page_limit=5');
});
```

Just call get from MovieService

Returns the resource

AngularJS Animations

AngularJS can also accommodate animations of various sorts, such as button clicks and mouse-overs, that result in changes to the screen. Such animations can prove useful for certain types of prompt-and-response screens. This entails the inclusion of an `ngAnimate` module as a possible dependency. `ngAnimate` modules enable you to hook animations for common directives such as `ngRepeat`, `ngSwitch`, and `ngView`. All of these directives are based on CSS classes, so if an HTML element has a class, you can animate it.

AngularJS adds special classes to your html-elements. A simple example of animation and its result is shown in Figure 12.

```
<body ng-controller="AnimateCtrl">
  <button ng-click="add()">Add</button>
  <button ng-click="remove()">Remove</button></p>
  <ul>
    <li ng-repeat="customer in
customers">{{customer.name}}</li>
  </ul>
</body>
```

Animation Test



FIGURE 12. AngularJS Animation Example

Animation Classes

When adding a new name to the model, ng-repeat knows the item that is either added or deleted.

CSS classes are added at runtime to the repeated element, such as when adding a new element,

```
-<li class="...ng-enter ng-enter-active">New Name</li>
```

or removing an existing element,

```
-<li class="...ng-leave ng-leave-active">New Name</li>
```

Compiling/Deploying

Compiling the theme will require the use of the gulp tool. Gulp commands are defined in the /gulp/tasks folder of your theme project.

For example,

```
gulp theme:build
```

will prompt user for the target theme to compile. The compiled theme will be placed in the /dist folder. The dist folder places the themes that you send to clients or copy into your testing environment's theme folder.

HINT: During the build, all the JavaScript codes from the views, directives, and services are move into a single file 'app.min.js' and it is minified. The minified version prevents normal humans from editing it on the fly.

There is a way to prevent minification by going to the gulp/tasks/theme.build.js and find the following code snippet:

```
gulp.task('theme:build:js', false, function () {
  return gulp.src(config.js)
    .pipe(concat('app.min.js'))
    .pipe(uglify({mangle: true}))
    .pipe(gulp.dest(config.source))
    .on('error', error);
});
Comment out //.pipe(uglify({mangle: true}))
```

This will prevent the minification of the app.min.js file and allow us to continue to modify the angular code when it is already in the working environment.

Troubleshooting Compiling and Deploying

Locating the current angular theme view and directive from your current place in the SecureAuth application is easier using an Inspection Tool which is found in all the major browsers.

In this example, we use Chrome:

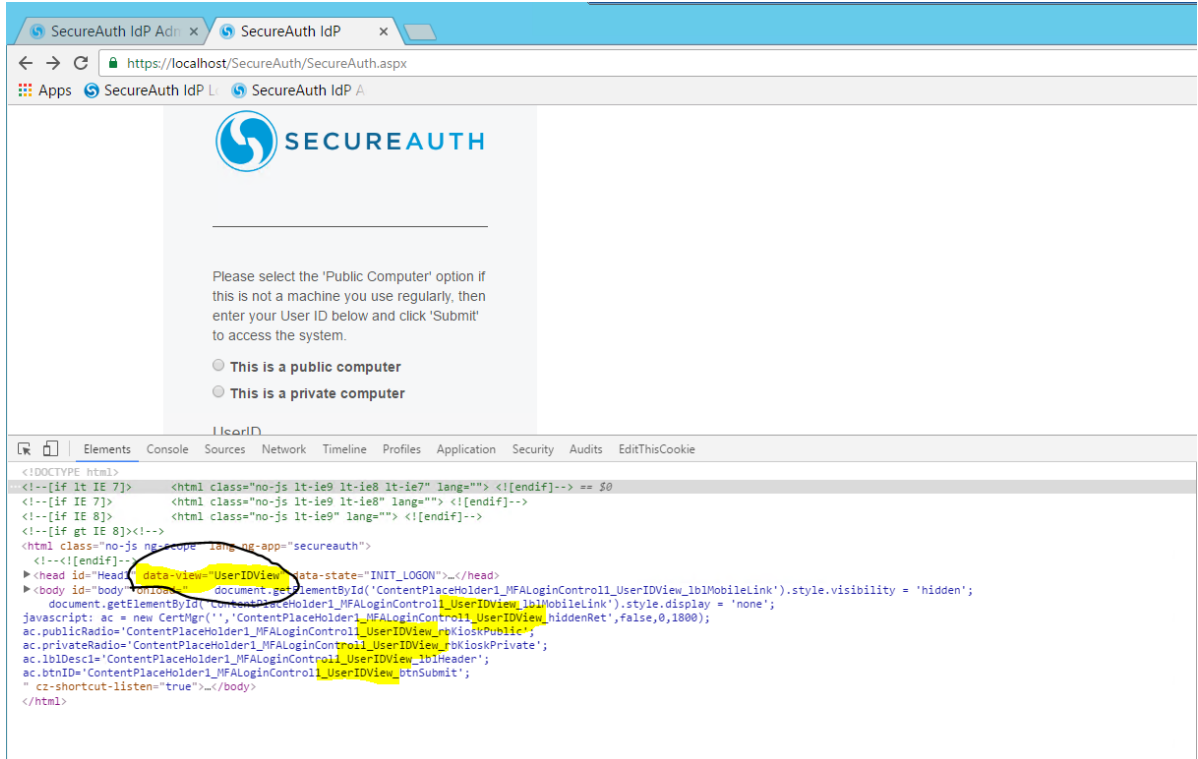


FIGURE 13. Chrome Example

The simplest way to locate the current view is by looking at the 'data-view' attribute in the <head> tag. If that doesn't exist, the app will try to locate state in 'ContentPlaceholder1_MFALoginControl1_' span tags by parse out the data-view value from that string. Lastly, if that fails, it will use the path in the URL.

To locate the directive in focus, use this example:

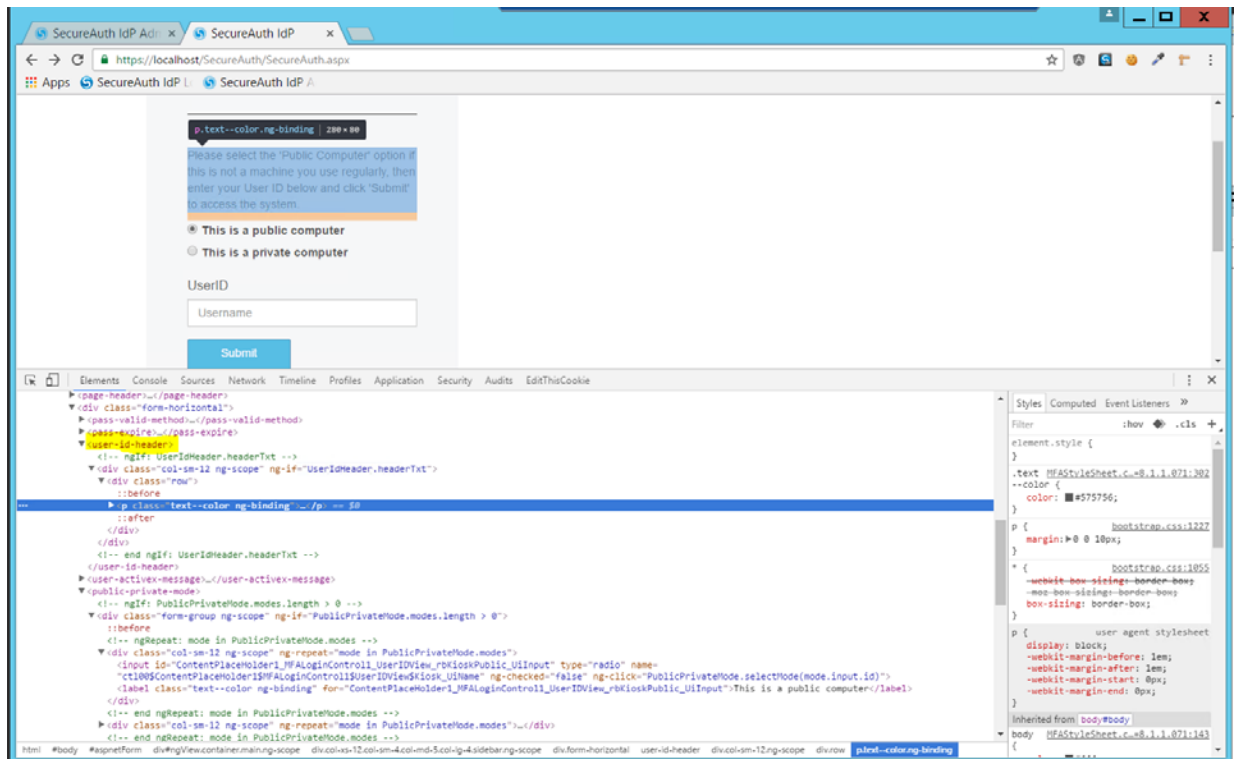


FIGURE 14. Locating the Directive

The directive is found in the DOM that encompasses the element you are targeting. In this example, we are targeting the heading text in the login page. The directive is “user-id-header”

Locating the angular view and directive to focus on is the first and possibly the most important step in troubleshooting or making modification in your theme.

Test Driving a Design

In general, this is the process you should take when preparing to test your code:

1. Write your tests first, then write your code.
2. AngularJS emphasizes modularity so it should be easy to test your code.
3. Code can be tested using several unit testing frameworks, including QUnit, Jasmine, and Mocha. One of the testing frameworks, QUnit, is described briefly in the following subsection.

QUnit

To install and use QUnit, do this:

1. Download qunit.js and qunit.css from the Qunit website (<https://qunitjs.com>).

2. Write a simple HTML page to run the tests.
3. Write the tests you require.

An example of using QUnit is shown in the following example:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit Example</title>
  <link rel="stylesheet" href="qunit=1.10.0.cas">
  <script src="qunit-1.10.0.js"></script>
</head>
<body>
  <div id="qunit">
    <script type="text/javascript">

      function calculate(a,b) {
        return a=b;
      }
      test( "calculate test", function() {
        ok( calculate(5,5) --- 10, "OK!" );
        ok( calculate(5,0) --- 5, "OK!" );
        ok( calculate(5,5) --- 0, "OK!" );
      });
    </script>
  </body>
</html>
```

There are three assertions you must make while writing test script in QUnit:

- + Basic -
 - `-ok(Boolean [,message]);`
- + If actual == expected
 - `-equal(actual, expected [, message]);`
- + If actual ===expected
 - `-deepEqual(actual, expected [, message]);`

For more information on automating test using qunit, please refer to: <http://quintjs.com/cookbook/#automating-unit-testing>.

Using these assertions, testing AngularJS Service using QUnit can look like this example:

```
var myApp = angular.module('myApp', []);
//One service
myApp.service("MyService", function() {
  this.add = function(a,b) {
    return a + b;
  };
});
/*TESTS*/
var injector = angular.injector(['ng', 'myApp']);
QUnit.test('MyService', function() {
  var MyService = injector.get('MyService');
  ok(2 == MyService.add(1,1));
});
```

Common Changes

This section provides information on some of the most common changes AngularJS can now support.

Styling Changes

- + Using the same example found in the Troubleshooting section, we want to change the font and color of the login heading text as shown in Figure 15.

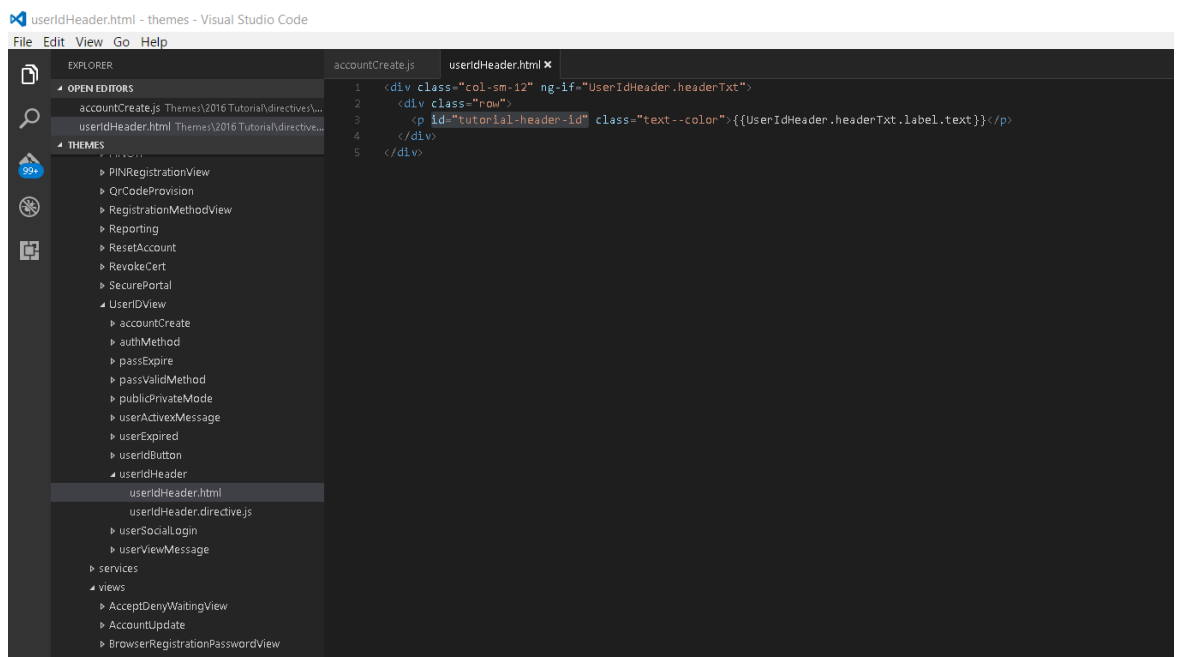


FIGURE 15. Changing the Font and Color of the Login Heading Text

- + There is an added `id="tutorial-header-id"` text element here to provide another way for you to change the styling. There are now three easy ways to change the styling for this text. The scss code below is added to the `app.scss` file.

```
#tutorial-header-id{
  font-weight: 800;
}
.text--color{
  color: blue;
}
user-id-header{
  p{
    font-size: 10px;
  }
}
```

- First method: we target ID directly and change the boldness of the text
- Second method: there was already a class `.text-color` that is associated with that text, we can use that to make a change to the color. This is NOT preferable since this change will affect other element that have this same class. (unless you want to do that)
- Third method: (personally preferred) we target the directive and target the tag within that directive to make the change. This prevents the CSS styling from leaking out of the intended target.

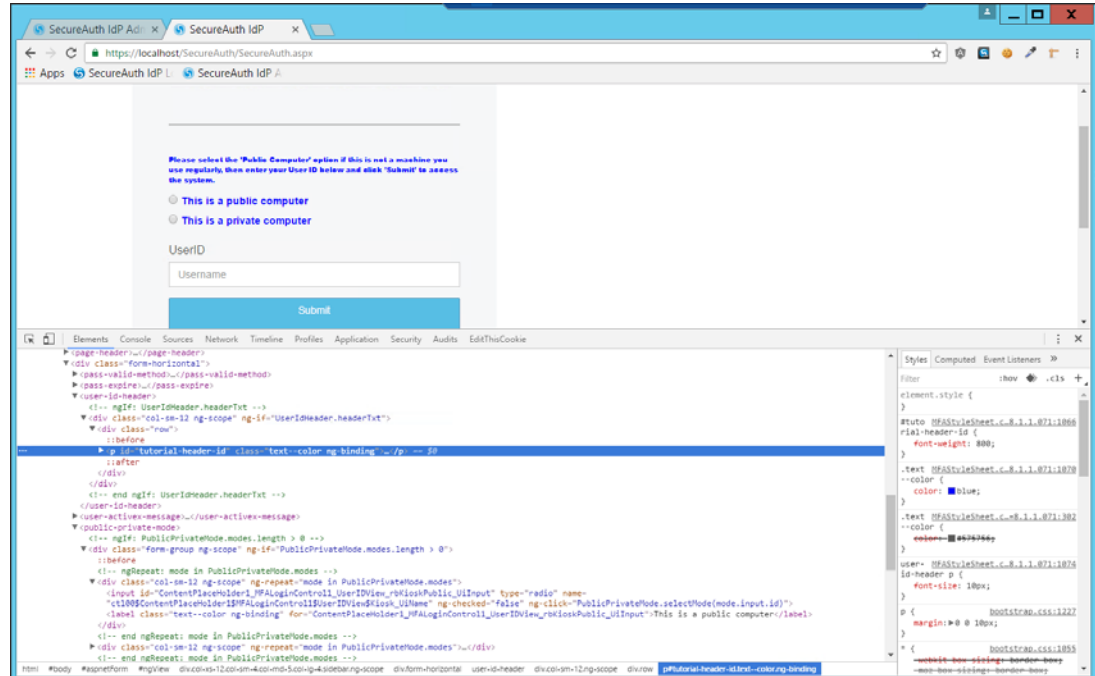


FIGURE 16. Styling Method Examples

As you can see, the CSS is applied in the screen shot from all three methods. But the only issue you will have is when you apply by the predefined class. This affects all elements with that class.

Text Changes

AngularJS now supports several types of text changes.

- + Using the same example in styling changes with the header text change. We want to create a custom header text.
- + The following changes have been entered to `userIdHeader.html`:

```
<div class="col-sm-12" ng-if="UserIdHeader.headerTxt">
  <div class="row">
    <!--<p id="tutorial-header-id" class="text--color">{{UserIdHeader.headerTxt.label.text}}</p-->
    <p class="text--color">This is the tutorial header text</p>
  </div>
</div>
```

The result of this change would be a page like the example in Figure 17.



FIGURE 17. Public/Private Computer Radio Button Example

Function Changes

To change the function of the submit button on the login page like this:

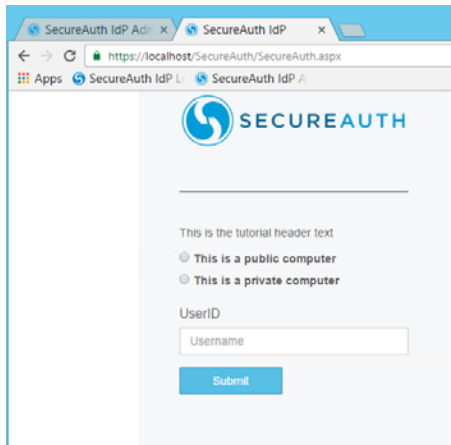
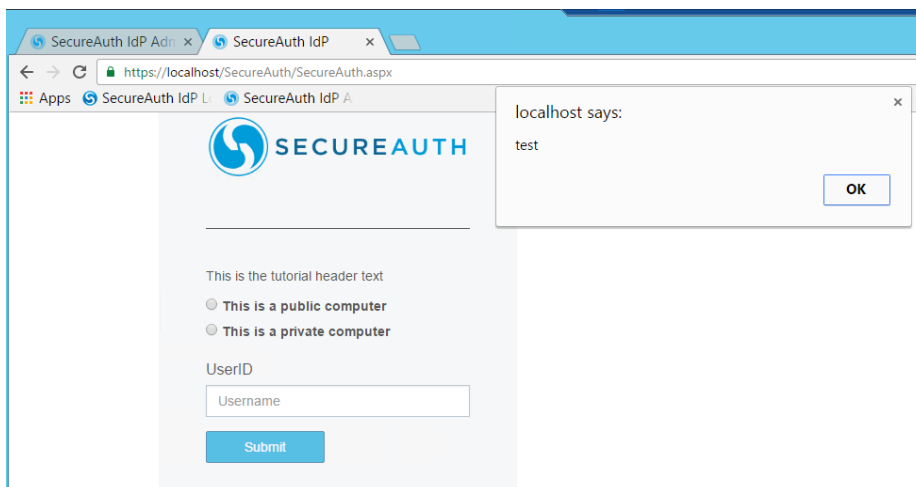


FIGURE 18. Login Page Example

Currently, this example is triggering the form-post. You can change the function to generate an alert message, such as in the following changes to the `userIdButton.directive.js` (see “Troubleshooting Compiling and Deploying” on page 39 for guidance on locating directives).



```
'use strict';

angular.module('secureauth')
  .directive('userIdButton', function (config, userIdView) {

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

```
angular.extend(vm, {
  submitted: 'false',
  userIdViewBtn: userIdView.getUserIdViewBtn()[0],
  click: function (id) {
    alert("test");
    // var valid = angular.element('#aspnetForm').hasClass('ng-
valid');
    // if (valid && vm.submitted === 'false') {
    //   angular.element('#' + id).trigger('click');
    //   vm.submitted = 'true';
    // }
  },
  dblclick: function () {
    return false;
  }
});

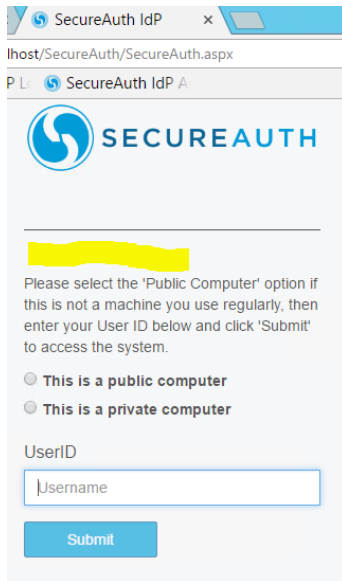
};

return {
  restrict: 'EA',
  controller: userIdButtonController,
  controllerAs: 'UserIdButton',
  templateUrl: config.theme + '/directives/UserIDView/userIdButton/use-
rIdButton.html',
  bindToController: true
};
});
```

Adding a New Element or Directive

Adding a new Element such as an `<a>` tag or a new line of text can be done in multiple ways.

Example: We want to add a link to the home login page right above the heading text.



Method One: Locate the directive in charge of printing the heading text `<user-id-header>` and know that this is the `UserIdView`. We can put the new link directly into the `<user-id-header>` directive html:

```
accountCreate.js  useridHeader.html x  SecureAuthLight.Master  userViewMessage.html
1  <a href="#">Tutorial Link</a>
2  <div class="col-sm-12" ng-if="UserIdHeader.headerTxt">
3    <div class="row">
4      <p class="text--color">{{UserIdHeader.headerTxt.label.text}}</p>
5    </div>
6  </div>
```

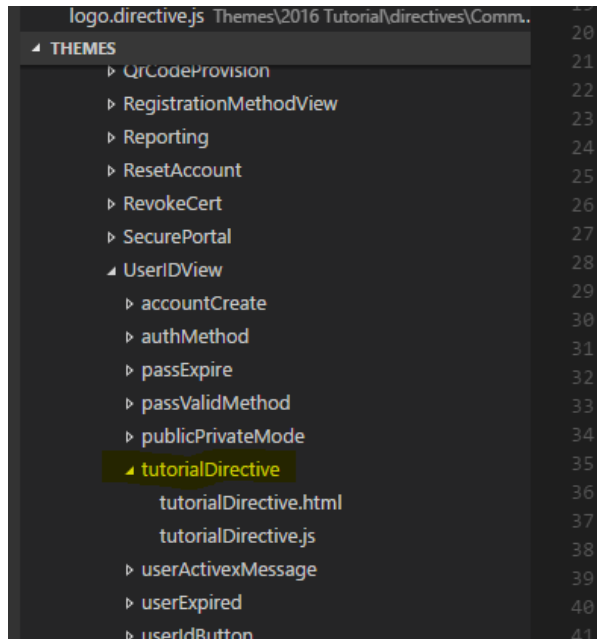
Method Two: Locate the UserIDView.html in the views folder and insert the new link there. This is NOT recommended because it strays from the structure of this framework.

```
accountCreate.js  userIDHeader.html  UserIDView.html x  Se
9
10 <page-header></page-header>
11
12 <div class="form-horizontal">
13
14 <pass-valid-method></pass-valid-method>
15 <pass-expire></pass-expire>
16 <a href="#">Tutorial Link</a>
17 <user-id-header></user-id-header>
18 <user-activex-message></user-activex-message>
19 <public-private-mode></public-private-mode>
20 <auth-method></auth-method>
21 <user-expired></user-expired>
22 <user-social-login></user-social-login>
23 <user-id-button></user-id-button>
24 <user-view-message></user-view-message>
25 </div>
26
27 <hr class="sidebar__hr hidden-xs">
28
```

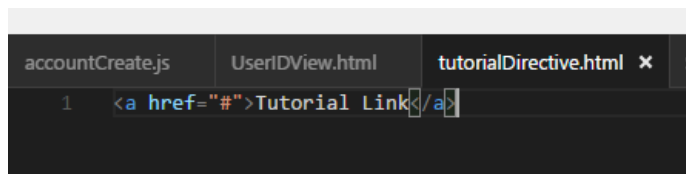
Method Three (Recommended): Create a new directive under UserIDView.

Using the gulp tool, type: `gulp theme:directive` and the tool prompts for the location where you would like to create the directive and the name of the directive. In this example, the name should be camelCased.

```
PS C:\Users\vtrung\Desktop\AngularTheme\themes\Themes\2016 Tutorial> gulp theme:directive
[08:42:57] Working directory changed to
(node:12120) fs: re-evaluating native module sources is not supported. If you are using the graceful
to a more recent version.
[08:42:59] Using gulpfile
[08:42:59] Starting 'theme:directive'...
[08:42:59] Starting 'theme:create:directive'...
[08:42:59] Finished 'theme:directive' after
] New directive for which theme? 2016 Tutorial
] New directive for which view? UserIDView
] what is your new directive name? tutorialDirective
```



The new directive named 'tutorialDirective.html' is created under the UserIDView folder in directives. In the tutorialDirective.html, insert the link code as shown below.



To add the directive to the UserIDView, go to the view folder and locate `useridview.html`. Add the new directive tag to it:

```
accountCreate.js  UserIDView.html x  tutorialDirective.html  SecureAuthLight.Master
9
10 <page-header></page-header>
11
12 <div class="form-horizontal">
13
14 <pass-valid-method></pass-valid-method>
15 <pass-expire></pass-expire>
16 <tutorial-directive></tutorial-directive>
17 <user-id-header></user-id-header>
18 <user-activex-message></user-activex-message>
19 <public-private-mode></public-private-mode>
20 <auth-method></auth-method>
21 <user-expired></user-expired>
22 <user-social-login></user-social-login>
23 <user-id-button></user-id-button>
24 <user-view-message></user-view-message>
25 </div>
26
27 <hr class="sidebar__hr hidden-xs">
28
29 <div class="sidebar__footer hidden-xs">
30 <forgot-username-link></forgot-username-link>
```

GitHub Tool

GitHub is a web-based Git repository hosting service. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. Version control systems like GitHub keep the revisions straight and store the modifications in a central repository. This allows developers to easily collaborate, as they can download a new version of the software, make changes, and upload the newest revision. Every developer can see these new changes, download them, and contribute.

Reproducing a Project

How do you clone and push from a personal repository back to a client branch? How do you add a non-forked project? There is a method we recommend for doing this.

1. Clone your IDP branch, version, or whatever you want copied to your new repository in this manner:



```
git clone git@sagithub-ent.secureauth.com:integration-development/idp.git
cd idp
git checkout <branch/tag/version>
```

2. Create a new repository in GitHub:

Create a new repository


A repository contains all the files for your project, including the revision history.


Owner **Repository name**

 tailoring-services ▾ / TeslaMotors 

Great repository names are short and memorable. Need inspiration? How about **turbo-disco**.

Description (optional)

 **Public**
Any logged in user can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

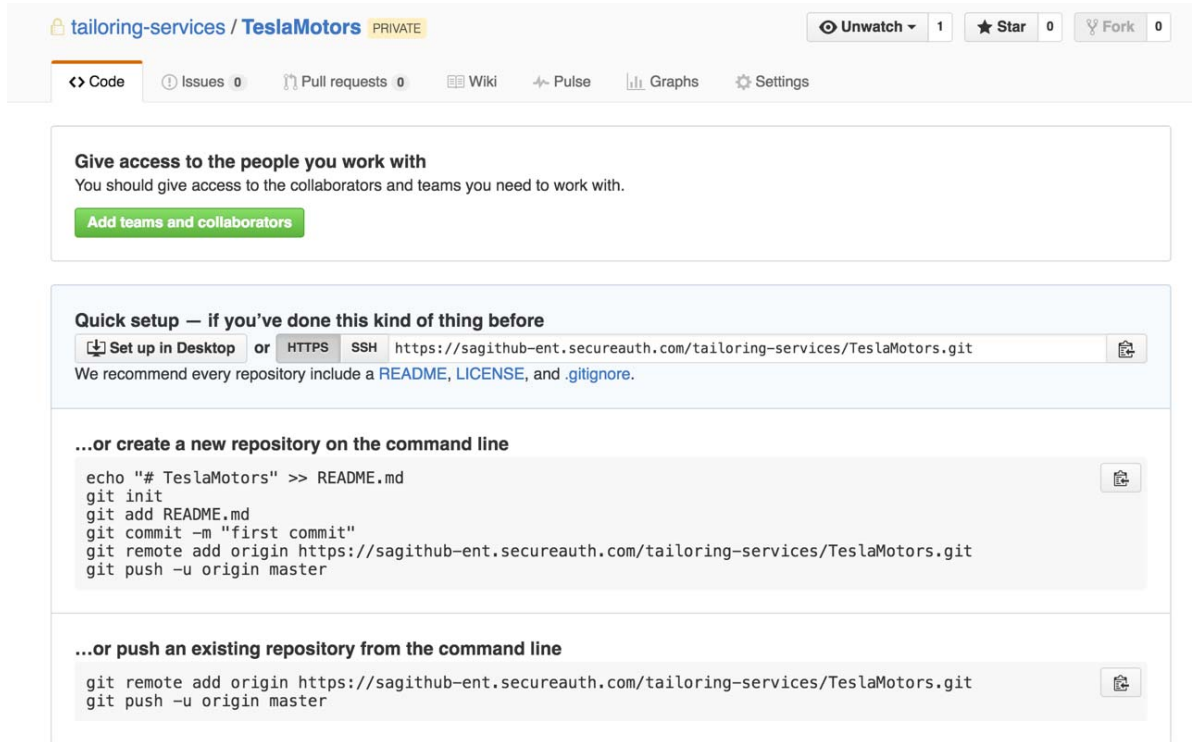
Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾

Create repository

FIGURE 19. GitHub Repository Example

3. Follow the steps provided to push an existing repository.



The screenshot shows the GitHub interface for a repository named 'tailoring-services / TeslaMotors' which is marked as 'PRIVATE'. The repository has 1 watch, 0 stars, and 0 forks. The main navigation bar includes 'Code', 'Issues 0', 'Pull requests 0', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. Below the navigation bar, there are three main sections:

- Give access to the people you work with**: A message stating 'You should give access to the collaborators and teams you need to work with.' with a green button labeled 'Add teams and collaborators'.
- Quick setup — if you've done this kind of thing before**: A section with a 'Set up in Desktop' button, followed by 'or' and 'HTTPS' and 'SSH' tabs. The URL 'https://sagithub-ent.secureauth.com/tailoring-services/TeslaMotors.git' is displayed in a text box. Below this, it says 'We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).'
- ...or create a new repository on the command line**: A code block containing the following commands:

```
echo "# TeslaMotors" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://sagithub-ent.secureauth.com/tailoring-services/TeslaMotors.git
git push -u origin master
```
- ...or push an existing repository from the command line**: A code block containing the following commands:

```
git remote add origin https://sagithub-ent.secureauth.com/tailoring-services/TeslaMotors.git
git push -u origin master
```

In this case, you might have to change “origin” to “new” in this manner:

```
git remote add new ....
git push -u new <branch/tag/version>
```

4. Now you have a “new” repository to use (and fork or change) and can create branches for each version.

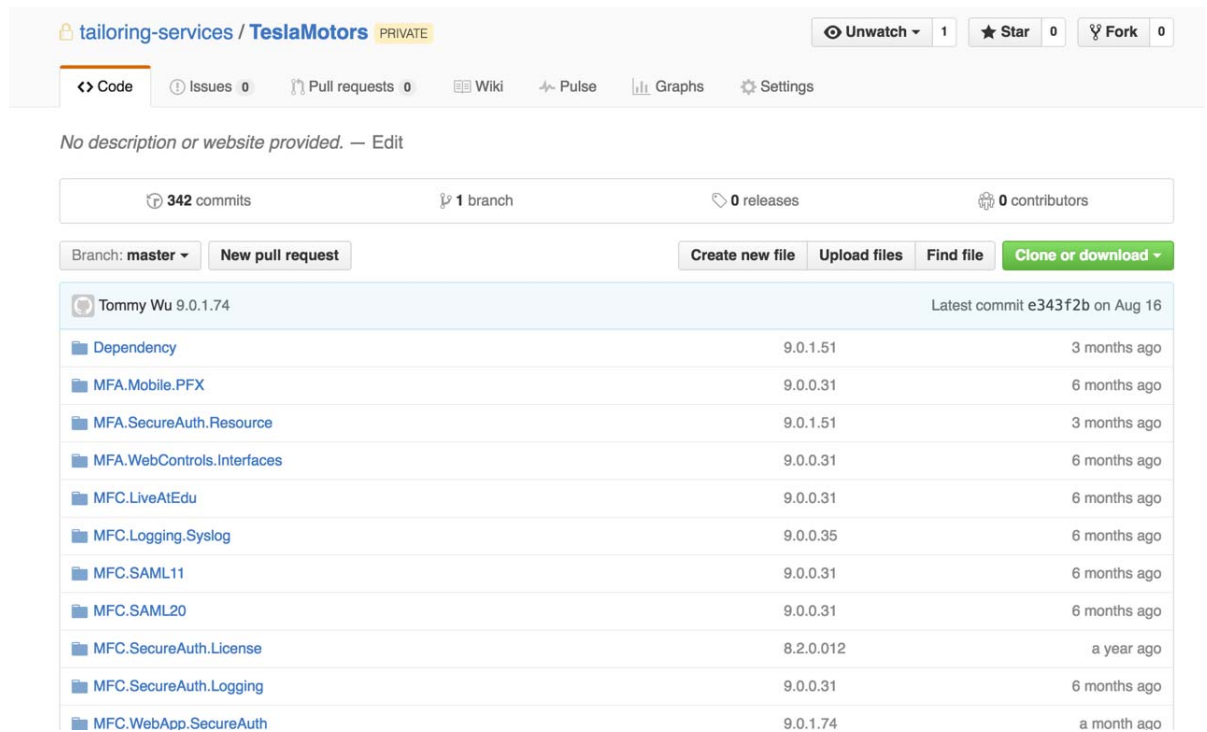


FIGURE 20. Branching and Forking Sample

GULP tools

GULP is one of the essential tools to create and compile code using AngularJS. There are several commands that you will use repeatedly.

TABLE 5. GULP Commands

Command	Uses / Reasons
gulp theme:view	Create a new view, such as for a new aspx form that requires a different view to display it
gulp theme:directive	Create a new directive, such as when a new element is needed to display new items/forms/text that shouldn't be added to another directive, or when new functionality is added to the aspx.
gulp theme:service	Create a new service, such as when a new view is created behind a new aspx page.
gulp theme:boilerplate	Create a new theme from default themes.
gulp theme:build	Compile a theme and place it in the distribution folder (/dist)

Route Object: Resolving Property

When routing to a view, the route object contains a “resolve” property that can accept a map of promises and wait for them to resolve before performing the route.

```
angular.module('moduleName', [])
  config(function($routeProvider){
    $routeProvider.when("/", {
      templateUrl: 'view.html',
      controller: 'controller.js',
      resolve: //promise
    })
  })
});
```

View Controller

If every view managed by a controller can reflect the project structure by packing it together and coming up with naming conventions, then views can comprise:

- + View-name
 - View-name.html
 - View-name.js
- + Another-view
 - Another-view-name.html
 - Another-view-name.js

The view-name.js is defined in the `.controller('viewNameCtrl'... argument`

Load Dependencies

Load dependencies occur as a reaction to an event. One approach is to load resources depending on the user behavior in this manner:

- + Load only when a user starts to fill a form
- + Load by mouse position
- + Load when the server returns a response

Conclusions

AngularJS is a modular JavaScript SPA framework with lots of great features, but there is a hard learning curve. The language is great for creating, reading, updating, and deleting apps, but unsuitable for every type of app. However, it works extremely well with some JS libraries, such as JQuery.

Questions and Answers

How do you load scripts asynchronously?

RequireJS provides a clean way to load and manage dependencies as shown in this example:

```
<script data-main="main" src="require.js"></script>
define(function() {
  //module code
})
require(['module'], function(module) {
  //use this module
})
```

How do you register components against a module after Bootstrap?

There are three essential points to remember when registering components.

- + Components register against the module in the *config phase* using providers. Register controllers manually using `$controllerProvider`. An example of registering components is shown in the following example:

```
angular.module('moduleName', [])
  .config(function($controllerProvider) {
    $controllerProvider.register('Ctrl', function() {
      // controller code
    })
  });
```

- + All components can be registered with their matching provider methods:

```
// services can register with $provide
$provide.service()
$provide.factory(),
$provide.value(),
$provide.constant(),
// other components use specific providers
$controllerProvider.register()
$animateProvider.register()
$filterProvider.register()
$compileProvider.directive()
```

- + Hold a reference to this provider in order to use it later in code.

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

```
.config(function($controllerProvider) {
    app.loadController = $controllerProvider.register;
})
});
app.loadController('comeCtrl', function($scope) {})
```

When does the actual loading occur?

Loading the program occurs when:

- + **routing** to a view - \$routeProvider
- + **loading** content -<ng-include>
- + In **response** to event - like click or hover

Best Practices

Some of the best practices we like to recommend are:

- + Controllers and services should not reference the DOM
- + Controllers should have view behavior
- + Services should have reusable logic, independent of the view
- + Scope should be read-only in the templates
- + Be careful with simple examples
- + Your architectures should reflect the system not the frameworks you used in your system

MVC Flow

In general, MVC flow should follow the pattern shown in Figure 21.

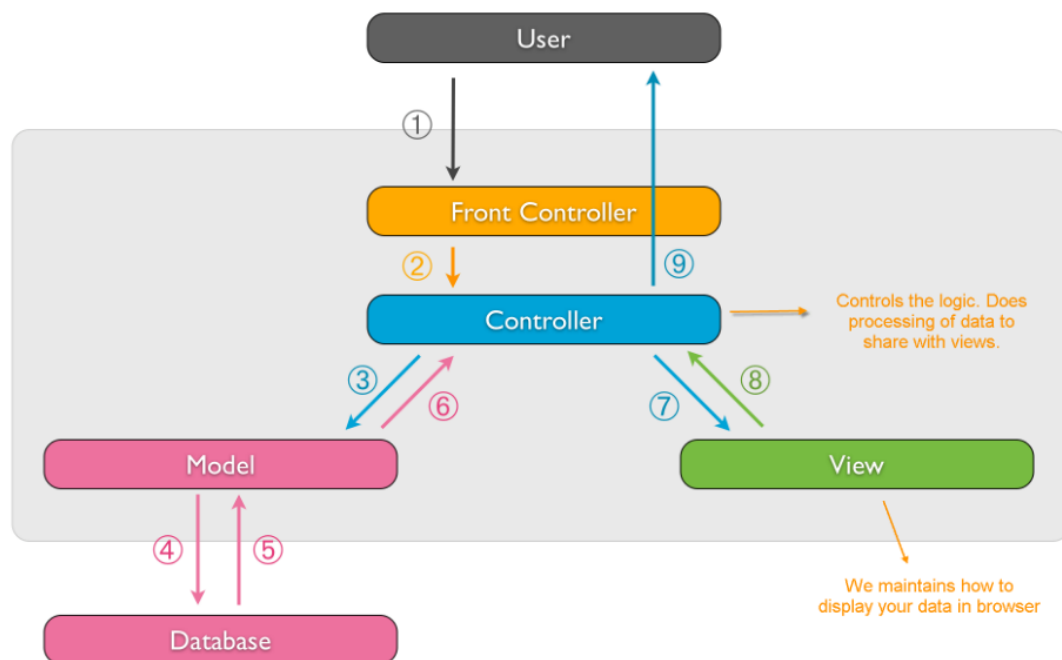


FIGURE 21. Recommended MVC Flow

Large Applications

Separating your code into multiple files is considered a best practice when building large applications with AngularJS. For example:

```
angular.module('myApp.controllers', [])
  controller('MyCtrl1', function() {
  controller('myCtrl2', function() {
  });
```

Define each module as a dependency, as shown in this example:

```
angular.module('myApp', ['ngRoute',
  'myApp.filters',
  'myApp.services',
  'myApp.directives',
  'myApp.controllers',
  'ngRoute',
  'ngResource',
  'ui.bootstrap',
])
```

Dependency injection (DI) is a software design pattern that deals with how components get hold of their dependencies. The angular injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

Components such as services, directives, filters, and animations are defined by an injectable factory method or constructor function – injected with either “service” or “value” components.

Controllers are defined by a constructor function which can be injected with any of the “service” and “value” components as dependencies, but they can also be provided with special dependencies.

The run method accepts a function which can be injected with “provider” and “constant” components as dependencies. You cannot inject “service” or “value” components into a configuration.

Factory Methods

Factory functions use a unique way of defining a directive, service, or filter and are registered with modules. The recommended way to do this is shown in the following example:

```
angular.module('myModule', [])
  .factory('serviceId', ['depService', function(depService) {
  //...
```

```
    }])
    .directive('directiveName', ['depService', function(depService) {
        //...
    }])
    .filter('filterName', ['depService', function(depService) {
        //...
    }]);
```

Module Methods

Specify functions to run at configuration and runtime for a module by calling the `config` and `run` methods. These functions are injectable with dependencies just like factory functions.

```
angular.module('myModule', [])
.config(['depProvider', function(depProvider) {
    //...
}])
.run(['depProvider', function(depProvider) {
    //...
}]);
```

Controllers

Controllers are “classes” or “constructor functions” that are responsible for providing the application behavior that supports the declarative markup in the template. The recommended way to handle this is:

```
someModule.controller('MyController', ['$scope', 'dep1', 'dep2', function(
    $scope, dep1, dep2) {
    ...
    $scope.aMethod = function() {
    ...
    }
    ...
}]);
```

Unlike services, there can be many instances of the same type of controller in an application. Additional dependencies are made available to controllers in this manner:

- + `$scope` – controllers are associated with an element in the DOM and so are provided with access to the scope
- + Other components (like services) only have access to the `$rootScope` service

If a controller is instantiated as part of a route, then any values that are resolved as part of the route are made available for injection into the controller.

Dependency Annotation

Angular invokes certain functions (like service factories and controllers) via the injector. You need to annotate these functions so the injector knows what services to inject into the function.

Three ways of annotating code with service information:

- + Using the inline array annotation (preferred)
- + Using the `$inject` property annotation
- + Implicitly from the function parameter names (has caveats)

The preferred approach, inline array annotation, involve the pass of an array whose elements consist of a list of strings (the names of the dependencies) followed by the function itself. When using this type of annotation, take care to keep the annotation array in sync with the parameters in the function declaration.

```
someModule.controller('MyController', ['$scope', 'greeter', function($scope, greeter) {
    // ...
}]);
```

\$Inject Property Annotation

To allow the minifiers to rename the function parameters and still be able to inject the right services, the function needs to be annotated with the `$inject` property. The `$inject` property is an array of service names to inject.

```
var MyController = function($scope, greeter) {
    // ...
}
MyController.$inject = ['$scope', 'greeter'];
someModule.controller('MyController', MyController);
```

For more on this topic, refer to <https://javascript-minifier.com/>.

Implicit Annotation

If minifying will be done to the code, the service names will be renamed and will break the app.

The simplest way to get hold of the dependencies is to assume that the function parameter names are the names of the dependencies.

```
someModule.controller('MyController', function($scope, greeter) {
  // ...
});
```

Strict Dependency Injection

The `Ng-strict-di` directive acts on the same elements as `ng-app` for strict DI mode in code such as this:

```
<!doctype html>
<html ng-app="myApp" ng-strict-di>
<body>
  I can add: {{ 1 + 2 }}.
  <script src="angular.js"></script>
</body>
</html>
```

'willBreak' Service

When the `willBreak` service is instantiated, Angular will generate an error because of strict mode. This can prove useful when using `ng-annotate` to ensure that all of application components have annotations.

```
angular.module('myApp', [])
.factory('willBreak', function($rootScope) {
  // $rootScope is implicitly injected
})
.run(['willBreak', function($rootScope) {
  // Angular will throw when this runs
}]);
```

Declarative Programming

Declarative programming creates user interfaces and connects software components. For more on this features, refer to: <https://www.youtube.com/watch?v=jfiMRue456w>.

Large AngularJS Application

When writing a large program using AngularJS, almost all components can be brought into play as shown in Figure 22.

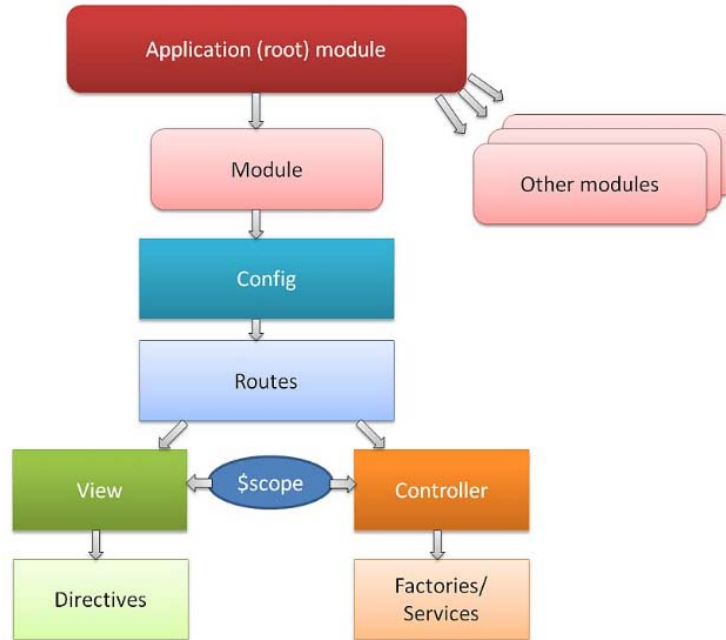


FIGURE 22. Large AngularJS Application Flowchart

This includes most of the objects and components previously defined and highlighted in this guide.

References

For more information, please refer to the following topics:

- + <http://docs.angularjs.org/guide>
- + <https://github.com/angular/angular.js/wiki>
- + <http://www.thinkster.io/pick/taQOoMGII>
- + <http://www.youtube.com/angularjs>
- + <https://egghead.io/>
- + <http://joehooks.com>