Agenda

Module1:

Introduction, Single Page Application, Download AngularJS

Module2:

Directives, Filters, Expressions and Data Binding

Module3:

Templates, Views, Model, Controllers, and Scope

Module4:

Modules, Routes, Services and Factories

Module5:

Dependency Injection, Sample Angular Application

Module6:

Angular UI and Nested Routing

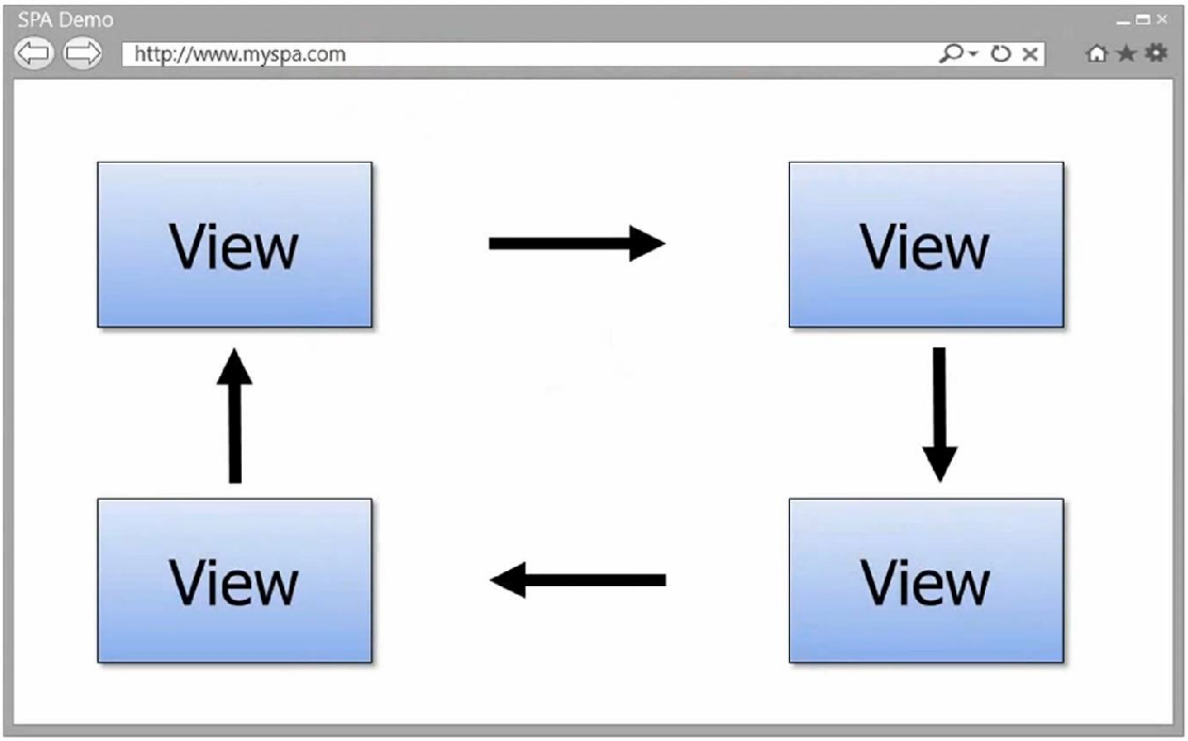
Module7:

More about AngularJS

Module8:

Angular UI Bootstrap

Single Page Application (SPA)



Introduction to Single Page Application (SPA):

What's a SPA?

A SPA is an application that runs inside a browser (or in a "compatible" environment) and does not require page reloading during use.

You can think of a SPA as a thick client that is loaded from a web server.

You most likely have already used this type of application: Gmail, Google Maps, Google Drive, iCloud, GitHub, Hotmail, Soundcloud, or Trello are common SPAs.

A Single Page Application is one in which we have a shell page and we can load multiple views into that.

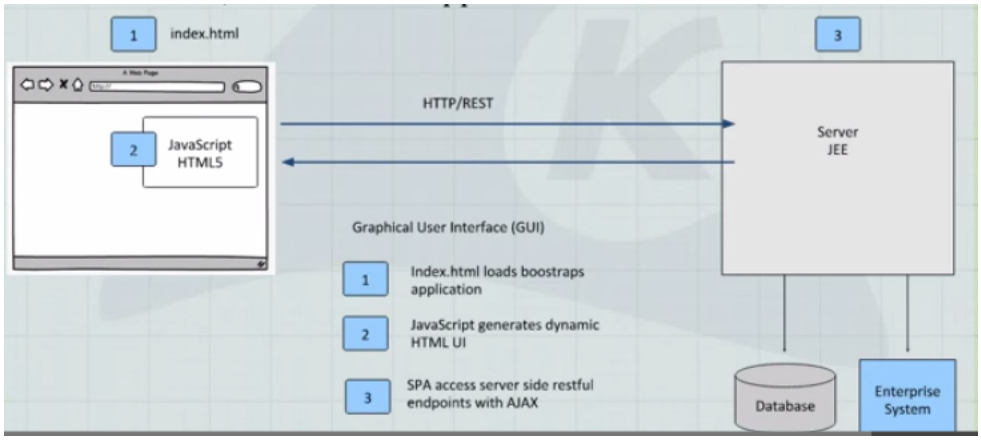
So a traditional app, as you know you typically blink and load everything again.

It's not very efficient on the bandwidth, especially in the mobile world.

In a SPA we can load the initial content upfront and then the different views or the little kind of mini- web pages can be loaded on the fly and embedded into the shell.

User Interface is implemented in 100% client (browser) side JavaScript.

Server Side, RESTful API is used as application data access.



Why SPAs?

- Richness and Responsiveness
- Manipulate UI elements asynchronously and without screen refresh
- Eliminates need for plug-ins
- Lot less bandwidth consume, UI more responsive, batteries last longer

Implementing SPAs

- Just JavaScript, jQuery, with UI components is not enough, can be done but, difficult to maintain and support
- MVC Framework Required
    - Separate UI from Application Logic
    - Dependency Management
    - Modularity

What Is AngularJS?

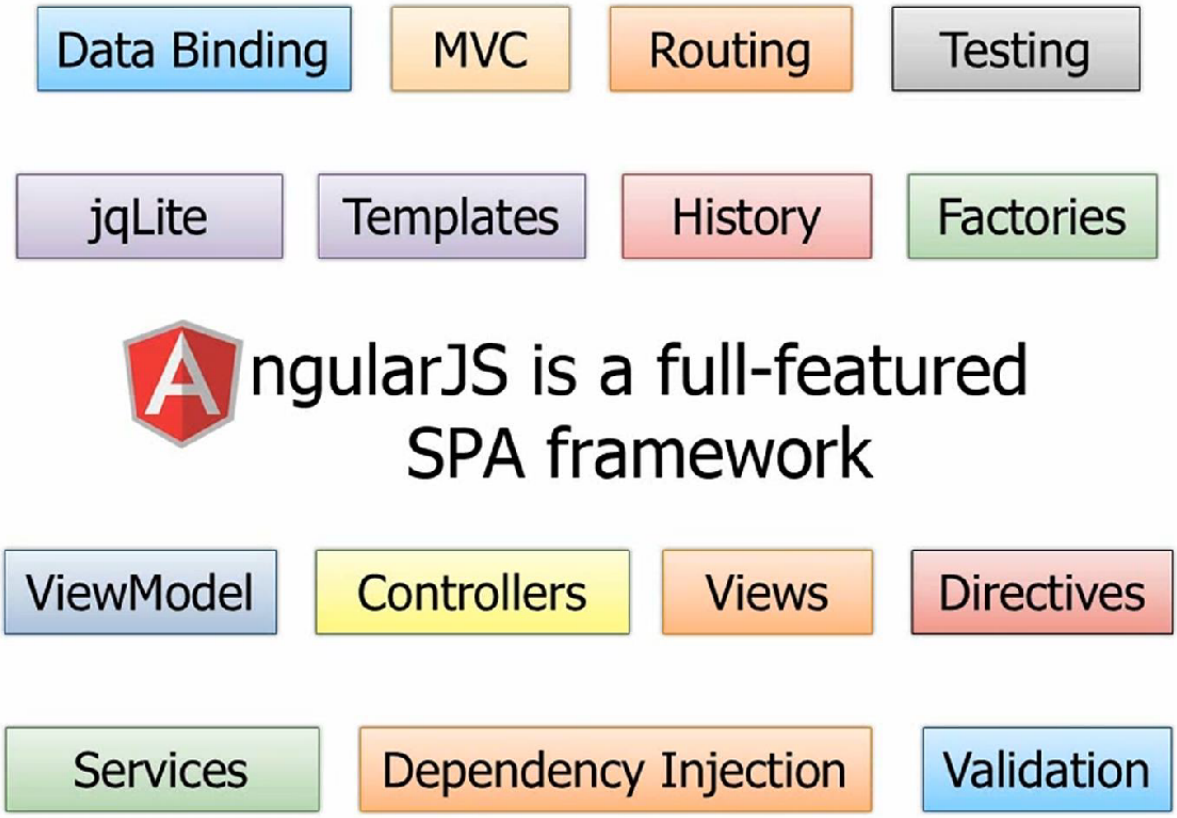
AngularJS is a JavaScript MVC framework developed by Google that lets you build well structured, easily testable, and maintainable front-end applications.

AngularJS is a structural framework for dynamic web apps.

It lets you use HTML as your template language and lets you extend HTML's syntax to express your application's components clearly

Angular data binding and dependency injection eliminate much of the code you would otherwise have to write.

And it all happens within the browser, making it an ideal partner with any server technology

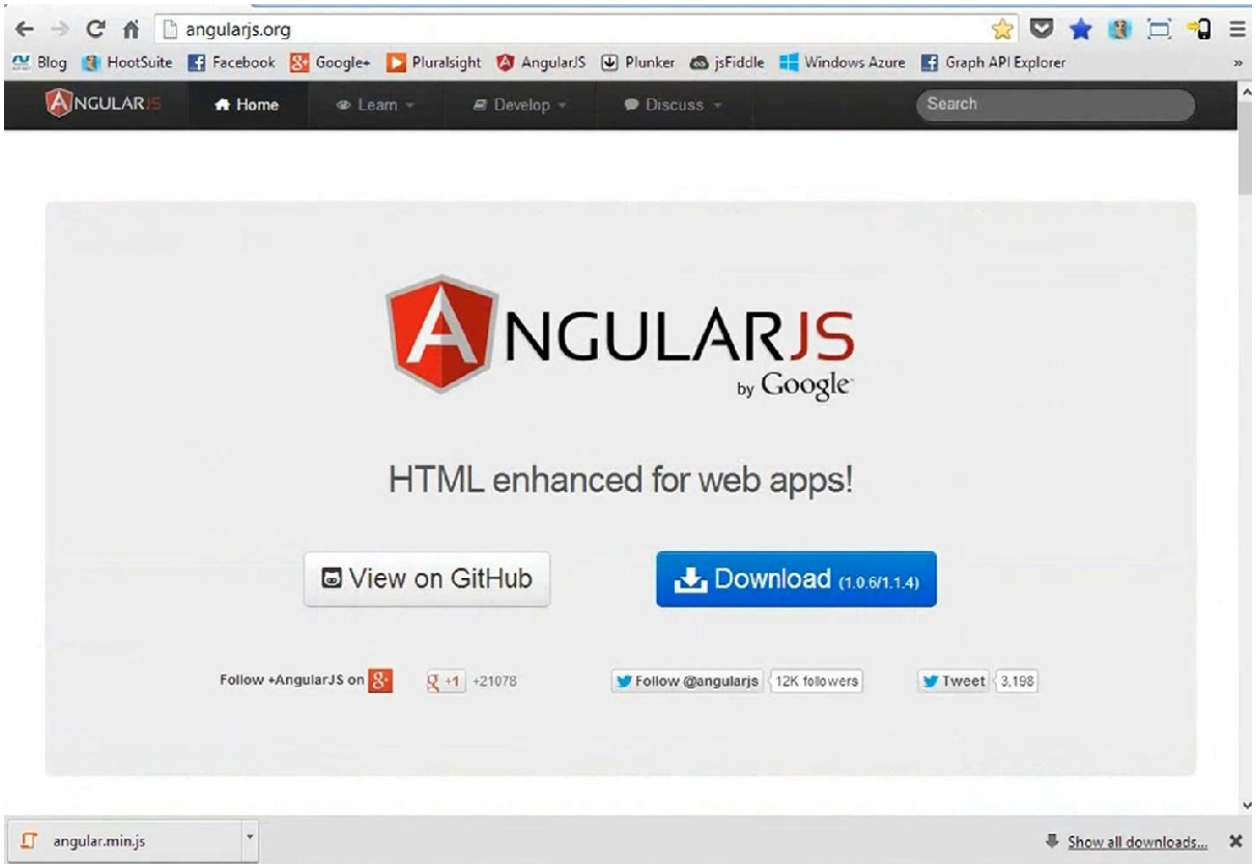


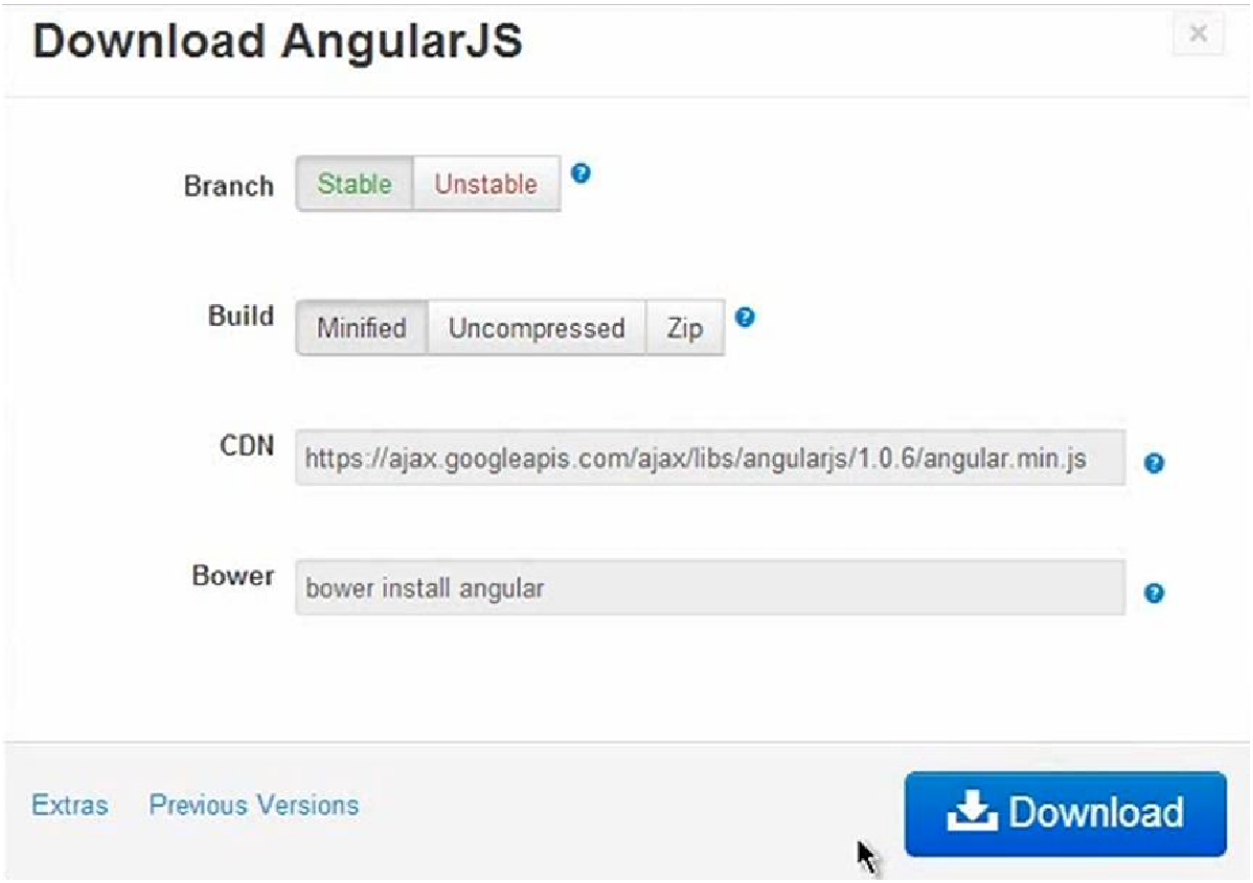
Overview of AngularJS:

Template	HTML with additional markup
Directives	extend HTML with custom attributes and elements
Model	the data shown to the user in the view and with which the user interacts
Scope	context where the model is stored so that controllers, directives and expressions can access it
Expressions	access variables and functions from the scope
Compiler	parses the template and instantiates directives and expressions

Filter	formats the value of an expression for display to the user
View	what the user sees (the DOM)
Data Binding	sync data between the model and the view
Controller	the business logic behind views
Dependency Injection	Creates and wires objects and functions
Injector	dependency injection container
Module	a container for the different parts of an app including controllers, services, filters, directives which configures the Injector
Service	reusable business logic independent of views

AngularJS.org


angularjs.org
Blog
HootSuite
Facebook
Google+
Pluralsight
AngularJS
Plunker
jsFiddle
Windows Azure
Graph API Explorer
ANGULARJS
Home
Learn
Develop
Discuss
Search
ANGULARJS
by Google
HTML enhanced for web apps!
View on GitHub
Download (1.0.6/1.1.4)
Follow +AngularJS on
+1
+21078
Follow @angularjs
12K followers
Tweet
3,198
angular.min.js
Show all downloads...


Download AngularJS
Branch
Stable
Unstable
Build
Minified
Uncompressed
Zip
CDN
https://ajax.googleapis.com/ajax/libs/angularjs/1.0.6/angular.min.js
Bower
bower install angular
Extras
Previous Versions
Download

Directives, Filters and Data Binding

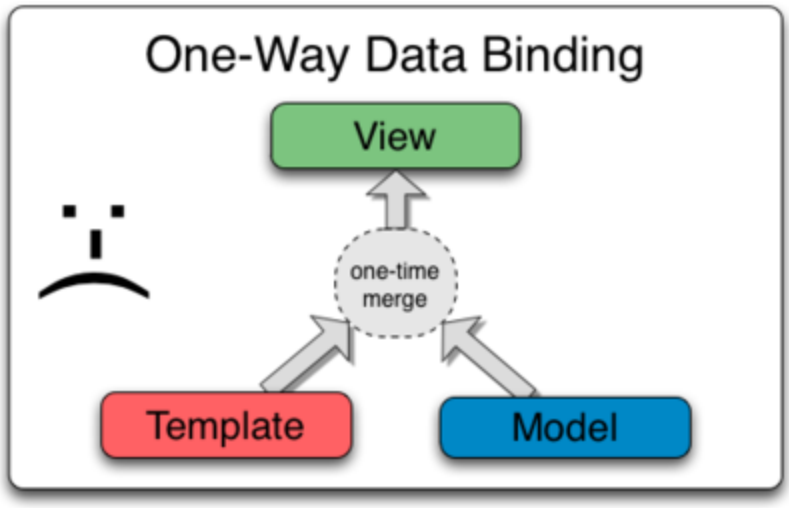
Data Binding in AngularJS

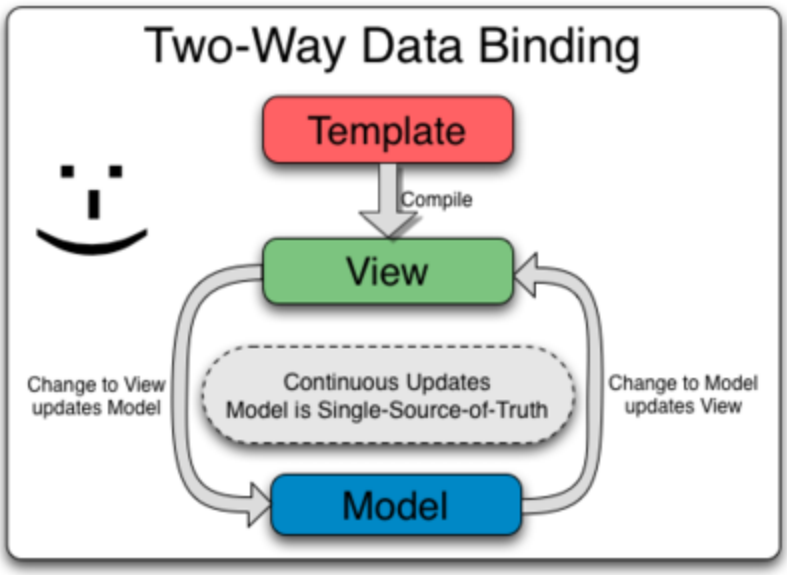
Data-binding in Angular apps is the automatic synchronization of data between the model and view components.

The way that Angular implements data-binding lets you treat the model as the single-source-of-truth in your application.

The view is a projection of the model at all times.

When the model changes, the view reflects the change, and vice versa.



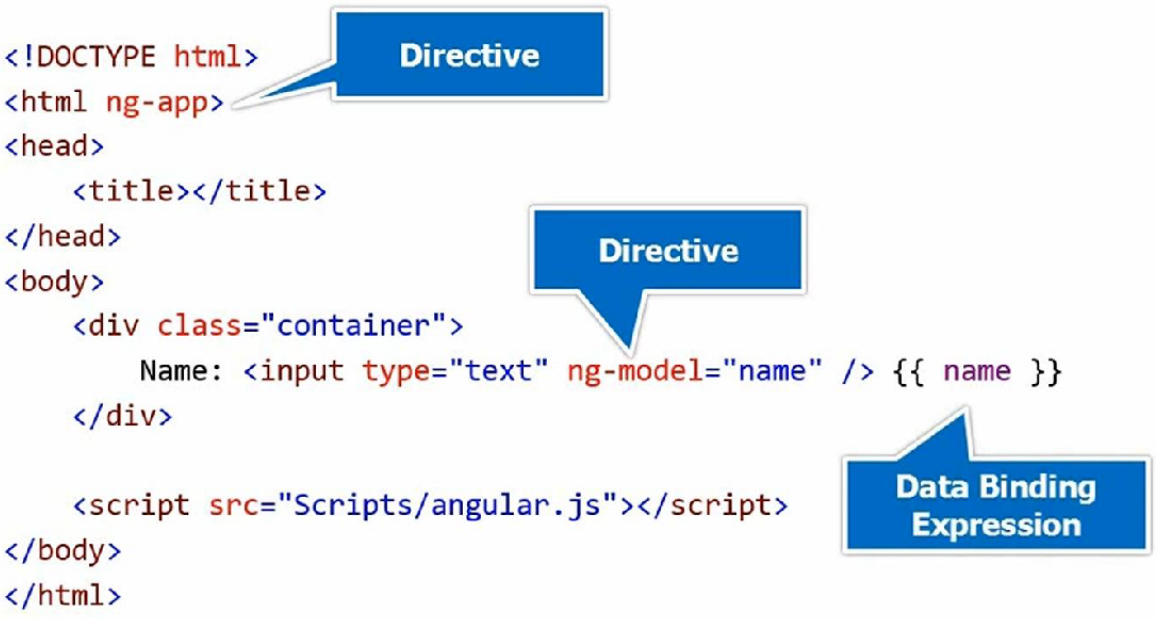


What are Directives?

At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's **HTML compiler** (`$compile`) to attach a specified behavior to that DOM element or even transform the DOM element and its children.

Angular comes with a set of these directives built-in, like `ngBind`, `ngModel`, and `ngClass`.

Using Directives and Data Binding Syntax




```
<!DOCTYPE html>
<html ng-app>
<head>
    <title></title>
</head>
<body>
    <div class="container">
        Name: <input type="text" ng-model="name" /> {{ name }}
    </div>

    <script src="Scripts/angular.js"></script>
</body>
</html>
```

Notice at the top we have **ng-app**.

Any time you see **ng-** that is an Angular directive.

It's a built-in directive.

You can also write custom ones.

This particular directive is very important because the script that's now loaded is going to kick off and this will initialize the Angular app.

Right now we don't have any particular module associated or any other code but we can still do stuff just by adding **ng-app**.

So for example, this is an example of another directive called **ng-model**.

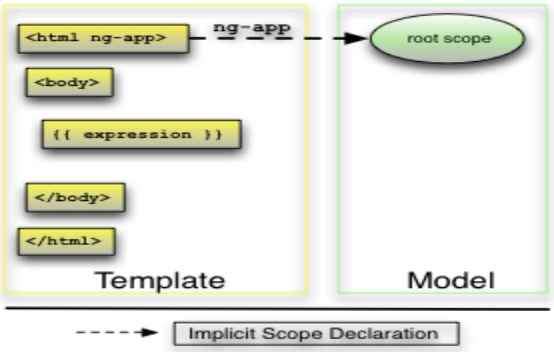
What **ng-model** does is behind the scenes it's going to add a property up in the memory called "name" into what's called "the **scope**".

You can see that as I type it automatically binds it.

Include the **ng-app**

Include the **ng-model**

Bind to that model.



Iterating with the ng-repeat Directive

```
<html data-ng-app="">
...

   <div class="container"
      data-ng-init="names=['Dave','Napur','Heedy','Shriva']">

       <h3>Looping with the ng-repeat Directive</h3>
       <ul>

       </ul>
   </div>

...
</html>
```

```
<div class="container"
   data-ng-init="names=['Dave','Napur','Heedy','Shriva']">

    <h3>Looping with the ng-repeat Directive</h3>
    <ul>
        <li data-ng-repeat="name in names">{{ name }}</li>
    </ul>
</div>
```

So again, we have the **ng-app**, the **ng-init**: these are two directives. Then the third is **ng-repeat** which will simply loop through all the names, and then data-bind or apply the value into the **<li>.**

Data binding

```
<div ng-app ng-init="qty=1;cost=2">
 <b>Invoice:</b>
 <div>
   Quantity: <input type="number" min="0" ng-model="qty">
 </div>
 <div>
   Costs: <input type="number" min="0" ng-model="cost">
 </div>
 <div>
   <b>Total:</b> {{qty * cost | currency}}
 </div>
</div>
```

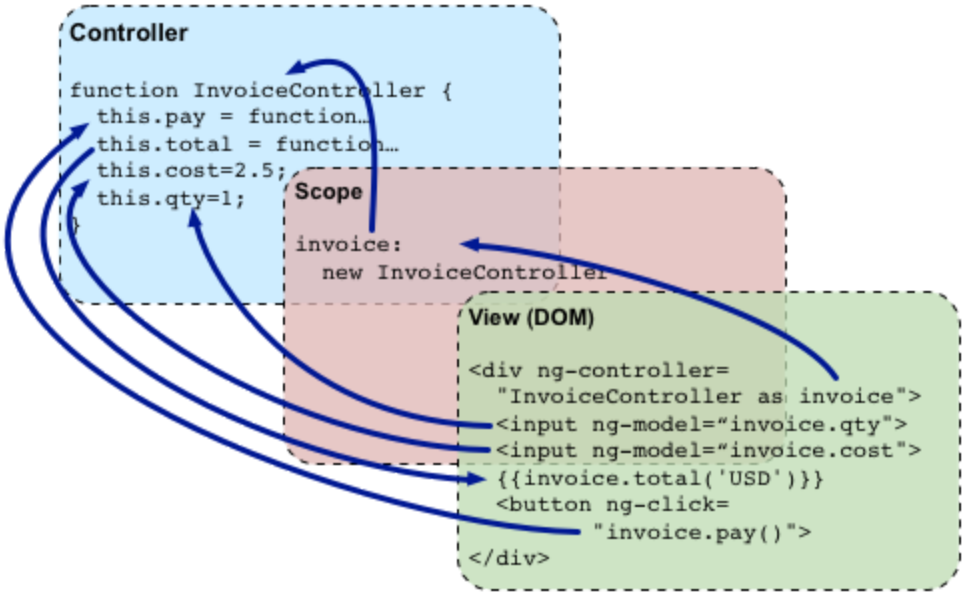
Adding UI logic: Controllers

```
angular.module('invoice1', [])
.controller('InvoiceController', function() {
 this.qty = 1;
 this.cost = 2;
 this.inCurr = 'EUR';
 this.currencies = ['USD', 'EUR', 'CNY'];
 this.usdToForeignRates = {
   USD: 1,
   EUR: 0.74,
   CNY: 6.09
 };

 this.total = function total(outCurr) {
   return this.convertCurrency(this.qty * this.cost, this.inCurr, outCurr);
 };
```

```
  this.convertCurrency = function(amount, inCurr, outCurr) {
    return amount * this.usdToForeignRates[outCurr] / this.usdToForeignRates[inCurr];
  };
  this.pay = function pay() {
    window.alert("Thanks!");
  };
});
```

Index.html

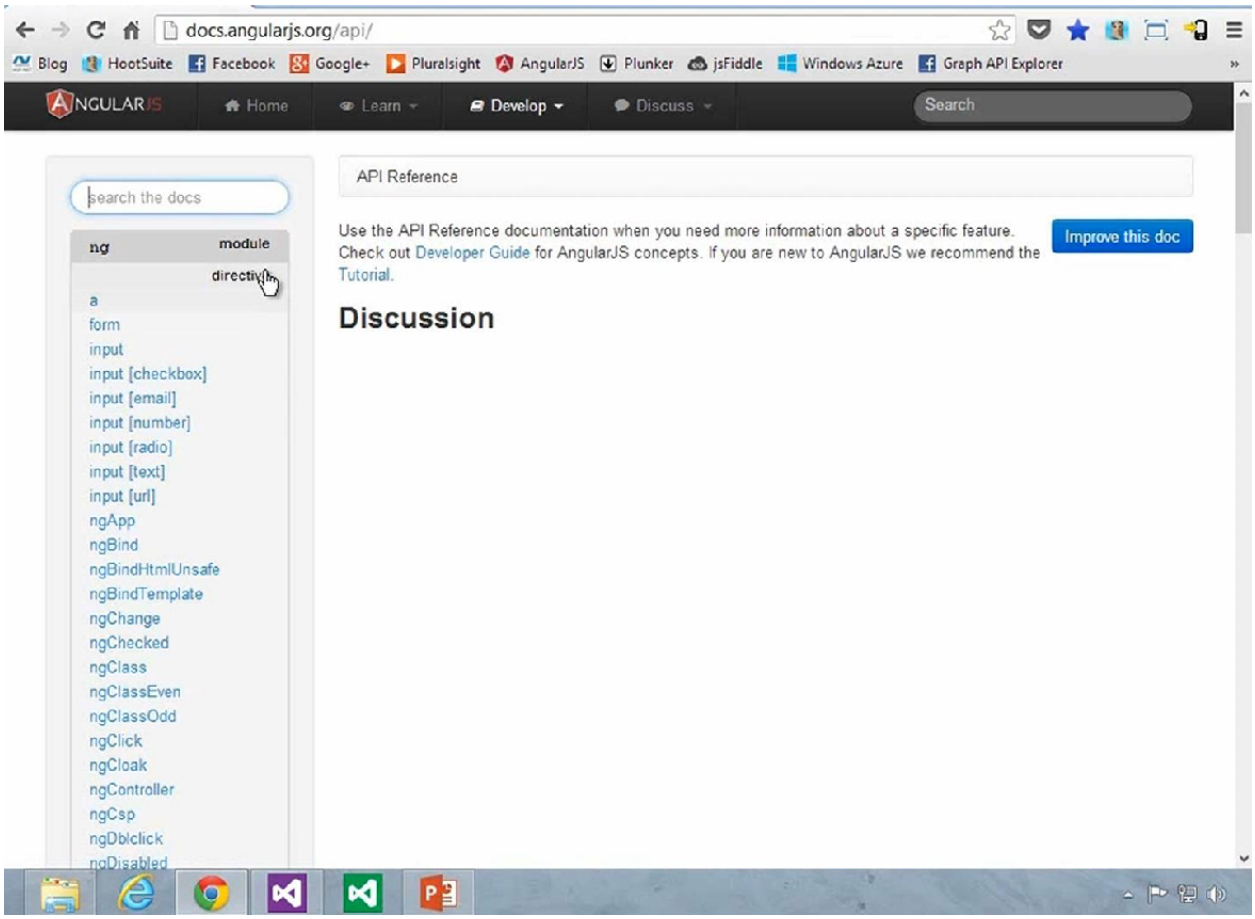
```
<div ng-app="invoice1" ng-controller="InvoiceController as invoice">
  <b>Invoice:</b>
  <div>
    Quantity: <input type="number" min="0" ng-model="invoice.qty" required >
  </div>
  <div>
    Costs: <input type="number" min="0" ng-model="invoice.cost" required >
    <select ng-model="invoice.inCurr">
      <option ng-repeat="c in invoice.currencies">{{c}}</option>
    </select>
  </div>
  <div>
    <b>Total:</b>
    <span ng-repeat="c in invoice.currencies">
      {{invoice.total(c) | currency:c}}
    </span>
    <button class="btn" ng-click="invoice.pay()">Pay</button>
  </div>
</div>
```


Controller
function InvoiceController {
this.pay = function…
this.total = function…
this.cost=2.5;
this.qty=1;
}
Scope
invoice:
new InvoiceController
View (DOM)
<div ng-controller=
"InvoiceController as invoice">
<input ng-model="invoice.qty">
<input ng-model="invoice.cost">
{{invoice.total('USD')}}
<button ng-click=
"invoice.pay()">

When you use directives one of the nice things you can do is go off to the documentation. One of the best things you need to know about is go to “Develop”…

.. and select “API Reference”.

The AngularJS API Reference for Directives



Using Filters

Using Filters

```
<ul>
    <li data-ng-repeat="cust in customers | orderBy:'name'">
        {{ cust.name | uppercase }}
    </li>
</ul>
```

```
<input type="text" data-ng-model="nameText" />
<ul>
    <li data-ng-repeat="cust in customers | filter:nameText |
                        orderBy:'name'">
      {{ cust.name }} - {{ cust.city }}</li>
</ul>
```

Again, if you go off to http://angularjs.org , go to “API Reference” and then scroll on down a little bit you’ll see a whole list of the filters.

Example for Filter:

```
Search: <input ng-model="query">
Sort by:
<select ng-model="orderProp">
  <option value="name">Alphabetical</option>
  <option value="age">Newest</option>
</select>


<ul class="phones">
  <li ng-repeat="phone in phones | filter:query | orderBy:orderProp">
    <span>{{phone.name}}</span>
    <p>{{phone.snippet}}</p>
  </li>
</ul>



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

phonecatApp.controller('PhoneListCtrl', function ($scope) {
  $scope.phones = [
    {'name': 'Nexus S',
     'snippet': 'Fast just got faster with Nexus S.',
     'age': 1},
    {'name': 'Motorola XOOM™ with Wi-Fi',
     'snippet': 'The Next, Next Generation tablet.',
     'age': 2},
    {'name': 'MOTOROLA XOOM™',
     'snippet': 'The Next, Next Generation tablet.',
     'age': 3}
  ];

  $scope.orderProp = 'age';
});
```

Angular Templates:

In Angular, templates are written with HTML that contains Angular-specific elements and attributes.

Angular combines the template with information from the model and controller to render the dynamic view that a user sees in the browser.

These are the types of Angular elements and attributes you can use:

- Directive — an attribute or element that augments an existing DOM element or represents a reusable DOM component.
- Markup — the double curly brace notation `{{ }}` to bind expressions to elements is built-in Angular markup.
- Filter — Formats data for display.
- Form controls — Validates user input.

The following code snippet shows a template with directives and curly-brace expression bindings:

```
<html ng-app>
 <!-- Body tag augmented with ngController directive  -->
 <body ng-controller="MyController">
   <input ng-model="foo" value="bar">
   <!-- Button tag with ng-click directive, and
        string expression 'buttonText'
        wrapped in "{{ }}" markup -->
   <button ng-click="changeFoo()">{{buttonText}}</button>
   <script src="angular.js">
 </body>
</html>
```

In a simple app, the template consists of HTML, CSS, and Angular directives contained in just one HTML file (usually `index.html`).

Views, Controllers and Scope

Understanding Controllers

In Angular, a Controller is a JavaScript **constructor function** that is used to augment the Angular Scope.

When a Controller is attached to the DOM via the ng-controller directive, Angular will instantiate a new Controller object, using the specified Controller's **constructor function**.

A new **child scope** will be available as an injectable parameter to the Controller's constructor function as `$scope`.

Use controllers to:

- Set up the initial state of the `$scope` object.
- Add behavior to the `$scope` object.

Setting up the initial state of a `$scope` object

Typically, when you create an application you need to set up the initial state for the Angular `$scope`.

You set up the initial state of a scope by attaching properties to the `$scope` object.

The properties contain the **view model** (the model that will be presented by the view).

All the `$scope` properties will be available to the template at the point in the DOM where the Controller is registered.

The following example demonstrates creating a `GreetingController`, which attaches a `greeting` property containing the string `'Hola!'` to the `$scope`:

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

myApp.controller('GreetingController', ['$scope', function($scope) {
  $scope.greeting = 'Hola!';
}]);
```

We attach our controller to the DOM using the `ng-controller` directive. The `greeting` property can now be data-bound to the template:

```
<div ng-controller="GreetingController">
  {{ greeting }}
</div>
```

Adding Behavior to a Scope Object

In order to react to events or execute computation in the view we must provide behavior to the scope.

We add behavior to the scope by attaching methods to the `$scope` object.

These methods are then available to be called from the template/view.

The following example uses a Controller to add a method, which doubles a number, to the scope:

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

myApp.controller('DoubleController', ['$scope', function($scope) {
  $scope.double = function(value) { return value * 2; };
}]);
```

Once the Controller has been attached to the DOM, the `double` method can be invoked in an Angular expression in the template:

```
<div ng-controller="DoubleController">
  Two times <input ng-model="num"> equals {{ double(num) }}
</div>
```

Scope Inheritance Example

It is common to attach Controllers at different levels of the DOM hierarchy.

Since the ng-controller directive creates a new child scope, we get a hierarchy of scopes that inherit from each other.

The `$scope` that each Controller receives will have access to properties and methods defined by Controllers higher up the hierarchy.

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

myApp.controller('MainController', ['$scope', function($scope) {
  $scope.timeOfDay = 'morning';
  $scope.name = 'Nikki';
}]);

myApp.controller('ChildController', ['$scope', function($scope) {
  $scope.name = 'Mattie';
}]);

myApp.controller('GrandChildController', ['$scope', function($scope) {
  $scope.timeOfDay = 'evening';
  $scope.name = 'Gingerbread Baby';
}]);
```

```
<div class="spicy">
  <div ng-controller="MainController">
    <p>Good {{timeOfDay}}, {{name}}!</p>

    <div ng-controller="ChildController">
      <p>Good {{timeOfDay}}, {{name}}!</p>

      <div ng-controller="GrandChildController">
        <p>Good {{timeOfDay}}, {{name}}!</p>
      </div>
    </div>
  </div>
</div>
```

View, Controllers and Scope

$scope is the "glue" (ViewModel) between a controller and a view

The way it works in Angular is you have a **View**, which is what we've been doing in the previous section with our **Directives**, our **Filters** and our **Data Binding**.

But we don't want to put all of our logic into the **View** because it's not very maintainable or testable or all those types of things.

Instead we're going to have a special little JavaScript object – a container - called a **Controller**. The **Controller** will drive things.

It's going to control ultimately what data gets bound into the **View**.

If the View passes up data to the controller it will handle passing off maybe to a service which then updates a back-end data store.

The glue between the View and the Controller is something called the **Scope**, and in Angular you're going to see a lot of objects or variables that start with $.

$scope represents the scope object.

When I say it's the glue, it literally is the thing that ties the controller to the view.

Note:

The view doesn't have to know about the controller, and the controller definitely doesn't want to know about the view.

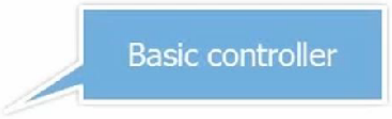
A **ViewModel** literally is the **model** – the data – for the view.

Well that's really all the scope is.

The scope is our ViewModel and it's the glue between the view and the controller.

Creating a View and Controller

```
<script>
    function SimpleController($scope) {

        $scope.customers = [
           { name: 'Dave Jones', city: 'Phoenix' },
           { name: 'Jamie Riley', city: 'Atlanta' },
           { name: 'Heedy Wahlin', city: 'Chandler' },
           { name: 'Thomas Winter', city: 'Seattle' }
        ];

    }
</script>
```



Creating a View and Controller

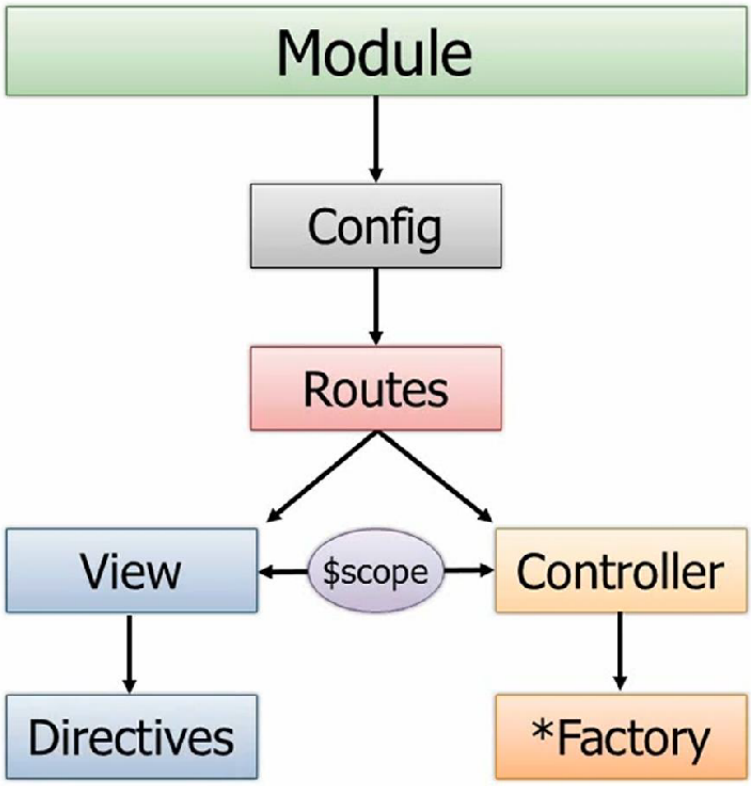
```
<div class="container" data-ng-controller="SimpleController">
    <h3>Adding a Simple Controller</h3>
    <ul>
        <li data-ng-repeat="cust in customers">
           {{ cust.name }} - {{ cust.city }}
        </li>
    </ul>
</div>

<script>
    function SimpleController($scope) {

        $scope.customers = [
           { name: 'Dave Jones', city: 'Phoenix' },
           { name: 'Jamie Riley', city: 'Atlanta' },
           { name: 'Heedy Wahlin', city: 'Chandler' },
           { name: 'Thomas Winter', city: 'Seattle' }
        ];

    }
</script>
```

$scope injected dynamically

```
<ul>
    <li data-ng-repeat="cust in customers">
       {{ cust.name }} - {{ cust.city }}
    </li>
</ul>
```

Access $scope

Modules, Routes and Factories


Module
Config
Routes
View
$scope
Controller
Directives
*Factory

What is a Module?

You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.

Why Module?

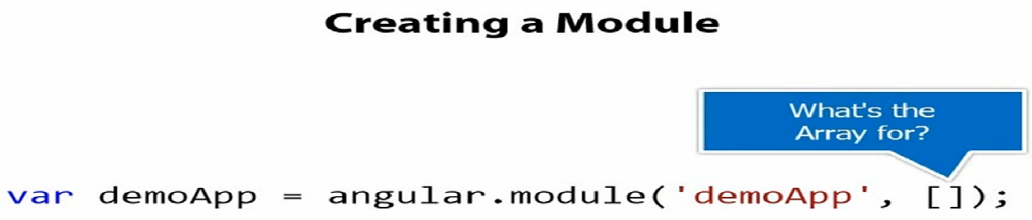
Most applications have a main method that instantiates and wires together the different parts of the application.

Angular apps don't have a main method.

Instead modules declaratively specify how an application should be bootstrapped.

There are several advantages to this approach:

- The declarative process is easier to understand.
- You can package code as reusable modules.
- The modules can be loaded in any order (or even in parallel) because modules delay execution.
- Unit tests only have to load relevant modules, which keeps them fast.
- End-to-end tests can use modules to override configuration.



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

Important things to notice:

- The Module API
- The reference to `myApp` module in `<div ng-app="myApp">`. This is what bootstraps the app using your module.
- The empty array in `angular.module('myApp', [])`. This array is the list of modules `myApp` depends on.

So a module can have something off of it called a **config** function, and it can be defined to use different **routes**.

Now routes again are really important in the SPA world.

Because if you have different views and those views need to be loaded into the shell page then we need a way to be able to track what route we're on and what view that's associated with and then what controller goes with that view and how we do all of that marrying together of these different pieces.

When you define a route in ***AngularJS*** you can define two things on that route:

One of those is the view. So what view when that route such as “unit.org/orders” then maybe go to “orderspartial.html” or “ordersfragment.html” or whatever you want to call it.

Then that view needs a controller.

Instead of hard-coding the controller into the view – which works and you can certainly do it.

A given controller would then of course have access to the scope object which then the view will bind to.

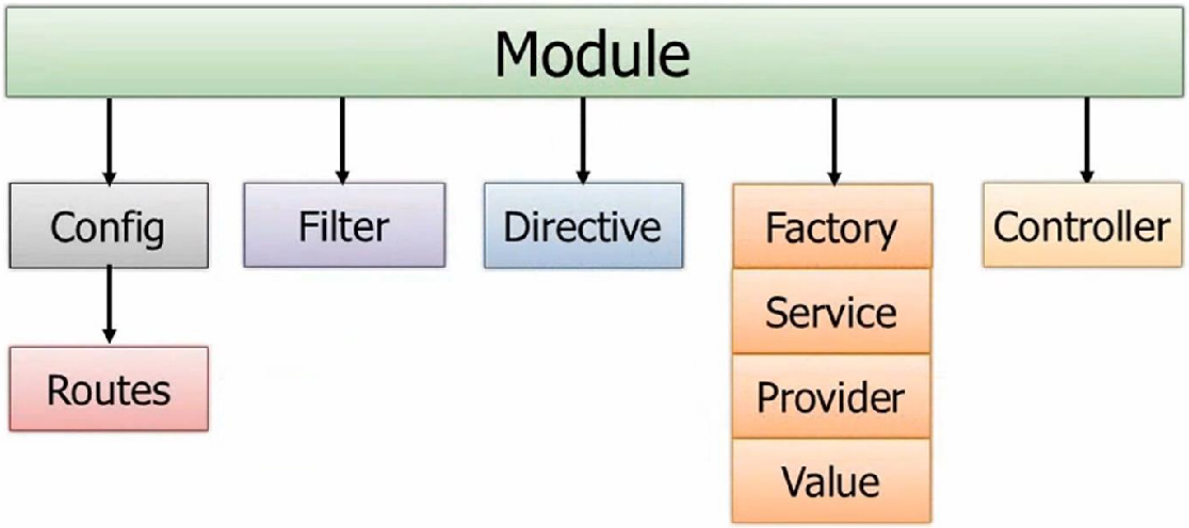
And then controllers rather than having all the functionality to get the data and update the data and perform CRUD operations and things like that, in a more real life application they’ll call out to factories or…

I put a star there because you might have services, providers or even values you want to get.

There are a lot of different ways you can access data.

Modules are Containers

```
<html ng-app="moduleName">
```

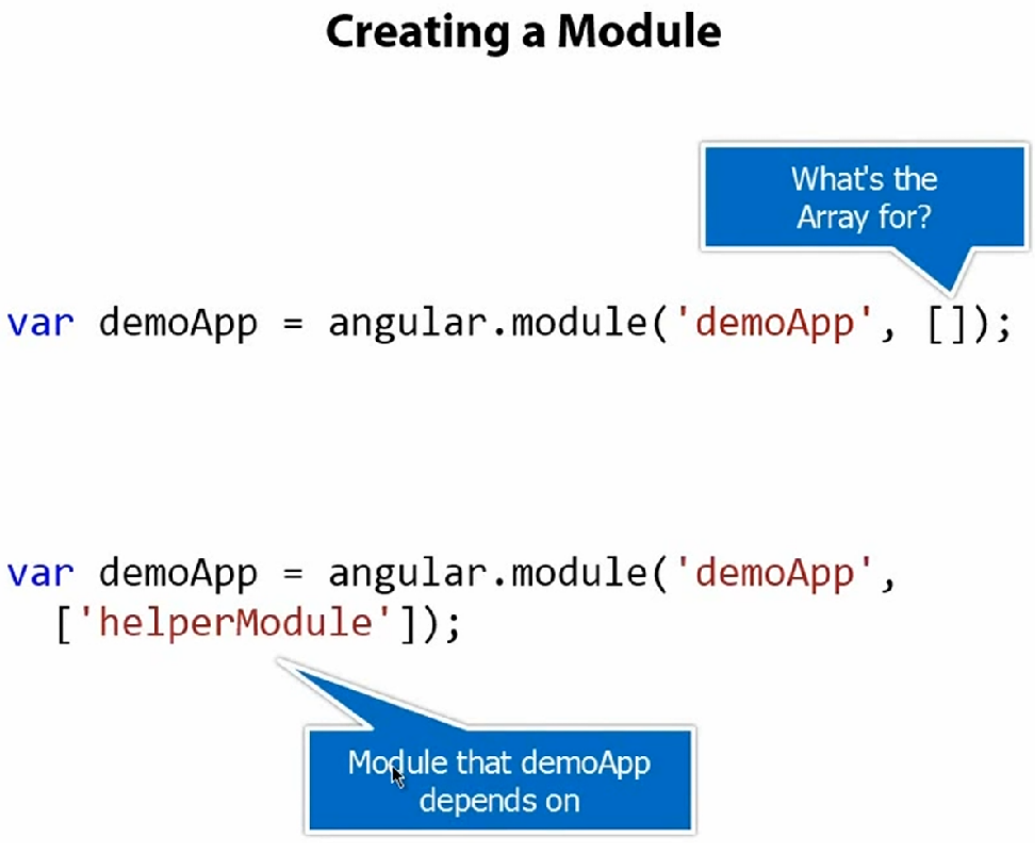


Modules are Containers:

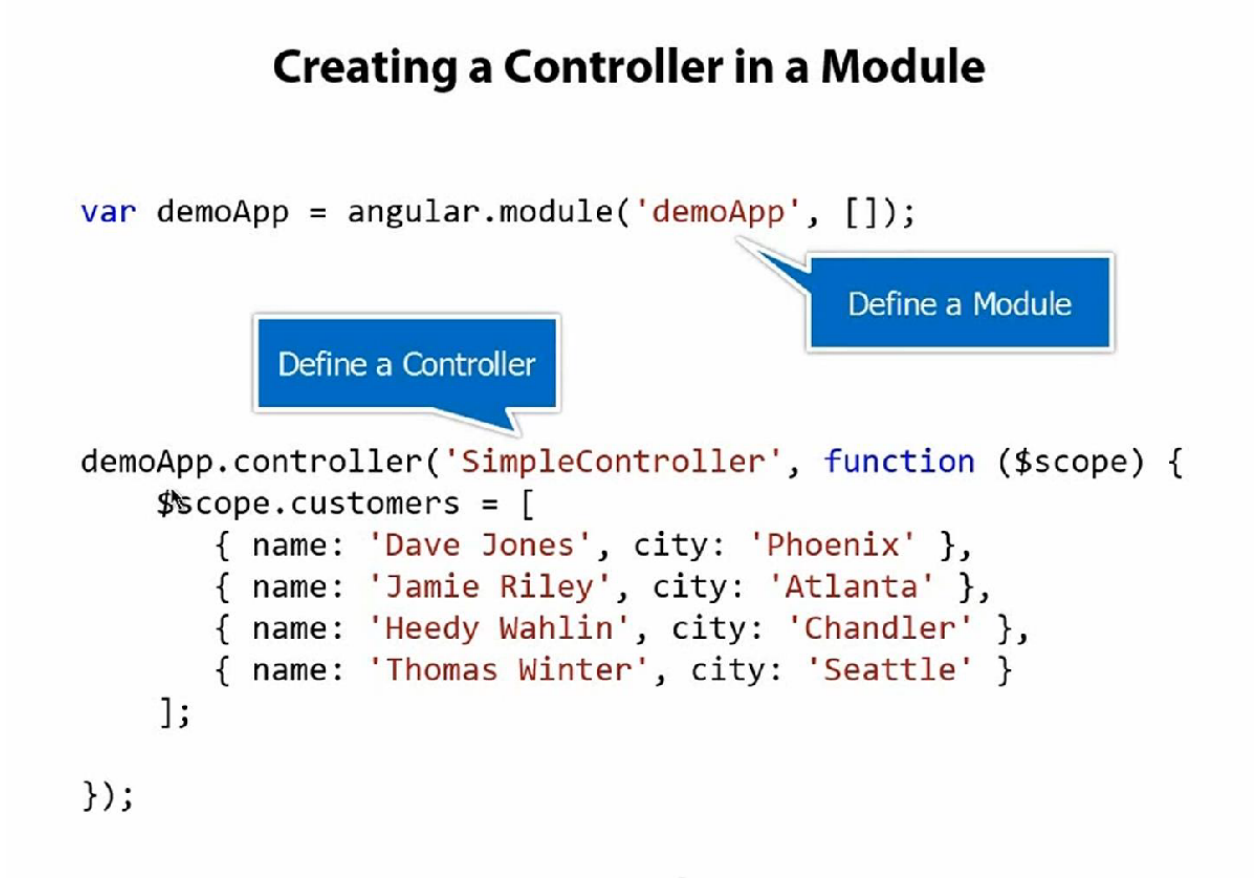
```
var demoApp = angular.module('demoApp', []);
```

You might wonder what exactly is the empty array for?

The answer is this is where **dependency injection** comes in because your module might actually rely on other modules to get data.



Creating a Controller in a Module



We can add controller to the module in different ways:

- Directly using Anonymous function
- Adding a named functions
- Adding multiple controllers

Adding Controller to the Module directly with Anonymous function:

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

demoApp.controller('SimpleController', function ($scope) {
   $scope.message = 'Welcome to AngularJs';

      $scope.persons = [
         { name: 'satya', city: 'hyderabad' },
         { name: 'ram', city: 'mysore' },
         { name: 'ganesh', city: 'bengalore' }
      ];

});
```

Adding a named function:

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

   function SimpleController($scope) {
      $scope.message = 'Welcome to AngularJs';

      $scope.persons = [
         { name: 'satya', city: 'hyderabad' },
         { name: 'ram', city: 'mysore' },
         { name: 'ganesh', city: 'bengalore' }
      ];
   }

   demoApp.controller('SimpleController', SimpleController);
```

Adding multiple controllers:

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

   function SimpleController($scope) {
      $scope.message = 'Welcome to AngularJs';

      $scope.persons = [
         { name: 'satya', city: 'hyderabad' },
         { name: 'ram', city: 'mysore' },
         { name: 'ganesh', city: 'bengalore' }
      ];
   }


   function TestController($scope) {
      $scope.controllerName = 'Test Controller';
      alert("test controller loaded");

   };

   demoApp.controller('SimpleController', SimpleController);
   demoApp.controller('TestController', TestController);
```

OR

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

   function SimpleController($scope) {
      $scope.message = 'Welcome to AngularJs';

      $scope.persons = [
         { name: 'satya', city: 'hyderabad' },
         { name: 'ram', city: 'mysore' },
         { name: 'ganesh', city: 'bengalore' }
      ];
   }


   function TestController($scope) {
      $scope.controllerName = 'Test Controller';
      alert("test controller loaded");

   };
```

```
var controllers = {};

controllers.SimpleController = SimpleController;
controllers.TestController = TestController;

demoApp.controller(controllers);
```

HTML Page:

Example1:

Simple example for Angular Directives:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" data-ng-app="">
<head  >
   <title>Demo</title>
   <script src="Scripts/angular.min.js"></script>
   <script>

   </script>
</head>
<body >
   <div>
      Name: <input  type="text" name="name" data-ng-model="name" /> <br />
     Hello... {{name}}
      <br />

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

Example2:

Global Scope (rootScope) Initialization using ngInit directive.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" data-ng-app="">
<head  >
   <title>Demo</title>
   <script src="Scripts/angular.min.js"></script>
   <script>

   </script>
</head>
<body >
   <div data-ng-init="names=['satya','james','ram']" >
   <ul>
   <li data-ng-repeat="name in names">
      {{name}}
   </li>

   </ul>
   </div>


   <div
data-ng-init="persons=[{name:'satya',city:'mysore'},{name:'ram',city:'Hyderabad'},{name:'ganes
h',city:'chennai'}]">
      <ul>
         <li data-ng-repeat="personName in persons">
           {{personName.name | uppercase}} - {{personName.city | lowercase}}

         </li>
      </ul>
   </div>


</body>
</html>
```

Example with Controller and Module:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" data-ng-app="demoApp">
<head>
  <title></title>
  <script src="Scripts/angular.min.js">
  </script>
  <script>

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

    var controllers = {};
    function SimpleController($scope) {
      $scope.message = 'Welcome to AngularJs';

      $scope.persons = [
        { name: 'satya', city: 'hyderabad' },
        { name: 'ram', city: 'mysore' },
        { name: 'ganesh', city: 'bengalore' }
      ];

    }

    controllers.SimpleController = SimpleController;

    controllers.TestController = function ($scope) {
      $scope.controllerName = 'Test Controller';

    };
    demoApp.controller(controllers);

  </script>

</head>

<body>
  <div data-ng-controller="SimpleController">
  <h3>{{message}}</h3>
    <br />
    Name : <input type="text" name="name" data-ng-model="filters.name" /> <br />
  <div>
    <ul>
      <li data-ng-repeat="personName in persons|filter:filters.name|orderBy:'city'">
        {{personName.name}}- {{personName.city}}
```

```
        </li>
      </ul>
    </div>
      </div>

    <div data-ng-controller="TestController">
      <h2>{{controllerName}}</h2>
    </div>
</body>
</html>
```

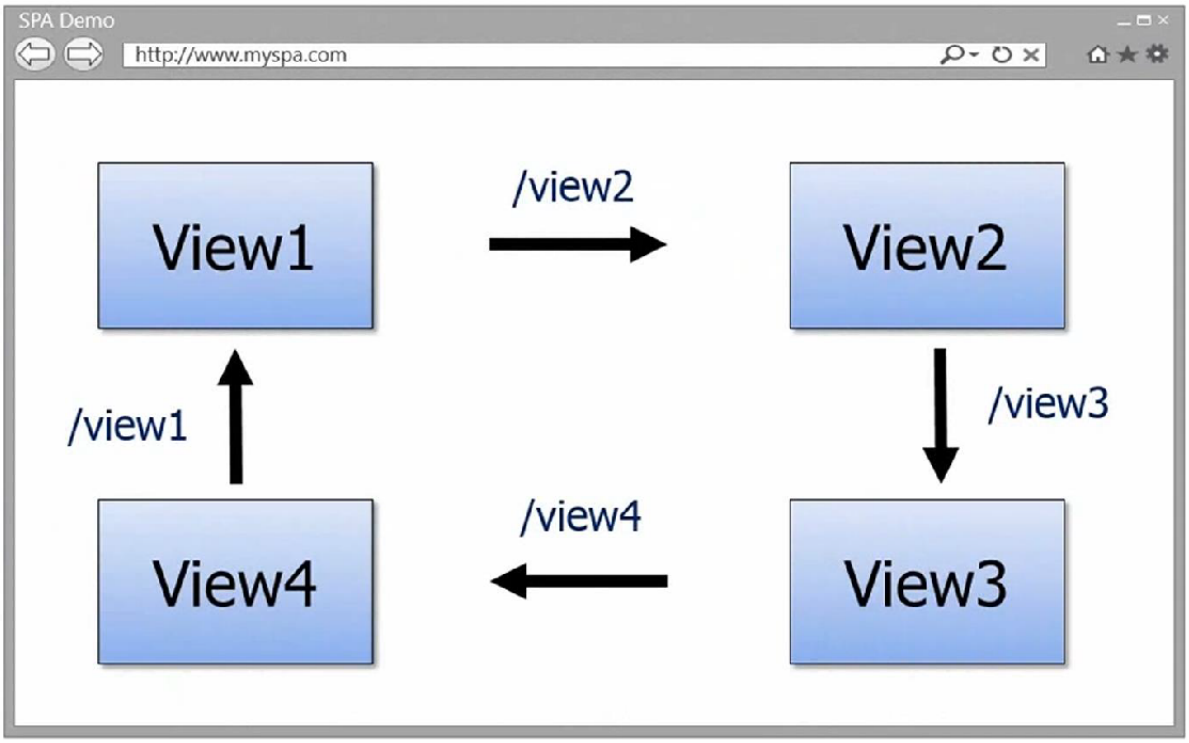
Example 4:

Controller with functions and Modules:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" data-ng-app="demoApp">
<head>
  <title></title>
  <script src="Scripts/angular.min.js">
  </script>
  <script>
    var demoApp = angular.module('demoApp', []);
    var controllers = {};
    function SimpleController($scope) {
      $scope.message = 'Welcome to AngularJs';

      $scope.persons = [
        { name: 'satya', city: 'hyderabad' },
        { name: 'ram', city: 'mysore' },
        { name: 'ganesh', city: 'bengalore' }
      ];


      $scope.addPerson = function () {

        $scope.persons.push({ name: $scope.newPerson.name, city: $scope.newPerson.city });
        alert("added");
      }


    }


    controllers.SimpleController = SimpleController;
    controllers.TestController = function ($scope) {
      $scope.controllerName = 'Test Controller';

    };
    demoApp.controller(controllers);

  </script>

</head>
```

```
<body>
  <div data-ng-controller="SimpleController">
  <h3>{{message}}</h3>
    <br />
    Name : <input type="text" name="name" data-ng-model="filters.name" /> <br />
  <div>
    <ul>
      <li data-ng-repeat="personName in persons|filter:filters.name|orderBy:'city'">
        {{personName.name}}- {{personName.city}}

      </li>
    </ul>
  </div>

    <div>
    <h3>Add New Person</h3>
    Name :  <input type="text" name="newName" data-ng-model="newPerson.name" /> <br
/>
    City: <input type="text" name="newCity" data-ng-model="newPerson.city" /><br />
    <br />
    <button data-ng-click="addPerson()">Add New Person</button>
  </div>
    </div>

  <div data-ng-controller="TestController">
    <h2>{{controllerName}}</h2>
  </div>
</body>
</html>
```

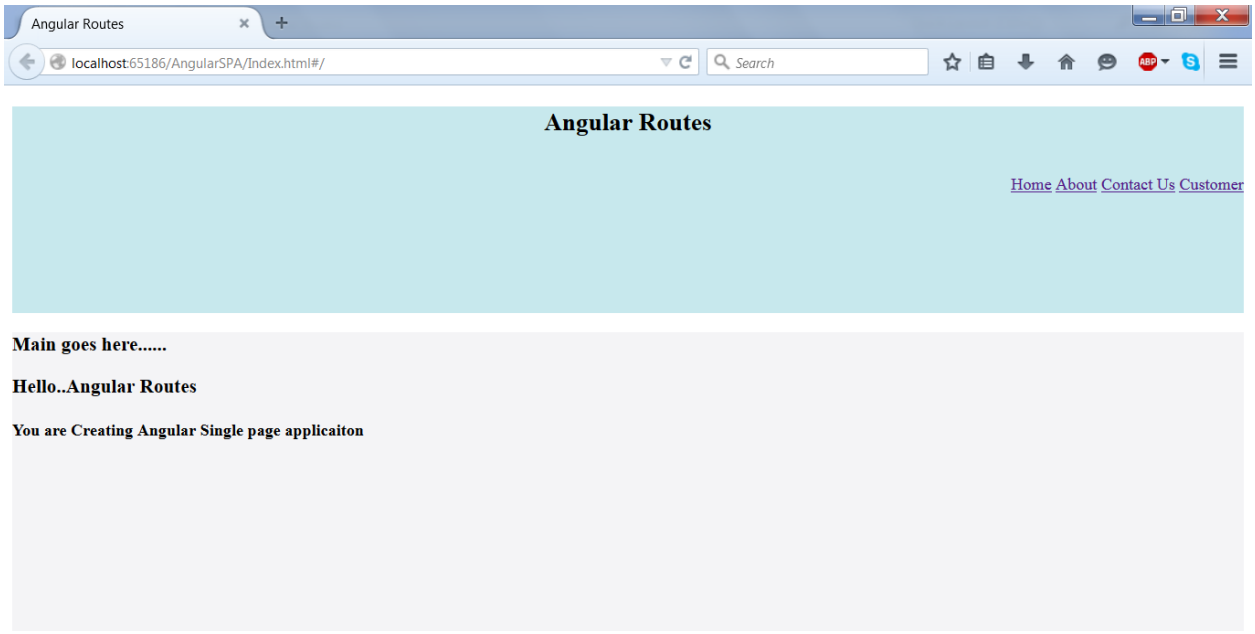
The Role of Routes



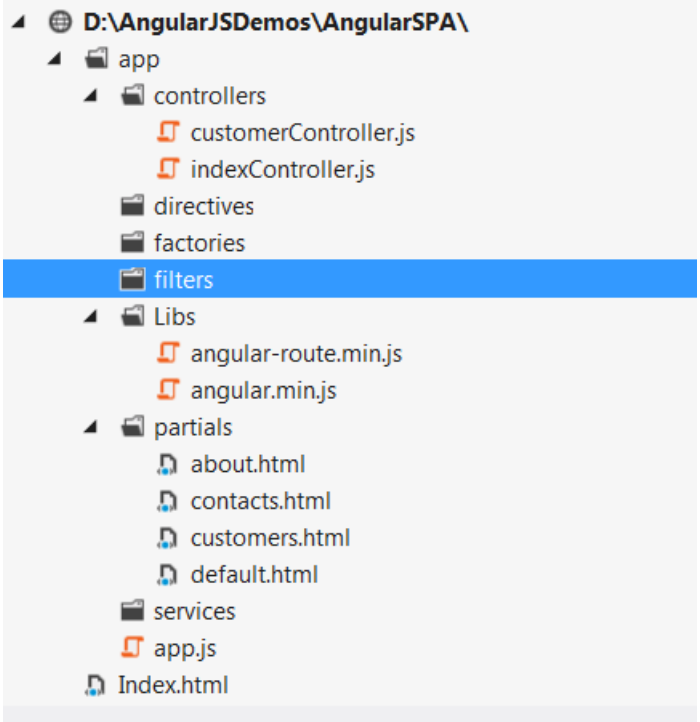
Defining Routes

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

demoApp.config(function ($routeProvider) {
    $routeProvider
        .when('/',
            {
                controller: 'SimpleController',
                templateUrl:'View1.html'
            })
        .when('/partial2',
            {
                controller: 'SimpleController',
                templateUrl:'View2.html'
            })
        .otherwise({ redirectTo: '/' });
});
```

A COMPLETE EXAMPLE WITH ROUTES, CONTROLLERS AND VIEWS



FOLDER STRUCTURE:



```
//app.js

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

myApp.config(function ($routeProvider) {

    $routeProvider
    .when('/',
    {
        controller: 'IndexController',
        templateUrl:'app/partials/default.html'
    })
    .when('/about',
    {
        controller: 'IndexController',
        templateUrl:'app/partials/about.html'
    })
     .when('/contacts',
     {
         controller: 'IndexController',
         templateUrl:'app/partials/contacts.html'
     })
     .when('/customers',
     {
         controller: 'CustomerController',
         templateUrl:'app/partials/customers.html'
     })
          .when('/customers/:customerId',
     {
         controller: 'CustomerController',
         templateUrl: 'app/partials/customer.html'
     })
    .otherwise({rediectTo:'/'});
});
```

//IndexController.js

```
function IndexController($scope) {

}

myApp.controller('IndexController', IndexController);
```

//customerController.js

```
function CustomerController($scope,$routeParams) {
    $scope.customers = [{ CustomerId: 1, CustomerName: 'satya', City: 'Edison', State:
'NJ' }, { CustomerId: 2, CustomerName: 'john', City: 'AT', State: 'PA' }];

    function getCustomerById() {
        var customerId = $routeParams.customerId;
          if (customerId != null) {
            for (var i = 0; i < $scope.customers.length; i++) {
                if (customerId == $scope.customers[i].CustomerId) {

                    $scope.selectedCustomer = $scope.customers[i];
                    break;
                }
            }

        }

    }
    getCustomerById();
}

myApp.controller('CustomerController', CustomerController);
```

INDEX.HTML

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>Angular Routes</title>
    <script src="app/Libs/angular.min.js"></script>
    <script src="app/Libs/angular-route.min.js"></script>

    <script src="app/app.js"></script>
    <script src="app/controllers/indexController.js"></script>
    <script src="app/controllers/customerController.js"></script>

    <style>
        #header {
            height:200px;
            background-color:#c7e8ed;
        }
        #header h1,h2 {
                text-align:center;
        }

        .top-nav {
            float:right;
        }

            .top-nav ul li {
                display:inline;
            }

        #content {
            min-height:300px;
            background-color:#f4f4f6;
        }

        #footer {
            text-align:center;
            font-weight:bold;
            background-color:#fbf57b;
        }
    </style>

</head>
<body>
    <section id="header">
        <h2>Angular Routes</h2>
        <nav class="top-nav">
            <ul>
                <li><a href="#/">Home</a></li>
                <li><a href="#/about">About</a></li>
                <li><a href="#/contacts">Contact Us</a></li>
                <li><a href="#/customers">Customer</a></li>
            </ul>
        </nav>
```

```
    </section>

    <section id="content">
        <h3>Main goes here......</h3>

        <div ng-view>

        </div>

    </section>

    <section id="footer">
        &copy;My Company-2015
    </section>
</body>
</html>
```

//CUSTOMERS.HTML

```
<div>
    <form>
        <h3>Customer Form</h3>
    </form>
    <div idg="grid">
        <table>
            <tr>
                <th>CustomerId</th>
                <th>CustomerName</th>
                <th>City</th>
                <th>State</th>
            </tr>
            <tr ng-repeat="cust in customers">
                <td>{{cust.CustomerId}}</td>
                <td>{{cust.CustomerName}}</td>
                <td>{{cust.City}}</td>
                <td>{{cust.State}}</td>
                <td><a href="#/customers/{{cust.CustomerId}}">View Details</a></td>
            </tr>
        </table>
    </div>
</div>
```

//CUSTOMER.HTML

```
<div>
    <h3>Customer Info</h3>
    Name : <span>{{selectedCustomer.CustomerName}}</span> <br />
    City: <span>{{selectedCustomer.City}}</span><br />
    State : <span>{{selectedCustomer.State}}</span> <br />
</div>
```

//DEFAULT.HTML

```
<div>
    <h3>Hello..Angular Routes</h3>
    <h4>You are Creating Angular Single page applicaiton</h4>
</div>
```

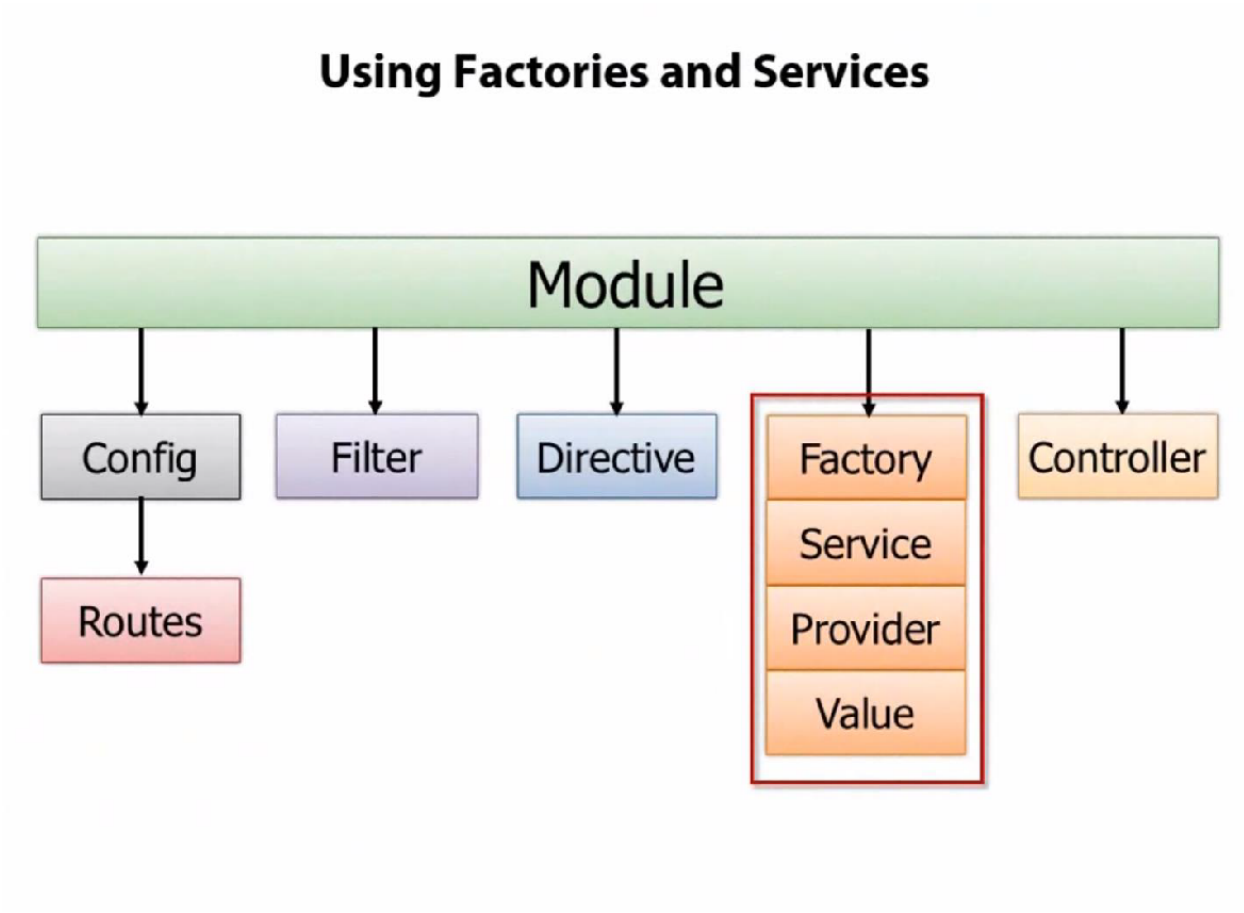
//ABOUT.HTML

```
<div>
    <p>
        We are from MyCompany COE.
    </p>
</div>
```

//CONTACTS.HTML

```
<div>

    <h5>One Corporate Place, South</h5>
    <h6>Piscataway, NJ-08854</h6>
```

```
</div>
```



Another feature of AngularJS is the ability to encapsulate data functionality into factory, services, provider or little value providers.

So for instance if I had to go and get customers and I need those customers in multiple controllers I wouldn't want to hard code that data in each controller.

It just wouldn't make sense and there'd be a lot of duplication there.

Instead what I'll do is I'll move that code out to a factory, service or provider.

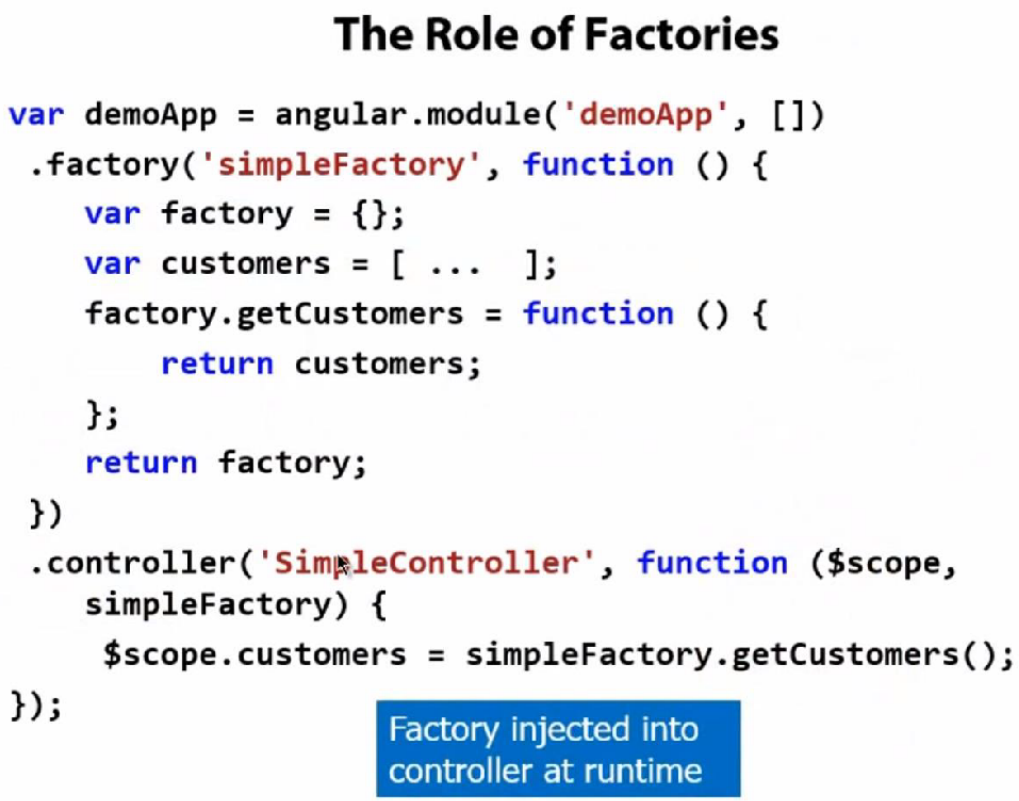
The difference between the three is just the way in which they create the object that goes and gets the data.

- With the **factory** you actually create an **object** inside of the factory and return it.
- With the **service** you just have a standard function that uses the **'this'** keyword to define function.
- With the **provider** there's a **$get** you define and it can be used to get the object that returns the data.
- A **value** is just a way to get for instance a config value.

A simple example of this you'll see on the Angular site is you might just want the version of a particular script.

So you'd have a name-value pair where the name of the value might be "version" and then the value might be say "1.4"

The Role of the Factory



Factory Demo

```
demoApp.factory('simpleFactory', function () {
    var customers = [
        { name: 'John Smith', city: 'Phoenix' },
        { name: 'John Doe', city: 'New York' },
        { name: 'Jane Doe', city: 'San Francisco' }
    ];

    var factory = {};
    factory.getCustomers = function () {
        return customers;
    };
    factory.postCustomer = function (customer) {

    };

    return factory;
});
```

In case of Service, we use as below:

```
this.getCustomers = function () {
    return customers;
};
this.postCustomer = function (customer) {
    this
};
```

Now, you can re-use the factory in the controller:

```
demoApp.controller('SimpleController', function ($scope, simp
    $scope.customers = [];

    init();

    function init() {
        $scope.customers = simpleFactory.getCustomers();
    }
```

More About Angular

AngularJS Services

> Angular services are substitutable objects that are wired together using dependency injection (DI). You can use services to organize and share code across your app.

Angular services are:

- Lazily instantiated – Angular only instantiates a service when an application component depends on it.
- Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.

Angular offers several useful services (like `$http`), but for most applications you'll also want to create your own.

Note: Like other core Angular identifiers, built-in services always start with `$` (e.g. `$http`).

Using a Service

To use an Angular service, you add it as a dependency for the component (controller, service, filter or directive) that depends on the service.

Sample Service:

```
demoApp.service('simpleService', function () {

    var persons = [
        { name: 'satya', city: 'hyderabad' },
        { name: 'ram', city: 'mysore' },
        { name: 'ganesh', city: 'bengalore' }
    ];

    this.getPersons = function () {

        return persons;
    }
```

```
    this.getPerson = function (customer) {

        for (var i = 0; i < persons.length; i++) {
            if (persons[i].name === customer.name) {
                return persons[i];
            }
        }
        return null;
    }



    this.deletePerson = function (customer) {

        for (var i = persons.length - 1; i >= 0; i--) {
            if (persons[i].name === customer.name) {
                persons.splice(i, 1);
                break;
            }
        }
    }

    this.updatePerson = function (customer) {
        for (var i = persons.length - 1; i >= 0; i--) {
            if (persons[i].name === customer.name) {
                persons[i].name = customer.name;
                persons[i].city = customer.city;
                break;
            }
        }
    }
    this.insertPerson = function (customer) {

        persons.push(customer);


    }

});
```

```
//Injecting service in controller

function SimpleController($scope, $routeParams, simpleService) {
    $scope.message = 'Welcome to AngularJs';

    $scope.persons = [];
    function init() {

        $scope.persons = simpleService.getPersons();
    }

    init();
    $scope.addNew = function () {
        $scope.isAddOrUpdate = true;
    }
    $scope.addPerson = function () {

        var newPerson = { name: $scope.selectedPerson.name, city:
$scope.selectedPerson.city }

        simpleService.insertPerson(newPerson);
        $scope.selectedPerson = null;
        $scope.isAddOrUpdate = false;
        alert("added");
    }

    $scope.updatePerson = function () {
        var newPerson = { name: $scope.selectedPerson.name, city:
$scope.selectedPerson.city }

        simpleService.updatePerson(newPerson);

        $scope.isAddOrUpdate = false;
        $scope.isAdded = true;
        $scope.isUpdated = false;
        $scope.selectedPerson = null;
    }

    $scope.editPerson = function (customer) {
        $scope.selectedPerson = simpleService.getPerson(customer);
        $scope.isAddOrUpdate = true;
        $scope.isAdded = false;
        $scope.isUpdated = true;

    };

    $scope.deletePerson = function (customer) {
        simpleService.deletePerson(customer);

    }

    $scope.status = false;
    $scope.isAdded = true;
    $scope.isUpdated = false;
    $scope.isAddOrUpdate = false;
    $scope.selectedPerson = {};
    $scope.getPerson = function (customer) {
```

```
        $scope.selectedPerson = simpleService.getPerson(customer);

        if ($scope.selectedPerson != null) {
            $scope.status = true;
        }

    }

    function getPersonByName() {
        var customerName = $routeParams.name;



        if (customerName != null) {
            $scope.message = 'Customer Details';
            var customer = { name: customerName, city: '' };
            $scope.selectedPerson = simpleService.getPerson(customer);
        }


    }
    getPersonByName();



}
```

View:

```
<div>
    <h2>Customer Controller</h2>
    <h3>
        {{message}}
    </h3>
    <table>
        <tr>
            <th>Id</th>
            <th>Name</th>
            <th>City</th>
            <th>Actions</th>
        </tr>
        <tr data-ng-repeat="cust in data">
            <td>
                {{cust.Id}}
            </td>
            <td>
                {{cust.Name}}
            </td>

            <td>
                {{cust.City}}
            </td>
            <td>
                <a href="#/customers/{{cust.Id}}">Details</a>
            </td>
        </tr>
    </table>
</div>
```

What are Scopes?

scope is an object that refers to the application model.

It is an execution context for expressions.

Scopes are arranged in hierarchical structure which mimic the DOM structure of the application.

Scopes can watch expressions and propagate events.

Scope as Data-Model

Scope is the glue between application controller and the view.

During the template linking phase the directives set up `$watch` expressions on the scope.

The `$watch` allows the directives to be notified of property changes, which allows the directive to render the updated value to the DOM.

Both controllers and directives have reference to the scope, but not to each other.

This arrangement isolates the controller from the directive as well as from the DOM.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title></title>
   <script src="Scripts/angular.js"></script>
</head>
<body>
   <script>
      angular.module('scopeExample', [])
.controller('MyController', ['$scope', function ($scope) {
   $scope.username = 'World';

   $scope.sayHello = function () {
      $scope.greeting = 'Hello ' + $scope.username + '!';
   };
      }]);


   </script>
   <div ng-app="scopeExample" ng-controller="MyController">
  Your name:
   <input type="text" ng-model="username">
   <button ng-click='sayHello()'>greet</button>
  <hr>
  {{greeting}}
</div>

</body>
</html>
```

Dependency Injection

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.

Factory Methods

The way you define a directive, service, or filter is with a factory function.

The factory methods are registered with modules.

The recommended way of declaring factories is:

```
angular.module('myModule', [])
.factory('sampleFactory', ['depService', function(depService) {
  // ...
}])
.directive('directiveName', ['depService', function(depService) {
  // ...
}])
.filter('filterName', ['depService', function(depService) {
  // ...
}]);
```

Module Methods

We can specify functions to run at configuration and run time for a module by calling the `config` and `run` methods.

These functions are injectable with dependencies just like the factory functions.

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

Controllers with dependency with Array Notation

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 of declaring Controllers using the array notation:

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

Angular Expressions

Angular expressions are JavaScript-like code snippets that are usually placed in bindings such as `{{ expression }}`.

For example, these are valid expressions in Angular:

- `1+2`
- `a+b`
- `user.name`
- `items[index]`

Angular Expressions vs. JavaScript Expressions

Angular expressions are like JavaScript expressions with the following differences:

- **Context:** JavaScript expressions are evaluated against the global `window`. In Angular, expressions are evaluated against a `scope` object.
- **Forgiving:** In JavaScript, trying to evaluate undefined properties generates `ReferenceError` or `TypeError`. In Angular, expression evaluation is forgiving to `undefined` and `null`.
- **No Control Flow Statements:** You cannot use the following in an Angular expression: conditionals, loops, or exceptions.
- **No Function Declarations:** You cannot declare functions in an Angular expression, even inside `ng-init` directive.
- **No RegExp Creation With Literal Notation:** You cannot create regular expressions in an Angular expression.
- **No Comma And Void Operators:** You cannot use `,` or `void` in an Angular expression.
- **Filters:** You can use filters within expressions to format data before displaying it.

```
<span>
 1+2={{1+2}}
</span>
```

```
angular.module('expressionExample', [])
.controller('ExampleController', ['$scope', function($scope) {
 var exprs = $scope.exprs = [];
 $scope.expr = '3*10|currency';
 $scope.addExp = function(expr) {
  exprs.push(expr);
 };

 $scope.removeExp = function(index) {
  exprs.splice(index, 1);
 };
}]);
```

```
<div ng-controller="ExampleController" class="expressions">
 Expression:
 <input type='text' ng-model="expr" size="80"/>
 <button ng-click="addExp(expr)">Evaluate</button>
 <ul>
  <li ng-repeat="expr in exprs track by $index">
   [ <a href="" ng-click="removeExp($index)">X</a> ]
   <tt>{{expr}}</tt> => <span ng-bind="$parent.$eval(expr)"></span>
  </li>
 </ul>
</div>
```

Filters in AngularJs

A filter formats the value of an expression for display to the user.

They can be used in view templates, controllers or services and it is easy to define your own filter.

The underlying API is the `filterProvider`.

Using filters in view templates

Filters can be applied to expressions in view templates using the following syntax:

`{{ expression | filter }}`

E.g. the markup `{{ 12 | currency }}` formats the number 12 as a currency using the `currency` filter. The resulting value is `$12.00`.

Filters can be applied to the result of another filter. This is called "chaining" and uses the following syntax:

`{{ expression | filter1 | filter2 | ... }}`

Filters may have arguments. The syntax for this is

`{{ expression | filter:argument1:argument2:... }}`

E.g. the markup `{{ 1234 | number:2 }}` formats the number 1234 with 2 decimal points using the `number` filter. The resulting value is`1,234.00`.

Using filters in controllers, services, and directives

You can also use filters in controllers, services, and directives.

For this, inject a dependency with the name `<filterName>Filter` to your controller/service/directive.

E.g. using the dependency `numberFilter` will inject the number filter.

The example below uses the filter called `filter`.

This filter reduces arrays into sub arrays based on conditions.

The filter can be applied in the view template with markup like `{{ctrl.array | filter:'a'}}`, which would do a fulltext search for "a".

However, using a filter in a view template will reevaluate the filter on every digest, which can be costly if the array is big.

```
angular.module('FilterInControllerModule', []).
controller('FilterController', ['filterFilter', function(filterFilter) {
  this.array = [
    {name: 'Tobias'},
    {name: 'Jeff'},
    {name: 'Brian'},
    {name: 'Igor'},
    {name: 'James'},
    {name: 'Brad'}
  ];
  this.filteredArray = filterFilter(this.array, 'a');
}]);
```

```
<div ng-controller="FilterController as ctrl">
  <div>
    All entries:
    <span ng-repeat="entry in ctrl.array">{{entry.name}} </span>
  </div>
  <div>
    Entries that contain an "a":
    <span ng-repeat="entry in ctrl.filteredArray">{{entry.name}} </span>
  </div>
</div>
```

Creating custom filters

Writing your own filter is very easy: just register a new filter factory function with your module.

Internally, this uses the `filterProvider`.

This factory function should return a new filter function which takes the input value as the first argument.

Any filter arguments are passed in as additional arguments to the filter function.

Note: filter names must be valid angular expression identifiers, such as `uppercase` or `orderBy`. Names with special characters, such as hyphens and dots, are not allowed.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title></title>
   <script src="Scripts/angular.js"></script>
</head>
<body>
   <script>
      angular.module('myReverseFilterApp', [])
  .filter('reverse', function () {
     return function (input, uppercase) {
        input = input || '';

        var out = "";
        for (var i = 0; i < input.length; i++) {
           out = input.charAt(i) + out;

        }

        // conditional based on optional argument
        if (uppercase) {
           out = out.toUpperCase();
        }
        return out;
     };
  })
  .controller('MyController', ['$scope', function ($scope) {
     $scope.greeting = 'hello';
  }]);

   </script>

 <div ng-app="myReverseFilterApp" ng-controller="MyController">
  <input ng-model="greeting" type="text"><br>
  No filter: {{greeting}}<br>
  Reverse: {{greeting|reverse}}<br>
  Reverse + uppercase: {{greeting|reverse:true}}<br>
</div>
</body>
</html>
```

Forms in AngularJs

Controls (`input`, `select`, `textarea`) are ways for a user to enter data.

A Form is a collection of controls for the purpose of grouping related controls together.

Form and controls provide validation services, so that the user can be notified of invalid input before submitting a form.

This provides a better user experience than server-side validation alone because the user gets instant feedback on how to correct the error

Simple form

The key directive in understanding two-way data-binding is ngModel.

The `ngModel` directive provides the two-way data-binding by synchronizing the model to the view, as well as view to the model.

```
<div ng-controller="ExampleController">
  <form novalidate class="simple-form">
    Name: <input type="text" ng-model="user.name" /><br />
    E-mail: <input type="email" ng-model="user.email" /><br />
    Gender: <input type="radio" ng-model="user.gender" value="male" />male
    <input type="radio" ng-model="user.gender" value="female" />female<br />
    <input type="button" ng-click="reset()" value="Reset" />
    <input type="submit" ng-click="update(user)" value="Save" />
  </form>
  <pre>form = {{user | json}}</pre>
  <pre>master = {{master | json}}</pre>
</div>
<script>
  angular.module('formExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.master = {};
      $scope.update = function(user) {
        $scope.master = angular.copy(user);
      };
      $scope.reset = function() {
        $scope.user = angular.copy($scope.master);
      };
      $scope.reset();
    }]);
</script>
```

Using CSS classes for form validation

To allow styling of form as well as controls, `ngModel` adds these CSS classes:

- `ng-valid`: the model is valid
- `ng-invalid`: the model is invalid
- `ng-valid-[key]`: for each valid key added by `$setValidity`
- `ng-invalid-[key]`: for each invalid key added by `$setValidity`
- `ng-pristine`: the control hasn't been interacted with yet
- `ng-dirty`: the control has been interacted with
- `ng-touched`: the control has been blurred
- `ng-untouched`: the control hasn't been blurred
- `ng-pending`: any `$asyncValidators` are unfulfilled

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title></title>
   <script src="Scripts/angular.js"></script>
</head>
<body>
  <div ng-app="formExample" ng-controller="ExampleController">
  <form  class="css-form">
    Name: <input type="text" ng-model="user.name" required /><br />
    E-mail: <input type="email" ng-model="user.email" required /><br />
    Gender: <input type="radio" ng-model="user.gender" value="male" />male
    <input type="radio" ng-model="user.gender" value="female" />female<br />
    <input type="button" ng-click="reset()" value="Reset" />
    <input type="submit" ng-click="update(user)" value="Save" />
  </form>

       <pre>form = {{user | json}}</pre>
  <pre>master = {{master | json}}</pre>

</div>

<style type="text/css">
  .css-form input.ng-invalid.ng-touched {
    background-color: #FA787E;
  }

  .css-form input.ng-valid.ng-touched {
    background-color: #78FA89;
  }
</style>

<script>
   angular.module('formExample', [])
     .controller('ExampleController', ['$scope', function ($scope) {
        $scope.master = {};

        $scope.update = function (user) {
           alert("HI");
           $scope.master = angular.copy(user);
        };

        $scope.reset = function () {
           alert("HI");
           $scope.user = angular.copy($scope.master);
        };

        $scope.reset();
     }]);
</script>

</body>
</html>
```

Binding to form and control state

A form is an instance of FormController.

The form instance can optionally be published into the scope using the `name` attribute.

Similarly, an input control that has the ngModel directive holds an instance of NgModelController.

Such a control instance can be published as a property of the form instance using the `name` attribute on the input control.

The name attribute specifies the name of the property on the form instance.

This implies that the internal state of both the form and the control is available for binding in the view using the standard binding primitives.

This allows us to extend the above example with these features:

- Custom error message displayed after the user interacted with a control (i.e. when `$touched` is set)
- Custom error message displayed upon submitting the form (`$submitted` is set), even if the user didn't interact with a control

```
angular.module('formExample', [])
.controller('ExampleController', ['$scope', function($scope) {
 $scope.master = {};
 $scope.update = function(user) {
   $scope.master = angular.copy(user);
 };
 $scope.reset = function(form) {
   if (form) {
     form.$setPristine();
     form.$setUntouched();
   }
   $scope.user = angular.copy($scope.master);
 };
 $scope.reset();
}]);


<div ng-controller="ExampleController">
 <form name="form" class="css-form" >
   Name:
```

```
  <input type="text" ng-model="user.name" name="uName" required />
  <br />
  <div ng-show="form.$submitted || form.uName.$touched">
    <div ng-show="form.uName.$error.required">Tell us your name.</div>
  </div>
  E-mail:
  <input type="email" ng-model="user.email" name="uEmail" required />
  <br />
  <div ng-show="form.$submitted || form.uEmail.$touched">
    <span ng-show="form.uEmail.$error.required">Tell us your email.</span>
    <span ng-show="form.uEmail.$error.email">This is not a valid email.</span>
  </div>

  Gender:
  <input type="radio" ng-model="user.gender" value="male" />male
  <input type="radio" ng-model="user.gender" value="female" />female
  <br />
  <input type="checkbox" ng-model="user.agree" name="userAgree" required="" />
  I agree:
  <input ng-show="user.agree" type="text" ng-model="user.agreeSign" required="" />
  <br />
  <div ng-show="form.$submitted || form.userAgree.$touched">
    <div ng-show="!user.agree || !user.agreeSign">Please agree and sign.</div>
  </div>
  <input type="button" ng-click="reset(form)" value="Reset" />
  <input type="button" ng-click="update(user)" value="Save" />
 </form>
</div>
```

Custom model update triggers

By default, any change to the content will trigger a model update and form validation.

You can override this behavior using the ngModelOptions directive to bind only to specified list of events. I.e.ng-model-options="{ updateOn: 'blur' }" will update and validate only after the control loses focus. You can set several events using a space delimited list. I.e.

ng-model-options="{ updateOn: 'mousedown blur' }"

If you want to keep the default behavior and just add new events that may trigger the model update and validation, add "default" as one of the specified events.

I.e. ng-model-options="{ updateOn: 'default blur' }"

```
angular.module('customTriggerExample', [])
.controller('ExampleController', ['$scope', function($scope) {
 $scope.user = {};
}]);
```

```
<div ng-controller="ExampleController">
 <form>
  Name:
  <input type="text" ng-model="user.name" ng-model-options="{ updateOn: 'blur' }" /><br />
  Other data:
  <input type="text" ng-model="user.data" /><br />
 </form>
 <pre>username = "{{user.name}}"</pre>
 <pre>userdata = "{{user.data}}"</pre>
</div>
```

Non-immediate (debounced) model updates

You can delay the model update/validation by using the debounce key with the ngModelOptions directive.

This delay will also apply to parsers, validators and model flags like $dirty or $pristine.

I.e. `ng-model-options="{ debounce: 500 }"` will wait for half a second since the last content change before triggering the model update and form validation.

If custom triggers are used, custom debouncing timeouts can be set for each event using an object in `debounce`. This can be useful to force immediate updates on some specific circumstances (like blur events).

I.e.`ng-model-options="{ updateOn: 'default blur', debounce: { default: 500, blur: 0 } }"`

If those attributes are added to an element, they will be applied to all the child elements and controls that inherit from it unless they are overridden.

```
angular.module('debounceExample', [])
.controller('ExampleController', ['$scope', function($scope) {
  $scope.user = {};
}]);
```

```
<div ng-controller="ExampleController">
  <form>
    Name:
    <input type="text" ng-model="user.name" ng-model-options="{ debounce: 250 }" /><br />
  </form>
  <pre>username = "{{user.name}}"</pre>
</div>
```

Creating Custom Directives:

At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's **HTML compiler** (`$compile`) to attach a specified behavior to that DOM element or even transform the DOM element and its children.

Angular comes with a set of these directives built-in, like `ngBind`, `ngModel`, and `ngClass`.

Much like you create controllers and services, you can create your own directives for Angular to use.

Normalization

Angular **normalizes** an element's tag and attribute name to determine which elements match which directives.

We typically refer to directives by their case-sensitive camelCase **normalized** name (e.g. `ngModel`).

However, since HTML is case-insensitive, we refer to directives in the DOM by lower-case forms, typically using dash-delimited attributes on DOM elements (e.g. `ng-model`).

The **normalization** process is as follows:

1. Strip `x-` and `data-` from the front of the element/attributes.
2. Convert the:, `-`, or `_`-delimited name to `camelCase`.

```
<div ng-controller="Controller">
 Hello <input ng-model='name'> <hr/>
 <span ng-bind="name"></span> <br/>
 <span ng:bind="name"></span> <br/>
 <span ng_bind="name"></span> <br/>
 <span data-ng-bind="name"></span> <br/>
 <span x-ng-bind="name"></span> <br/>
</div>
```

Best Practice:

Prefer using the dash-delimited format (e.g.ng-bind for ngBind). If you want to use an HTML validating tool, you can instead use the data-prefixed version (e.g. data-ng-bind for ngBind).

Directive types

$compile can match directives based on element names, attributes, class names, as well as comments.

All of the Angular-provided directives match attribute name, tag name, comments, or class name.

The following demonstrates the various ways a directive (myDir in this case) can be referenced from within a template:

```
<my-dir></my-dir>
<span my-dir="exp"></span>
<!-- directive: my-dir exp -->
<span class="my-dir: exp;"></span>
```

Best Practice: Prefer using directives via tag name and attributes over comment and class names. Doing so generally makes it easier to determine what directives a given element matches.

Text and attribute bindings

During the compilation process the compiler matches text and attributes using the $interpolate service to see if they contain embedded expressions.

These expressions are registered as watches and will update as part of normal digest cycle.

An example of interpolation is shown below:

```
<a ng-href="img/{{username}}.jpg">Hello {{username}}!</a>
```

Creating Directives

Much like controllers, directives are registered on modules.

To register a directive, you use the `module.directive` API.

`module.directive` takes the normalized directive name followed by a **factory function.**

This factory function should return an object with the different options to tell `$compile` how the directive should behave when matched.

The factory function is invoked only once when the compiler matches the directive for the first time.

You can perform any initialization work here.

```
angular.module('docsSimpleDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    template: 'Name: {{customer.name}} Address: {{customer.address}}'
  };
});
```

```
<div ng-controller="Controller">
  <div my-customer></div>
</div>
```

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

```
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    templateUrl: 'my-customer.html'
  };
});
```

My-customer.html

```
Name: {{customer.name}} Address: {{customer.address}}
```

templateUrl can also be a function which returns the URL of an HTML template to be loaded and used for the directive.

Angular will call the templateUrl function with two parameters: the element that the directive was called on, and an attr object associated with that element.

```
angular.module('docsTemplateUrlDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])

.directive('myCustomer', function() {
```

```
  return {
    templateUrl: function(elem, attr){
      return 'customer-'+attr.type+'.html';
    }
  };
});

<div ng-controller="Controller">
  <div my-customer type="name"></div>
  <div my-customer type="address"></div>
</div>

Customer-name.html
Name: {{customer.name}}

Customer-address.html
Address: {{customer.address}}
```

Note: When you create a directive, it is restricted to attribute and elements only by default.

In order to create directives that are triggered by class name, you need to use the `restrict` option.

The `restrict` option is typically set to:

- `'A'` - only matches attribute name
- `'E'` - only matches element name
- `'C'` - only matches class name

These restrictions can all be combined as needed:

- `'AEC'` - matches either attribute or element or class name

Let's change our directive to use `restrict: 'E'`:

```
angular.module('docsRestrictDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    templateUrl: 'my-customer.html'
  };
});
```

```
<div ng-controller="Controller">
  <my-customer></my-customer>
</div>
```

My-customer.html

```
Name: {{customer.name}} Address: {{customer.address}}
```

Creating a Directive that Manipulates the DOM

```
angular.module('docsTimeDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.format = 'M/d/yy h:mm:ss a';
}])
.directive('myCurrentTime', ['$interval', 'dateFilter', function($interval, dateFilter) {

  function link(scope, element, attrs) {
    var format,
        timeoutId;

    function updateTime() {
      element.text(dateFilter(new Date(), format));
    }

    scope.$watch(attrs.myCurrentTime, function(value) {
      format = value;
      updateTime();
    });
```

```
  element.on('$destroy', function() {
    $interval.cancel(timeoutId);
  });

  // start the UI update process; save the timeoutId for canceling
  timeoutId = $interval(function() {
    updateTime(); // update DOM
  }, 1000);
 }

 return {
  link: link
 };
}]);

<div ng-controller="Controller">
 Date format: <input ng-model="format"> <hr/>
 Current time is: <span my-current-time="format"></span>
</div>
```

AngularJS Routing and Views

The magic of Routing is taken care by a service provider that Angular provides out of the box called $routeProvider.

An Angular service is a singleton object created by a service factory.

These service factories are functions which, in turn, are created by a service provider.

The service providers are constructor functions. When instantiated they must contain a property called $get, which holds the service factory function.

When we use AngularJS's dependency injection and inject a service object in our Controller, Angular uses $injector to find corresponding service injector.

Once it get a hold on service injector, it uses $get method of it to get an instance of service object.

Sometime the service provider needs certain info in order to instantiate service object.

Application routes in Angular are declared via the $routeProvider, which is the provider of the $route service.

This service makes it easy to wire together controllers, view templates, and the current URL location in the browser. Using this feature we can implement deep linking, which lets us utilize the browser's history (back and forward navigation) and bookmarks.

Syntax to add Routing

Below is the syntax to add routing and views information to an angular application.

We defined an angular app "sampleApp" using angular.module method.

Once we have our app, we can use config()method to configure $routeProvider.

$routeProvider provides method .when() and .otherwise()which we can use to define the routing for our app.

```
var sampleApp = angular.module('phonecatApp', ['ngRoute']);

sampleApp .config(['$routeProvider',
  function($routeProvider) {
    $routeProvider.
```

```
    when('/addOrder', {
      templateUrl: 'templates/add-order.html',
      controller: 'AddOrderController'
    }).
    when('/showOrders', {
      templateUrl: 'templates/show-orders.html',
      controller: 'ShowOrdersController'
    }).
    otherwise({
      redirectTo: '/addOrder'
    });
  }]);
```

How to pass Parameters in Route Urls

```
when('/ShowOrder/:orderId', {
    templateUrl: 'templates/show_order.html',
    controller: 'ShowOrderController'
});



...
$scope.order_id = $routeParams.orderId;
...
```

How to Load local views (Views within script tag)

It is not always that you want to load view templates from different files.

Sometimes the view templates are small enough that you might want them ship with main html instead of keeping them in separate html files.

ng-template directive

You can use ng-template to define small templates in your html file. For example:

```
<script type="text/ng-template" id="add_order.html">
   <h2> Add Order </h2>
   {{message}}
</script>
```

AngularJS ui-router (nested routes)

To work with Nested routes we need below scripts:

- AngularJs (core)
- Angular-ui-router

You can either download from angular.org or you can use CDN urls.

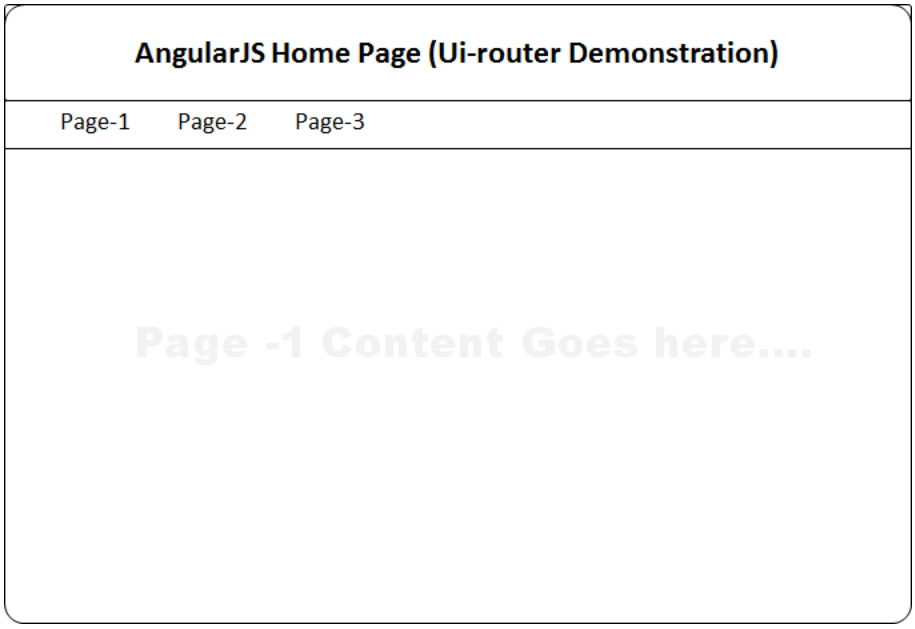
URL: https://code.angularjs.org/ you can download any file from here.

<script src="//ajax.googleapis.com/ajax/libs/angularjs/1.4.0-beta.6/angular.min.js"></script>

<script src="//ajax.googleapis.com/ajax/libs/angularjs/1.4.0-beta.6/angular-route.js"></script>

<script src="//angular-ui.github.io/ui-router/release/angular-ui-router.js"></script>

The Ui-router provides the features where we can do the nested routing and named views vice versa.



As per the above screen design, we will be designing our page, to navigate from one page to another page, when clicking on the **page-1** we will show the content of the page-1 below, same goes to other pages on click.

MainPage.html

```
<!DOCTYPE html>
<html>
<head>
   <title>router demo</title>
 <!-- Angular -->
   <script src="app/Libs/angular.min.js"></script>


 <!-- UI-Router -->
   <script src="app/Libs/angular-ui-router.js"></script>

   <!--app-->
 <script src="app/app.js"></script>


</head>
<body data-ng-app="myApp">
 <h2>AngularJS Ui router - Demonstration</h2>
 <div data-ui-view=""></div>
</body>
</html>
```

app.js

```
var myApp = angular.module("myApp", ['ui.router']);

myApp.config(function ($stateProvider, $urlRouterProvider) {
   $urlRouterProvider.when("", "/PageTab/Page1");
   $stateProvider
      .state("PageTab", {
         url: "/PageTab",
         templateUrl: "PageTab.html"
      })
      .state("PageTab.Page1", {
         url: "/Page1",
         templateUrl: "Page1.html"
      })
      .state("PageTab.Page2", {
         url: "/Page2",
         templateUrl: "Page2.html"
      })
      .state("PageTab.Page3", {
         url: "/Page3",
         templateUrl: "Page3.html"
      });
});
```

PageTab.html

```
<div>
   <div>
      <span style="width:100px" ui-sref=".Page1"><a href="">Page-1</a></span>
      <span style="width:100px" ui-sref=".Page2"><a href="">Page-2</a></span>
      <span style="width:100px" ui-sref=".Page3"><a href="">Page-3</a></span>
   </div>
   <div>
      <div ui-view=""></div>
   </div>
</div>
```

Page1.html

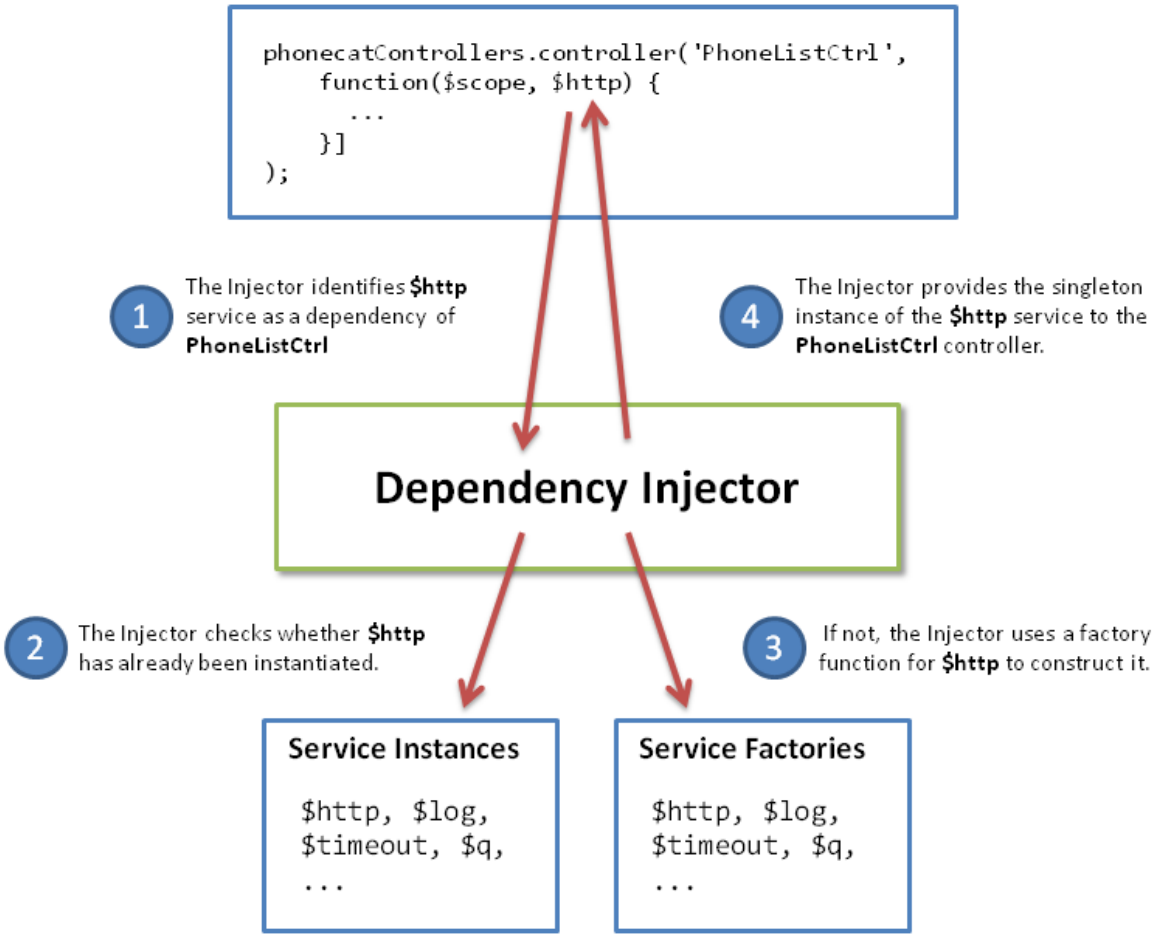
```
<div>
   <div>
      <h1>Page 1 content goes here...</h1>
   </div>
</div>
```

AngularJS and Ajax

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

phonecatApp.controller('PhoneListCtrl', function ($scope, $http) {
  $http.get('phones/phones.json').success(function(data) {
    $scope.phones = data;
  });

  $scope.orderProp = 'age';
});
```

Phones.js

```
[
 {
  "age": 13,
  "id": "motorola-defy-with-motoblur",
  "name": "Motorola DEFY\u2122 with MOTOBLUR\u2122",
  "snippet": "Are you ready for everything life throws your way?"
  ...
 },
...
]
```



You can include dependency injection explicitly:

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

phonecatApp.controller('PhoneListCtrl', ['$scope', '$http',
  function ($scope, $http) {
    $http.get('phones/phones.json').success(function(data) {
      $scope.phones = data;
    });

    $scope.orderProp = 'age';
  }]);
```

Templating Links & Images

```
[
  {
    ...
    "id": "motorola-defy-with-motoblur",
    "imageUrl": "img/phones/motorola-defy-with-motoblur.0.jpg",
    "name": "Motorola DEFY\u2122 with MOTOBLUR\u2122",
    ...
  },
  ...
]
```

```
...
      <ul class="phones">
        <li ng-repeat="phone in phones | filter:query | orderBy:orderProp" class="thumbnail">
          <a href="#/phones/{{phone.id}}" class="thumb"><img ng-src="{{phone.imageUrl}}"></a>
          <a href="#/phones/{{phone.id}}">{{phone.name}}</a>
          <p>{{phone.snippet}}</p>
        </li>
      </ul>
...
```

Custom Filter

```
angular.module('phonecatFilters', []).filter('checkmark', function() {
  return function(input) {
    return input ? '\u2713' : '\u2718';
  };
});
```

```
...
angular.module('phonecatApp', ['ngRoute','phonecatControllers','phonecatFilters']);
...
```

```
    <dl>
      <dt>Infrared</dt>
      <dd>{{phone.connectivity.infrared | checkmark}}</dd>
      <dt>GPS</dt>
      <dd>{{phone.connectivity.gps | checkmark}}</dd>
    </dl>
...
```

Event handling

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

phonecatControllers.controller('PhoneDetailCtrl', ['$scope', '$routeParams', '$http',
  function($scope, $routeParams, $http) {
    $http.get('phones/' + $routeParams.phoneId + '.json').success(function(data) {
      $scope.phone = data;
      $scope.mainImageUrl = data.images[0];
    });

    $scope.setImage = function(imageUrl) {
      $scope.mainImageUrl = imageUrl;
    };
  }]);
```

```
<img ng-src="{{mainImageUrl}}" class="phone">

...

<ul class="phone-thumbs">
  <li ng-repeat="img in phone.images">
    <img ng-src="{{img}}" ng-click="setImage(img)">
  </li>
</ul>
...
```