AngularJS

MV* Framework for creating Single Page Application

AngularJS

(v1.2.20)

Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
25/08/2014	1.0	AngularJS v1.2.20	Karthik Muthukrishnan	

Course Goals and Non Goals

- **Course Goals**
  - Learning the fundamentals of AngularJS.

- **Course Non Goals**
  - Comparison with other MV* frameworks like Backbone.JS, Knockout, EmberJS, Meteor, ExtJS
  - Writing unit tests and Integrated end to end test for Angular components
  - Creating RESTful Services.

Pre-requisites

- **HTML, JavaScript, AJAX Basics & jQuery Basics**

Intended Audience

- **Web application developers**

Day Wise Schedule

- **Day 1**

  Lesson 1: Introduction to AngularJS

- **Day 2**

  Lesson 2: AngularJS Directives

- **Day 3**

  Lesson 3: AngularJS Filters

  Lesson 4 : AngularJS Services

- **Day 4**

  Lesson 4 (Contd.): AngularJS Services

  Lesson 5: AngularJS Routing

Table of Contents

Table of Contents

Table of Contents

Table of Contents

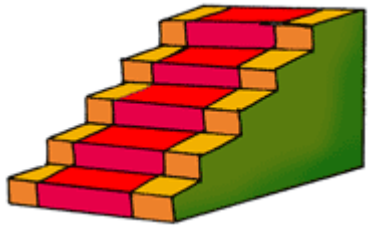
Table of Contents

References

- ng-book The Complete Book on AngularJS by Ari Lerner
- O'REILLY AngularJS by Brad Green & Shyam Seshadri
- APress Pro AngularJS by Adam Freeman
- PACKT Publishing Instant AngularJS Starter by Dan Menard
- PACKT Publishing Dependency Injection with AngularJS by Alex Knol
- PACKT Publishing AngularJS Directives by Alex Vanston
- https://docs.angularjs.org/guide
- https://docs.angularjs.org/api

Next Step Courses (if applicable)

- **Creating Web application using MEAN (MongoDB, ExpressJS, AngularJS, NodeJS) Stack**



Other Parallel Technology Areas

- **Knockout**
- **Backbone.JS**
- **EmberJS**
- **Meteor**
- **ExtJS**

AngularJS

Introduction to AngularJS

Lesson Objectives

- **JavaScript fundamentals**
- **MV* Frameworks**
- **AngularJS Fundamentals**

JavaScript fundamentals

- **Browser gets the HTML text of the page, parses it into DOM structure, lays out the content of the page, and styles the content before it gets displayed.**
- **HTML is great for declaring static documents, but it falters when we try to use it for declaring dynamic views in web-applications.**
- **JavaScript has become one of the most popular client side scripting language on the web which is used to create dynamic views in web-applications.**
- **JavaScript plays a major role in the usage of ajax, user experience and responsive web design.**
- **DOM manipulation libraries like jQuery simplifies client side scripting, but it is not solving the problem of handling separation of concerns.**
- **Fortunately there are few libraries and frameworks are available to accomplish this task.**

Objects in JavaScript

- **JavaScript is an object oriented language. In JavaScript we can define our own objects and assign methods, properties to it.**
- **In JavaScript objects are also associative arrays (or) hashes (key value pairs).**
  - Assign keys with obj[key] = value or obj.name = value
  - Remove keys with delete obj.name
  - Iterate over keys with for(key in obj), iteration order for string keys is always in definition order, for numeric keys it may change.
- **Properties, which are functions, can be called as obj.method(). They can refer to the object as *this*. Properties can be assigned and removed any time.**
- **A function can create new objects when run in constructor mode as new Func(params). Names of such functions are usually capitalized**

Creating objects

- **An empty object can be creating using**
  - obj = new Object(); (or) obj = { };
  - It stores values by key, with that we can assign or delete it using "dot notation" or "Square Brackets" (associative arrays).

using dot notation

```
> var employee = {};
  undefined
> employee.Id = 714709;
  714709
> employee.Name = "Karthik"
  "Karthik"
> employee.Name
  "Karthik"
> delete employee.Name
  true
> employee
  Object {Id: 714709}
```

key: 'Name'
value: 'Karthik'

employee.Name deleted

using square brackets

```
> var employee = {};
  undefined
> employee["Id"] = 714709;
  714709
> employee["Name"] = "Karthik"
  "Karthik"
> employee
  Object {Id: 714709, Name: "Karthik"}
> delete employee["Name"]
  true
> employee
  Object {Id: 714709}
```

Checking for non existing property in object

- **If the property does not exist in the object , then *undefined* is returned**
- **To check whether key existence we can use *in* operator**

```
> var employee = {}
  undefined
> employee.Id          //Checking non existing Property
  undefined
> employee.Id === undefined    // strict comparison
  true
> "Id" in employee     // "in" operator to check for keys existence
  false
> employee.Id = 714709
  714709
> "Id" in employee
  true
```

Iterating over object keys

- **We can iterate over keys using *for .. In***

```
> var employee = {}
  undefined
> employee.Id = 714709
  714709
> employee.Name = "Karthik"
  "Karthik"
> for(key in employee) { console.log("Key : " + key + " Value : " + employee[key]) }
  Key : Id Value : 714709
  Key : Name Value : Karthik
```

Object reference

- **A variable which is assigned to object actually keeps reference to it.**
- **It acts like a pointer which points to the real data. Using reference variable we can change the properties of object.**
- **Variable is actually a reference, not a value when we pass an object to a function.**

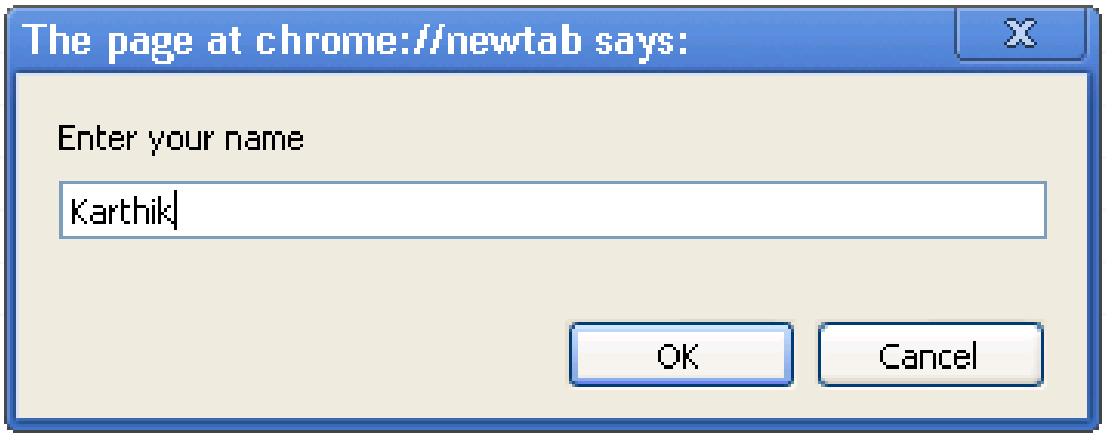
```
> var employee = {};
  undefined
> employee.Id = 714709;
  714709
> var obj = employee;  // now obj points to same object
  undefined
> obj.Id = 707224;
  707224
> employee.Id
  707224
```

this keyword

- **When a function is called from the object, *this* becomes a reference to this object.**

```
> var foo = {
      name : "Guest",
      setName : function(){
     this.name = prompt('Enter your name');  //this acts as a reference to foo object
     },
     getName : function(){
         console.log("Your name is : "+this.name);
     }
};
undefined
> foo.getName();
Your name is : Guest
< undefined
> foo.setName()
undefined
> foo.getName();
Your name is : Karthik
< undefined
```

prompts for name when foo.setName() is called



Constructor Function

- **We can create an object using obj = {…..}**
- **Another way of creating an object in JavaScript is to construct it by calling a function with new directive (Constructor function). Constructor functions should be in Pascal case.**
- **It takes this, which is initially an empty object, and assigns properties to it. The result is returned (unless the function has explicit return).**

Constructor Function

```
> function Calculator(firstVar,secondVar){
      this.firstVar = firstVar;
      this.secondVar = secondVar;
      this.sum = function(){
          return this.firstVar + this.secondVar;
      }
  }
  undefined
> new Calculator(5,5);    // returns this
  ▶ Calculator {firstVar: 5, secondVar: 5, sum: function}
> var calcObj1 = new Calculator(5,5);
  undefined
> var calcObj2 = new Calculator(15,15);
  undefined
> calcObj1.sum();
  10
> calcObj2.sum();
  30
```

Prototypal inheritance

- **In JavaScript, the inheritance is prototype-based. Instead of class inherits from other class, an object inherits from another object.**
- **object inherits from another object using the following syntax.**
- ***childObject.__proto__ = baseObject***
  - Above mentioned syntax provided by Chrome / FireFox. In other browsers the property still exists internally, but it is hidden
- ***childObject = Object.create(baseObject)***
- ***ConstructorFunction.prototype = baseObject***
  - Above mentioned syntax works with all modern browsers.

Prototypal inheritance using __proto__

```
> var foo = {
      fooVar : "Foo Variable",
      fooMethod : function(){
          console.log(this.fooVar);
      }
  }

  var bar = {
      barVar : "Bar Variable"
  }
< undefined
> bar.__proto__ = foo;        // bar object inherits from foo
< ▶ Object {fooVar: "Foo Variable", fooMethod: function}
> bar
< ▶ Object {barVar: "Bar Variable", fooVar: "Foo Variable", fooMethod: function}
```

Prototypal inheritance using Object.create()

```
> var foo = {
      fooVar : "Foo Variable",
      fooMethod : function(){
          console.log(this.fooVar);
      }
  }

< undefined
> var bar = Object.create(foo)     //bar object inherits from foo object
< undefined
> bar
< ▶ Object {fooVar: "Foo Variable", fooMethod: function}
> bar.barVar = "Bar Variable";
< "Bar Variable"
> bar
< ▶ Object {barVar: "Bar Variable", fooVar: "Foo Variable", fooMethod: function}
```

Prototypal inheritance using prototype

```
> function Employee(){
      this.Id = 0;
      this.Name = "";
  }

  function Manager(){ }
//Manager Inherits Employee object
> Manager.prototype = new Employee();
< Employee {Id: 0, Name: ""}
> var anil  = new Manager();
< undefined
> anil
< Manager {Id: 0, Name: ""}     // All objects created by new Manager will have
                                // Id and Name
> anil.Id = 5085;
< 5085
> anil.Name = "Anil Patil";
< "Anil Patil"
> anil
< Manager {Id: 5085, Name: "Anil Patil"}
```

Prototypal inheritance

- ***Object.getPrototypeOf(obj)*** **returns the value of obj.__proto__.**

```
> var foo = {fooVar : "Foo Variable"};
  var bar = Object.create(foo);
< undefined
> Object.getPrototypeOf(bar)
< Object {fooVar: "Foo Variable"}
> Object.getPrototypeOf(bar) === foo
< true
```

- **for..in loop lists properties in the object and its prototype chain.**

  obj.hasOwnProperty(prop) returns true if property belongs to that object.

```
> var foo = {fooVar : "Foo Variable"};
  var bar = {barVar : "Bar Variable"};
  bar.__proto__ = foo;
  for(property in bar){
      if(bar.hasOwnProperty(property))
          console.log("Own Property : "+property);
      else
      console.log("Inherited Property : "+property);
  }
  Own Property : barVar
  Inherited Property : fooVar
```

Static variables and methods

- **In JavaScript we can directly put data into function object which acts like Static member.**
- **Static Members need to be accessed directly by Object name, cannot be accessed by reference variable. Static members gets created when the first object gets created.**

```
> var Employee = function(){
      Employee.CompanyName = "IGATE";
      Employee.doWork = function(){
          console.log('Work Implementation');
      }
  }
< undefined
> Employee.CompanyName
< undefined
> new Employee();
< Employee {}
> Employee.CompanyName
< "IGATE"
> Employee.doWork()
  Work Implementation
```

JavaScript Functions

- **JavaScript treats functions as objects(first-class functions).**
- **In JavaScript functions can be instantiated, returned by other functions, stored as elements of arrays and assigned to variables.**
- **A function with no name is called an anonymous function.**
- **Closure is a function to which the variables of the surrounding context are bound by reference.**
- **JavaScript function acts as a constructor when we use it together with the new operator**

Working with JavaScript Functions

- **Declaring the function anonymously**

```
function(){
        console.log('IGATE');
}
```

- **Invoking the anonymous function. Function executes immediately after declaration.**

```
(function(){
        console.log('IGATE');
})();
```

Working with JavaScript Functions

- **Declaring a named function. function doSomething will be available inside the scope in which it's declared.**

```
function doSomething(){
        console.log('IGATE');
}

/* Inner Scope */
(function(){
        doSomething();
})();
```

- **Assigning function to a variable.**

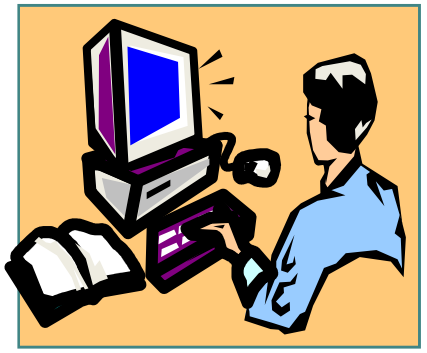
```
var doSomething = function(){
        console.log('IGATE');
}
```

Working with JavaScript Functions

```
/*Anonymous Closures*/
(function(){
        var data = "Closing the variables inside the function from the rest of
the world"
        console.log('Closure Invoked');
})();

var employee = function(){
        this.employeeId = 0;
        this.name = "";
};
/* JavaScript function acts as a constructor */
var emp = new  employee();
```

Demo

- **Working-with-Javascript-Functions**
- **Closure-Demo**



MV* Frameworks

- **MV* Frameworks are designed to make our code easier to maintain and to improve the user experience**
- **MV* framework is nothing but the popular patterns like**
  - Model-View-Controller(MVC)
  - Model-View-ViewModel(MVVM)
  - Model-View-Presenter(MVP)
  - MVW(hatever works for you)
- **Idea of all the patterns is to separate Model, View and the Controller (the logic that hooks up model and view)**
- **AngularJS, Backbone.JS, Knockout, EmberJS, Meteor, ExtJS are some of the famous MV* framework libraries.**

Model, View and Controllers

Model

- Contains the data which we are using in our application

View

- Displays the data to the user and read the user input.

Controller

- Format the data for views and handle application state.

Introduction to AngularJS

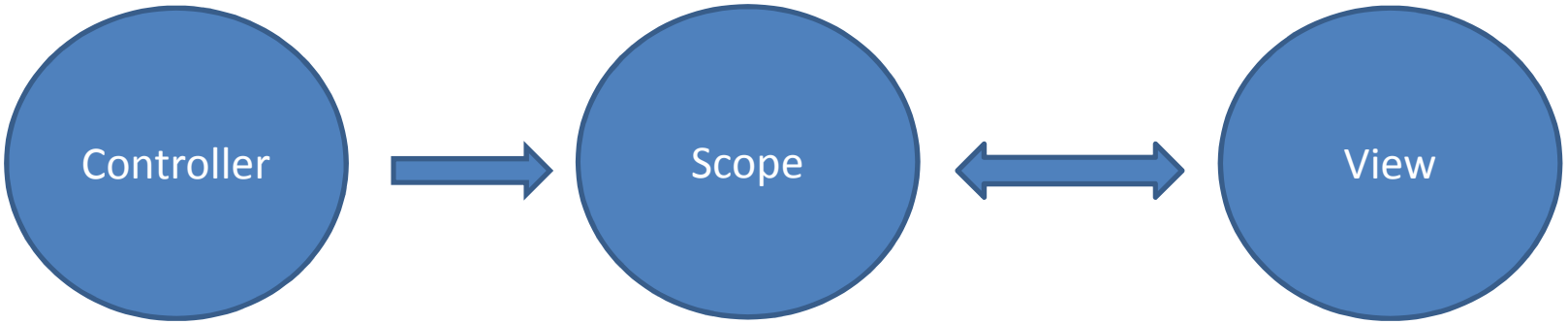
- **AngularJS is an open source JavaScript library that is sponsored and maintained by Google.**
- **Developed in 2009 by Misko Hevery. Publicly released as version 0.9.0 in Oct 2010.**
- **AngularJS makes it easy to build interactive, modern web applications by increasing the level of abstraction between the developer and common web app development tasks by following Model–View–Controller (MVC) pattern.**
- **AngularJS lets you to extend HTML vocabulary for your application. The resulting environment is extraordinarily expressive, readable, and quick to develop.**
- **AngularJS helps us to create single page applications easily.**
  - No page refresh on page change and different data on each page

AngularJS Features

- **Extending HTML to add dynamic nature so that we can build modern web applications with separation of application logic, data models, and views templates**
- **Two way binding**
  - It synchronize the data between model and view, view component gets updated when the model get change and vice versa, no need for events to accomplish this
- **Templates can be created using HTML itself**
- **Testability is the primary consideration in AngularJS. It supports both isolated unit tests and Integrated end to end test**
- **It also supports Routing, Filtering, Ajax calls, data binding, caching, history, and DOM manipulation.**

AngularJS Controller and Scope

- **Controllers primary responsibility is to create scope object ($scope), It also constructs the model on $scope and provides commands for the view to act upon $scope.**
- **Scope communicate with view in two way communication**
- **Scope exposes model to view, but scope is not a model. Model is nothing but the data present in the scope.**
- **View can be binded to the functions on the scope.**
- **We can modify the model using the methods available on the scope.**



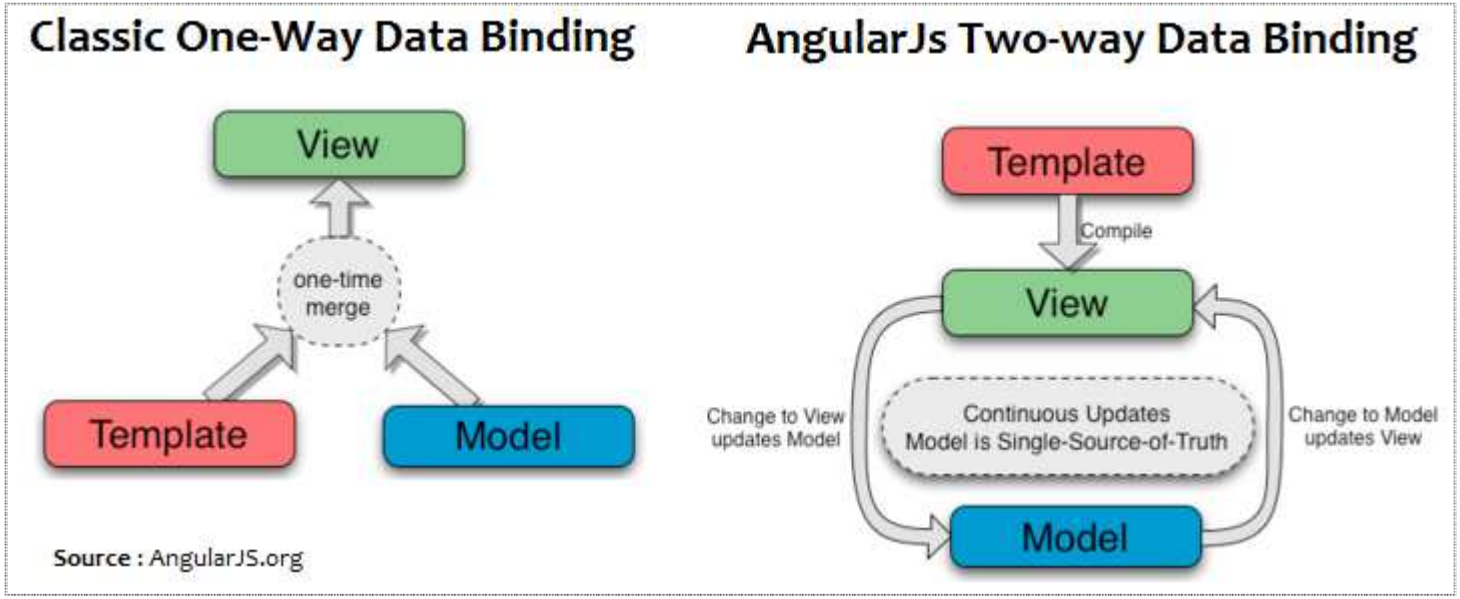
$scope is the glue between Controller and Model

AngularJS Model

- **The model is simply a plain old JavaScript object, does not use getter/setter methods or have any special framework-specific needs.**
- **Changes are immediately reflected in the view via the two-way binding feature.**
- **All model objects stem from scope object.**
- **Typically model objects are initialized in controller code with syntax like:**
  - *$scope.companyName = "IGATE";*
- **In the HTML template, that model variable would be referenced in curly braces such as: {{companyName}} without the $scope prefix.**

AngularJS View and Templates

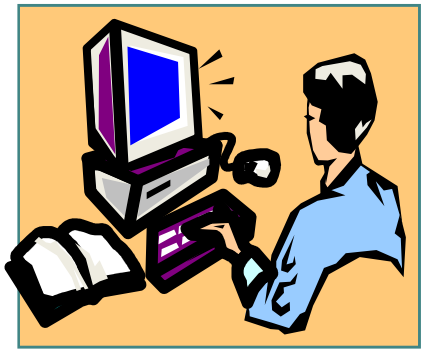
- **The View in AngularJS is the compiled DOM.**
- **View is the product of $compile merging the HTML template with $scope.**
- **In Angular, templates are written with HTML that contains Angular-specific elements and attributes. Angular combines the template with information from the model and controller to render the dynamic view that a user sees in the browser**



IGATE
Speed. Agility. Imagination

Demo

- **AngularJs-MVC**



AngularJS Modules

- **A module is the overall container used to group AngularJS code. It consists of compiled services, directives, views controllers, etc.**
- **Module is like a main method that instantiates and wires together the different parts of the application.**
- **Modules declaratively specify how an application should be bootstrapped.**
- **The Angular module API allows us to declare a module using the angular.module() API method.**
- **When declaring a module, we need to pass two parameters to the method. The first is the name of the module we are creating. The second is the list of dependencies, otherwise known as injectables.**
  - angular.module('myApp', []); // setter method for defining Angular Module.
  - angular.module('myApp'); // getter method for referencing Angular Module.

AngularJs Expressions

- **Expressions {{expression}} are JavaScript like code snippets.**
- **In Angular, expressions are evaluated against a scope object.**
- **AngularJS let us to execute expressions directly within our HTML pages.**
- **Expressions are generally placed inside a binding and typically it has variable names set in the scope object.**
- **Expression can also hold computational codes like {{3 * 3}}, but we cannot directly use JavaScript syntax like {{Math.random()}}, conditionals, loops or exceptions inside it.**

```
<div>{{3 * 3}}</div> returns 9

<div>{{'Karthik'+' '+'Muthukrishnan'}}</div> returns  Karthik Muthukrishnan

<div>{{['Ganesh','Abishek','Karthik','Anil'][2]}}</div> returns Karthik
```

$rootScope

- **When Angular starts to run and generate the view, it will create a binding from the root ng-app element to the $rootScope.**
- **$rootScope is the eventual parent of all $scope objects and it is set when the module initializes via run method.**
- **The $rootScope object is the closest object we have to the global context in an Angular app. It's a bad idea to attach too much logic to this global context.**

```
<div ng-app="myApp">     <h1>{{companyName}}</h1>   </div>
<script>
        var app= angular.module("myApp",[]);
        app.run(function($rootScope){
                        $rootScope.companyName = "IGATE";
                        $rootScope.printCompanyName = function() {
                                        console.log($rootScope.companyName);
                        }
        });
</script>
```

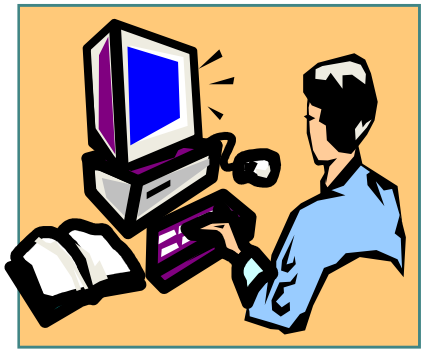
Steps for Coding Hello World in AngularJs

- **Step 1: Declare the module**
- **Step 2: Declare the controller and set the properties (or) function to the scope.**
- **Step 3: Bootstrap angularjs using *ng-app* and define the controller, so that the properties which we have set in the controller can be consumed in the view(HTML).**

```
<html ng-app="sampleApp">    Step - 3
<head>
<script type="text/javascript" src="angular.js"></script>
<script type="text/javascript">
    angular.module('sampleApp',[])  Step - 1
    .controller('SampleController',function($scope){
    $scope.greet = "Hello World";
                                    Step - 2
});
</script>
<head>
<body>
<div ng-controller="SampleController">  Step - 3
    <h2>{{greet}}</h2>
</div>
</body>
</html>
```

Demo

- **AngularJs-Modules**



Dependency Injection

- **Dependency injection is a design pattern that allows for the removal of hard-coded dependencies, thus making it possible to remove or change them at run time.**

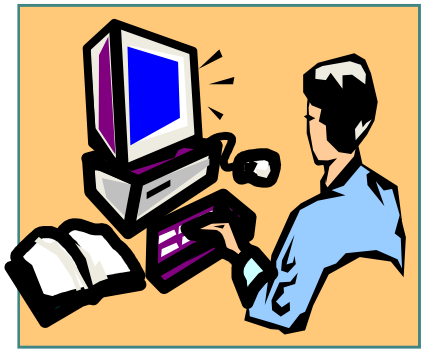
```
function Foo(object) {
   this.object = object;
}
Foo.prototype.showDetails = function(data) {
  this.object.display(data);
}

var greeter = {
        display : function(msg){
          alert(msg);
        }
}

var foo = new Foo(greeter);
foo.showDetails("IGATE");
```

Dependency Injection

- **At runtime, the Foo doesn't care how it gets the dependency, so long as it gets it. In order to get that dependency instance into Foo, the creator of Foo is responsible for passing in the Foo dependencies when it's created.**
- **This ability to modify dependencies at run time allows us to create isolated environments that are ideal for testing. We can replace real objects in production environments with mocked ones for testing environments.**
- **In AngularJS at run time, an injector will create instances of the dependencies and pass them along to the dependent consumer.**

Demo

- **Dependency Demo**



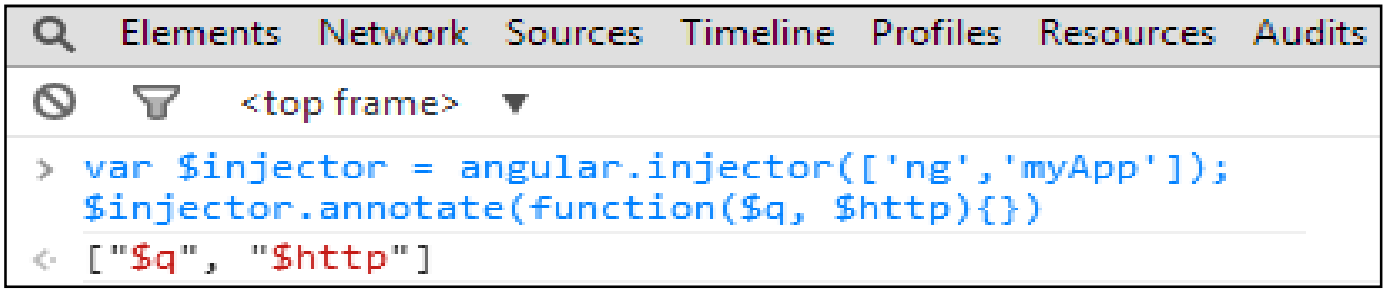
AngularJS Services

- **Services provide a method for us to keep data around for the lifetime of the app and communicate across controllers in a consistent manner.**
- **Services are singleton objects that are instantiated only once per app (by the $injector) and lazyloaded (created only when necessary).**
- **We need to put our business logic in Services.**

We will discuss in detail about Services, later in this course.

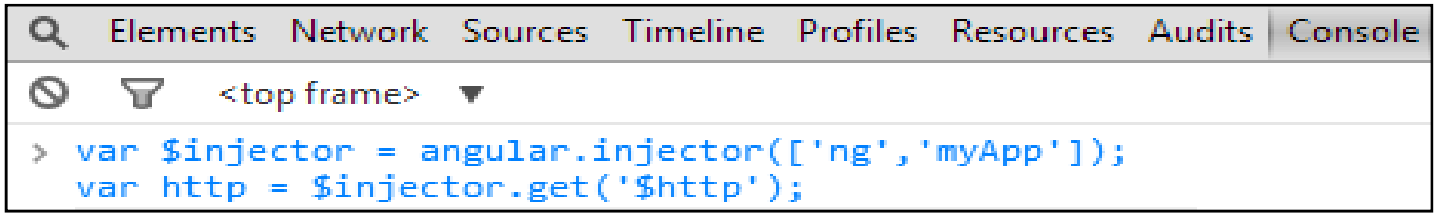
injector Service

- **Angular uses the injector for managing lookups and instantiation of dependencies. We will very rarely work directly with injector service**
- **injector is responsible for handling all instantiations of our Angular components, including app modules, directives, controllers, etc.**
- **injector is responsible for instantiating the instance of the object and passing in any of its required dependencies. Injector API has following methods.**
- **annotate()**
  - The annotate() function returns an array of service names that are to be injected into the function when instantiated.

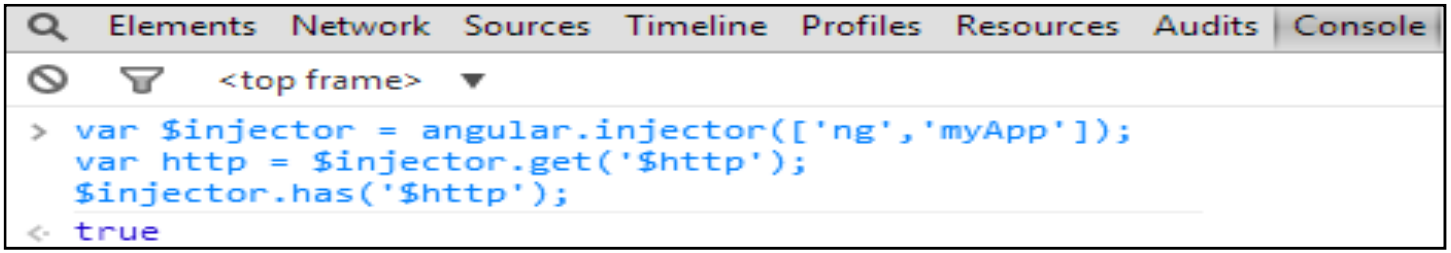


injector Service

- **get()**
  - The get() method returns an instance of the service which takes the name argument. (the name of the instance we want to get).

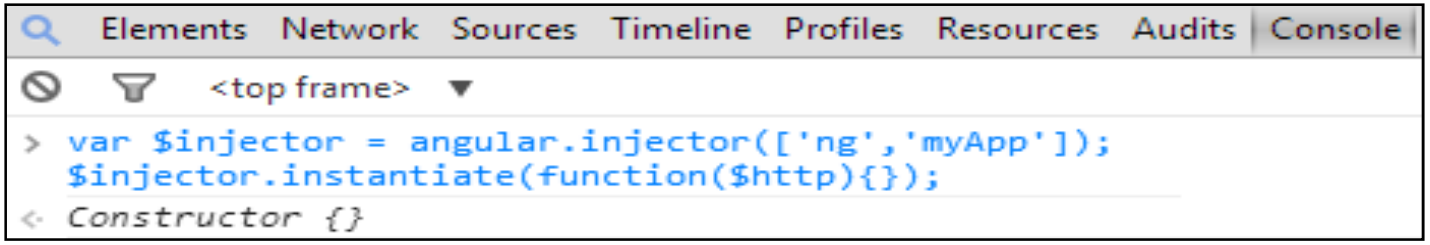


- **has()**
  - The has() method returns true if the injector knows that a service exists in its registry and false if it does not.

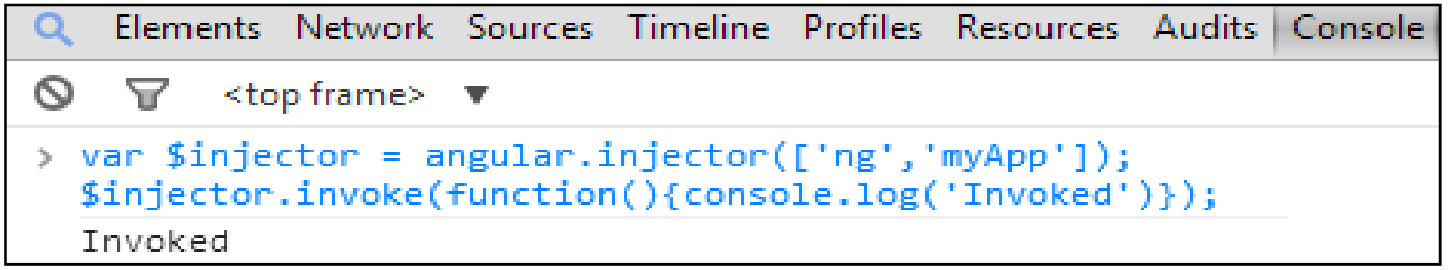


injector Service

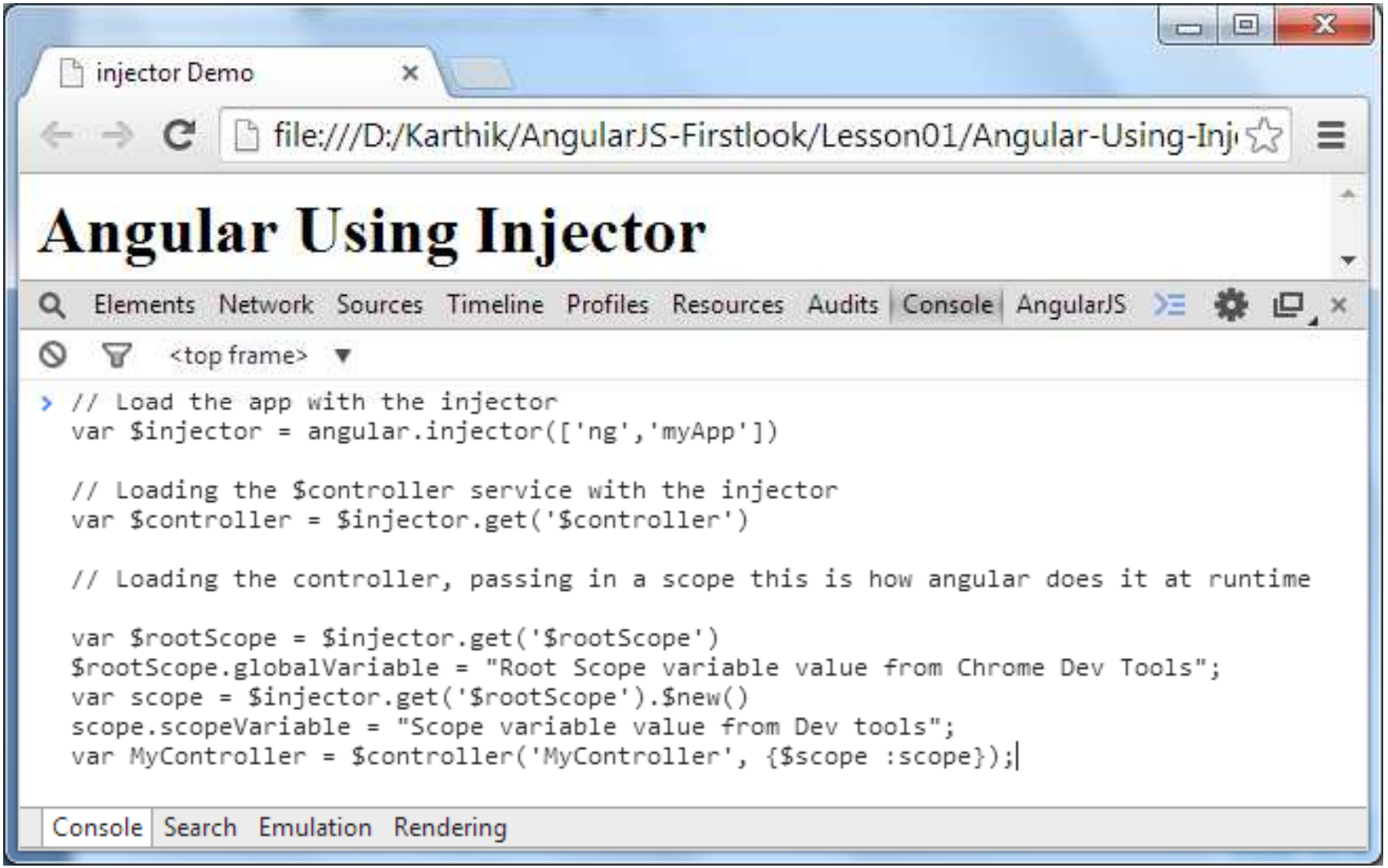
- **instantiate()**
  - The instantiate() method creates a new instance of the JavaScript type. It takes a constructor and invokes the new operator with all of the arguments specified.



- **invoke()**
  - The invoke() method invokes the method and adds the method arguments from the $injector. The invoke() method returns the value that the fn function returns

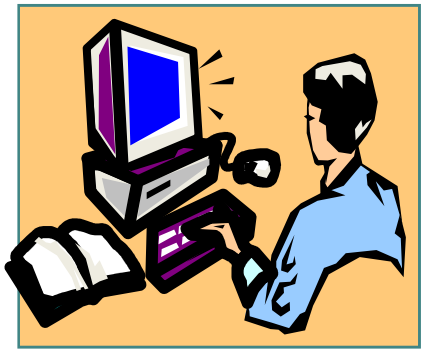


How Angular uses injector Service



Demo

- **Angular-Using-Injector**
- **Injector-Demo**

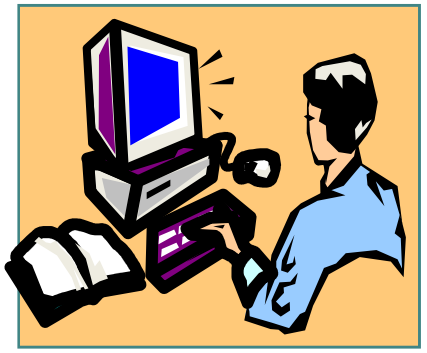


Config and Run Method

- **angular.Module type has config() and run() method**
- **config(configFn)**
  - We can use this method to register the work which needs to be performed on module loading.
  - It will be very useful for configuring the service.
- **run(initializationFn)**
  - We can use this method to register the work which needs to be performed when the injector is done loading all modules.

Demo

- **Config-Demo**

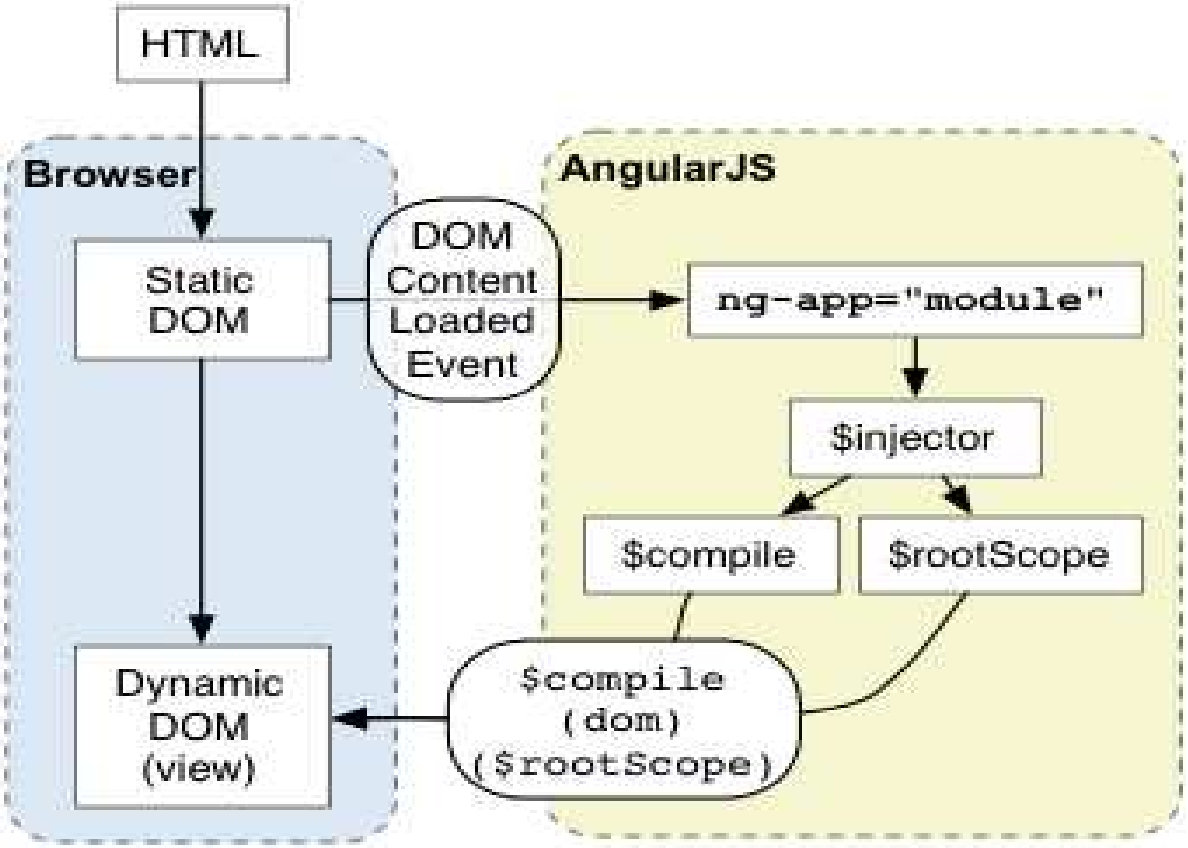


jqLite

- **Angular.js comes with a simple compatible implementation of jQuery called jqLite**
- **Angular doesn't depend on jQuery. In order to keep Angular small, Angular implements only a subset of the selectors in jqLite, so error will occurs when a jqLite instance is invoked with a selector other than this subset.**
- **We can include a full version of jQuery, which Angular will automatically use. So that all the selectors will be available.**
- **If jQuery is available, *angular.element* is an alias for the jQuery function. If jQuery is not available, *angular.element* delegates to Angular's built-in subset of jQuery, called "jQuery lite" or "jqLite."**
- **All element references in Angular are always wrapped with jQuery or jqLite; they are never raw DOM references.**

How AngularJs Works

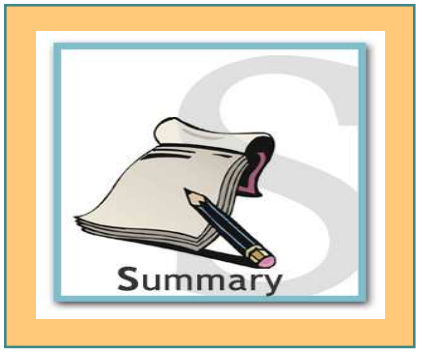
- **$compile compiles DOM into a template function that can be used to link scope and the view together.**



Source : Angularjs.org

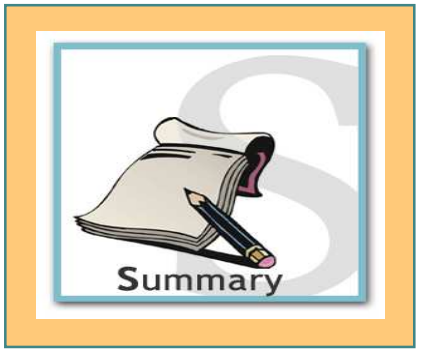
Summary

- JavaScript objects are also associative arrays
- In JavaScript, the inheritance is prototype-based.
- JavaScript treats functions as objects(first-class functions).
- JavaScript function acts as a constructor when we use it together with the new operator.
- Angular thinks of HTML as if it had been designed to build applications instead of documents.
- Angular supports unit tests and end to end tests.
- Controller is the central component in an angular application.



Summary

- Creating the scope is the primary responsibility of the controller.
- $scope is the glue between Controller and Model
- View can bind to the properties as well as functions on the scope.
- Expressions only supports a subset of JavaScript. We can create an array in expression.
- Using Dependency Injection we can replace real objects in production environments with mocked ones for testing environments



AngularJS

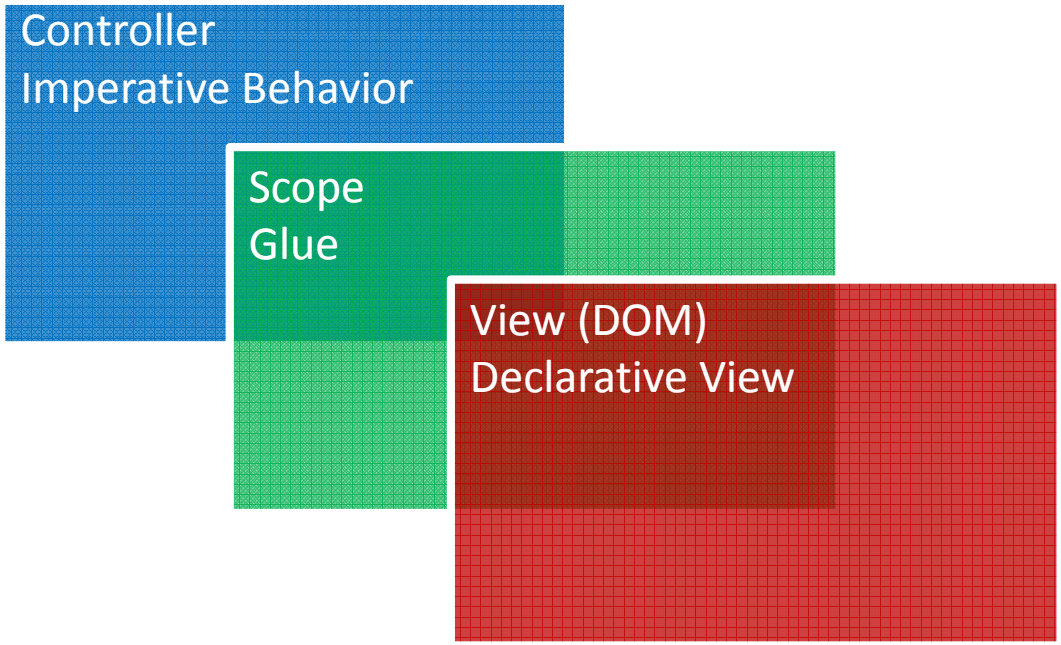
AngularJS Directives

Lesson Objectives

- **Directives**
- **Built-In Directives**
- **Creating Custom Directives**

Controllers

- **Controller is used to provide the business logic behind the view and construct and value the model.**
- **The goal is to not manipulate the DOM at all in the controller, which we have done using other frameworks.**



Controllers – Best practices

- **Controllers should not know anything about the view they control.**
- **Controllers should be small and focused.**
- **Controllers should not talk to other controllers**
- **Controllers should not own the domain model.**

Directives

- **Directives are ways to transform the DOM through extending HTML and provides new functionality to it.**
- **As a good practice DOM manipulations need to be done in directives.**
- **Directive is simply a function that we run on a particular 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.**
- **Directives are actually defined with camelCase in the JavaScript, but applied with a dash to the HTML.**
- **Angular comes with a set of built-in directives along with that we can create our own directives.**

Directives

- **Angular directive can be specified in 3 ways**
- **As a tag**
  - <ng-form />
- **As an attribute**
  - <div ng-form />
- **As a class**
  -
- **All the in-built directives in angular cannot be specified with all the 3 ways, some of them can be specified in 1 or 2 ways only.**

Built-In Directives

- **Angular provides a suite of built-in directives.**
- **ngApp**
  - Placing ng-app on any DOM element marks that element as the beginning of the $rootScope.
  - $rootScope is the beginning of the scope chain, and all directives nested under the ng-app in your HTML inherit from it.
  - $rootScope can be accessed via the run method
  - Using $rootScope is like using global scope hence it is not a best practice.
  - We can use ng-app once per document.
- **ngController**
  - This directive is used to place a controller on a DOM element.
  - Instead of defining actions and models on $rootScope, use ng-controller

Built-In Directives

- **ngBind**
  - The ngBind attribute tells Angular to replace the text content of the specified HTML element with the value of a given expression and to update the text content when the value of that expression changes.
  - It is preferable to use ngBind instead of {{ expression }}. if a template is momentarily displayed by the browser in its raw state before Angular compiles it. Since ngBind is an element attribute, it makes the bindings invisible to the user while the page is loading.
- **ngBindTemplate**
  - The ngBindTemplate directive specifies that the element text content should be replaced with the interpolation of the template in the ngBindTemplate attribute. Unlike ngBind, the ngBindTemplate can contain multiple {{ }} expressions..

Built-In Directives

- **ngBindHtml**
  - Creates a binding that will innerHTML the result of evaluating the expression into the current element in a secure way.
  - **ngSanitize (angular-sanitize.js)** need to be included as module's dependencies to evaluate in a secure way, else it will throw error "Attempting to use an unsafe value in a safe context".
  - We can bypass sanitization by binding to an explicitly trusted value via **$sce.trustAsHtml**. Same can be done using **ng-bind-html-unsafe** which has been removed in Angular 1.2
- **ngShow**
  - The ngShow directive shows or hides the given HTML element based on the expression provided to the ngShow attribute.

… Continued

Built-In Directives

- When the ngShow expression evaluates to a falsy value then the ng-hide CSS class is added to the class attribute on the element causing it to become hidden. When truthy, the ng-hide CSS class is removed from the element causing the element not to appear hidden.

- **ngHide**
  - The ngHide directive shows or hides the given HTML element based on the expression provided to the ngHide attribute.
  - When the ngHide expression evaluates to a truthy value then the .ng-hide CSS class is added to the class attribute on the element causing it to become hidden. When falsy, the ng-hide CSS class is removed from the element causing the element not to appear hidden.

Built-In Directives

- When the ngShow expression evaluates to a falsy value then the ng-hide CSS class is added to the class attribute on the element causing it to become hidden. When truthy, the ng-hide CSS class is removed from the element causing the element not to appear hidden.

- **ngCloak**
  - It's an alternative to **ngBind** directive. It is used to prevent the Angular html template from being briefly displayed by the browser in its raw (uncompiled) form while your application is loading.
  - Use this directive to avoid the undesirable flicker effect caused by the html template display. The directive can be applied to the <body> element, but the preferred usage is to apply multiple ngCloak directives to small portions of the page to permit progressive rendering of the browser view.

…Continued

Built-In Directives

- We need to add the following styles in our HTML to make ngCloak to work
- *[ng\:cloak], [ng-cloak], [data-ng-cloak], [x-ng-cloak], .ng-cloak, .x-ng-cloak {*

  display: none !important;}
- In Legacy browsers, like IE7 we need to add the css class *ng-cloak* in addition to the ngCloak directive

- **ngStyle**
  - ngStyle directive allows you to set CSS style on an HTML element.
  - CSS style names and values must be quoted.
- **ngClass**
  - ngClass directive allows you to dynamically set CSS classes on an HTML element by databinding an expression that represents all classes to be added.

Built-In Directives

- **ngRepeat**
  - ngRepeat directive instantiates a template once per item from a collection. Each template instance gets its own scope, where the given loop variable is set to the current collection item, and $index is set to the item index or key
- **ngClassEven**
  - Works exactly as ngClass, except they work in conjunction with ngRepeat and take effect only on even row elements i.e. 0,2,4,6 …
- **ngClassOdd**
  - Works exactly as ngClass, except they work in conjunction with ngRepeat and take effect only on odd row elements i.e. 1,3,5,7…

Built-In Directives

- **ngSrc**
  - Angular will tell the browser not to fetch the image via the given URL until all expressions provided to ng-src have been interpolated.
  - It is recommended to use in place of src.
- **ngHref**
  - Angular waits for the interpolation to take place and then activates the link's behavior.
  - It is recommended to use in place of href.
- **ngDisabled, ngChecked, ngReadonly & ngSelected**
  - Those directives works with HTML boolean attributes.

Built-In Directives

- **ngNonBindable**
  - ngNonBindable directive tells Angular not to compile or bind the contents of the current DOM element.
  - It will be useful to display Angular Code snippets
- **ngIf**
  - ng-if directive is used to completely remove or recreate an element in the DOM based on an expression. If the expression assigned to ng-if evaluates to a false value, then the element is removed from the DOM, otherwise a clone of the element is reinserted into the DOM.
  - Using ng-if when an element is removed from the DOM, its associated scope is destroyed. when it comes back into being, a new scope is created

Built-In Directives

- **ngModel**
  - ngModel directive provides the two-way data-binding by synchronizing the model to the view, as well as view to the model.
  - ngModel will try to bind to the property given by evaluating the expression on the current scope. If the property doesn't already exist on this scope, it will be created implicitly and added to the scope.
  - ngModel is responsible for:
    - Binding the view into the model, which other directives such as input, textarea or select.
    - Providing validation behavior (i.e. required, number, email, url).
    - Keeping the state of the control (valid/invalid, dirty/pristine, touched/untouched, validation errors).
    - Setting related css classes on the element (ng-valid, ng-invalid, ng-dirty, ng-pristine, ng-touched, ng-untouched) including animations.
    - Registering the control with its parent form.

Built-In Directives

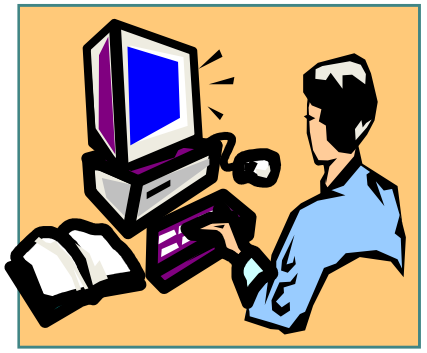
- **ngSwitch**
  - The ngSwitch directive is used to conditionally swap DOM structure on your template based on a scope expression
  - It is used in conjunction with ng-switch-when and on="propertyName" to switch which directives render in our view when the given propertyName changes.
- **ngInit**
  - The ngInit directive is used to set up the state inside the scope of a directive when that directive is invoked.
  - It is a shortcut for using ng-bind without needing to create an element; therefore, it is most commonly used with inline text

Built-In Directives

- **ngInclude**
  - ngInclude directive is used to fetch, compile and include an external HTML fragment into your current application.
  - By default, the template URL is restricted to the same domain and protocol as the application document. This is done by calling *$sce.getTrustedResourceUrl* on it
  - The URL of the template is restricted to the same domain and protocol as the application document unless white listed or wrapped as trusted values.
  - To access file locally in chrome use *chrome.exe -allow-file-access-from-files -disable-web-security*

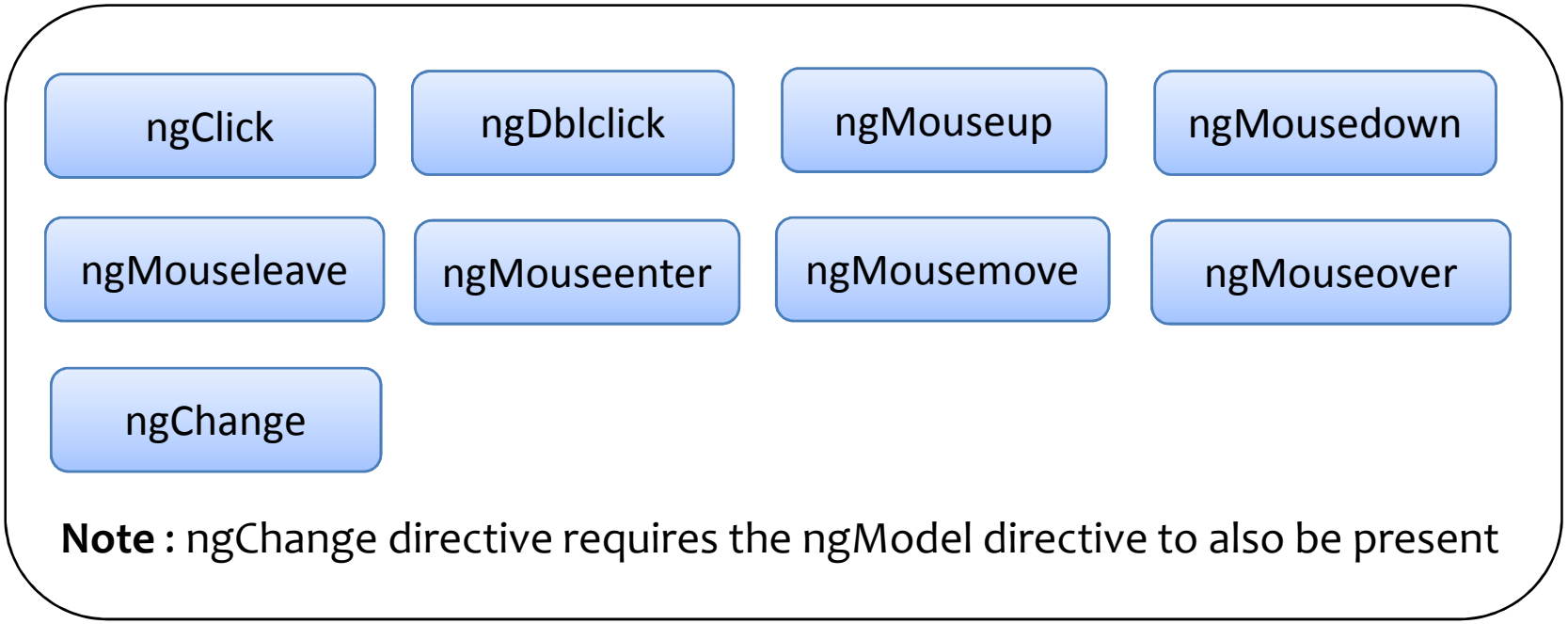
Demo

- **Built-InDirectives**



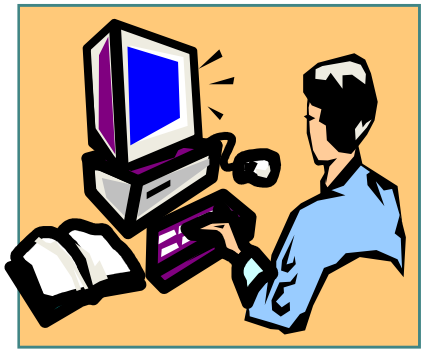
Built-In Event directives

- **When Angular parses the HTML, it look for directive and takes action based on that, when it looks for event directives it register the event on the DOM object.**



Demo

- **EventDirectives**

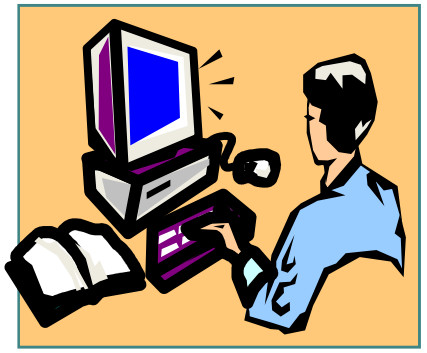


Form Validation

- **ngForm directive can be used, when we need to nest a form within another form, which is not allowed in normal HTML.**
- **The outer form is valid when all of the child forms are valid. It will be very useful when dynamically generating forms using ngRepeat directive. Angular will not submit the form to the server unless the form has an action attribute specified.**
- **The following CSS classes are set automatically, depending on the validity of the form:**
  - ng-valid when form is valid (is set if the form is valid)
  - ng-invalid when form is invalid
  - ng-pristine when form is pristine (the field has not been modified by user)
  - ng-dirty when form is dirty (the field has been modified by user)

Demo

- **FormValidation**

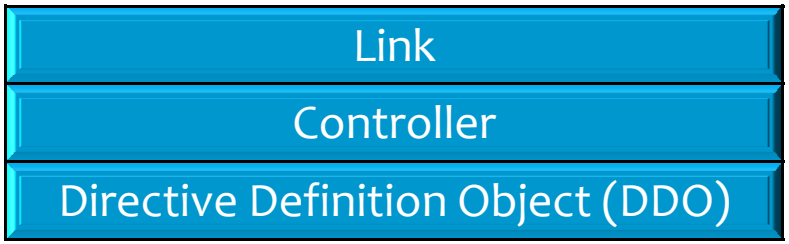


Custom Directives

- **Directives makes the Angular framework so powerful, we can also create our own directives. A directive is defined using the .directive() method on application's Angular module.**
- **Directives can be implemented in the following ways:**
  - Element directives : activated when AngularJS finds a matching HTML element in the HTML template
  - Attribute directives : activated when AngularJS finds a matching HTML element attribute
  - CSS class directives : activated when AngularJS finds a matching CSS Class
  - Comment directives : activated when AngularJS finds a matching HTML comment
- **AngularJS recommends to use element and attribute directives, and leave the CSS class and comment directives (unless absolutely necessary).**

Custom Directives

- **Directives consists of three (or less) important things**



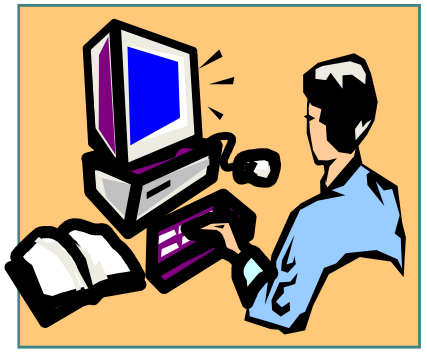
- **Link function is where the DOM manipulation occurs.**
- **Controller is constructed during the pre-linking phase and receives the $scope for the element.**
- **Directive Definition Object (DDO) tells the compiler how a Directive needs to be assembled. Common properties include the link function, controller function, restrict, template and templateUrl**

Applying restrictions to directives

- **Custom directive is restricted to attribute by default.**
- **In order to create directives that are triggered by element, class name & comment we 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
  - 'M' - only matches comment
- **These restrictions can also be combined**
  - 'AEC' - matches either attribute or element or class name

Demo

- **Directive01**



Custom Directives - template

- **template is an inline template specified using html as a string / function which gets appended / replaced (by setting replace:true) within the element where the directive was invoked.**
- **template has a scope that can be accessed using double curly markup, like {{ expression }}. Backslashes is used at the end of the each line to denote multi line string**
- **When a template string must be wrapped in a parent element. i.e, a root DOM element must exist.**

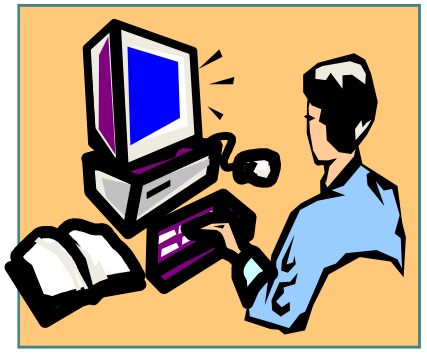
```
app.directive("helloworldattr",function(){
        return {
                replace:true,
                template: "<h1>Hello World Attribute from Directive</h1>"
         }
});
```

Custom Directives - templateUrl

- **templateUrl comes in handy when our template gets too big to inline.**
- **We can specify the path of an HTML file / a function which returns the path of an HTML file.**
- **By default, the HTML file will be requested on demand via Ajax when the directive is invoked.**
  - When developing locally, we should run a server in the background to serve up the local HTML templates from our file system. Failing to do so will raise a Cross Origin Request Script (CORS) error.
  - Start the chrome by typing **chrome.exe -allow-file-access-from-files -disable-web-security** in run(Win + R Key) to avoid CORS error, Mozilla Firefox wont give any errors. I.E doesn't support CORS
  - We can load templates directly into the cache in a script tag, or by consuming the $templateCache service directly.

Demo

- **Directive09**
- **Directive10**



Custom Directives – compile() & link function()

- **The compile() and link() functions define how the directive is to modify the HTML that matched the directive.**
- **When the directive is first compiled by AngularJS (first found in the HTML), the compile() function is called. The compile() function can then do any one-time configuration of the element needed.**
- **The compile() function finishes by returning the link() function. The link() function is called every time the element is to be bound to data in the $scope object.**
- **compile() function has to return the link() function when executed.**
- **We can even set only a link() function also for the custom directives.**

Custom Directives – compile() & link function()

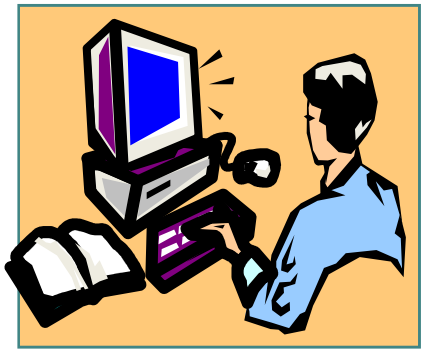
```
<div ng-controller="MyController" >
   <userinfo >This will be replaced</userinfo>
</div>

<script>
   var myapp = angular.module("myapp", []);
   myapp.directive('userinfo', function() {
      var directive = {};
      directive.restrict = 'E'; /* restrict this directive to elements */
      directive.compile = function(element, attributes) {
         element.css("border", "1px solid #cccccc");
         var linkFunction = function($scope, element, attributes) {
            element.html("This is the new content: " + $scope.firstName);
            element.css("background-color", "#ffff00");
         }
         return linkFunction;
      }
      return directive;
   });
   myapp.controller("MyController", function($scope, $http) {
                  $scope.firstName = "Karthik";
   });
</script>
```

Custom Directives – compile() & link function()

- **The template produced by a directive is meaningless unless it's compiled against the right scope. By default a directive does not get a new child scope. Rather, it gets the parent's scope. This means that if the directive is present inside a controller it will use that controller's scope. To utilize the scope, we can make use of a function called link.**
- **link takes a function with the following signature, function link(scope, element, attrs) { ... } where:**
  - scope is an Angular scope object.
  - element is the jqLite-wrapped element that this directive matches.
  - attrs is a hash object with key-value pairs of normalized attribute names and values.
- **The link function is mainly used for attaching event listeners to DOM elements, watching model properties for changes, and updating the DOM**

Demo

- **Directive02**

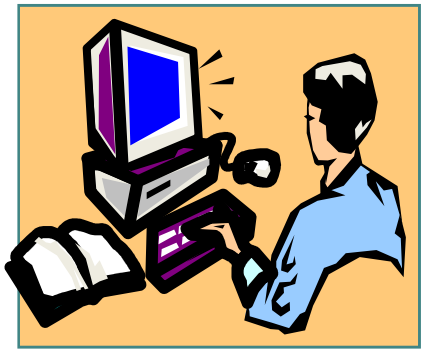


Custom Directives - controller Function and require

- **The controller function of a directive is used to establish the communication between two directives.**
- **controller function is used to create a UI component by combining two directives.**
- **require is used to inject the controller of the required directive as the fourth parameter of the current directive's linking function.**
- **require uses '^' prefix when the directive looks for the controller on its parents otherwise it looks for the controller on its own element.**

Demo

- **Directive04**



Custom Directives – Directive's Scope

- **By default a directive gets the parent's scope, so that they are free to modify the parent's controller scope properties.**
- **If directives need to add properties and functions for internal use there is no need to add it to the parent's scope.**
- **The scope can be configured with the scope property of the directive definition object**
  - A child scope – This scope prototypically inherits the parent's scope. (scope is set to true)
  - An isolated scope – A new scope that does not inherit from the parent and exists on its own. These bindings are specified by the attribute defined in HTML and the definition of the scope property in the directive definition object.

Custom Directives – Directive's Scope

```
<div ng-init="myProperty = 'Controller's Parent Scope'"></div>
 Surrounding scope: {{ myProperty }}
 <div my-inherit-scope-directive="SomeCtrl">
  Inside an directive with inherited scope: {{ myProperty }}
 </div>
 <div my-directive>
  Inside myDirective, isolate scope: {{ myProperty }}
 <div>
</div>
<script>
  angular.module('myApp', []).directive('myDirective', function() {
   return {
            scope: {}     //Isolated Scope
     };
  }).directive('myInheritScopeDirective', function() {
   return {
            scope: true   // child Scope
     };
  })
</script>
```

Custom Directives – Isolated Scope using '@ = &'

- **There are 3 types of binding options which are defined as prefixes in the scope property. The prefix is followed by the attribute name of HTML element.**
  - One Way Text Binding (Prefix: @)
  - Two-way Binding (Prefix: =)
  - Execute Functions in the Parent Scope(Prefix: &)
- **Text bindings are prefixed with @, and they are always strings. Whatever we write as attribute value, it will be parsed and returned as strings. If the parent scope changes, the isolated scope will reflect that change, but not the other way around.**
- **Two-way bindings are prefixed by = and can be of any type. Whenever the parent scope property changes, the corresponding isolated scope property also changes, and vice versa**

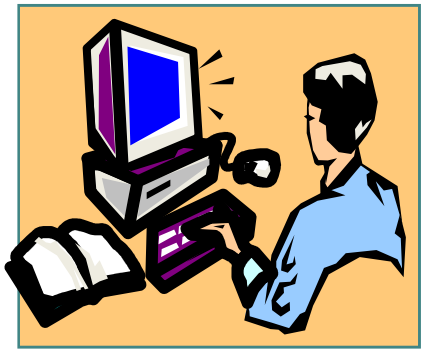
Custom Directives – Isolated Scope using '@ = &'

- **We can call a function on the parent scope from isolated scope using '&'**

```
<div ng-controller="MyCntrl">
  <div my-directive company="{{company}}"></div>
</div>

<script>
var app = angular.module("myApp",[]);
app.directive("myDirective", function(){
   return {
                restrict:"A",
                 scope: {
                          company:"@"
                 },
                 template: '{{company}}'
    }
});
app.controller("MyCntrl",function($scope){
          $scope.company = "IGATE";
});
</script>
```

Demo

- **Directive05**
- **Directive06**
- **Directive07**

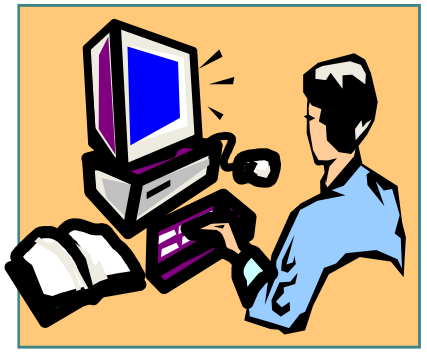


Custom Directives – Transclusion

- **Transclusion is a feature which lets us to wrap a directive around arbitrary content. We can later extract and compile it against the correct scope, and finally place it at the specified position in the directive template.**
- **Transclude allows us to pass in an entire template, including its scope, to a directive. Doing so gives us the opportunity to pass in arbitrary content and arbitrary scope to a directive. If the scope option is not set, then the scope available inside the directive will be applied to the template passed in.**
- **Transclusion is most often used for creating reusable widgets.**
- **ngTransclude directive marks the insertion point for the transcluded DOM of the nearest parent directive that uses transclusion.**
  - Any existing content of the element that this directive is placed on will be removed before the transcluded content is inserted.

Demo

- **Directive08**



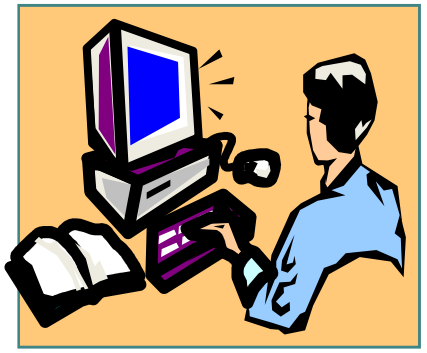
Working with jQuery UI

- **We can wrap the jQuery UI Datepicker into an Angular.js easily with the help of a custom directive**

```
<div>
        <input data-igate-date-picker="true" type="text"/>
</div>
<script>
var app = angular.module("myApp",[]);
app.directive('igateDatePicker',function(){
        return function(scope,element,attrs){
                element.datepicker({
                        changeMonth: true,
                        changeYear: true
                });
        }
});
</script>
```

Demo

- **jQueryUI-DatePicker**



Digest Cycle and $scope

- **Digest cycle can be considered as a loop, during which Angular checks if there are any changes occured to the variables watched being watched.**
- **Angular sets up a watcher on the scope model, which in turn updates the view whenever the model changes.**

```
$scope.$watch('modelVariable', function(newValue, oldValue) {
        //update the DOM with newValue
});
```

- **$digest cycle fires the watchers. When the $digest cycle starts, it fires each of the watchers. These watchers checks the current value of the scope model is different from old value. If there is a change, then the corresponding listener function executes.As a result any expressions in the view gets updated.**

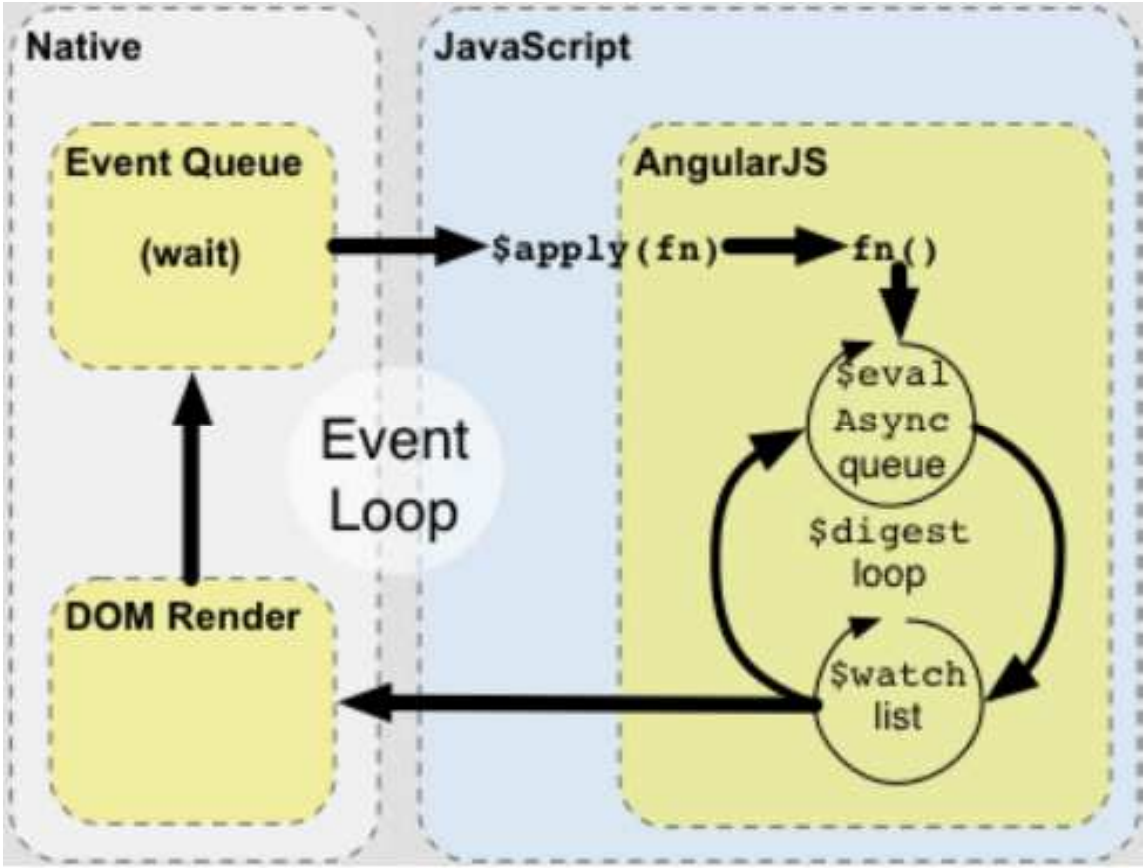
Digest Cycle and $scope

- **$digest cycle starts as a result of a call to $scope.$digest(). Angular doesn't directly call $digest(). Instead, it calls $scope.$apply(), which in turn calls $rootScope.$digest(). As a result of this, a digest cycle starts at the $rootScope, and subsequently visits all the child scopes calling the watchers along the way.**
- **AngularJS wraps the function calls (which updates the model) from view within $scope.$apply()**
- **The $apply() function comes in two flavors.**
  - The first one takes a function as an argument, evaluates it, and triggers a $digest cycle.
  - The second version does not take any arguments and just starts a $digest cycle.

Digest Cycle and $scope

- **Built-in directives/services (like ng-click, ng-repeat, ng-model, $timeout,$http etc) which changes the models automatically trigger a $digest cycle.i.e. It calls $apply() automatically and creates implicit watches to the model variables.**
- **If we change any model outside of the Angular context, then we need to inform Angular of the changes made by calling $apply() manually. For instance, if setTimeout() function updates a scope model, Angular wont have any idea about the model change then it becomes our responsibility to call $apply() manually, which in turn triggers a $digest cycle.**
- **If a directive that sets up a DOM event listener and changes models inside the handler function, we need to call $apply() to ensure the changes take effect.**

Digest Cycle and $scope



Source : AngularJS.org

$watch

- **$watches can be used to watch any value, and trigger a function call when that value changes. A $watch can be set up from any $scope by calling $scope.$watch().**
- **There are two ways to set up a watch (no difference between two)**
  - By Expression.

```
$scope.$watch('modelVariable', function(newValue, oldValue) {
        //update the DOM with newValue
});
```

  - By Function

```
$scope.$watch( function() { return $scope.modelVariable; }, function(newValue, oldValue) {
        //update the DOM with newValue
} });
```

$digest

- **$digest loops through all watchers on the scope on which it is called and it's child scopes. It evaluates them and executing the handlers if any changes found.**
- **To call $digest**
  - $scope.$digest();

$apply

- **It's a wrapper around $rootScope.$digest that evaluates any expression passed to it prior to calling $digest().**
- **Different ways to calling $apply:**
  - $scope.$apply('modelVariable= "test"');
  - $scope.$apply(function(scope) {

    scope. modelVariable = 'test';

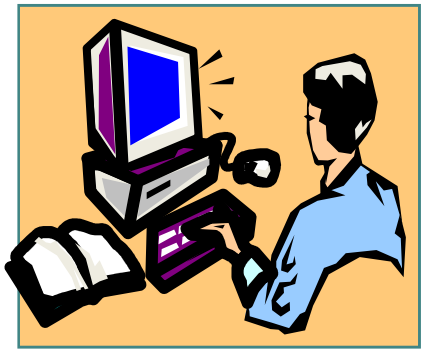
    });
  - $scope.$apply(function(){

    scope. modelVariable = 'test';

    });
  - $scope.$apply(); //Similar to $digest()

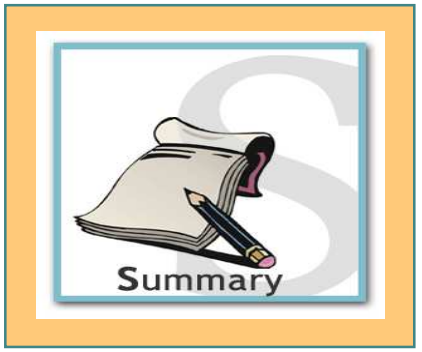
Demo

- **Directive03**



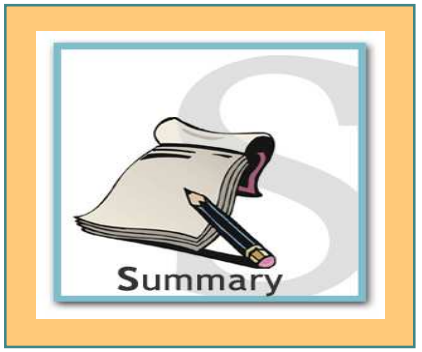
Summary

- scope is not a model actually it contains the model
- We can use JavaScript objects as model in angular.
- Double curly brace is the markup indicator for binding data to view
- ngSrc is used to bind and image's src. It delays fetching an image until binding has occured.
- angular directives can be written in three different ways : tag, attribute & class and we cannot write all the inbuilt directives in all the 3 ways.
- ngChange directive requires the ngModel directive to also be present



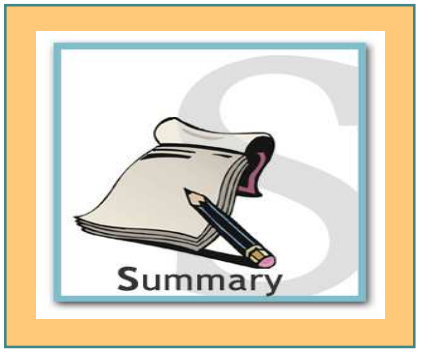
Summary

- ngCloak directive is used to avoid a flash of unbound html
- ngBind does not support multiple bindings where as ngBindTemplate supports multiple bindings
- ngClass directives has two companion directives : ngClassEven & ngClassOdd
- ngForm allow us to Nest forms
- Avoid custom tag name directives to make Angular support with older version of IE.
- Two way binding will update on every key stroke. Two way binding is supported by Input, Select and Textarea HTML elements



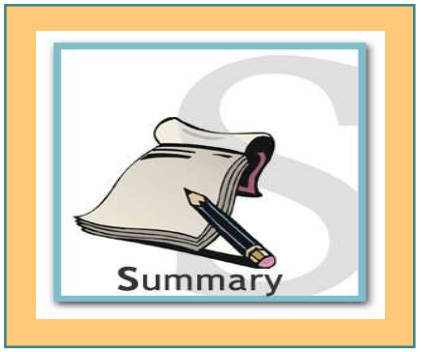
Summary

- Two way binding will update on every key stroke. Two way binding is supported by Input, Select and Textarea HTML elements
- The property will be created automatically, when we refer a property that doesn't exist in a ngModel directive.
- ngPattern directive allow us to create a regex for validation.
- form tag should have a name property to check the validity of form.
- restrict property of a directive can take the following values : E , A, C and M



Summary

- **template and templateUrl is used to specify the html which will be used by our directives.**
- **If we change any model outside of the Angular context, then we need to inform Angular of the changes made by calling $apply() manually.**



AngularJS

AngularJS Filters

Lesson Objectives

- **Built-In Filters**
- **Creating Custom Filters**

Filters

- **A filter formats the value of an expression for display to the user.**
- **Filters can be invoked in HTML with the | (pipe) character inside the template.**
- **We can also use filters from within JavaScript by using the $filter service.**
- **To pass an argument to a filter in the HTML form, we pass it with a colon after the filter name (for multiple arguments, we can simply append a colon after each argument)**
- **Angular gives us several built-in filters as well as an easy way to create our own.**

Built-In Filters

- **uppercase**
  - Converts string to uppercase.
  - {{'igate'|uppercase}} <!-- Displays: IGATE -->

```
angular.module('filterApp',[])
.controller('FilterController', function($scope,$filter) {
        $scope.companyName = $filter('uppercase')('igate');
});
```

- **lowercase**
  - Converts string to lowercase.
  - {{'IGATE'|lowercase}} <!-- Displays: igate -->
- **number**
  - Formats a number as text. If the input is not a number an empty string is returned.
  - {{ 123.456789 | number:2 }} <!-- Displays: 123.46 -->

Built-In Filters

- **currency**
  - The currency filter formats a number as currency. Currency gives us the option of displaying a currency symbol or identifier.
  - The default currency option is that of the current locale we can also pass in a currency to display.
  - {{ 30 | currency}} <!-- Displays: $30.00 -->
  - {{ 30 | currency : "Rs."}} <!-- Displays: Rs.30.00 -->
- **json**
  - Convert s JavaScript object into JSON string.
  - This filter is mostly useful for debugging
  - {{{ 'Id':714709,'Name':'Karthik'} | json }}

<!-- Displays: { "Id": 714709, "Name": "Karthik"} -->

Built-In Filters

- **date**
  - Formats date to a string based on the requested format.
  - Date to be formatted can be either a Date object or milliseconds (string or number)
  - The date formatter provides us several built-in options. If no date format is passed, then it set to the default mediumDate.

```
angular.module('filterApp',[])
.controller('FilterController', function($scope,$filter) {
        $scope.today = new Date();
        $scope.formattedDate = $filter('date')(new Date(),'EEEE dd, MMMM yyyy');
});
```

  - {{today | date:'medium'}} <!-- Displays: Jul 30, 2014 5:13:19 PM -->
  - {{today | date:'short'}} <!-- Displays: 7/30/14 5:13 PM -->
  - {{today | date:‘fullDate'}} <!-- Displays: Wednesday, July 30, 2014-->
  - {{today | date:‘longDate'}} <!-- Displays: July 30, 2014-->

Built-In Filters

- {{today | date:'mediumDate'}} <!-- Displays: Jul 30, 2014-->
- {{today | date:‘shortDate'}} <!-- Displays: 7/30/14 -->
- {{today | date:‘mediumTime'}} <!-- Displays: 5:22:21 PM -->
- {{today | date:‘mediumTime'}} <!-- Displays: 5:23 PM -->
- {{today | date:'d-M-y'}} <!-- Displays: 30-7-2014 -->
- {{today | date:'d-M-yyyy'}} <!-- Displays: 30-7-2014 -->
- {{today | date:'dd-MM-yy'}} <!-- Displays: 30-07-14 -->
- {{today | date:'EEEE dd, MMMM yyyy'}} <!-- Displays: Wednesday 30, July 2014 -->
- {{today | date:'EEE dd MMM yyyy'}} <!-- Displays: Wed 30 Jul 2014 -->
- {{ today | date:'hh:mm:ss.sss a' }} <!-- Displays: 05:35:31.951 PM -->
- {{ today | date:'hh:mm:ss a' }} <!-- Displays: 05:35:31 PM -->
- {{ today | date:'H:m:s a' }} <!-- Displays: 17:35:31 PM -->
- {{ today | date:'Z' }} <!-- Displays: +0530 -->

Built-In Filters

- **filter**
  - filter selects a subset of items from an array of items and returns a new array.
  - The filter method takes a string, object, or function that it will run to select or reject array elements.

filter based on a string :

```
{{ ['Anil', 'Latha', 'Mahima', 'Sachin', 'Veena'] | filter:'e' }}
<!-- returns array ["Veena"]      'e' returns other than 'Veena'    -->
```

filter based on a function:

```
angular.module('filterApp',[])
.controller('FilterController', function($scope,$filter) {
        var pattern = /^\d{6}$/;
        $scope.getSixDigitsPattern = function(item) {
                return pattern.test(item);
        }
});
```

Built-In Filters

```
{{ ['714709', '562A', '044-235', '801234','ABC'] | filter:getSixDigitsPattern }}
<!-- return array ["714709","801234"] -->
```

filter based on a object:

```
{{
    [
        {"Id":1,"Name":"Anil","Location":"Mumbai"},
        {"Id":2,"Name":"Latha","Location":"Bangalore"},
        {"Id":3,"Name":"Mahima","Location":"Pune"},
        {"Id":4,"Name":"Sachin","Location":"Mumbai"},
        {"Id":5,"Name":"Veena","Location":"Pune"}
    ] | filter:{"Location":"Mumbai"}
}}
<!-- returns array
[
    {"Id":1,"Name":"Anil","Location":"Mumbai"},
    {"Id":4,"Name":"Sachin","Location":"Mumbai"}
]
-->
```

Built-In Filters

- **limitTo**
  - limitTo filter creates a new array or string that contains only the specified number of elements, either taken from the beginning or end, depending on whether the value is positive or negative.

```
{{ 'IGATE GLOBAL SOLUTIONS' | limitTo:5 }}
<!-- returns  first 5 characters: IGATE -->

{{ 'IGATE GLOBAL SOLUTIONS' | limitTo:-9 }}
<!-- returns  last 9 characters: SOLUTIONS -->

{{
        ['Bangalore','Chennai','Hyderabad','Gandhinagar','Mumabai','Noida','Pune']
       | limitTo:2
}}
<!--  returns  first 2 array elements :  ["Bangalore","Chennai"] -->
```

Built-In Filters

- **orderBy**
  - The orderBy filter orders the specific array using an expression.
  - It is ordered alphabetically for strings and numerically for numbers.
  - It takes 2 parameters. First parameter is the predicate used to determine the order of the sorted array. Second parameter(optional) is a boolean value, if it is true it will sort the data in reverse order.

```
{{['Chennai','Bangalore','Pune','Mumbai'] | orderBy:'toString()'}}
<!-- returns ["Bangalore","Chennai","Mumbai","Pune"] -->


{{[{'Id':1,'Location':'Bangalore'},{'Id':2,'Location':'Chennai'}] | orderBy:'Id':true}}
<!-- returns [{"Id":2,"Location":"Chennai"},{"Id":1,"Location":"Bangalore"}] -->
```

Custom Filters

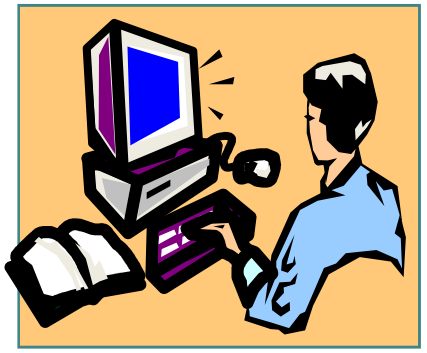
- We can easily create custom filters in AngularJS.

```
angular.module('filterApp',[])
.filter('getIntegers',function(){
        return function(numberArray){
                var size = numberArray.length;
                var evenNumbers = [];
                for(counter=0;counter<size;counter++)
                {
                        if(numberArray[counter]%2==0)
                            evenNumbers.push(numberArray[counter]);
                }
                return evenNumbers;
        }
})

{{[1,2,3,4,5,6,7,8,9,10] | getIntegers}}        <!- - returns [2,4,6,8,10]  - ->
```

- Filters are just functions to which we pass input. In the function above, we simply take the input as the array on which we are calling the filter.

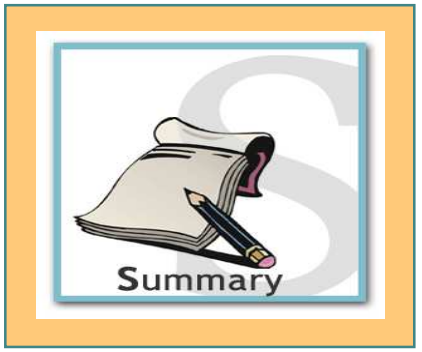
Demo

- **FiltersDemo**



Summary

- Filters are just functions to which we pass input.
- Filters can be invoked in HTML with the | (pipe) character inside the template.
- We can indicate a parameter to a filter using a colon (:)
- We can limit a filter to only search in a specific field
- We can specify custom date formats using date filter



AngularJS

AngularJS Services

Lesson Objectives

- **Creating and Registering Services**
- **Built-In Services**

Service Introduction

- **Service is just a simple JavaScript object that does some sort of work. It is typically stateless and encapsulates some sort of functionality.**
- **As a best practice we need to place the business logic into a service instead of placing it in the controller. It help us adhere to the Single Responsibility Principle(SRP) and Dependency Inversion Principle (DIP) as well as make the service reusable.**
- **Service can be used with controller, directive to construct model.**
- **Services provide a method for us to keep data around for the lifetime of the app and communicate across controllers in a consistent manner.**
- **Services are singleton objects that are instantiated only once per app (by the $injector) and lazyloaded (created only when necessary).**

Creating and Registering a Service

- **Angular comes with several built-in services along with that we can create our own services.**
- **Angular compiler can reference service and load it as a dependency for runtime once it is registered.**
- **We can create service using five different ways**
  - factory()
  - service()
  - provider()
  - constant()
  - value()

Registering service using factory() function

- **The most common method for registering a service with our Angular app is through the factory() method. This method is a quick way to create and configure a service.**
- **The factory() function takes two arguments.**
  - Name of the service which we want to register
  - Function which runs when Angular creates the service. It will be invoked once for the duration of the app lifecycle, as the service is a singleton object. It can return anything from a primitive value to a function to an object.

```
var app = angular.module("serviceApp",[]);
app.factory("companyService",function(){
        return {
                company : {"Name":"IGATE", "Location":"India"}
        };
});
```

Registering service using service() function

- **The service() function is used to register an instance of a service using a constructor function.**
- **The service() function will instantiate the instance using the new keyword when creating the instance.**
- **The service() function takes two arguments.**
  - Name of the service which we want to register
  - The constructor function that we'll call to instantiate the instance.

```
var app = angular.module("serviceApp",[]);
/*constructor  function*/
var Company = function(){
        this.getCompanyInfo = function(){
                return {"Name":"IGATE", "Location":"India"};
        };
};
app.service('companyService', Company);    // service() function
```

Registering service using provider() function

- **A provider is an object with a $get() method. The $injector calls the $get method to create a new instance of the service. The provider() method is responsible for registering services in the $providerCache.**
- **factory() function is shorthand for creating a service through the provider() method wherein we assume that the $get() function is the function passed**
- **$provide service is responsible for instantiating these providers at runtime.**
- **The provide() function takes two arguments. Name & (object/function/array)**

```
var app = angular.module("serviceApp",[]);
app.provider("companyService", {
        $get : function(){
            return {
                    company : {"Name":"IGATE", "Location":"India"}
             };
        }
});
```

Registering a Service with $provide

- **We can also register services via the $provide service inside of a module's config function.**

```
<div ng-app="serviceApp" ng-controller="ServiceCntrl">
        <div>{{company.Name}} - {{company.Location}}</div>
</div>

<script>
var app = angular.module("serviceApp",[]);
app.config(function($provide){
   $provide.factory('companyService',function(){
       return {
           company : {"Name":"IGATE", "Location":"India"}
        };
    });
});

app.controller('ServiceCntrl',function($scope,companyService){
        $scope.company = companyService.company;
});
</script>
```

Registering service using constant() function

- **constant() function is used to register an existing value as a service so that we can inject it into a module configuration function**
- **The constant() funtion returns a registered service instance.**
- **The constant() function takes two arguments**
  - The name with which to register the constant.
  - The value to register as the constant

```
var app = angular.module("serviceApp",[]);
app.constant('companyName','IGATE');
app.config(function(companyName){
        //We can resolve companyName here
});
```

Registering service using value() function

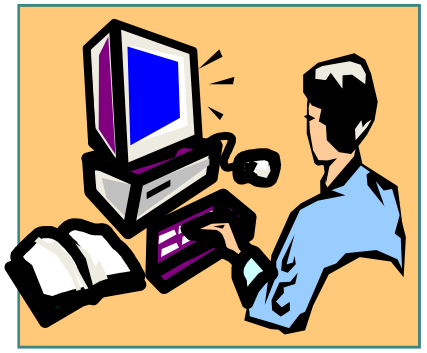
- **value() function is used to register the service, when the return value is just a constant.**
- **The value() funtion returns a registered service instance.**
- **we cannot inject a value into a config function.**
- **The value() function takes two arguments**
  - The name with which to register the value.
  - The injectable instance

```
var app = angular.module("serviceApp",[]);
app.value('employee',{Id: 714709, Name:'Karthik'});
```

- **Use value() to register a service object or function and use constant() for configuration data.**

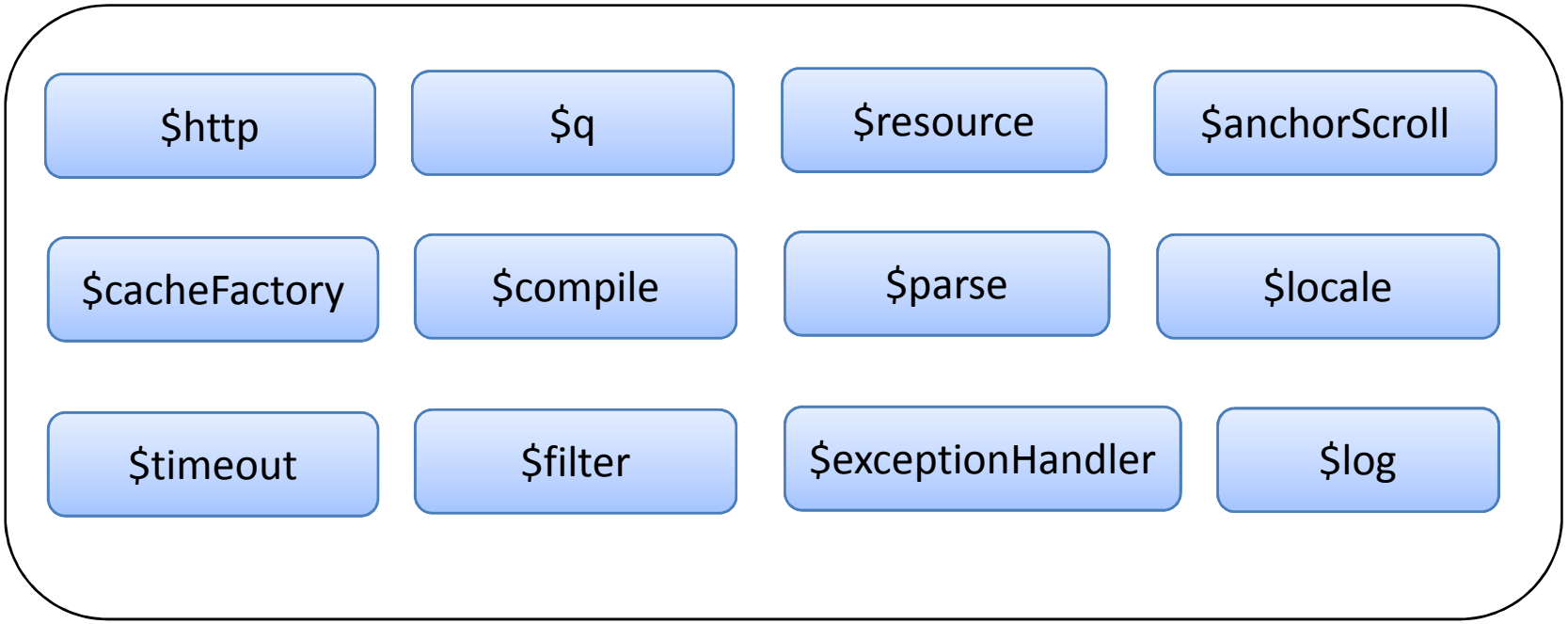
Demo

- **CustomService**



Built-In Services

- **Angular services are substitutable objects that are wired together using dependency injection (DI). We can use those services to organize and share code across the application.**
- **Angular ships with lot of Built-In services. Some of important services are :**



AngularJS promise

- **We used to implement asynchronous code is by using callback functions, but it is too complicated when we have to compose multiple asynchronous calls and make decisions depending upon the outcome of this composition.**
- **A promise represents the eventual result of an asynchronous operation.**
- **A promise is an object with a then method, then() function accepts 2 functions as parameters:**
  - function to be executed (onSuccess) when the promise is fulfilled.
  - function to be executed (onFailure) when the promise is rejected.

promise.then(onSuccess,onFailure);

- **Both functions onSuccess and onFailure takes one parameter i.e. response outcome of an asynchronous service. For each promise only one of the functions (onSuccess or onFailure) can be called.**

$q Service

- **$q service in AngularJS provides us deferred and promise implementations.**
- **Deferred represents a task that will finish in the future. We can get a new deferred object by calling the defer() function on the $q service.**
  - var deferred = $q.defer();
- **The purpose of the deferred object is to expose the associated Promise instance as well as APIs that can be used for signaling the successful or unsuccessful completion, as well as the status of the task.**
  - resolve(…) method of deferred object is used to signal that the task has succeeded.
  - reject(…) method is used to signal that the task has failed.
- **We can pass any type of information to resolve and reject methods which becomes the result of the task when it is called. The deferred object has a promise property which represents the promise of this task.**

$q Service

```
<div ng-app="serviceApp" ng-controller="ServiceCntrl">
  <div><input type="button" value="$q Service Test" ng-click="test()"/></div>
</div>
<script>
var app = angular.module("serviceApp",[]);
app.controller('ServiceCntrl',function($scope,$q){
        $scope.test = function(){
                var deferred = $q.defer();
                var promise = deferred.promise;
                promise.then(function(response){
                        alert('Success :'+response.message);
                },function(response){
                        alert('Error :'+response.message);
                });
                /*uncomment to resolve the promise*/
                deferred.resolve({message:'Promise will gets fullfilled'});
                /*uncomment to reject the promise*/
                //deferred.reject({message:'Promise gets rejected'});
        }
});
</script>
```

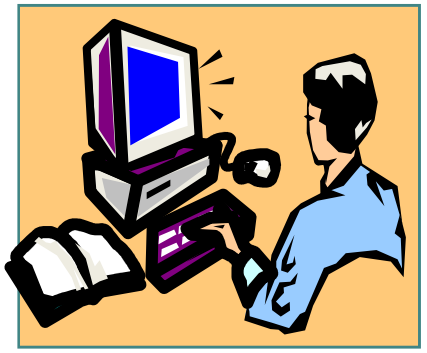
$http Service

- **$http service is a core Angular service that facilitates communication with the remote HTTP servers via the browser's XMLHttpRequest object or via JSONP.**
- **The $http service is a function which takes a single argument i.e. a configuration object which is used to generate an HTTP request and returns a promise with two $http specific methods: success and error.**

```
$http({method: 'GET', url: '/someUrl'}).
  success(function(data, status, headers, config) {
    // this callback will be called asynchronously  when the response is available
  }).
  error(function(data, status, headers, config) {
    // returns response with an error status.
  });
```

Demo

- **httpService**
- **httpService-using-qService**



$resource Service

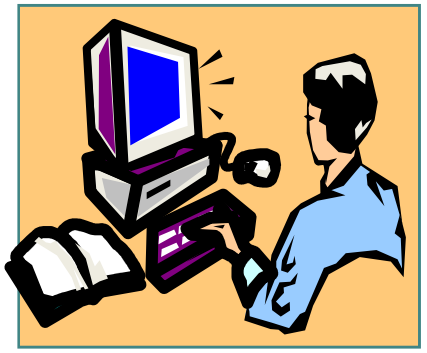
- **AngularJS's $resource service allows us to create convenience methods for dealing with typical RESTful APIs.**
- **RESTful functionality is provided by Angular in the ngResource module, which is distributed separately from the core Angular framework so we need to refer angular-resource.js and define ngResource in our module.**

```
<script src="angular-resource.js"></script>
var app = angular.module('myApp', ['ngResource']);
```

- **$resource service returns resource object it has action methods which provides high-level behaviors so that we can interact with that methods without the need to interact with the low level $http service.**

Demo

- **resourceService**



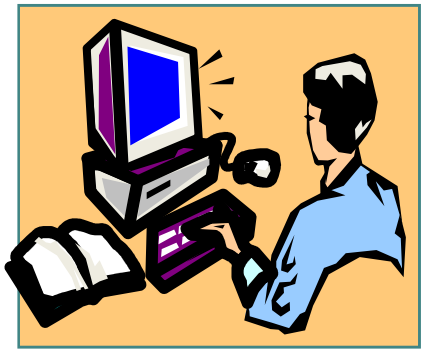
$anchorScroll Service

- **$anchorScrollService checks current value of $location.hash() and scrolls to the related element.**
- **By calling $anchorScrollProvider.disableAutoScrolling() we can disable this feature.**

```
<div id="scrollArea" ng-controller="ScrollController">
        <a ng-click="gotoBottom()">Go to bottom</a>
                        ….
                        ….
        <a id="bottom"></a> You're at the bottom!
</div>

app.controller('ScrollController',function ($scope, $location, $anchorScroll) {
   $scope.gotoBottom = function() {
                    $location.hash('bottom');
                      $anchorScroll();
     };               });
```

Demo

- **anchorScrollService**

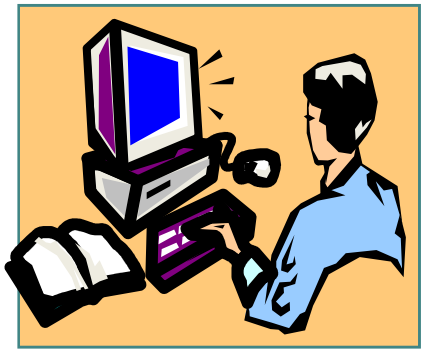


$cacheFactory Service

- **We can cache the objects using $cacheFactory service**
- **cache object has the following set of methods:**
  - {object} info() — Returns id, size, and options of cache.
  - {{*}} put({string} key, {*} value) — Puts a new key-value pair into the cache and returns it.
  - {{*}} get({string} key) — Returns cached value for key or undefined for cache miss.
  - {void} remove({string} key) — Removes a key-value pair from the cache.
  - {void} removeAll() — Removes all cached values.
  - {void} destroy() — Removes references to this cache from $cacheFactory.
- **We can limit the number of items in the cache**
  - $cacheFactory('customCache',{capacity:2})

Demo

- **cacheFactoryService**



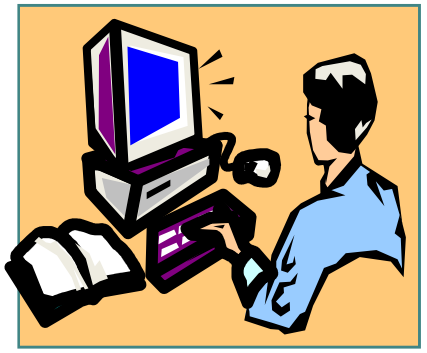
$compile Service

- **$compile service compiles an HTML string or DOM into a template and produces a template function, which can then be used to link scope and the template together.**

```
<div  ng-controller="ServiceController">
                    <div id="target"></div>
   Markup : <input type="text" ng-model="markup"/><br/>
<input type="button" ng-click="appendToDivElement(markup)" value="Append"/>
</div>

var app = angular.module('serviceApp',[]);
app.controller("ServiceController",function($scope,$compile){
        $scope.appendToDivElement= function(markup){
                    return
angular.element('#target').append($compile(markup)($scope));
        }
});
```

Demo

- **compileService**



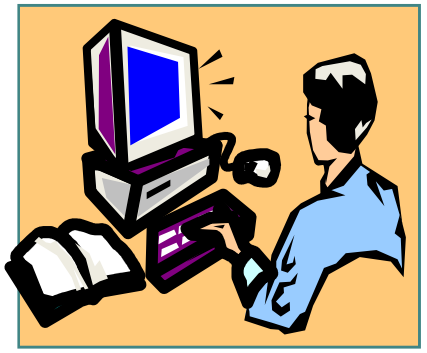
$locale Service

- **$locale service provides localization rules for Angular components.**
- **locale id formatted as languageId-countryId (e.g. en-us)**
- **locale script files are available under i18n folder it needs to be referred in html.**
  - angular-locale_hi-in.js (hindi)
  - angular-locale_ta-in.js (tamil)

```
<div  ng-app=" serviceApp "  ng-controller="ServiceController">
        <h2>{{ todayDate | date:dateFormat }}</h2>
</div>

var app = angular.module('serviceApp',[])
app.controller("ServiceController",function($scope,$locale){
        $scope.todayDate = new Date();
        $scope.dateFormat = $locale.DATETIME_FORMATS.fullDate;
});
```

Demo

- **localeService**



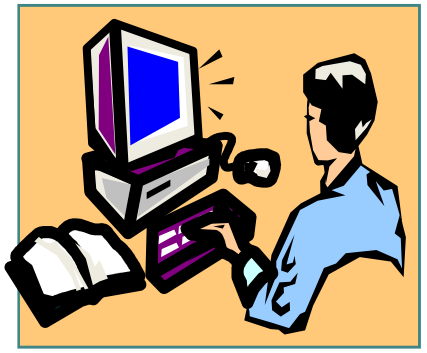
$timeout Service

- **$timeout is a wrapper for window.setTimeout function.**
- **The return value of registering a timeout function is a promise, which will be resolved when the timeout is reached and the timeout function is executed.**
- **To cancel a timeout request, call $timeout.cancel(promise).**

```
<div ng-app="serviceApp" ng-controller="ServiceController">
        <h2>Display Company Name in 5 sec(s): {{ companyName }}</h2>
</div>

var app = angular.module('serviceApp',[])
app.controller("ServiceController",function($scope,$timeout,$interval){
        var timoutPromise = $timeout(function(){
                        $scope.companyName = "IGATE";
        },5000);
});
```

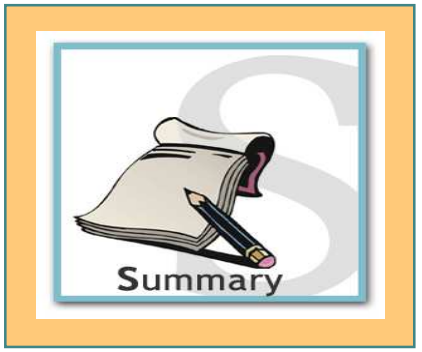
Demo

- **timeoutService**



Summary

- Service is just a simple JavaScript object that does some sort of work.
- We can create service using five different ways
- Angular services are substitutable objects that are wired together using dependency injection (DI).
- A promise is an object with a then method, then() function accepts 2 functions as parameters which gets executed when the promise gets fulfilled and rejected respectively.
- AngularJS's $resource service allows us to create convenience methods for dealing with typical RESTful APIs



AngularJS

AngularJS Routing

Lesson Objectives

- AngularJS Routing Basics
- Understanding Routing Modes
- Working with $routeParams Service
- Working with $route Service
- Working with $location Service
- Working with Routing Events

Routing

- **It is very important to navigate from one page view to another in single page application.**
- **We can achieve this by including multiple templates in the view using ng-include directive, but this will be unmanageable and also make it difficult to allow other developers to join in the development.**
- **We can break out the view into a layout and template views and only show the view which we want to show based upon the URL the user is accessing. Routing means loading sub-templates depending upon the URL of the page.**
- **Routes are a way for multiple views to be used within a single HTML page. This enables you page to look more "app-like" because users are not seeing page reloads happen within the browser.**

AngularJS Routes

- **AngularJS routes enable us to create different URLs for different content in our application. Having different URLs for different content enables the user to bookmark URLs to specific content. In AngularJS each such bookmarkable URL is called a route.**
- **AngularJS routes enables us to show different content depending on what route is chosen. A route is specified in the URL after the # sign**
  - http://igate.com/index.html#/training

Setting up page for routing

- **To setup a page for routing we need to follow the 4 steps given below**
- **AngularJS requires the route service, which is not part of the default Angular library. We need to load angular-route.js, as part of your script loading.**
  - <script type="text/javascript" src="Scripts/angular-route.js"></script>
- **We need to inject the route service in our app module**
  - var app = angular.module('routeApp',['ngRoute']);
- **Use ngView directive in the HTML tag(use div tag) to display the given route.**
  - <div ng-view />
- **Configure $routeProvider in the module's config() function via calls to the when() and otherwise() functions.**

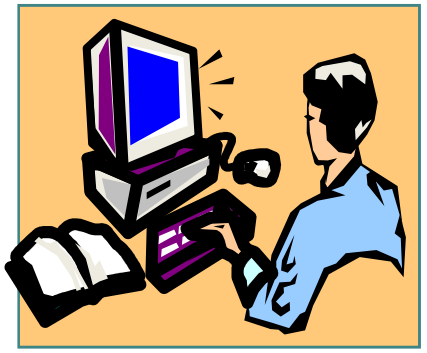
Setting up page for routing

```
var app = angular.module('routeApp',['ngRoute']);
app.config(function($routeProvider){
	$routeProvider
		.when('/',
		{
			template:'<h1>Home Page</h1>'
		})
		.when('/company',
		{
			template:'<h1>IGATE</h1>'
		})
		.otherwise({
			redirectTo:'/'
		})
});
```

- **When the browser loads the Angular app, it will default to the URL set as the default route. Unless we load the browser with a different URL, the default is the '/' route.**

Demo

- **RouteBasics**



Routing Modes

- **Routing mode refers specifically to the format of the URL in the browser address bar. It determines the look of the URL. AngularJS has 2 routing modes**
- **Hashbang Mode**
  - The default behavior of the $location service is to route using the hashbang mode. It provides deep-linking capabilities to Angular apps. URL paths take a prepended '#' character. We can configure hashbang mode in the config function on an app module. We can also configure the hashPrefix, which is part of the fallback mechanism that Angular uses for older browsers.

```
var app = angular.module('routeApp',['ngRoute']);
app.config(function($routeProvider,$locationProvider){
        $locationProvider.html5Mode(false);
        $locationProvider.hashPrefix('!');
});
```

Routing Modes

- **HTML5 Mode**
  - This mode makes URLs look like regular URLs (except that in older browsers they will look like the hashbang URL).
  - $location service automatically falls back to using hashbang URLs if the browser doesn't support the HTML5 history API and also rewrites the URL.
  - For example, with the tag: Employee, a legacy browser's URL will be rewritten to the hashbang URL equivalent: /index.html#! /employee/36?show=true

```
var app = angular.module('routeApp',['ngRoute']);
app.config(function($routeProvider,$locationProvider){
        $locationProvider.html5Mode(true);
});
```

Route Parameters

- **AngularJS will parse a route param with a colon (:) and pass it to $routeParams.**

```
var app = angular.module('routeApp',['ngRoute']);
app.config(function($routeProvider){
        $routeProvider
        .when('/Employees/:id',
        {
                templateUrl:'partials/employees.html',
                controller:'EmployeeController'
        });
});
```

- **Angular will populate the $routeParams with the key of :id, and the value of key will be populated with the value of the loaded URL. If the browser loads the URL /Employees/714709, then the $routeParams object will look like: {id:714709}**

Route Parameters

- **Parameter samples**
  - '/igate' : Matches exactly **igate**
  - '/employee/:id' : Matches employee /714709, employee /desigan
- **We can also specify parameters as query parameters following a '?'**
  - '/employee/:id' : Matches employee /714709?department=training&company=IGATE

```
var app = angular.module('routeApp',['ngRoute']);
app.config(function($routeProvider){
        $routeProvider .when('/employee/:id',
        {
                redirectTo:function(routeParams,path,search){
                    console.log(routeParams);  // Object {id: "714709"}
                     console.log(path);              //   /employee/714709
console.log(search);    // Object {department: "training", company: "IGATE"}
                      return "/";
                }
        })
});
```

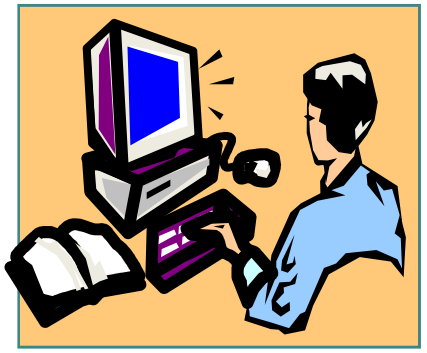
$routeParams

- **The $routeParams service allows you to retrieve the current set of route parameters. Controller functions can get access to route parameters via the AngularJS $routeParams service**

```
var app = angular.module('routeApp',['ngRoute']);
app.config(function($routeProvider){
	$routeProvider
		.when('/:company',
		{
			templateUrl:'partials/map.html',
			controller:'RouteController'
		})
});
app.controller("RouteController",function($scope,$routeParams){
	$scope.model = $routeParams.company;

});
```

Demo

- **RouteParams**

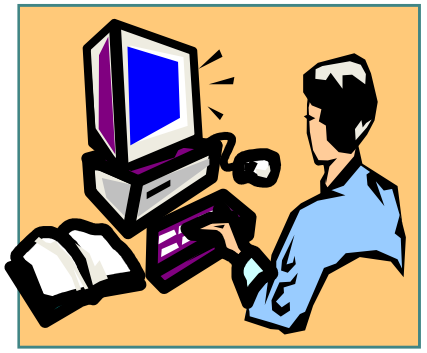


$route

- **$route service is used for deep-linking URLs to controllers and views (HTML partials).**
- **It watches $location.url() and tries to map the path to an existing route definition.**
- **Using route service in the controller, we can access**
  - custom property defined in routing ($route.current.propertyName)
  - URL of the template($route.current.templateUrl)
  - contents of the template($route.current.locals.$template)
  - route parameters($route.current.pathParams)
  - query string passed in URL ($route.current.params)
- **We can reload the partial page using $route.reload()**

Demo

- **RouteService**

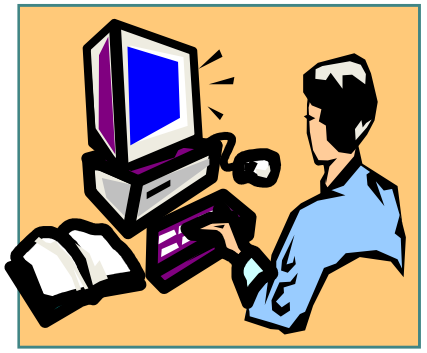


resolve property

- **Resolve is a property on the routing configuration, and each property on resolve can be an injectable function (it can ask for service dependencies). The function should return a promise.**
- **A resolve contains one or more promises that must resolve successfully before the route changes. i.e. We can wait for the data available before showing a view**
- **It simplifies the initialization of the model inside a controller because the initial data is given to the controller instead of the controller needing to go out and fetch the data.**
- **When the promise completes successfully, the resolve property is available to inject into a controller function**

Demo

- **RouteService-Resolve**

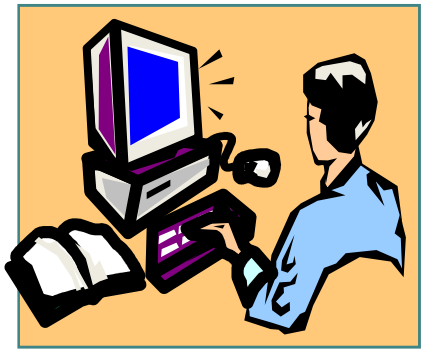


$location service

- **$location service parses the URL in the browser address bar and makes the URL available to the application. The route parameters are a combination of $location's search() and path()**
- **Route service watches $location.url() and tries to map the path to an existing route definition.**
- **$location service is used when application needs to react to a change in the current URL or there is a need to change the current URL in the browser.**

Demo

- **Route-UsingLocationService**



Route Events

- **$route service fires events at different stages of the routing flow. We can set up event listeners for these different routing events and react.**
- **It is useful when we want to manipulate events based upon routes and is particularly useful for detecting when users are logged in and authenticated.**
- **Using $rootScope, we can set up an event listener to listen for routing events.**
- **$routeChangeStart**
  - Angular broadcasts $routeChangeStart before the route changes. This step is where the route services begin to resolve all of the dependencies necessary for the route change to happen and where templates and the resolve keys are resolved.
  - The $routeChangeStart event fires with two parameters:
    - The next URL to which we are attempting to navigate
    - The URL that we are on before the route change

Route Events

- **$routeChangeSuccess**
  - Angular broadcasts the $routeChangeSuccess event after the route dependencies have been resolved.
  - The $routeChangeSuccess event fires with three parameters:
    - The raw Angular evt object
    - The route where the user currently is
    - The previous route (or undefined if the current route is the first route)
- **$routeChangeError**
  - Angular broadcasts the $routeChangeError event if any of the promises are rejected or fail.
  - The $routeChangeError event fires with three parameters:
    - The current route information
    - The previous route information
    - The rejection promise error

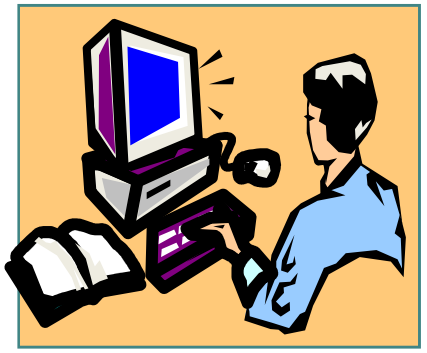
Route Events

- **$routeUpdate**
  - Angular broadcasts the $routeUpdate event if the reloadOnSearch property has been set to false and we're reusing the same instance of a controller.

```
angular.module('myApp', [])
.run(['$rootScope', '$location', function($rootScope, $location) {
        $rootScope.$on('$routeChangeStart', function(evt, next, current) {
        })
}])

angular.module('myApp', [])
.run(['$rootScope', '$location', function($rootScope, $location) {
        $rootScope.$on('$routeChangeSuccess', function(evt, next, previous) {
        })
}])

angular.module('myApp', [])
.run(['$rootScope', '$location', function($rootScope, $location) {
    $rootScope.$on('$routeChangeError', function(current, previous, rejection) {
    })
}])
```

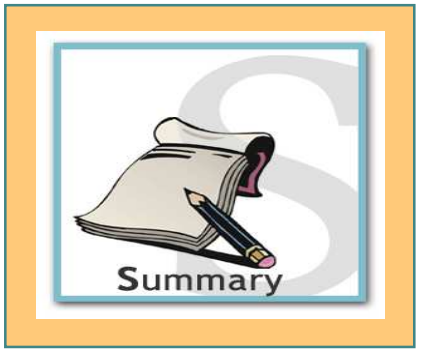
Demo

- **Route-RouteEvents**



Summary

- **$routeProvider service use to create routes.**
- **otherwise $routeProvider function allows us to set a default route.**
- **Using $routeParams service we can access the parameters passed on a route.**
- **$route.reload() allows us to refresh a view without refreshing the entire app.**
- **We can enable HTML5 routing using $locationProvider service *$locationProvider.html5Mode(true)*, but it requires server-side configuration.**
- **resolve route property allows us to delay loading a view until the data which needs to get render is loaded.**



Summary

- **We can navigate to a new view from the code using $location.url('newUrl').**
- **$location.search() gives us the access to the query string parameters on the URL.**