AngularJS

AngularJS:

* **Client-side technology, written entirely in JavaScript**
* It works with the long-established technologies of the web (HTML, CSS, and JavaScript) to make the development of web apps easier and faster than ever before.
* It is used in Single Page Application (SPA) projects.
* It extends HTML DOM with additional attributes and makes it more responsive to user actions.
* AngularJS is open source, completely free.
* It is now maintained by Google
* Angular 1 is called Angularjs while Angular version 2 and above are called Angular

Example:

<!doctype html>

<html ng-app>

<head>

<script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.3/angular.min.js"></script>

</head>

<body>

<div>

<label>Name:</label>

<input type = "text" ng-model = "yourName" placeholder = "Enter a name here">

<hr />

<h1>Hello {{yourName}}!</h1>

</div>

</body>

</html>

Features:

* It lets you use HTML as your template language and lets you extend HTML's
* Angular's data binding and dependency injection eliminate much of the code you currently have to write
* AngularJS is a powerful JavaScript based development framework to create RICH Internet Application(RIA).
* AngularJS provides developers options to write client side application (using JavaScript) in a clean MVC(Model View Controller) way.
* Application written in AngularJS is cross-browser compliant. AngularJS automatically handles JavaScript code suitable for each browser.
* AngularJS is open source, completely free, and used by thousands of developers around the world.

Core Features:

* **Data-binding** − It is the automatic synchronization of data between model and view components.
* **Scope** − These are objects that refer to the model. They act as a glue between controller and view.
* **Controller** − These are JavaScript functions that are bound to a particular scope.
* **Services** − AngularJS come with several built-in services for example $http to make a XMLHttpRequests. These are singleton objects which are instantiated only once in app.
* **Filters** − These select a subset of items from an array and returns a new array.
* **Directives** − Directives are markers on DOM elements (such as elements, attributes, css, and more). These can be used to create custom HTML tags that serve as new, custom widgets. AngularJS has built-in directives (ngBind, ngModel...)
* **Templates** − These are the rendered view with information from the controller and model. These can be a single file (like index.html) or multiple views in one page using "partials".
* **Routing** − It is concept of switching views.
* **Model View Whatever** − MVC is a design pattern for dividing an application into different parts (called Model, View and Controller), each with distinct responsibilities. AngularJS does not implement MVC in the traditional sense, but rather something closer to MVVM (Model-View-ViewModel). The Angular JS team refers it humorously as Model View Whatever.
* **Deep Linking** − Deep linking allows you to encode the state of application in the URL so that it can be bookmarked. The application can then be restored from the URL to the same state.
* **Dependency Injection** − AngularJS has a built-in dependency injection subsystem that helps the developer by making the application easier to develop, understand, and test.

| **Concept** | **Description** |
| --- | --- |
| [Template](https://docs.angularjs.org/guide/concepts#template) | HTML with additional markup |
| [Directives](https://docs.angularjs.org/guide/concepts#directive) | extend HTML with custom attributes and elements |
| [Model](https://docs.angularjs.org/guide/concepts#model) | the data shown to the user in the view and with which the user interacts |
| [Scope](https://docs.angularjs.org/guide/concepts#scope) | context where the model is stored so that controllers, directives and expressions can access it |
| [Expressions](https://docs.angularjs.org/guide/concepts#expression) | access variables and functions from the scope |
| [Compiler](https://docs.angularjs.org/guide/concepts#compiler) | parses the template and instantiates directives and expressions |
| [Filter](https://docs.angularjs.org/guide/concepts#filter) | formats the value of an expression for display to the user |
| [View](https://docs.angularjs.org/guide/concepts#view) | what the user sees (the DOM) |
| [Data Binding](https://docs.angularjs.org/guide/concepts#databinding) | sync data between the model and the view |
| [Controller](https://docs.angularjs.org/guide/concepts#controller) | the business logic behind views |
| [Dependency Injection](https://docs.angularjs.org/guide/concepts#di) | Creates and wires objects and functions |
| [Injector](https://docs.angularjs.org/guide/concepts#injector) | dependency injection container |
| [Module](https://docs.angularjs.org/guide/concepts#module) | a container for the different parts of an app including controllers, services, filters, directives which configures the Injector |
| [Service](https://docs.angularjs.org/guide/concepts#service) | reusable business logic independent of views |

How Is AngularJS different from other JavaScript framework?

* In other JavaScript frameworks, we are forced to extend from custom JavaScript objects and manipulate the DOM from the outside in.

var btn = $("Hi");

btn.on('click', function(evt) { console.log("Clicked button") }); $("#checkoutHolder").append(btn);

* It requires the developer to have knowledge of the entire DOM and force our complex logic inside JavaScript code to manipulate a foreign DOM.
* AngularJS, on the other hand, augments HTML to give it native Model-View-Controller (MVC) capabilities.
* It enables you, the developer, to encapsulate a portion of your entire page as one application, rather than forcing the entire page to be an AngularJS application.

SPA(Single Page Application):

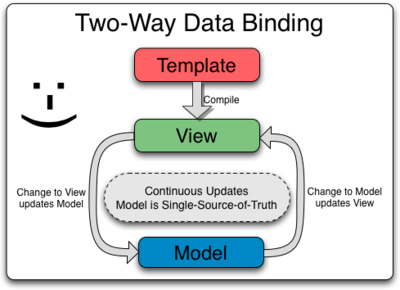
* A single-page application (SPA) is a web application or web site that fits on a single web page with the goal of providing a more fluid user experience similar to a desktop application.
* AngularJs provides **ng-route** and **ng-view** directive and by using them we can easily develop our single page web app.
* Advantages:
  + ***SPA is extremely good for very responsive sites:***
    - Server-side rendering is not as hard as it used to be with simple techniques like keeping a #hash in the URL, or more recently [HTML5 pushState](https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Manipulating_the_browser_history). With this approach the exact state of the web app is embedded in the page URL.
    - As in GMail every time you open a mail a special hash tag is added to the URL. If copied and pasted to other browser window can open the exact same mail (provided they can authenticate).
  + ***With SPA we don't need to use extra queries to the server to download pages:***
    - It can download all the html templates at one go and then we need to only do ajax call to populate the template.
  + Performance Improvement, Single Page Application can improve performance in many ways, Single time file load each of HTML, CSS, JS.
* Disadvantages:
  + ***Client must enable javascript***
  + ***Security:***
    - Javascript is at client side so it is not that secured***.***

# Data Binding

Data-binding in AngularJS apps is the automatic synchronization of data between the model and view components. The way that AngularJS implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa.

**What is bi-directional Data binding in AngularJS?**

Any time the model is changed in the client-side model, the view reflects these changes without writing any custom code this is automatic Data binding.



Bi-directional in this context means that if the view changes the value, the model observes the change through dirty checking, and if the model changes the value, the view update with the change.

To set up this binding, we used the ng-model function on the input, like so:

<input ng-model="person.name" type="text" placeholder="Your name">

<h1>Hello {{ person.name }}</h1>

Now we can see that we’re setting up a bi-directional binding purely in the view. To illustrate the bidirectional binding from the other way (back end to front end), we’ll have to dive into Controllers

<!doctype html>

<html ng-app>

<head>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.0-rc.2/angular.js"></script>

</head>

<body>

<div ng-controller="MyController">

<h1>Hello {{ clock }}!</h1>

</div>

<script type="text/javascript">

function MyController($scope) {

$scope.clock = new Date();

var updateClock = function() {

Data Binding and Your First AngularJS Web Application 15

$scope.clock = new Date();

};

setInterval(function() {

$scope.$apply(updateClock);

}, 1000);

updateClock();

};

</script>

</body>

</html>

**When the page is loaded in the browser, following things happen:**

1. All the Html files are loaded
2. All the JavaScript files are loaded.
3. AngularJS global variable is created.
4. JavaScript which registers controller functions is executed
5. Next AngularJS scans through the HTML to look for AngularJS apps and views
6. Once view is located, it connects that view to the corresponding controller function
7. Next, AngularJS executes the controller functions.
8. It then renders the views with data from the model populated by the controller. The page is now ready.

**MVC Architecture**

**M**odel **V**iew **C**ontroller or MVC as it is popularly called, is a software design pattern for developing web applications. A Model View Controller pattern is made up of the following three parts −

* **Model** − It is the lowest level of the pattern responsible for maintaining data.
* **View** − It is responsible for displaying all or a portion of the data to the user.
* **Controller** − It is a software Code that controls the interactions between the Model and View.

**AngularJS - Directives**

* **ng-app** − This directive starts an AngularJS Application.
* **ng-init** − This directive initializes application data.
* **ng-model** − This directive defines the model that is variable to be used in AngularJS.
* **ng-repeat** − This directive repeats html elements for each item in a collection.

**Expressions**

* Expressions are used to bind application data to html.
* Expressions are written inside double braces like {{ expression}}.
* Expressions behave in same way as ng-bind directives.
* AngularJS application expressions are pure javascript expressions and outputs the data where they are used.
* An expression that starts with :: is considered a one-time expression. One-time expressions will stop recalculating once they are stable, which happens after the first digest if the expression result is a non-undefined value (see value stabilization algorithm below).

{{ ::vm.user }}

* The main purpose of one-time binding expression is to provide a way to create a binding that gets deregistered and frees up resources once the binding is stabilized. Reducing the number of expressions being watched makes the digest loop faster and allows more information to be displayed at the same time.

# Interpolation and data-binding

* Interpolation markup with embedded [expressions](https://docs.angularjs.org/guide/expression) is used by AngularJS to provide data-binding to text nodes and attribute values.
* During the compilation process the [compiler](https://docs.angularjs.org/api/ng/service/$compile) uses the [$interpolate](https://docs.angularjs.org/api/ng/service/$interpolate) service to see if text nodes and element attributes contain interpolation markup with embedded expressions.
* If that is the case, the compiler adds an interpolateDirective to the node and registers [watches](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) on the computed interpolation function, which will update the corresponding text nodes or attribute values as part of the normal [digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) cycle.

### How the string representation is computed

If the interpolated value is not a String, it is computed as follows:

* undefined and null are converted to ''
* if the value is an object that is not a Number, Date or Array, $interpolate looks for a custom toString() function on the object, and uses that. Custom means that myObject.toString !== Object.prototype.toString.
* if the above doesn't apply, JSON.stringify is used

#### **Why mixing interpolation and expressions is bad practice:**

<div ng-show="form{{$index}}.$invalid"></div>

* It increases the complexity of the markup
* There is no guarantee that it works for every directive, because interpolation itself is a directive. If another directive accesses attribute data before interpolation has run, it will get the raw interpolation markup and not data.
* It impacts performance, as interpolation adds another watcher to the scope.
* Since this is not recommended usage, we do not test for this, and changes to AngularJS core may break your code.

**Controllers**

* AngularJS application mainly relies on controllers to control the flow of data in the application between model and view.
* A controller is defined using ng-controller directive. A controller is a JavaScript object containing attributes/properties and functions.
* Each controller accepts $scope as a parameter which refers to the application/module that controller is to control.

Use controllers to:

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

Do not use controllers to:

* Manipulate DOM — Controllers should contain only business logic. Putting any presentation logic into Controllers significantly affects its testability. AngularJS has [databinding](https://docs.angularjs.org/guide/databinding) for most cases and [directives](https://docs.angularjs.org/guide/directive) to encapsulate manual DOM manipulation.
* Format input — Use [AngularJS form controls](https://docs.angularjs.org/guide/forms) instead.
* Filter output — Use [AngularJS filters](https://docs.angularjs.org/guide/filter) instead.
* Share code or state across controllers — Use [AngularJS services](https://docs.angularjs.org/guide/services) instead.
* Manage the life-cycle of other components (for example, to create service instances).

<html>

<head>

<title>Angular JS Controller</title>

<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp" ng-controller = "studentController">

Enter first name: <input type = "text" ng-model = "student.firstName"><br><br>

Enter last name: <input type = "text" ng-model = "student.lastName"><br>

<br>

You are entering: {{student.fullName()}}

</div>

<script>

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

mainApp.controller('studentController', function($scope) {

$scope.student = {

firstName: "Mahesh",

lastName: "Parashar",

fullName: function() {

var studentObject;

studentObject = $scope.student;

return studentObject.firstName + " " + studentObject.lastName;

}

};

});

</script>

</body>

</html>

**Filters**

* Filters are used to change modify the data and can be clubbed in expression or directives using pipe character. Following is the list of commonly used filters.
* Filters format the value of an expression for display to the user. They can be used in view templates, controllers or services. AngularJS comes with a collection of [built-in filters](https://docs.angularjs.org/api/ng/filter), but it is easy to define your own as well.

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Name** | **Description** |
| 1 | uppercase | Converts a text to upper case text. |
| 2 | lowercase | Converts a text to lower case text. |
| 3 | currency | Formats text in a currency format. |
| 4 | filter | filter the array to a subset of it based on provided criteria. |
| 5 | orderby | orders the array based on provided criteria. |

Example:

<html>

<head>

<title>Angular JS Filters</title>

<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp" ng-controller = "studentController">

<table border = "0">

<tr>

<td>Enter first name:</td>

<td><input type = "text" ng-model = "student.firstName"></td>

</tr>

<tr>

<td>Enter last name: </td>

<td><input type = "text" ng-model = "student.lastName"></td>

</tr>

<tr>

<td>Enter fees: </td>

<td><input type = "text" ng-model = "student.fees"></td>

</tr>

<tr>

<td>Enter subject: </td>

<td><input type = "text" ng-model = "subjectName"></td>

</tr>

</table>

<br/>

<table border = "0">

<tr>

<td>Name in Upper Case: </td><td>{{student.fullName() | uppercase}}</td>

</tr>

<tr>

<td>Name in Lower Case: </td><td>{{student.fullName() | lowercase}}</td>

</tr>

<tr>

<td>fees: </td><td>{{student.fees | currency}}

</td>

</tr>

<tr>

<td>Subject:</td>

<td>

<ul>

<li ng-repeat = "subject in student.subjects | filter: subjectName |orderBy:'marks'">

{{ subject.name + ', marks:' + subject.marks }}

</li>

</ul>

</td>

</tr>

</table>

</div>

<script>

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

mainApp.controller('studentController', function($scope) {

$scope.student = {

firstName: "Mahesh",

lastName: "Parashar",

fees:500,

subjects:[

{name:'Physics',marks:70},

{name:'Chemistry',marks:80},

{name:'Math',marks:65}

],

fullName: function() {

var studentObject;

studentObject = $scope.student;

return studentObject.firstName + " " + studentObject.lastName;

}

};

});

</script>

</body>

</html>

### When filters are executed

In templates, filters are only executed when their inputs have changed. This is more performant than executing a filter on each [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) as is the case with [expressions](https://docs.angularjs.org/guide/expression).

There are two exceptions to this rule:

1. In general, this applies only to filters that take [primitive values](https://developer.mozilla.org/docs/Glossary/Primitive) as inputs. Filters that receive [Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#Objects) as input are executed on each $digest, as it would be too costly to track if the inputs have changed.
2. Filters that are marked as $stateful are also executed on each $digest. See [Stateful filters](https://docs.angularjs.org/guide/filter" \l "stateful-filters) for more information. Note that no AngularJS core filters are $stateful.

## Using filters in controllers, services, and directives

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

For this, inject a dependency with the name <filterName>Filter into your controller/service/directive. E.g. a filter called number is injected by using the dependency numberFilter. The injected argument is a function that takes the value to format as first argument, and filter parameters starting with the second argument.

angular.module('FilterInControllerModule', []).

controller('FilterController', ['filterFilter', function FilterController(filterFilter) {

this.array = [

{name: 'Tobias'},

{name: 'Jeff'},

{name: 'Brian'},

{name: 'Igor'},

{name: 'James'},

{name: 'Brad'}

];

this.filteredArray = filterFilter(this.array, 'a');

}]);

# [What is stateful filtering in AngularJS?](http://stackoverflow.com/questions/25877704/what-is-stateful-filtering-in-angularjs)

"State" is referring to variables/properties/etc that are set throughout the application. These values have the potential to change at any given time. The docs are saying that the filter shouldn't depend on external "state". Anything the filter needs to know about should be passed in as an argument when filtering, and the filter should then have everything it needs to do the filtering and return the result Look over the demo in the docs and you'll see that in the "stateful" filter, the filter has a dependency on a service which it uses to do the filtering. That service value could change during a $digest cycle, so the $stateful property has to be set on the filter so that Angular will run the filter again to be sure that dependency hasn't changed state, this changing the filter's result.

So, all "state" should be in the arguments, like this:

<p>{{myData | multiplyBy:multiplier}}</p>

With a filter like:

.filter('multiplyBy', function() {

function filter(input, multiplier) {

return input \* multiplier;

}

return filter;

})

If the data or arguments change, the filter will run again.

The stateful version would be something like this (not recommended!):

<p>{{myData | myFilter}}</p>

And the filter gets it's needed information from external sources:

.filter('myFilter', ['someDependency', function(someDependency) {

function filter(input) {

// let's just say `someDependency = {multiplier: 3}`

return input \* someDependency.multiplier;

}

filter.$stateful = true;

return filter;

}])

In that sample filter, someDependency.multiplier should have been passed in as an argument to the filter (as in the first example), rather than being a dependency of the filter.

To further clarify the problem: If you called a function like this: foo(20) and get a result of 40, you should get the same result if you repeat the process. If you called foo(20) again and got 92, that would be rather confusing, right? Assuming foo isn't a function that is made to return random values, the only way it could return different numbers each time is if it performs differently based on a hidden state (something changing internally, rather than being passed in as an argument). The idea that the function would return the same each time given the same arguments is called being "idempotent".

**Note: $stateful seems to be new in Angular 1.3**

**HTML DOM**

Following directives can be used to bind application data to attributes of HTML DOM Elements.

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Name** | **Description** |
| 1 | ng-disabled | disables a given control. |
| 2 | ng-show | shows a given control. |
| 3 | ng-hide | hides a given control. |
| 4 | ng-click | represents a AngularJS click event. |

**Modules**

* You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.
* In JavaScript, placing functional code in the global namespace is rarely a good idea. It can cause collisions that are tough to debug and cost us precious development time
* AngularJS supports modular approach. Modules are used to separate logics say services, controllers, application etc. and keep the code clean. We define modules in separate js files and name them as per the module.js file. In this example we're going to create two modules.
* **Application Module** − used to initialize an application with controller(s).
  + In Angular, a module is the main way to define an AngularJS app. The module of an app is where we’ll contain all of our application code. An app can contain several modules, each one containing code that pertains to specific functionality.

angular.module('myApp', []);

* + name (string):The name property on the modules gives us the name of the module as a string.
  + requires (array of strings): The requires property contains a list of modules (as strings) that the injector loads before the module itself is loaded.
* **Controller Module** − used to define the controller.
* Using modules gives us a lot of advantages, such as:   
  • Keeping our global namespace clean

• Making tests easier to write and keeping them clean so as to more easily target isolated functionality

• Making it easy to share code between applications   
• Allowing our app to load different parts of the code in any order

## Why?

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

AngularJS apps don't have a main method. Instead modules declaratively specify how an application should be bootstrapped. There are several advantages to this approach:

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

## Recommended Setup

While the example above is simple, it will not scale to large applications. Instead we recommend that you break your application to multiple modules like this:

* A module for each feature
* A module for each reusable component (especially directives and filters)
* And an application level module which depends on the above modules and contains any initialization code.

<html>

<head>

<title>Angular JS Modules</title>

<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

<script src = "/angularjs/src/module/mainApp.js"></script>

<script src = "/angularjs/src/module/studentController.js"></script>

<style>

table, th , td {

border: 1px solid grey;

border-collapse: collapse;

padding: 5px;

}

table tr:nth-child(odd) {

background-color: #f2f2f2;

}

table tr:nth-child(even) {

background-color: #ffffff;

}

</style>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp" ng-controller = "studentController">

<table border = "0">

<tr>

<td>Enter first name:</td>

<td><input type = "text" ng-model = "student.firstName"></td>

</tr>

<tr>

<td>Enter last name: </td>

<td><input type = "text" ng-model = "student.lastName"></td>

</tr>

<tr>

<td>Name: </td>

<td>{{student.fullName()}}</td>

</tr>

<tr>

<td>Subject:</td>

<td>

<table>

<tr>

<th>Name</th>

<th>Marks</th>

</tr>

<tr ng-repeat = "subject in student.subjects">

<td>{{ subject.name }}</td>

<td>{{ subject.marks }}</td>

</tr>

</table>

</td>

</tr>

</table>

</div>

</body>

</html>

*mainApp.js*

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

*studentController.js*

mainApp.controller("studentController", function($scope) {

$scope.student = {

firstName: "Mahesh",

lastName: "Parashar",

fees:500,

subjects:[

{name:'Physics',marks:70},

{name:'Chemistry',marks:80},

{name:'Math',marks:65},

{name:'English',marks:75},

{name:'Hindi',marks:67}

],

fullName: function() {

var studentObject;

studentObject = $scope.student;

return studentObject.firstName + " " + studentObject.lastName;

}

};

});

**Forms:**

AngularJS enriches form filling and validation. We can use ng-click to handle AngularJS click on button and use $dirty and $invalid flags to do the validations in seamless way. Use no validate with a form declaration to disable any browser specific validation. Forms controls make heavy use of Angular events. Let's have a quick look on events first.

Validate data

Following can be used to track error.

* **$dirty** − states that value has been changed.
* **$invalid** − states that value entered is invalid.
* **$error** − states the exact error.
* **$pristine**: It will be TRUE, if the user has not interacted with the form yet
* **$dirty**: It will be TRUE, if the user has already interacted with the form.
* **$valid**: It will be TRUE, if all containing form and controls are valid
* **$invalid**: It will be TRUE, if at least one containing form and control is invalid.
* **$error**: Is an object hash, containing references to all invalid controls or forms, where:
  + - keys are validation tokens (error names)
    - values are arrays of controls or forms that are invalid with given error.

## Using CSS classes

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

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

There are some built in validation tokens, that can help in validating form:

* email
* max
* maxlength
* min
* minlength
* number
* pattern
* required
* url

<html>

<head>

<title>Angular JS Forms</title>

<script src = "angular.min.js"></script>

<style>

table, th , td {

border: 1px solid grey;

border-collapse: collapse;

padding: 5px;

}

table tr:nth-child(odd) {

background-color: #f2f2f2;

}

table tr:nth-child(even) {

background-color: #ffffff;

}

</style>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp" ng-controller = "studentController">

<form name = "studentForm" novalidate>

<table border = "0">

<tr>

<td>Enter first name:</td>

<td><input name = "firstname" type = "text" ng-model = "firstName" required>

<span style = "color:red" ng-show = "studentForm.firstname.$dirty && studentForm.firstname.$invalid">

<span ng-show = "studentForm.firstname.$error.required">First Name is required.</span>

</span>

</td>

</tr>

<tr>

<td>Enter last name: </td>

<td><input name = "lastname" type = "text" ng-model = "lastName" required>

<span style = "color:red" ng-show = "studentForm.lastname.$dirty && studentForm.lastname.$invalid">

<span ng-show = "studentForm.lastname.$error.required">Last Name is required.</span>

</span>

</td>

</tr>

<tr>

<td>Email: </td><td><input name = "email" type = "email" ng-model = "email" length = "100" required>

<span style = "color:red" ng-show = "studentForm.email.$dirty && studentForm.email.$invalid">

<span ng-show = "studentForm.email.$error.required">Email is required.</span>

<span ng-show = "studentForm.email.$error.email">Invalid email address.</span>

</span>

</td>

</tr>

<tr>

<td>

<button ng-click = "reset()">Reset</button>

</td>

<td>

<button ng-disabled = "studentForm.firstname.$dirty &&

studentForm.firstname.$invalid || studentForm.lastname.$dirty &&

studentForm.lastname.$invalid || studentForm.email.$dirty &&

studentForm.email.$invalid" ng-click="submit()">Submit</button>

</td>

</tr>

</table>

</form>

</div>

<script>

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

mainApp.controller('studentController', function($scope) {

$scope.reset = function(){

$scope.firstName = "Mahesh";

$scope.lastName = "Parashar";

$scope.email = "MaheshParashar@tutorialspoint.com";

}

$scope.reset();

});

</script>

</body>

</html>

## Custom model update triggers

By default, any change to the content will trigger a model update and form validation. You can override this behavior using the [ngModelOptions](https://docs.angularjs.org/api/ng/directive/ngModelOptions) directive to bind only to specified list of events. I.e. ng-model-options="{ updateOn: 'blur' }" will update and validate only after the control loses focus. You can set several events using a space delimited list. I.e. ng-model-options="{ updateOn: 'mousedown blur' }"

## Non-immediate (debounced) model updates

You can delay the model update/validation by using the debounce key with the [ngModelOptions](https://docs.angularjs.org/api/ng/directive/ngModelOptions) directive. This delay will also apply to parsers, validators and model flags like $dirty or $pristine

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

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

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

## Custom Validation

With a custom directive, you can add your own validation functions to the $validators object on the [ngModelController](https://docs.angularjs.org/api/ng/type/ngModel.NgModelController). To get a hold of the controller, you require it in the directive as shown in the example below.

An integer directive that validates whether the input is a valid integer. For example, 1.23 is an invalid value, since it contains a fraction. Note that we validate the viewValue (the string value of the control), and not the modelValue. This is because input[number] converts the viewValue to a number when running the $parsers.

var app = angular.module('form-example1', []);

var INTEGER\_REGEXP = /^-?\d+$/;

app.directive('integer', function() {

return {

require: 'ngModel',

link: function(scope, elm, attrs, ctrl) {

ctrl.$validators.integer = function(modelValue, viewValue) {

if (ctrl.$isEmpty(modelValue)) {

// consider empty models to be valid

return true;

}

if (INTEGER\_REGEXP.test(viewValue)) {

// it is valid

return true;

}

// it is invalid

return false;

};

}

};

});

<form name="form" class="css-form" novalidate>

<div>

<label>

Size (integer 0 - 10):

<input type="number" ng-model="size" name="size"

min="0" max="10" integer />{{size}}</label><br />

<span ng-show="form.size.$error.integer">The value is not a valid integer!</span>

<span ng-show="form.size.$error.min || form.size.$error.max">

The value must be in range 0 to 10!</span>

</div>

</form>

**Includes**

Using AngularJS, we can embedded HTML pages within a HTML page using ng-include directive.

<div ng-app = "" ng-controller = "studentController">

<div ng-include = "'main.htm'"></div>

<div ng-include = "'subjects.htm'"></div>

</div>

**Ajax**

AngularJS provides $http control which works as a service to read data from the server. The server makes a database call to get the desired records. AngularJS needs data in JSON format. Once the data is ready, $http can be used to get the data from server in the following manner –

function studentController($scope,$http) {

var url = "data.txt";

$http.get(url).success( function(response) {

$scope.students = response;

});

}

**Views**

AngularJS supports Single Page Application via multiple views on a single page. To do this AngularJS has provided ng-view and ng-template directives and $routeProvider services.

ng-view

ng-view tag simply creates a place holder where a corresponding view (html or ng-template view) can be placed based on the configuration.

ng-template

ng-template directive is used to create an html view using script tag. It contains "id" attribute which is used by $routeProvider to map a view with a controller.

<div ng-app = "mainApp">

...

<script type = "text/ng-template" id = "addStudent.htm">

<h2> Add Student </h2>

{{message}}

</script>

</div>

**$routeProvider**

$routeProvider is the key service which set the configuration of urls, maps them with the corresponding html page or ng-template, and attaches a controller with the same.

Following are the important points to be considered in above example.

* $routeProvider is defined as a function under config of mainApp module using key as '$routeProvider'.
* $routeProvider.when defines a url "/addStudent" which then is mapped to "addStudent.htm". addStudent.htm should be present in the same path as main html page.If htm page is not defined then ng-template to be used with id="addStudent.htm". We've used ng-template.
* "otherwise" is used to set the default view.
* "controller" is used to set the corresponding controller for the view.

Example SPA:

<html>

<head>

<title>Angular JS Views</title>

<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular-route.min.js"></script>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp">

<p><a href = "#addStudent">Add Student</a></p>

<p><a href = "#viewStudents">View Students</a></p>

<div ng-view></div>

<script type = "text/ng-template" id = "addStudent.htm">

<h2> Add Student </h2>

{{message}}

</script>

<script type = "text/ng-template" id = "viewStudents.htm">

<h2> View Students </h2>

{{message}}

</script>

</div>

<script>

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

mainApp.config(['$routeProvider', function($routeProvider) {

$routeProvider.

when('/addStudent', {

templateUrl: 'addStudent.htm',

controller: 'AddStudentController'

}).

when('/viewStudents', {

templateUrl: 'viewStudents.htm',

controller: 'ViewStudentsController'

}).

otherwise({

redirectTo: '/addStudent'

});

}]);

mainApp.controller('AddStudentController', function($scope) {

$scope.message = "This page will be used to display add student form";

});

mainApp.controller('ViewStudentsController', function($scope) {

$scope.message = "This page will be used to display all the students";

});

</script>

</body>

</html>

**Scopes**

* Scope is a special javascript object which plays the role of **joining controller with the views**.
* **Scope contains the model data**. In controllers, model data is accessed via $scope object.
* $scopes in AngularJS are arranged in a hierarchical structure that mimics the DOM and thus are nestable: We can reference properties on parent $scopes.
* Every part of an AngularJS application has a parent scope (as we’ve seen, at the ng-app level, this scope is called the $rootScope)
* There is one exception: A scope created inside of a directive is called the isolate scope.
* With the exception of isolate scopes, all scopes are created with prototypal inheritance, meaning that they have access to their parent scopes

*Scope Inheritance*

Scope are controllers specific. If we defines nested controllers then child controller will inherit the scope of its parent controller.

<html>

<head>

<title>Angular JS Forms</title>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp" ng-controller = "shapeController">

<p>{{message}} <br/> {{type}} </p>

<div ng-controller = "circleController">

<p>{{message}} <br/> {{type}} </p>

</div>

<div ng-controller = "squareController">

<p>{{message}} <br/> {{type}} </p>

</div>

</div>

<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

<script>

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

mainApp.controller("shapeController", function($scope) {

$scope.message = "In shape controller";

$scope.type = "Shape";

});

mainApp.controller("circleController", function($scope) {

$scope.message = "In circle controller";

});

mainApp.controller("squareController", function($scope) {

$scope.message = "In square controller";

$scope.type = "Square";

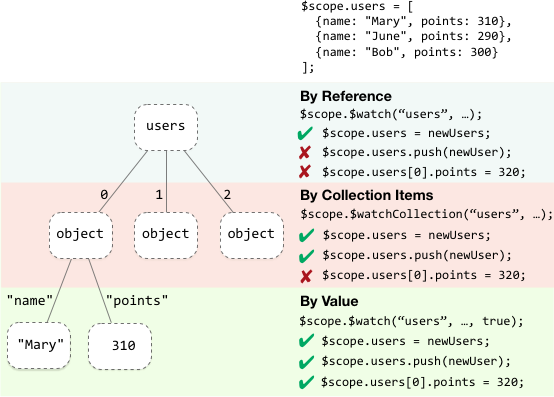
});

</script>

</body>

</html>

### Scope $watch Depths



* Watching by reference ([scope.$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope" \l "$watch)(watchExpression, listener)) detects a change when the whole value returned by the watch expression switches to a new value. If the value is an array or an object, changes inside it are not detected. This is the most efficient strategy.
* Watching collection contents([scope.$watchCollection](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watchCollection)(watchExpression, listener)) detects changes that occur inside an array or an object: When items are added, removed, or reordered. The detection is shallow - it does not reach into nested collections. Watching collection contents is more expensive than watching by reference, because copies of the collection contents need to be maintained. However, the strategy attempts to minimize the amount of copying required.
* Watching by value ([scope.$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope" \l "$watch) (watchExpression, listener, true)) detects any change in an arbitrarily nested data structure. It is the most powerful change detection strategy, but also the most expensive. A full traversal of the nested data structure is needed on each digest, and a full copy of it needs to be held in memory.

**$rootScope**

* When Angular starts to run and generate the view, it will create a binding from the root ng-app element to the $rootScope.
* This $rootScope is the eventual parent of all $scope objects.
* 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, in the same way that it’s not a good idea to dirty the JavaScript global scope.

# Dependency Injection

Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies.

The AngularJS injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

## Using Dependency Injection

Dependency Injection is pervasive throughout AngularJS. You can use it when defining components or when providing run and config blocks for a module.

* [Services](https://docs.angularjs.org/api/ng/type/angular.Module#service), [directives](https://docs.angularjs.org/api/ng/type/angular.Module#directive), [filters](https://docs.angularjs.org/api/ng/type/angular.Module#filter), and [animations](https://docs.angularjs.org/api/ng/type/angular.Module#animation) are defined by an injectable factory method or constructor function, and can be injected with "services", "values", and "constants" as dependencies.
* [Controllers](https://docs.angularjs.org/api/ng/service/$controller) are defined by a constructor function, which can be injected with any of the "service" and "value" as dependencies, but they can also be provided with "special dependencies". See [Controllers](https://docs.angularjs.org/guide/di#controllers) below for a list of these special dependencies.
* The [run](https://docs.angularjs.org/api/ng/type/angular.Module#run) method accepts a function, which can be injected with "services", "values" and, "constants" as dependencies. Note that you cannot inject "providers" into run blocks.
* The [config](https://docs.angularjs.org/api/ng/type/angular.Module#config) method accepts a function, which can be injected with "providers" and "constants" as dependencies. Note that you cannot inject "services" or "values" into configuration.
* The [provider](https://docs.angularjs.org/api/ng/type/angular.Module#provider) method can only be injected with other "providers". However, only those that have been **registered beforehand** can be injected. This is different from services, where the order of registration does not matter.

## Dependency Annotation

AngularJS invokes certain functions (like service factories and controllers) via the injector. You need to annotate these functions so that the injector knows what services to inject into the function. There are three ways of annotating your code with service name information:

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

### Inline Array Annotation

This is the preferred way to annotate application components. This is how the examples in the documentation are written.

For example:

someModule.controller('MyController', ['$scope', 'greeter', function($scope, greeter) {

// ...

}]);

Here we pass an array whose elements consist of a list of strings (the names of the dependencies) followed by the function itself.

When using this type of annotation, take care to keep the annotation array in sync with the parameters in the function declaration.

### $inject Property Annotation

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

var MyController = function($scope, greeter) {

// ...

}

MyController.$inject = ['$scope', 'greeter'];

someModule.controller('MyController', MyController);

In this scenario the ordering of the values in the $inject array must match the ordering of the parameters in MyController.

Just like with the array annotation, you'll need to take care to keep the $inject in sync with the parameters in the function declaration.

### Implicit Annotation

**Careful:** If you plan to [minify](http://en.wikipedia.org/wiki/Minification_(programming)) your code, your service names will get renamed and break your app.

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

someModule.controller('MyController', function($scope, greeter) {

// ...

});

Given a function, the injector can infer the names of the services to inject by examining the function declaration and extracting the parameter names. In the above example, $scope and greeter are two services which need to be injected into the function.

One advantage of this approach is that there's no array of names to keep in sync with the function parameters. You can also freely reorder dependencies.

However this method will not work with JavaScript minifiers/obfuscators because of how they rename parameters.

*If you decide to take this approach, you probably want to use ng-strict-di.*

*Because of these caveats, we recommend avoiding this style of annotation.*

## Using Strict Dependency Injection

You can add an ng-strict-di directive on the same element as ng-app to opt into strict DI mode:

<!doctype html>

<html ng-app="myApp" ng-strict-di>

<body>

I can add: {{ 1 + 2 }}.

<script src="angular.js"></script>

</body>

</html>

Strict mode throws an error whenever a service tries to use implicit annotations.

Consider this module, which includes a willBreak service that uses implicit DI:

angular.module('myApp', [])

.factory('willBreak', function($rootScope) {

// $rootScope is implicitly injected

})

.run(['willBreak', function(willBreak) {

// AngularJS will throw when this runs

}]);

# Templates

In AngularJS, templates are written with HTML that contains AngularJS-specific elements and attributes. AngularJS combines the template with information from the model and controller to render the dynamic view that a user sees in the browser.

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

* [Directive](https://docs.angularjs.org/guide/directive) — An attribute or element that augments an existing DOM element or represents a reusable DOM component.
* [Markup](https://docs.angularjs.org/api/ng/service/$interpolate) — The double curly brace notation {{ }} to bind expressions to elements is built-in AngularJS markup.
* [Filter](https://docs.angularjs.org/guide/filter) — Formats data for display.
* [Form controls](https://docs.angularjs.org/guide/forms) — Validates user input.

**Services**

* AngularJS supports the concepts of "Separation of Concerns" using services architecture. Services are javascript functions and are responsible to do a specific task only.
* This makes them an individual entity which is maintainable and testable.
* Controllers, filters can call them as on requirement basis. Services are normally injected using dependency injection mechanism of AngularJS.
* You can use services to organize and share code across your app.
* Like other core AngularJS identifiers, built-in services always start with $ (e.g. $http).

AngularJS services are:

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

There are two ways to create a service.

* factory
* service

<html>

<head>

<title>Angular JS Services</title>

<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp" ng-controller = "CalcController">

<p>Enter a number: <input type = "number" ng-model = "number" /></p>

<button ng-click = "square()">X<sup>2</sup></button>

<p>Result: {{result}}</p>

</div>

<script>

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

mainApp.factory('MathService', function() {

var factory = {};

factory.multiply = function(a, b) {

return a \* b

}

return factory;

});

mainApp.service('CalcService',function(MathService){

this.square = function(a) {

return MathService.multiply(a,a);

}

});

mainApp.controller('CalcController', function($scope, CalcService) {

$scope.square = function() {

$scope.result = CalcService.square($scope.number);

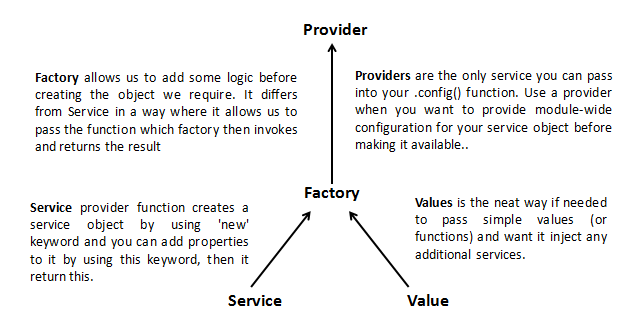
}

});

</script>

</body>

</html>



**Provider**:

A provider is used to create a configurable service object. It returns value by using $get() function.

### Creating service using provider method

1. **<script>**
2. ***//define a provider using provider() function***
3. **app.provider('configurableService', function () {**
4. **var name = '';**
5. **this.setName = function (newName) {**
6. **name = newName;**
7. **};**
8. **this.$get = function () {**
9. **return name;**
10. **};**
11. **});**
13. ***//configuring provider using config() function***
14. **app.config(function (configurableService) {**
15. **configurableService.setName('www.dotnet-tricks.com');**
16. **});**
17. **</script>**

**Dependency Injection**

Dependency Injection is a software design pattern in which components are given their dependencies instead of hard coding them within the component. This relieves a component from locating the dependency and makes dependencies configurable. This helps in making components reusable, maintainable and testable.

AngularJS provides a supreme Dependency Injection mechanism. It provides following core components which can be injected into each other as dependencies.

* value
* factory
* service
* provider
* constant

## What are Directives?

At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's **HTML compiler** ([$compile](https://docs.angularjs.org/api/ng/service/$compile)) to attach a specified behavior to that DOM element (e.g. via event listeners), or even to transform the DOM element and its children.

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

**Best Practice:** Comment directives were commonly used in places where the DOM API limits the ability to create directives that spanned multiple elements (e.g. inside <table> elements). AngularJS 1.2 introduces [ng-repeat-start and ng-repeat-end](https://docs.angularjs.org/api/ng/directive/ngRepeat) as a better solution to this problem. Developers are encouraged to use this over custom comment directives when possible.

**Best Practice:** In order to avoid collisions with some future standard, it's best to prefix your own directive names. For instance, if you created a <carousel> directive, it would be problematic if HTML7 introduced the same element. A two or three letter prefix (e.g. btfCarousel) works well. Similarly, do not prefix your own directives with ng or they might conflict with directives included in a future version of AngularJS.

**Custom Directives**

Custom directives are used in AngularJS to extend the functionality of HTML. Custom directives are defined using "directive" function. A custom directive simply replaces the element for which it is activated. AngularJS application during bootstrap finds the matching elements and do one time activity using its compile() method of the custom directive then process the element using link() method of the custom directive based on the scope of the directive. AngularJS provides support to create custom directives for following type of elements.

* **Element directives** − Directive activates when a matching element is encountered.
* **Attribute** − Directive activates when a matching attribute is encountered.
* **CSS** − Directive activates when a matching css style is encountered.
* **Comment** − Directive activates when a matching comment is encountered.

These restrictions can all be combined as needed:

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

**When should I use an attribute versus an element?**

Use an element when you are creating a component that is in control of the template. The common case for this is when you are creating a Domain-Specific Language for parts of your template. Use an attribute when you are decorating an existing element with new functionality.

**scope**

The scope property can be false, true, or an object:

* **false (default):** No scope will be created for the directive. The directive will use its parent's scope.
* **true:** A new child scope that prototypically inherits from its parent will be created for the directive's element. If multiple directives on the same element request a new scope, only one new scope is created.
* **{...} (an object hash):** A new "isolate" scope is created for the directive's template. The 'isolate' scope differs from normal scope in that it does not prototypically inherit from its parent scope. This is useful when creating reusable components, which should not accidentally read or modify data in the parent scope. Note that an isolate scope directive without a template or templateUrl will not apply the isolate scope to its children elements.

The 'isolate' scope object hash defines a set of local scope properties derived from attributes on the directive's element. These local properties are useful for aliasing values for templates. The keys in the object hash map to the name of the property on the isolate scope; the values define how the property is bound to the parent scope, via matching attributes on the directive's element:

* @ or @attr - bind a local scope property to the value of DOM attribute. The result is always a string since DOM attributes are strings. If no attr name is specified then the attribute name is assumed to be the same as the local name. Given <my-component my-attr="hello {{name}}"> and the isolate scope definition scope: { localName:'@myAttr' }, the directive's scope property localName will reflect the interpolated value of hello {{name}}. As the name attribute changes so will the localName property on the directive's scope. The name is read from the parent scope (not the directive's scope).
* = or =attr - set up a bidirectional binding between a local scope property and an expression passed via the attribute attr. The expression is evaluated in the context of the parent scope. If no attr name is specified then the attribute name is assumed to be the same as the local name. Given <my-component my-attr="parentModel"> and the isolate scope definition scope: { localModel: '=myAttr' }, the property localModel on the directive's scope will reflect the value of parentModel on the parent scope. Changes to parentModel will be reflected in localModel and vice versa. Optional attributes should be marked as such with a question mark: =? or =?attr. If the binding expression is non-assignable, or if the attribute isn't optional and doesn't exist, an exception ([$compile:nonassign](https://docs.angularjs.org/error/$compile/nonassign)) will be thrown upon discovering changes to the local value, since it will be impossible to sync them back to the parent scope. By default, the [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) method is used for tracking changes, and the equality check is based on object identity. However, if an object literal or an array literal is passed as the binding expression, the equality check is done by value (using the [angular.equals](https://docs.angularjs.org/api/ng/function/angular.equals) function). It's also possible to watch the evaluated value shallowly with [$watchCollection](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watchCollection): use =\* or =\*attr (=\*? or =\*?attr if the attribute is optional).
* < or <attr - set up a one-way (one-directional) binding between a local scope property and an expression passed via the attribute attr. The expression is evaluated in the context of the parent scope. If no attr name is specified then the attribute name is assumed to be the same as the local name. You can also make the binding optional by adding ?: <? or <?attr.

For example, given <my-component my-attr="parentModel"> and directive definition of scope: { localModel:'<myAttr' }, then the isolated scope property localModel will reflect the value of parentModel on the parent scope. Any changes to parentModel will be reflected in localModel, but changes in localModel will not reflect in parentModel. There are however two caveats:

* 1. one-way binding does not copy the value from the parent to the isolate scope, it simply sets the same value. That means if your bound value is an object, changes to its properties in the isolated scope will be reflected in the parent scope (because both reference the same object).
  2. one-way binding watches changes to the **identity** of the parent value. That means the [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) on the parent value only fires if the reference to the value has changed. In most cases, this should not be of concern, but can be important to know if you one-way bind to an object, and then replace that object in the isolated scope. If you now change a property of the object in your parent scope, the change will not be propagated to the isolated scope, because the identity of the object on the parent scope has not changed. Instead you must assign a new object.

One-way binding is useful if you do not plan to propagate changes to your isolated scope bindings back to the parent. However, it does not make this completely impossible.

* & or &attr - provides a way to execute an expression in the context of the parent scope. If no attr name is specified then the attribute name is assumed to be the same as the local name. Given <my-component my-attr="count = count + value"> and the isolate scope definition scope: { localFn:'&myAttr' }, the isolate scope property localFn will point to a function wrapper for the count = count + value expression. Often it's desirable to pass data from the isolated scope via an expression to the parent scope. This can be done by passing a map of local variable names and values into the expression wrapper fn. For example, if the expression is increment(amount) then we can specify the amount value by calling the localFn as localFn({amount: 22}).

How directives are compiled

It's important to note that Angular operates on DOM nodes rather than strings. Usually, you don't notice this restriction because when a page loads, the web browser parses HTML into the DOM automatically.

HTML compilation happens in three phases:

1. [$compile](https://code.angularjs.org/1.5.7/docs/api/ng/service/$compile) traverses the DOM and matches directives.

If the compiler finds that an element matches a directive, then the directive is added to the list of directives that match the DOM element. A single element may match multiple directives.

1. Once all directives matching a DOM element have been identified, the compiler sorts the directives by their priority.

Each directive's compile functions are executed. Each compile function has a chance to modify the DOM. Each compile function returns a link function. These functions are composed into a "combined" link function, which invokes each directive's returned link function.

1. $compile links the template with the scope by calling the combined linking function from the previous step. This in turn will call the linking function of the individual directives, registering listeners on the elements and setting up [$watchs](https://code.angularjs.org/1.5.7/docs/api/ng/type/$rootScope.Scope#$watch) with the [scope](https://code.angularjs.org/1.5.7/docs/api/ng/type/$rootScope.Scope) as each directive is configured to do.

The result of this is a live binding between the scope and the DOM. So at this point, a change in a model on the compiled scope will be reflected in the DOM.

Below is the corresponding code using the $compile service. This should help give you an idea of what Angular does internally.

var $compile = ...; // injected into your code

var scope = ...;

var parent = ...; // DOM element where the compiled template can be appended

var html = '<div ng-bind="exp"></div>';

// Step 1: parse HTML into DOM element

var template = angular.element(html);

// Step 2: compile the template

var linkFn = $compile(template);

// Step 3: link the compiled template with the scope.

var element = linkFn(scope);

// Step 4: Append to DOM (optional)

parent.appendChild(element);

Comparing **compile** vs **link** vs **controller** :

* Every directive is **compiled** only once and **link** function is retained for re-use. Therefore, if there's something applicable to all instances of a directive should be performed inside directive's compile function.
* Now, after compilation we have link function which is executed while attaching the **template** to the **DOM**. So, therefore we perform everything that is specific to every instance of the directive. For eg: **attaching events**, **mutating the template based on scope**, etc.
* Finally, the **controller** is meant to be available to be live and reactive while the directive works on the DOM (after getting attached). Therefore:

(1) After setting up the view[**V**] (i.e. template) with link. $scope is our [**M**] and $controller is our [**C**] in **M V C**

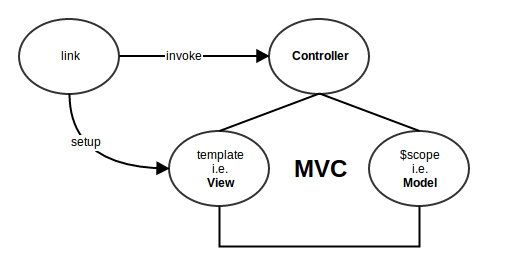
(2) Take advantage the **2-way** binding with **$scope** by setting up watches.

(3) $scope watches are expected to be added in the controller since this is what is watching the template during run-time.

(4) Finally, controller is also used to be able to communicate among related directives. (Like myTabs example in <https://docs.angularjs.org/guide/directive>)

(5) It's true that we could have done all of this in the link function as well, but it's about **separation of concerns**.

Therefore, finally we have the following which fits all the pieces perfectly:



<html>

<head>

<title>Angular JS Custom Directives</title>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp" ng-controller = "StudentController">

<student name = "Mahesh"></student><br/>

<student name = "Piyush"></student>

</div>

<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

<script>

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

mainApp.directive('student', function() {

var directive = {};

directive.restrict = 'E';

directive.template = "Student: <b>{{student.name}}</b> , Roll No: <b>{{student.rollno}}</b>";

directive.scope = {

student : "=name"

}

directive.compile = function(element, attributes) {

element.css("border", "1px solid #cccccc");

var linkFunction = function($scope, element, attributes) {

element.html("Student: <b>"+$scope.student.name +"</b> , Roll No: <b>"+$scope.student.rollno+"</b><br/>");

element.css("background-color", "#ff00ff");

}

return linkFunction;

}

return directive;

});

mainApp.controller('StudentController', function($scope) {

$scope.Mahesh = {};

$scope.Mahesh.name = "Mahesh Parashar";

$scope.Mahesh.rollno = 1;

$scope.Piyush = {};

$scope.Piyush.name = "Piyush Parashar";

$scope.Piyush.rollno = 2;

});

</script>

</body>

</html>

# [What's the meaning of require: 'ngModel'?](https://stackoverflow.com/questions/20930592/whats-the-meaning-of-require-ngmodel)

The require instruction gives you the controller for the directive you name as the fourth argument to your link function. (You can use ^ to look for the controller on a parent element; ? makes it optional.) So require: 'ngModel' gives you the controller for the ngModel directive, [which is an ngModelController](http://docs.angularjs.org/api/ng.directive%3angModel.NgModelController).

Directive controllers can be written to provide APIs that other directives can use; with ngModelController, you get access to special functionality that's built into ngModel, including getting and setting the value. Consider the following example:

<input color-picker ng-model="project.color">

app.directive('colorPicker', function() {

return {

require: 'ngModel',

link: function(scope, element, attrs, ngModel) {

element.colorPicker({

// initialize the color to the color on the scope

pickerDefault: scope.color,

// update the ngModel whenever we pick a new color

onColorChange: function(id, newValue) {

scope.$apply(function() {

ngModel.$setViewValue(newValue);

});

}

});

// update the color picker whenever the value on the scope changes

ngModel.$render = function() {

element.val(ngModel.$modelValue);

element.change();

};

}

}

});

This directive uses the ngModel controller to get and set the value of the color from the colorpicker. See this JSFiddle example: <http://jsfiddle.net/BinaryMuse/AnMhx/>

If you're using require: 'ngModel', you probably shouldn't also be using ngModel: '=' in your isolate scope; the ngModelController gives you all the access you need to change the value.

The bottom example on [the AngularJS homepage](http://angularjs.org/) also uses this functionality (except using a custom controller, not ngModel).

# Directive to Directive communication with "require"

As we all know, the DDO (Directive Definition Object) returned by the Angular 1.x directive callback function has a require property, that sort of establishes dependency of one directive on others. This is a convenient way of building components that contain multiple directives where one directive may depend on the functionality of others. *In practice, the require property simply tells Angular compiler where a particular directive controller should be search for. More details:* <https://demisx.github.io/angularjs/directives/2014/11/25/angular-directive-require-property-options.html>

**Bootstrapping in AngularJS:**

**Bootstrapping**in AngularJS is nothing but just the initialization of Angular app.

When you add ng-app directive to the root of your application, typically on the <html> tag or <body> tag if you want angular to auto-bootstrap your application.

|  |  |
| --- | --- |
|  | <html ng-app="myApp"> |

When angularJS finds ng-app directive, it loads the module associated with it and then compile the DOM.

Another way to bootstrapping is manually initializing using script. Manual initialization provides you more control on how and when to initialize angular App. It is useful when you want to perform any other operation before Angular wakes up and compile the page. Below is the script which allows to manually bootstrap angularJS.

|  |  |
| --- | --- |
|  | <script> |
|  | angular.element(document).ready(function() { | |

|  |  |  |
| --- | --- | --- |
|  | angular.bootstrap(document, ['myApp']); | |
|  | }); |

|  |  |
| --- | --- |
|  | </script> |

* Remember angular.bootstrap function will not create modules on the go. So you need to have your modules define, before you manually bootstrap. So below is the correct approach. First define the module and then bootstrap.

|  |  |
| --- | --- |
| 01 | <script> |
| 02 | angular.module('myApp', []) | |

|  |  |  |
| --- | --- | --- |
| 03 | .controller('MyController', ['$scope', function ($scope) { | |
| 04 | $scope.message= 'Hello World'; |

|  |  |  |
| --- | --- | --- |
| 05 | }]); | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | angular.element(document).ready(function() { | |
| 08 | angular.bootstrap(document, ['myApp']); |

|  |  |
| --- | --- |
| 09 | }); |
| 10 | </script> | |

* You cannot add controllers, services, directives, etc. after an application bootstraps.

**$apply and $watch**

**Digest cycle and $scope**

* First and foremost, AngularJS defines a concept of a so-called **digest cycle**. This cycle can be considered as a loop, during which AngularJS checks if there are any changes to all the variables **watched** by all the $scopes.
* So if you have $scope.myVar defined in your controller and this variable was **marked for being watched**, then you are implicitly telling AngularJS to monitor the changes on myVar in each iteration of the loop.
* A natural follow-up question would be: Is everything attached to $scope being watched? Fortunately, **no.** If you would watch for changes to every object in your $scope, then quickly a digest loop would take ages to evaluate and you would quickly run into performance issues.
* That is why the AngularJS team gave us two ways of declaring some $scope variable as being watched (read below).

$watch helps to listen for $scope changes

There are two ways of declaring a $scope variable as being watched.

1. **By using it in your template** via the expression <span>{{myVar}}</span>
2. **By adding it manually via the $watch service**

Ad 1) This is the most common scenario and I'm sure you've seen it before, but you didn't know that this has created a watch in the background. Yes, it had! Using AngularJS directives (such as ng-repeat) can also create implicit watches.

Ad 2) This is how you create your own **watches**. $watch service helps you to run some code when some value attached to the $scope has changed. It is rarely used, but sometimes is helpful. For instance, if you want to run some code each time 'myVar' changes, you could do the following:

function MyController($scope) {

$scope.myVar = 1;

$scope.$watch('myVar', function() {

alert('hey, myVar has changed!');

});

$scope.buttonClicked = function() {

$scope.myVar = 2; // This will trigger $watch expression to kick in

};

}

**$apply enables to integrate changes with the digest cycle**

You can think of the **$apply function as of an integration mechanism**. You see, each time you change some **watched variable attached to the $scope** object directly, AngularJS will know that the change has happened. This is because AngularJS already knew to monitor those changes. So if it happens in code managed by the framework, the digest cycle will carry on.

However, sometimes you want to **change some value outside of the AngularJS world** and see the changes propagate normally. Consider this - you have a $scope.myVar value which will be modified within a jQuery's $.ajax() handler. This will happen at some point in future. AngularJS can't wait for this to happen, since it hasn't been instructed to wait on jQuery.

To tackle this, $apply has been introduced. It lets you to start the digestion cycle explicitly. However, you should only use this to migrate some data to AngularJS (integration with other frameworks), but never use this method combined with regular AngularJS code, as AngularJS will throw an error then.

How is all of this related to the DOM?

Well, you should really follow the tutorial again, now that you know all this. The digest cycle will make sure that the UI and the JavaScript code stays synchronized, by evaluating every watcher attached to the all $scopes as long as nothing changes. If no more changes happen in the digest loop, then it's considered to be finished.

You can attach objects to the $scope object either explicitly in the Controller, or by declaring them in {{expression}} form directly in the view.

**run() function in angular?**

Here's the calling order:

1. app.config()
2. app.run()
3. directive's compile functions (if they are found in the dom)
4. app.controller()
5. directive's link functions (again, if found)

Here's a [simple demo](http://jsfiddle.net/ysq3m/) where you can watch each one executing (and experiment if you'd like).

From [Angular's module docs](https://github.com/angular/angular.js/blob/ce669edfa14dc7eb7c389d2f82c9c98399a9009b/docs/content/guide/module.ngdoc):

**Run blocks** –

* Unlike the configuration blocks, run blocks are executed after the injector is created and are the first methods that are executed in any Angular app.
* Run blocks are the closest thing in Angular to the main method. The run block is code that is typically hard to unit test and is related to the general app.
* Typically, these run blocks are places where we’ll set up event listeners that should happen at the global scale of the app

Let’s say that we want to run a function that validates that we have an authenticated user every time that we change our route. The only logical place to set this functionality is in the run method:

*angular.module('myApp', []).run(function($rootScope, AuthService) {*

*$rootScope.$on('$routeChangeStart',*

*function(evt, next, current) {*

*// If the user is NOT logged in*

*if (!AuthService.userLoggedIn()) {*

*if (next.templateUrl === "login.html") {*

*// Already heading to the login route so no need to redirect*

*} else {*

*$location.path('/login');*

*}*

*}*

*});*

*});*

One situation where run blocks are used is during [authentications](http://arthur.gonigberg.com/2013/06/29/angularjs-role-based-auth).

Only rootScope can be injected in run function.

rootScope can be injected in run() and services. For other it is already available if you inject it, it will cause ambiguity.

List at least three ways to communicate between modules of your application using core AngularJS functionality.

* Using services
* By assigning models on $rootScope
* Directly between controllers, using $parent, $$childHead, $$nextSibling, etc.
* Directly between controllers, using ControllerAs, or other forms of inheritance

MVVM:

Model:

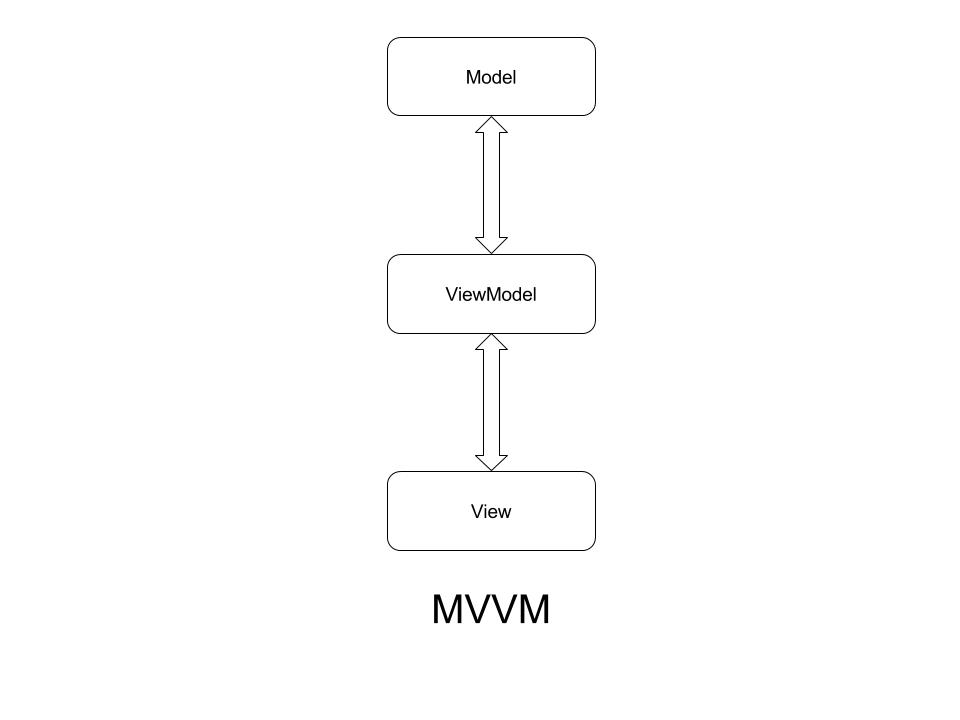
Same as MVC

View:

Same as MVC

## View Model:

* The view model acts as an intermediary between the view and the model, and is responsible for handling the view logic.
* The view model then provides data from the model in a form that the view can easily use.
* The view model retrieves data from the model and then makes the data available to the view, and may reformat the data in some way that makes it simpler for the view to handle.
* For example, when a user clicks a button in the UI, that action can trigger a command in the view model. The view model may also be responsible for defining logical state changes that affect some aspect of the display in the view, such as an indication that some operation is pending.



**Example of css Directive:**

ng-show, ng-hide, ng-disabled

Promise in angular JS?

## Promises

Promises are objects that help make working with async code feel like we’re writing synchronous code. Angular uses promises extensively, so it is important to get familiar with how to use them.

We use primarily only three methods when we use promises:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | promise.then(function(data) {    // Called when no errors have occurred with data  })  .catch(function(err) {    // Called when an error has occurred  })  .finally(function(data) {    // Called always, regardless of the output result  }) |

When we have a promise object, we can depend upon the .then() method to get called when we have a non-failure response, the catch() method to get called when there is an error, and the finally() method to get called regardless of the result of the function.

The $http object returns a promise when it’s completed the XHR request. Now, to interact with our request, we’ll simply use the .then() function to load the data on our $scope:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var promise = $http({    method: 'GET',    url: '/v1/api',    params: {      api\_key: 'abc'    }  });  promise.then(function(obj) {    // obj is the raw request object generated by Angular    // and contains status codes, the raw data, headers,    // and the config function used to make the request    $scope.data = obj.data;  }); |

## What is a promise?

A promise in the Javascript and AngularJS world is an assurance that we will get a result from an action at some point in the future, let’s see the two possible results of a promise:

* A promise is said to be **fulfilled** when we get a result from that action (meaning that we get a response, regardless of whether the response is good or bad)
* A promise is said to be **rejected** when we don’t get a response(for instance if we were retrieving some data from an API and for some reason we never got a response because the API endpoint was down etc.)

## Why do we need promises?

We need promises because we need to make decisions based on the possible results of our call (or the possibility that we don’t get a response from that call at all) probably an example will better help describe this:

Our program contacts an external API to get the list of clients  
while the response is received the program works on something else  
Once the response is received (if received) the program displays the client info on the screen  
If the response was not received (the API was down) then we display a message to the end user.  
Here is a really good example of what promises are and it’s explained as cartoon

Using Angular’s $q service to deal with promises

Angular JS provides a service called $q which allows you to work with asynchronous functions and user their return values when the execution has been completed, and what its really cool about it is that it will let you write your custom promises as well (so you can resolve or reject a promise when appropriate).

Let’s have a look at a simple example

|  |  |
| --- | --- |
|  | var deferred = $q.defer();  // deferred contains the promise to be returned  // to resolve (fulfill) a promise use .resolve  deferred.resolve(data);  // to reject a promise use .reject  deferred.reject(error); |

Now let’s have a look at how this would be implemented inside an AngularJS service:

|  |  |
| --- | --- |
|  | app.service("githubService", function ($http, $q) {        var deferred = $q.defer();        this.getAccount = function () {          return $http.get('https://api.github.com/users/haroldrv')              .then(function (response) {                  // promise is fulfilled                  deferred.resolve(response.data);                  // promise is returned                  return deferred.promise;              }, function (response) {                  // the following line rejects the promise                  deferred.reject(response);                  // promise is returned                  return deferred.promise;              })          ;      };  }); |

Finally, the AngularJS controller will use the service and either display the results on the page (if the promise was fulfilled and the data received)  or will display a message indicating that there was an error when attempting to retrieve the data from github

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | app.controller("promiseController", function ($scope, $q, githubService) {        githubService.getAccount()          .then(              function (result) {                  // promise was fullfilled (regardless of outcome)                  // checks for information will be peformed here                  $scope.account = result;              },              function (error) {                  // handle errors here                  console.log(error.statusText);              }          );  }); |

A plunker with a copy of the code can be found [here](http://plnkr.co/edit/kACAcbCUIGSLRHV0qojK?p=preview)

Also read : <http://www.ng-newsletter.com/posts/how-to-learn-angular.html>

Resolve in AngularJS

When a new route is requested in an AngularJS application, it is often necessary to retrieve data from the server to fill the template. If the server request takes too long to complete, it is therefore possible that an incomplete view is presented to the user.

To avoid this problem it is possible to delay the appearance of the new view until all the pending XHR requests have completed.

In a regular AngularJS web app, the routes are configured via methods in the $routeProvider object, as below:

$routeProvider.when('/library', {

templateUrl: 'partials/library.html',

controller: 'LibraryCtrl',

resolve: {

books: function(srvLibrary) {

return srvLibrary.getBooks();

},

movies: function(srvLibrary) {

return srvLibrary.getMovies();

}

}

});

[The resolve property is] an optional map of dependencies which should be injected into the controller. If any of these dependencies are promises, they will be resolved and converted to a value before the controller is instantiated and the $routeChangeSuccess event is fired.

Finally, to be able to use the resolved promises data in the controller it is necessary to inject these data into the controller, and attach them to the $scope variable.

angular.module('myApp.controllers', [])

.controller('LibraryCtrl', ['$scope', 'books', 'movies',

function($scope, books, movies) {

$scope.books = books.data;

$scope.movies = movies.data;

}]);

This is all that is required to delay the loading of a new route until the promises are resolved. As a result, however, a user could be surprised by the fact that nothing happens after they click a link. It is therefore a good idea to implement an immediate visual feedback

var app = angular.module('myApp',

['myApp.services', 'myApp.controllers']);

app.run(['$rootScope', function($root) {

$root.$on('$routeChangeStart', function(e, curr, prev) {

if (curr.$$route && curr.$$route.resolve) {

// Show a loading message until promises aren't resolved

$root.loadingView = true;

}

});

$root.$on('$routeChangeSuccess', function(e, curr, prev) {

// Hide loading message

$root.loadingView = false;

});

}]);

The following is one possible way to implement a loading message while waiting for the new route to appear.

<div class="modal" ng-show="loadingView">

<!-- loadingView is a variable defined in the $rootScope -->

<!-- The loading animation is inspired by

http://codepen.io/joni/details/FiKsd -->

<ul id="loading">

<li ng-repeat="i in [0,1,2,3,4,5,6,7,8,9]"></li>

</ul>

</div>

Emit and broadcast

Using $scope.$emit will fire an event up the $scope. Using $scope.$broadcast will fire an event down the $scope. Using $scope.$on is how we listen for these events. A quick example:

// firing an event upwards

$scope.$emit('myCustomEvent', 'Data to send');

// firing an event downwards

$scope.$broadcast('myCustomEvent', {

someProp: 'Sending you an Object!' // send whatever you want

});

// listen for the event in the relevant $scope

$scope.$on('myCustomEvent', function (event, data) {

console.log(data); // 'Data to send'

});

The key thing to remember when using $scope to fire your events, is that they will communicate only with immediate parent or child scopes only! Scopes aren’t always child and parent. We might have sibling scopes. Using $scope to fire an event will miss out sibling scopes, and just carry on up! They do not go sideways!

The simplest way to emulate parent and child scopes are to use Controllers. Each Controller creates new $scope, which Angular neatly outputs an ng-scope class on newly scoped elements for us:

<div ng-controller="ParentCtrl as parent" class="ng-scope">

{{ parent.data }}

<div ng-controller="SiblingOneCtrl as sib1" class="ng-scope">

{{ sib1.data }}

</div>

</div>

We could fire an event down from ParentCtrl to SiblingOneCtrl using $broadcast:

app.controller('ParentCtrl',

function ParentCtrl ($scope) {

$scope.$broadcast('parent', 'Some data'); // going down!

});

app.controller('SiblingOneCtrl',

function SiblingOneCtrl ($scope) {

$scope.$on('parent', function (event, data) {

console.log(data); // 'Some data'

});

});

If we wanted to communicate upwards, from SiblingOneCtrl to ParentCtrl, you guessed it, we can use $emit.

app.controller('ParentCtrl',

function ParentCtrl ($scope) {

$scope.$on('child', function (event, data) {

console.log(data); // 'Some data'

});

});

app.controller('SiblingOneCtrl',

function SiblingOneCtrl ($scope) {

$scope.$emit('child', 'Some data'); // going up!

});

To demonstrate how $scope works when firing the events, here’s a simple hierarchy:

<div ng-controller="ParentCtrl as parent" class="ng-scope">

<div ng-controller="SiblingOneCtrl as sib1" class="ng-scope"></div>

<div ng-controller="SiblingTwoCtrl as sib2" class="ng-scope"></div>

</div>

If SiblingTwoCtrl fired $scope.$broadcast, then SiblingOneCtrl would never know it happened. This can be an annoyance, but a (slightly hacky-feely) remedy can be done:

$scope.$parent.$broadcast('myevent', 'Some data');

What this does is jump up to ParentCtrl and then fire the $broadcast from there.

## $apply and $digest Explored

AngularJS offers an incredibly awesome feature known as two way data binding which greatly simplifies our lives. Data binding means that when you change something in the view, the scope model automagically updates. Similarly, whenever the scope model changes, the view updates itself with the new value. How does does AngularJS do that? When you write an expression ({{aModel}}), behind the scenes Angular sets up a watcher on the scope model, which in turn updates the view whenever the model changes. This watcher is just like any watcher you set up in AngularJS:

$scope.$watch('aModel', function(newValue, oldValue) {

//update the DOM with newValue

});

The second argument passed to $watch() is known as a listener function, and is called whenever the value of aModel changes. It is easy for us to grasp that when the value of aModel changes this listener is called, updating the expression in HTML. But, there is still one big question! How does Angular figure out when to call this listener function? In other words, how does AngularJS know when aModel changes so it can call the corresponding listener? Does it run a function periodically to check whether the value of the scope model has changed? Well, this is where the $digest cycle steps in.

It’s the $digest cycle where the watchers are fired. When a watcher is fired, AngularJS evaluates the scope model, and if it has changed then the corresponding listener function is called. So, our next question is when and how this $digest cycle starts.

The $digest cycle starts as a result of a call to $scope.$digest(). Assume that you change a scope model in a handler function through the ng-click directive. In that case AngularJS automatically triggers a $digest cycle by calling $digest(). When the $digest cycle starts, it fires each of the watchers. These watchers check if the current value of the scope model is different from last calculated value. If yes, then the corresponding listener function executes. As a result if you have any expressions in the view they will be updated. In addition to ng-click, there are several other built-in directives/services that let you change models (e.g. ng-model, $timeout, etc) and automatically trigger a $digest cycle.

So far, so good! But, there is a small gotcha. In the above cases, 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.

Now, let’s assume you attach an ng-click directive to a button and pass a function name to it. When the button is clicked, AngularJS wraps the function call within $scope.$apply(). So, your function executes as usual, change models (if any), and a $digest cycle starts to ensure your changes are reflected in the view.

**Note**: $scope.$apply() automatically calls $rootScope.$digest(). 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 when called. We will see why the former one is the preferred approach shortly.

## When Do You Call $apply() Manually?

If AngularJS usually wraps our code in $apply() and starts a $digest cycle, then when do you need to do call $apply() manually? Actually, AngularJS makes one thing pretty clear. It will account for only those model changes which are done inside AngularJS’ context (i.e. the code that changes models is wrapped inside $apply()). Angular’s built-in directives already do this so that any model changes you make are reflected in the view. However, if you change any model outside of the Angular context, then you need to inform Angular of the changes by calling $apply() manually. It’s like telling Angular that you are changing some models and it should fire the watchers so that your changes propagate properly.

For example, if you use JavaScript’s setTimeout() function to update a scope model, Angular has no way of knowing what you might change. In this case it’s your responsibility to call $apply()manually, which triggers a $digest cycle. Similarly, if you have a directive that sets up a DOM event listener and changes some models inside the handler function, you need to call $apply() to ensure the changes take effect.

***Use below URL for Best practices in AngularJS(MUST READ):***

<http://bguiz.github.io/js-standards/angularjs/single-responsibility/>

**What are component in Angularjs?**

In Angular, a Component is a special kind of directive that uses a simpler configuration which is suitable for a component-based application structure.

Advantages of Components:

* simpler configuration than plain directives
* promote sane defaults and best practices
* optimized for component-based architecture
* writing component directives will make it easier to upgrade to Angular 2

When not to use Components:

* for directives that rely on DOM manipulation, adding event listeners etc, because the compile and link functions are unavailable
* when you need advanced directive definition options like priority, terminal, multi-element
* when you want a directive that is triggered by an attribute or CSS class, rather than an element

**Diffrence between component and directive?**

|  | **Directive** | **Component** |
| --- | --- | --- |
| bindings | No | Yes (binds to controller) |
| bindToController | Yes (default: false) | No (use bindings instead) |
| compile function | Yes | No |
| controller | Yes | Yes (default function() {}) |
| controllerAs | Yes (default: false) | Yes (default: $ctrl) |
| link functions | Yes | No |
| multiElement | Yes | No |
| priority | Yes | No |
| require | Yes | Yes |
| restrict | Yes | No (restricted to elements only) |
| scope | Yes (default: false) | No (scope is always isolate) |
| template | Yes | Yes, injectable |
| templateNamespace | Yes | No |
| templateUrl | Yes | Yes, injectable |
| terminal | Yes | No |
| transclude | Yes (default: false) | Yes (default: false) |

When to use factory vs service?

## Explanation

You got different things here:

**First:**

* If you use a service you will get **the instance of a function** ("this" keyword).
* If you use a factory you will get **the value that is returned by invoking the function reference**(the return statement in factory).

ref: [angular.service vs angular.factory](http://stackoverflow.com/questions/14324451/angular-service-vs-angular-factory)

**Second:**

Keep in mind all providers in AngularJS (value, constant, services, factories) are singletons!

**Third:**

Using one or the other (service or factory) is about code style. But, the **common way** in AngularJS is to use **factory**.

Why ?

**Because** "The factory method is the most common way of getting objects into AngularJS dependency injection system. It is very flexible and can contain sophisticated creation logic. Since factories are regular functions, we can also take advantage of a new lexical scope to simulate "private" variables. This is very useful as we can hide implementation details of a given service."

(ref: [http://www.amazon.com/Mastering-Web-Application-Development-AngularJS/dp/1782161821](http://rads.stackoverflow.com/amzn/click/1782161821)).

## Usage

Service : Could be useful for sharing utility functions that are useful to invoke by simply appending () to the injected function reference. Could also be run with injectedArg.call(this) or similar.

Factory : Could be useful for returning a ‘class’ function that can then be new`ed to create instances.

So, **use a factory when you have complex logic** in your service and **you don't want expose this complexity**.

In other cases **if you want to return an instance of a service just use service**.

But you'll see with time that you'll use factory in 80% of cases I think.

For more details: <http://blog.manishchhabra.com/2013/09/angularjs-service-vs-factory-with-example/>

**UPDATE :**

Excellent post here : <http://iffycan.blogspot.com.ar/2013/05/angular-service-or-factory.html>

"If you want your function **to be called like a normal function**, use **factory**. If you want your function to be instantiated with the new operator, use service. If you don't know the difference, use factory."

**UPDATE :**

AngularJS team does his work and give an explanation: <http://docs.angularjs.org/guide/providers>

And from this page :

"Factory and Service are the most commonly used recipes. The only difference between them is that Service recipe works better for objects of custom type, while Factory can produce JavaScript primitives and functions."

ng-strict-di

You can add an ng-strict-di directive on the same element as ng-app to opt into strict DI mode:

<!doctype html>

<html ng-app="myApp" ng-strict-di>

<body>

I can add: {{ 1 + 2 }}.

<script src="angular.js"></script>

</body>

</html>

Strict mode throws an error whenever a service tries to use implicit annotations.

Consider this module, which includes a willBreak service that uses implicit DI:

angular.module('myApp', [])

.factory('willBreak', function($rootScope) {

// $rootScope is implicitly injected

})

.run(['willBreak', function(willBreak) {

// AngularJS will throw when this runs

}]);

When the willBreak service is instantiated, AngularJS will throw an error because of strict mode. This is useful when using a tool like [ng-annotate](https://github.com/olov/ng-annotate) to ensure that all of your application components have annotations.

If you're using manual bootstrapping, you can also use strict DI by providing strictDi: true in the optional config argument:

angular.bootstrap(document, ['myApp'], {

strictDi: true

});

## Setting HTTP Headers

The $http service will automatically add certain HTTP headers to all requests. These defaults can be fully configured by accessing the $httpProvider.defaults.headers configuration object, which currently contains this default configuration:

* $httpProvider.defaults.headers.common (headers that are common for all requests):
  + Accept: application/json, text/plain, \*﻿/﻿\*
* $httpProvider.defaults.headers.post: (header defaults for POST requests)
  + Content-Type: application/json
* $httpProvider.defaults.headers.put (header defaults for PUT requests)
  + Content-Type: application/json

### Default Transformations

The $httpProvider provider and $http service expose defaults.transformRequest and defaults.transformResponse properties. If a request does not provide its own transformations then these will be applied.

Request transformations ($httpProvider.defaults.transformRequest and $http.defaults.transformRequest) is an array with one function that does the following:

* If the data property of the request configuration object contains an object, serialize it into JSON format.

## Caching

[$http](https://docs.angularjs.org/api/ng/service/$http) responses are not cached by default. To enable caching, you must set the config.cache value or the default cache value to TRUE or to a cache object (created with [$cacheFactory](https://docs.angularjs.org/api/ng/service/$cacheFactory)). If defined, the value of config.cache takes precedence over the default cache value.

In order to:

* cache all responses - set the default cache value to TRUE or to a cache object
* cache a specific response - set config.cache value to TRUE or to a cache object

If caching is enabled, but neither the default cache nor config.cache are set to a cache object, then the default $cacheFactory("$http")object is used.

The default cache value can be set by updating the [$http.defaults.cache](https://docs.angularjs.org/api/ng/service/$http#defaults) property or the [$httpProvider.defaults.cache](https://docs.angularjs.org/api/ng/provider/$httpProvider#defaults) property.

When caching is enabled, [$http](https://docs.angularjs.org/api/ng/service/$http) stores the response from the server using the relevant cache object. The next time the same request is made, the response is returned from the cache without sending a request to the server.

## Interceptors

The interceptors are service factories that are registered with the $httpProvider by adding them to the $httpProvider.interceptors array. The factory is called and injected with dependencies (if specified) and returns the interceptor.

There are two kinds of interceptors (and two kinds of rejection interceptors):

* request: interceptors get called with a http [config](https://docs.angularjs.org/api/ng/service/$http#usage) object. The function is free to modify the config object or create a new one. The function needs to return the config object directly, or a promise containing the config or a new config object.
* requestError: interceptor gets called when a previous interceptor threw an error or resolved with a rejection.
* response: interceptors get called with http response object. The function is free to modify the response object or create a new one. The function needs to return the response object directly, or as a promise containing the response or a new response object.
* responseError: interceptor gets called when a previous interceptor threw an error or resolved with a rejection.