Dependency Injection

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

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

Using Dependency Injection

DI is pervasive throughout Angular. You can use it when defining components or when providing run andconfig blocks for a module.

* Components such as services, directives, filters, and animations are defined by an injectable factory method or constructor function. These components can be injected with "service" and "value" components as dependencies.
* Controllers are defined by a constructor function, which can be injected with any of the "service" and "value" components as dependencies, but they can also be provided with special dependencies. See[Controllers](https://docs.angularjs.org/guide/di#controllers) below for a list of these special dependencies.
* The run method accepts a function, which can be injected with "service", "value" and "constant" components as dependencies. Note that you cannot inject "providers" into run blocks.
* The config method accepts a function, which can be injected with "provider" and "constant" components as dependencies. Note that you cannot inject "service" or "value" components into configuration.

See [Modules](https://docs.angularjs.org/guide/module#module-loading-dependencies) for more details about run and config blocks.

Factory Methods

The way you define a directive, service, or filter is with a factory function. The factory methods are registered with modules. The recommended way of declaring factories is:

angular.module('myModule', [])

.factory('serviceId', ['depService', function(depService) {

// ...

}])

.directive('directiveName', ['depService', function(depService) {

// ...

}])

.filter('filterName', ['depService', function(depService) {

// ...

}]);

Module Methods

We can specify functions to run at configuration and run time for a module by calling the config and runmethods. These functions are injectable with dependencies just like the factory functions above.

angular.module('myModule', [])

.config(['depProvider', function(depProvider) {

// ...

}])

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

// ...

}]);

Controllers

Controllers are "classes" or "constructor functions" that are responsible for providing the application behavior that supports the declarative markup in the template. The recommended way of declaring Controllers is using the array notation:

someModule.controller('MyController', ['$scope', 'dep1', 'dep2', function($scope, dep1, dep2) {

...

$scope.aMethod = function() {

...

}

...

}]);

Unlike services, there can be many instances of the same type of controller in an application.

Moreover, additional dependencies are made available to Controllers:

* [$scope](https://docs.angularjs.org/guide/scope): Controllers are associated with an element in the DOM and so are provided with access to the [scope](https://docs.angularjs.org/guide/scope). Other components (like services) only have access to the [$rootScope](https://docs.angularjs.org/api/ng/service/$rootScope) service.
* [resolves](https://docs.angularjs.org/api/ngRoute/provider/$routeProvider#when): If a controller is instantiated as part of a route, then any values that are resolved as part of the route are made available for injection into the controller.

Dependency Annotation

Angular invokes certain functions (like service factories and controllers) via the injector. You need to annotate these functions so 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.

Tools like [ng-annotate](https://github.com/olov/ng-annotate) let you use implicit dependency annotations in your app and automatically add inline array annotations prior to minifying. If you decide to take this approach, you probably want to useng-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) {

// Angular will throw when this runs

}]);

When the willBreak service is instantiated, Angular 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

});

Understanding Controllers

In Angular, a Controller is defined by a JavaScript **constructor function** that is used to augment the[Angular Scope](https://docs.angularjs.org/guide/scope).

When a Controller is attached to the DOM via the [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) directive, Angular will instantiate a new Controller object, using the specified Controller's **constructor function**. A new **child scope** will be created and made available as an injectable parameter to the Controller's constructor function as $scope.

If the controller has been attached using the controller as syntax then the controller instance will be assigned to a property on the new scope.

Use controllers to:

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

## What is Routing in AngularJS?

If you want to navigate to different pages in your application, but you also want the application to be a SPA (Single Page Application), with no page reloading, you can use the ngRoute module.

The ngRoute module routes your application to different pages without reloading the entire application.

What do I Need?

To make your applications ready for routing, you must include the AngularJS Route module:

<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular-route.js"></script>

Then you must add the ngRoute as a dependency in the application module::

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

Now your application has access to the route module, which provides the $routeProvider.

Use the $routeProvider to configure different routes in your application:

Where Does it Go?

Your application needs a container to put the content provided by the routing.

This container is the ng-view directive.

There are three different ways to include the ng-view directive in your application:

<div ng-view></div>Try it Yourself »

<ng-view></ng-view>Try it Yourself »

<div class="ng-view"></div>Try it Yourself »

Applications can only have one ng-view directive, and this will be the placeholder for all views provided by the route.

Define the $routeProvider using the config method of your application. Work registered in the config method will be performed when the application is loading.

var app = angular.module("myApp", ["ngRoute"]);  
app.config(function($routeProvider) {  
    $routeProvider  
    .when("/", {  
        templateUrl : "main.htm"  
    })  
    .when("/london", {  
        templateUrl : "london.htm",  
        controller : "londonCtrl"  
    })  
    .when("/paris", {  
        templateUrl : "paris.htm",  
        controller : "parisCtrl"  
    })..otherwise({  
        template : "<h1>None</h1><p>Nothing has been selected,</p>"  
    });  
});  
app.controller("londonCtrl", function ($scope) {  
    $scope.msg = "I love London";  
});  
app.controller("parisCtrl", function ($scope) {  
    $scope.msg = "I love Paris";  
});

## The otherwise method

In the previous examples we have used the when method of the $routeProvider.

You can also use the otherwise method, which is the default route when none of the others get a match.

Note-<a href="">tag, in UI-Router you have to only keep state in mind. You provide links like <a ui-sref=""> alternative for href.

Difference between routeprovider and stateprovider

## 1. Angular Routing - per [$routeProvider docs](https://docs.angularjs.org/api/ngRoute/provider/$routeProvider)

URLs to controllers and views (HTML partials). It watches $location.url() and tries to map the path to an existing route definition.

**HTML**

<div ng-view></div>

Above tag will render the template from the $routeProvider.when() condition which you had mentioned in .config (configuration phase) of angular

**Limitations:-**

* The page can only contain single ng-view on page
* If you SPA has multiple part on page which you need to render on some condition, In this $routeProvider fails.(In such cases we need to go for directive like ng-include, ng-switch, ng-if, ng-show actually which looks bad as thinking of SPA)
* You can not relate to routes with each other like who is parent who is child.

## 2. ui-router - per [$stateProvider docs](http://angular-ui.github.io/ui-router/site/#/api/ui.router.state.$stateProvider)

AngularUI Router is a routing framework for AngularJS, which allows you to organize the parts of your interface into a state machine. UI-Router is organized around states, which may optionally have routes, as well as other behavior, attached.

**Multiple & Named Views**

Another great feature is the ability to have multiple ui-views view per template.

While multiple parallel views are a powerful feature, you'll often be able to manage your interfaces more effectively by nesting your views, and pairing those views with nested states.

**HTML**

<div ui-view>

<div ui-view='header'></div>

<div ui-view='content'></div>

<div ui-view='footer'></div>

</div>

The majority of ui-router's power is it can manages nested state & views.

**Pros**

* You can have multiple ui-view on single page
* various view can be nested in each other and maintain by defining state in routing phase.
* We can have child & parent relationship here, simply like inheritance in state, also you could define sibling states.
* You could change the ui-view="some" of state just by using absolute routing using @ with state name.
* Other way you could relative routing using only @ to change ui-view="some", This will replace the ui-view rather than checking it is nested or not.
* Here you could use ui-sref to create a href URL dynamically on the basis of URL mentioned in a state, also you could give a state params in the json format.

$routeProvider is linking specific URL to a controller and assigning a template .

$routeProvider

.when('/contact/', {

templateUrl**:** 'app/views/core/contact/contact.html',

controller**:** 'ContactCtrl'

})

});

$stateProvider allows us to give names for routes. Having a name we can duplicate the route with another name assign different controller, view, well.. we can do whatever we want,(more flexible).

$stateProvider

.state("contact", {

url**:** "/contact/",

templateUrl**:** '/app/Aisel/Contact/views/contact.html',

controller**:** 'ContactCtrl'

});

**$route:** This is used for deep-linking URLs to **controllers** and **views** (HTML partials) and watches **$location.url()** in order to map the path from an existing definition of route.   
**ui-router** - It is used to enhance and to improve the capabilities of routing, It is basically a framework.  
When we use **ngRoute**, the route is configured with **$routeProvider** and when we use **ui-router**, the route is configured with **$stateProvider** and **$urlRouterProvider**.

**ui-router** supports everything the normal **ngRoute** has as well as many extra functions.

There are some powerful reasons to use **ui-router** over **ngRoute**:

1. It is used for large application where the user have many pages and that are inheriting from different section. So ui-router is used for the nesting views.
2. It allows the user to have a strong linking between the state. So on changing the url in a particular place, it will update every link to that state.
3. User can also pass information between states with the help of $stateParams.

$Stateparam vs $state

With ui-router, it's possible to inject either $state or $stateParams into a controller to get access to parameters in the URL. However, accessing parameters through $stateParams only exposes parameters belonging to the state managed by the controller that accesses it, and its parent states, while $state.params has all parameters, including those in any child states.

1. Given the following code, if we directly load the URL http://path/1/paramA/paramB, this is how it goes when the controllers load:
2. $stateProvider.state('a', {
3. url: 'path/:id/:anotherParam/',
4. controller: 'ACtrl',
5. });
6. $stateProvider.state('a.b', {
7. url: '/:yetAnotherParam',
8. controller: 'ABCtrl',
9. });
10. module.controller('ACtrl', function($stateParams, $state) {
11. $state.params; // has id, anotherParam, and yetAnotherParam
12. $stateParams; // has id and anotherParam
13. }
14. module.controller('ABCtrl', function($stateParams, $state) {
15. $state.params; // has id, anotherParam, and yetAnotherParam
16. $stateParams; // has id, anotherParam, and yetAnotherParam
17. }