CONTENTS



# AngularJS Essentials

By Gil Fink

## WHAT IS ANGULARJS?

AngularJS is a very popular MVW (Model/View/Whatever) framework maintained by Google. AngularJS enables web developers to impose MVC structures on their client-side code and to build dynamic web applications more easily.

AngularJS introduces component-based architecture. It also includes a variety of conventions that developers should know and use, which will be presented in this Refcard. AngularJS includes a wide range of features that help to build Single Page Applications (SPAs) faster and is considered a "one-stop shop" for creating SPAs. Another good aspect of AngularJS is that it was written with testability in mind.

You will learn that AngularJS is expressive, readable, extensible, and quick to develop. All of these qualities make AngularJS one of the best JavaScript frameworks for application development

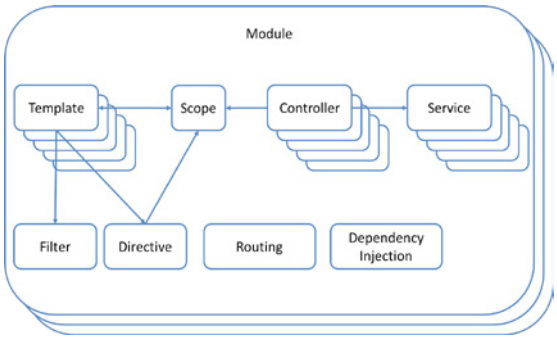
In this Refcard, you will get to know the essential parts of AngularJS. The Refcard will also try to help you speed up your development process when you use AngularJS.

## ANGULARJS BUILDING BLOCKS

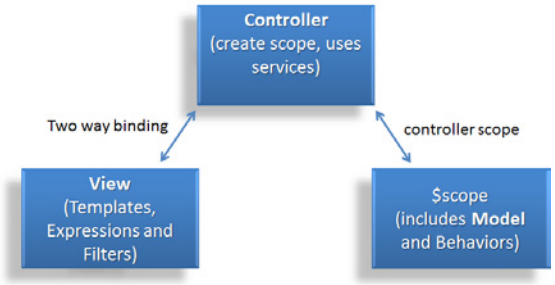
AngularJS includes a few building blocks that you should know and understand:

| VERSION | IMPLEMENTATIONS |
|---|---|
| **Modules** | Logical containers for different parts of AngularJS objects |
| **Controllers** | A constructor functions that are used to hold scopes and expose behaviors that views can use |
| **Scopes** | Objects that reference the application model |
| **Views** | HTML that presents the models and interacts with the user |
| **Templates** | HTML that contains AngularJS-specific elements and attributes |
| **Services** | Objects that enable you to organize and share code in an AngularJS application |
| **Filters** | Transform the model data to an appropriate representation in the view |
| **Directives** | Reusable web components that tell the AngularJS HTML compiler how to render them and how the components behave. Can help you enrich the existing HTML elements or create your own custom ones. |
| **Routing** | A module that maps URLs to application states / handles state and view switching |
| **Dependency Injection** | Design pattern that deals with how to inject dependencies to a dependent object. AngularJS uses this pattern and enables you to inject dependencies in a declarative way. |

The following diagram shows how these building blocks work together to build the AngularJS ecosystem:




Let's explore how some of these parts relate to the MVC design pattern:



### THE MODEL

The model is the data structure that holds some business data. It will usually reside from the server side using a REST API or from a service. The model is exposed to the AngularJS view using the $scope object.

### THE VIEW

The view is the HTML part that presents the data structure and enables user interactions. The view binds to a model using the $scope object. It can call controller functions through the $scope object as well. It uses filters to transform the model data to an appropriate representation. It also uses directives as reusable web components and for DOM manipulation.

### THE CONTROLLER

The controller performs all the business logic and orchestrates operations. It is responsible for initializing the $scope, can interact with AngularJS services, and exposes functions to the view using the $scope object. The controller is also responsible for updating the model based on user's view interactions.

## SETTING UP THE ENVIRONMENT

The first thing that you will need to do is set up the environment. AngularJS can be downloaded from www.angularjs.org and included in your web page or it can be included directly from the Google Content Delivery Network (CDN):

```
<script src="https://ajax.googleapis.com/ajax/libs/
angularjs/1.2.9/angular.min.js"></script>
```

Another option is to install AngularJS using your favorite package manager and then include its script. For example, you can use Bower to install AngularJS using the following syntax:

```
bower install angular
```

A third option is to clone the GitHub angular-seed repository. Angular-seed is an application skeleton for the AngularJS web application.

Once AngularJS is installed in your web application, you can start working with it.

### THE NG-APP DIRECTIVE

One of the most important AngularJS directives is the ng-app directive. ng-app bootstraps the application, and instructs AngularJS to load the main module that is given as the directive parameter. The next code example shows how to setup the ng-app directive to the myApp module:

```
<!doctype html>
<html ng-app="myApp">
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/
angularjs/1.2.9/angular.min.js"></script>
  </head>
  <body>
    <div>
    </div>
  </body>
</html>
```

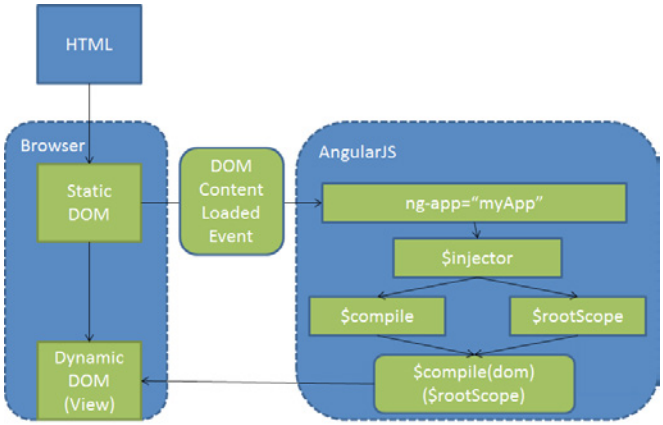
## ANGULARJS BOOTSTRAP

Once the browser loads a web page and it includes the AngularJS script along with the ng-app directive, AngularJS is bootstrapping. There are two options for bootstrapping: automatic bootstrap and manual bootstrap. Let's explore both of these options:

### AUTOMATIC BOOTSTRAP

Three main things happen when AngularJS is automatically bootstrapping. First, AngularJS creates an injector object, which will be used later on for dependency injection. Then, AngularJS creates the root scope, which is the root for all the scopes in the application. Finally, AngularJS compiles the Document Object Model (DOM) starting from the ng-app directive, compiling and processing all the directives and bindings that it finds in the DOM.

The following diagram shows the process:




### MANUAL BOOTSTRAP

Sometimes you want to perform operations before AngularJS starts. These operations can be configurations, retrieving relevant data or anything else you might think about. AngularJS enables you to manual bootstrap the application. You will need to remove the ng-app directive from the HTML and use the angular.bootstrap function instead. Pay attention, as you will need to declare your modules before using them in the angular.bootstrap function. Manual AngularJS bootstrapping is not so common. Now that we know about how to start an AngularJS application, let's explore the first AngularJS building block – modules.

## MODULES

### WHAT IS A MODULE?

AngularJS modules are logical containers for different parts of AngularJS objects. AngularJS applications are typically created with one or more modules. Modules can have dependencies on other modules. These dependencies are resolved automatically by AngularJS when a module is loaded into memory.

### CREATING A MODULE

You create modules using the angular.module function. Each created module should have a unique name that is passed as the first argument to the angular.module function. The second angular.module argument is an array of dependencies, which can have zero to many dependencies. Here is an example of using the angular.module function to declare a myApp module, which is dependent on a storageService:

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

### CONFIGURING A MODULE

You can add module configuration using the config function once you declare a module. The configuration will get executed when the module is loading. You can inject providers or constants (which will be discussed later in the Refcard) to the configuration block you are writing. Here is an example of a call for the config function:

```
angular.module('myApp', ['storageService'])
   .config(function ($provide, $filterProvider) {
      $provide.value('myValue', 1);
      $filterProvider.register('myFilter', ...);
   });
```

In the previous code, there are two injectable objects - $provide and $filterProvider. In the config function body, we create a constant value called myValue, which is set to 1. We also register a filter called myFilter into the $filterProvider.

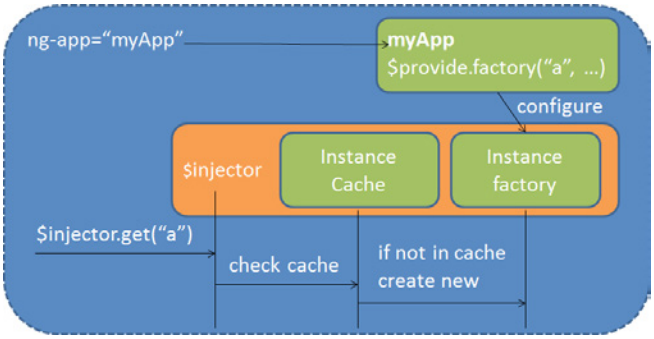
Another option to interact with an AngularJS module execution is to use the run function. The run function runs after all the modules have been loaded. The function is used to literally run the AngularJS application. You can inject services and constants into the run function callback. The following code shows how to use the run function:

```
angular.module('myApp', ['storageService'])
   .run(function (someInstance) {
      someInstance.doSomething();
   });
```

Now that we understand how to create and use modules, let's explore the AngularJS dependency injection (DI) mechanism.

### ANGULARJS DEPENDENCY INJECTION

Each AngularJS application includes an $injector object. The $injector is responsible for dependency management and for dependency lookups. Here is a diagram that shows how DI works in AngularJS:




Let's analyze what is going on in the diagram. Once the root module is created, AngularJS will create a new $injector object. While the application is running, a lot of objects such as controllers, services, and filters are registered in the $injector container. Later on, if an object has a dependency, AngularJS will use the $injector object to search it. If an instance of the dependency is available, it is returned by the $injector. If there is no instance of the dependency, a new object is created using the $injector instance factory. The new object is returned and also stored in the instance cache.

Some of the main AngularJS objects such as modules, controllers, and services can declare that they need a dependency. In order to do that, you need to add function arguments with the exact name of the dependency. That means that if you have an object named obj and you want to inject it, you will have to have a function argument in the AngularJS object with the same name. Here is a controller example that declares it needs a $scope injected object:

```
var myApp = angular.module('myApp',[]);
myApp.controller('MyController ', function($scope) {
});
```

You can see the use of explicit DI syntax ['$scope', function($scope) {}] which helps to avoid minification issues. This is the preferred way to use DI in AngularJS, and in the code example we use it to create a controller. Speaking of controllers, let's explore what these AngularJS objects are.

## CONTROLLERS

### WHAT IS A CONTROLLER?

AngularJS controllers are just JavaScript constructor functions that are used to hold scopes and to expose behaviors that views can use. You can think about controllers as the logic behind the view. Each controller has a new child scope that is available as the $scope argument in the constructor function. When a controller is created during runtime, the $injector will inject the $scope argument. Controllers add properties and functions to the $scope.

### CREATING A CONTROLLER

In order to create a controller, all you need to do is to declare a constructor function. The following code shows a simple controller:

```
var myApp = angular.module('myApp',[]);
myApp.controller('MyController', ['$scope',
function($scope) {
}]);
```

The way to wire a controller constructor function to a view is using the ng-controller directive:

```
<div ng-controller="MyController">
</div>
```

A few notes about controller best practices:

- Controllers should contain only business logic that relates to the view that they are bound to.

- Business logic that isn't related to the controller bound view should be injected into controllers as services.
- Don't use controllers for DOM manipulation, formatting, or filtering. We will discuss the ways to do those operations later in the Refcard.

Now that you are familiar with controllers, let's talk about scopes.

## SCOPES

### WHAT IS A SCOPE?

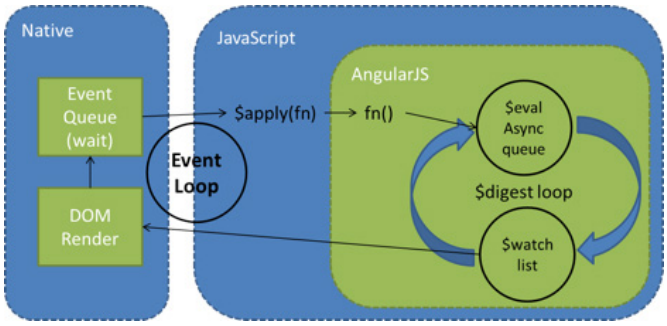
Scopes are objects that include properties and the functions set by controllers that a view can later use. You can think about scopes as the glue between a controller and a single view.

Once a controller is attached to a view using the ng-controller directive, all the scope properties and functions are data bound to the view, and it can use them. Changes to scope properties are reflected in the bounded view and user interactions can also change the bound properties.

### WHAT IS THE DIGEST CYCLE?

The digest cycle in AngularJS is code that runs in intervals and its main idea is to enable data binding. Its purpose is to observe model mutations (using $watch API) and to propagate any model changes (using $apply API).

The next diagram shows what the digest cycle in AngularJS looks like:




### USING SCOPES IN CODE

AngularJS runtime injects a scope object into a controller, and then the controller can set the $scope with properties and behaviors. The following example shows how to add a property called message and a function called myFunc to a scope object:

```
var myApp = angular.module('myApp',[]);
myApp.controller('MyController ', function($scope) {
   $scope.message = 'hello';
   $scope.myFunc = function() {
      // do something
   }
});
```

You can also put full model objects inside scope properties that can later be used by a view.

### THE $ROOTSCOPE

All AngularJS scopes inherit their functionality from the $rootScope. Every AngularJS application has a single $rootScope. The $rootScope includes several utility functions that you can use which are also available on its child scopes such as $new, $watch, and $destroy. You can find more details about these functions in the AngularJS documentation.

## VIEWS

A view is dynamic HTML that presents models and interacts with the user. AngularJS views can be bound to a controller using the ng-controller directive. Once a binding is set, you can use scope properties and functions in the view. In order to harness the full potential of views, you should understand two AngularJS concepts that relate to views: templates and expressions.

### TEMPLATES

A template is a declarative specification that contains HTML parts. It also contains AngularJS specific elements and attributes that are called directives and AngularJS expressions. The directives "explain" to AngularJS how to transform the template into a dynamic view. The expressions are placeholders that are used for data binding. The following example shows a template:

```
<div ng-controller="MyController">
        <input ng-model="message">
        <button ng-click="changeMessage()">{{btnTe
xt}}</button>
</div>
```

### EXPRESSIONS

In the previous template example you can see that we used curly brackets to wrap btnText. This is an example of an expression. Expressions are JavaScript code used for binding. Expressions are placed in double curly brackets. When AngularJS processes a template, it searches for expressions, then parses and evaluates them. The evaluation of expressions is being done against the controller scope object. If the scope doesn't contain the expression value, AngularJS will not throw an exception. This behavior enables you to perform lazy binding.

The following code example shows a simple expression that evaluates to 5:

```
<div> 3+2={{3+2}} </div>
```

AngularJS expressions don't have access to global variables and can only use scope-exposed functionality. Expressions can't use control flow statements except from the ternary operator (a ? b : c).

In AngularJS 1.3 you can also use expressions for one-time binding. Any expression that starts with :: will evaluate only once. The following code example shows you how to use one-time binding on a message property:

```
<div>
One-time binding: {{::message}}
</div>
```

## SERVICES

### WHAT IS A SERVICE?

Services in AngularJS are singleton objects that are used to perform a specific task. Services are UI independent, meaning that they shouldn't manipulate or use UI elements (this is the role of view and directives). You can inject services to modules, controllers, filters, directives, and more.

### CREATING A CUSTOM SERVICE

There are four options for service creation in AngularJS:

| SERVICE TYPE | DESCRIPTION |
|---|---|
| **value** | Use the module's value function with a given name and value. The injector will return the exact value. |
| **factory** | Use the module's factory function with a given name and a factory function. The injector will invoke the given function. |
| **service** | Use the module's service function with a given name and constructor function. The injector will use the new keyword to create the service single instance |
| **provider** | Use the module's provider function with a given name and provider function. The injector will invoke the provider's $get function. |

Here are usage examples for the first three options:

```
var myApp = angular.module('myApp',[]);
myApp.value('myValue', 'a constant value);

myApp.factory('myFactory', function(name) {
   return 'Hi ' + name;
}
myApp.service('myFactory', function(name) {
   return 'Hi ' + name;
}
```

When you want to use the previous services you will inject them to a dependent object by their name and use them. For example, here is a controller that uses the previous services:

```
myApp.controller('myController', function($scope,
myValue, myFactory) {
   $scope.value = myValue;
   $scope.greet = myFactory;
});
```

### BUILT-IN ANGULARJS SERVICES

AngularJS comes with several built-in services. Here are the descriptions of some of them:

| BUILT-IN SERVICE | DESCRIPTION |
|---|---|
| **$http** | Used for communicating over HTTP with remote servers. The service receives as argument a configuration object for configuring the HTTP request. |
| **$resource** | Abstraction on top of $http for interaction with REST services. Require a dependency on ngResource module. Exposes functions such as get, save and delete. |
| **$q** | Exposes the ability to create promises (deferred objects) for cleaner design of asynchronous code. |
| **$log** | Enable to log messages |
| **$location** | Includes the current parsed URL. Changes to $location are reflected to the browser address bar. |

There are many other services such as $animate, $window, $rootScope, $interval, and $document You can read about those services in the AngularJS documentation: https://docs.angularjs.org/api/ng/service.

## FILTERS

### WHAT IS A FILTER?

Filters format an evaluated expression value to an appropriate representation in the view. When you want to use a filter you just "pipe" it using the "|" sign to the expression like in the following code example:

```
{{ message | filter }}
```

For example, if you want to use a currency filter with a number value you just state the name of the filter:

```
{{ 5 | currency }}
```

The output of the expression will be 5$.

Filters can have arguments using the semicolon sign, and they can be chainable. On the other hand, filters shouldn't depend on global state and should be stateless.

### CREATING A CUSTOM FILTER

You can create your custom filters in two ways: using the module's filter function or registering a filter in the $filterProvider service. The first argument that a filter receives in its callback function is the input value. All the other arguments are considered additional arguments, which can be used by the filter function.

The following example shows how to register a filter that turns any string to its upper case representation:

```
var myApp = angular.module('myApp',[]);
myApp.filter('uppercase', function() {
   return function(str) {
      return (text || '').toUpperCase();
   };
}
```

### BUILT-IN ANGULARJS FILTERS

AngularJS comes with a set of built-in filters that you can use. Here are their descriptions:

| BUILT-IN FILTER | DESCRIPTION |
|---|---|
| currency | Used to format numbers as currency |
| date | Used to format dates as string |
| number | Used to format numbers as string |
| json | Converts JavaScript object to its string representation |
| lowercase | Transforms the given string into its lowercase representation |
| uppercase | Transforms the given string into its uppercase representation |
| limitTo | Creates a new array or string according to the specified number of elements |
| orderBy | Orders a given array by a given expression predicate |
| filter | Selects a subset of items form the given array |

## DIRECTIVES

### WHAT IS A DIRECTIVE?

Directives are one of the most powerful features in AngularJS. Directives are custom HTML elements and attributes that AngularJS recognizes as part of its ecosystem. Directives allow you to extend the existing HTML with your own elements or attributes and to add behavior to existing elements or attributes. You already saw a few of the AngularJS built-in directives such as ng-app and ng-controller.

When an AngularJS bootstrap runs, the AngularJS's HTML compiler ($compile) is attaching a specified behavior to DOM elements according to directives. The process is done in two phases: compile and link. The compiler is traversing on the DOM and collects all the directives. The linker combines directives with their scope and produces a dynamic view.

### CREATING A CUSTOM DIRECTIVE

Like controllers, services, and filters, directives are also registered to modules. In order to register a directive, you will use the module's directive function, which receives a directive name and a factory function. The factory function should return a directive definition object. The following example will create a bare directive with the name myDirective:

```
var myApp = angular.module('myApp',[]);
myApp.directive('myDirective', function() {
   return {
   };
});
```

The returned directive definition object can include various configuration options.

| DIRECTIVE CONFIGURATION | DESCRIPTION |
|---|---|
| restrict | By default directives are restricted to attributes. Using restrict you can restrict the directive to A (attribute), E (element), C (class) and M (comment) |
| template | Appends a given HTML as the child element of the element |
| replace | If set to true will replace the entire element with a given template (used with the template or templateUrl configurations) |
| templateUrl | A URL for a given template that would be loaded and appended to the element |
| scope | By default, directive has the scope of its parent controller. If Scope is set to {}, an isolated scope is created for the directive, You can map and bind parent properties to an isolated scope using @ (one directional mapping), = (bidirectional mapping), and & (executes an expression in the parent scope) |
| transclude | If set to true, the directive can wrap content. In the template of the directive you need to use the ng-transclude directive in order to mark the place to wrap content |
| link | A function that receives the scope, directive element, the element attributes as parameters and controllers. The function should be used to add event listeners or to manipulate the DOM |

The following code example shows a simple directive declaration:

```
myApp.directive('myDialog', function() {
   return {
      restrict: 'EA',
      scope: {},
      template: '<div class="dialog">{{message}}</div>',
      link: function(scope, element, attrs) {
         scope.message = 'hello';
      }
   };
});
```

When you want to use this directive you can write in your HTML:

```
<my-dialog></my-dialog>
```

or:

```
<div my-dialog></div>
```

### BUILT-IN USEFUL DIRECTIVES

AngularJS includes many built-in and useful directives that you can use.

| BUILT-IN DIRECTIVE | DESCRIPTION |
|---|---|
| ngModel | Binds an HTML element to a property in the scope |
| ngEventName (replace EventName with any DOM event) | These directives enable you to add custom behavior on the specified event |
| ngValue | Binds a given expression to a value of an input. When the input changes the bound model (which is the ngModel) changes as well |
| ngBind | Binds an expression to the text content of a HTML element. When the expression changes the text, content changes as well |
| ngClass | Dynamically sets CSS classes on HTML elements by evaluating the given expression |
| ngInclude | Retrieves an external HTML fragments, compile it and add it to the DOM |
| ngRepeat | Collection iterator. In each item in the collection, the template that is bound to the directive is instantiated with its own scope |
| ngShow | Shows or hides an HTML element according to the evaluated expression |
| ngSwitch | Enables to create DOM evaluation conditions like a switch case statement |

You can find more built-in directives in the following AngularJS documentation: https://docs.angularjs.org/api/ng/directive.

## ANGULARJS AND FORMS

AngularJS adds two way binding, state management, and validation to HTML forms. You can bind HTML input types to models using the ngModel directive and AngularJS will provide all the mechanisms.

When AngularJS locates form elements in the DOM, it will make them available as a property of the scope. The name of the property will be the name of the form. Each input type in the form will be a sub-property of the scope form property

### FORM VALIDATION DIRECTIVES

AngularJS provides a few built-in validation directives which can set the form validation $error property. The available directives are:

| VALIDATION DIRECTIVE | DESCRIPTION |
|---|---|
| required | A required field |
| min | Validates the value is greater than the minimum value |
| max | Validates the value is less than the maximum value |
| minlength | Validates the value has length bigger than the provided minimum |
| maxlength | Validates the value has length smaller than the provided maximum |
| pattern | Validates the value against a given regular expression |

Here is an example of using form validation with AngularJS:

```
<form name="myForm">
   <div>
      <input type="text" name="username" ng-
model="user.name" required/>
   </div>
   <div>
      <input type="password" name="password" ng-
model="user.password" required />
   </div>
   <button ng-click="login(user)">
        Login
   </button>
</form>
```

In this form, the scope will have a property with the name myForm that will be bound to the form. The myForm property will include the username and password sub-properties, which will include references to the input types. If the user name and password isn't supplied, the $error of the form will be set to an error and the form wouldn't submit.

AngularJS also decorates form elements and input types with CSS classes. You can use those classes to change the presented style of an element according to its state. The CSS classes that you can use are:

- ng-valid
- ng-invalid
- ng-invalid-[validation name]
- ng-pristine
- ng-dirty

## ROUTING

One of the most important mechanisms in Single Page Applications (SPA) is client-side routing. Client-side routing enables developers to intercept route changes and instead of rendering a new page in the server, render a document fragment in the client and replace a shell element. This is how you move from one page to another in SPAs. AngularJS provides a routing feature that can be used to create this behavior to inject views into an ngView decorated element.

### THE ngROUTE MODULE

The ngRoute module is a router module for AngularJS (it isn't built-in to AngularJS). There are other options (such as ui-router), which are also very commonly used. In order to use a module, you will have to reference the angular-route script and add ngRoute as a dependency of your ngApp module.

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

In order to register routes, you use the $routeProvider service. The $routeProvider service provides a *when* function to register routes and a default route that is set using the otherwise function. Each registered route includes a path and a configuration object.

The configuration object includes the following main configurations:

| ROUTE CONFIGURATION OPTION | DESCRIPTION |
|---|---|
| **template/ templateUrl** | The path/template string that represents the template that is going to be attached to an ngView |
| **controller** | The controller that is bound to the template's scope |
| **redirectTo** | Name of the route to redirect to when you want to redirect to a different route |

Here is an example:

```
myApp.config(function($routeProvider) {
   $routeProvider
      .when('/about', {
         templateUrl: 'views/about.html',
         controller: 'myController'
      }).when('/cart', {
         templateUrl: 'views/cart.html'
      }).otherwise({
         templateUrl: 'views/home.html',
         controller: 'myController'
   });
});
```

## ADDITIONAL RESOURCES

For further research about AngularJS, explore the following web sites:

- AngularJS web site: https://angularjs.org/
- AngularJS Developer Guide: https://docs.angularjs.org/guide
- PhoneCat Tutorial App: https://docs.angularjs.org/tutorial
- AngularJS official forum: https://groups.google.com/forum/#!forum/angular
- AngularJS official Twitter handle: https://twitter.com/angularjs

## ABOUT THE AUTHOR

**Gil Fink** is a web development expert, ASP.NET/IIS Microsoft MVP and the founder ofsparXys. He is currently consulting for various enterprises and companies, where he helps to develop web and RIA-based solutions. He conducts lectures and workshops for individuals and enterprises who want to specialize in infrastructure and web development. He is also co-author of several Microsoft Official Courses (MOCs) and training kits, co-author of "Pro Single Page Application Development" book (Apress), the founder of Front-End.IL Meetup and co-organizer of GDG Rashlatz Meetup. You can read his publications in his blog: http://blogs.microsoft.co.il/gilf/.

## RECOMMENDED BOOK


Pro Single Page Application Development walks you through building a SPA step-by-step. SPA development comes with its own particular challenges, including tracking history, user interface performance, and how to handle search engine optimization. This book will be your one-stop shop for creating fluid, modern applications on the web.

BUY NOW






**DZONE, INC.**
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

**REFCARDZ FEEDBACK WELCOME**
refcardz@dzone.com

**SPONSORSHIP OPPORTUNITIES**
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

**VERSION 1.0** **$7.95**