Proffesional development program

AngularJS

Contents

[Chapter 1. Prerequisite 5](#_Toc438478908)

[HTML 5](#_Toc438478909)

[CSS 5](#_Toc438478910)

[JavaScript 5](#_Toc438478911)

[jQuery 6](#_Toc438478912)

[S.P.A. Theory 6](#_Toc438478913)

[Chapter2. The AngularJS Philosophy 8](#_Toc438478914)

[What Is MVC (Model-View-Controller)? 8](#_Toc438478915)

[Core beliefs 8](#_Toc438478916)

[Data-driven (via data-binding) 8](#_Toc438478917)

[Declarative 9](#_Toc438478918)

[Separate your concerns 9](#_Toc438478919)

[Dependency Injection 9](#_Toc438478920)

[Extensible 10](#_Toc438478921)

[Test first, test again, keep testing 10](#_Toc438478922)

[Chapter 3. Starting out with AngularJS 11](#_Toc438478923)

[Bootstrapping AngularJS 11](#_Toc438478924)

[Modules 11](#_Toc438478925)

[Using ng-bind Versus Double Curlies 12](#_Toc438478926)

[First Controller 12](#_Toc438478927)

[$scope Versus controllerAs Syntax 13](#_Toc438478928)

[How AngularJS is working behind the scenes 13](#_Toc438478929)

[More directives 14](#_Toc438478930)

[AngularJS $watch() , $digest() and $apply() 15](#_Toc438478931)

[Forms 16](#_Toc438478932)

[One/two-way data-binding 16](#_Toc438478933)

[Working with Forms 16](#_Toc438478934)

[Form Validation and States 16](#_Toc438478935)

[Nested Forms with ng-form 18](#_Toc438478936)

[ngMessages 18](#_Toc438478937)

[Working with Filters 19](#_Toc438478938)

[Services.Factories.Providers 19](#_Toc438478939)

[Using the Factory Method 21](#_Toc438478940)

[Using the Service Method 22](#_Toc438478941)

[Using the Provider Method 22](#_Toc438478942)

[Global Object Services 23](#_Toc438478943)

[Exceptions 23](#_Toc438478944)

[Working with Dangerous Data 23](#_Toc438478945)

[Services for Ajax and Promises 24](#_Toc438478946)

[Services for Views 26](#_Toc438478947)

[Managing Injection 27](#_Toc438478948)

[Understanding AngularJS $rootScope and $scope 27](#_Toc438478949)

[Routing Using ngRoute 27](#_Toc438478950)

[Routing Options 28](#_Toc438478951)

[Events 29](#_Toc438478952)

[Using the $routeParams Service 29](#_Toc438478953)

[Things to Watch Out For 29](#_Toc438478954)

[Alternatives: ui-router 30](#_Toc438478955)

[Events 30](#_Toc438478956)

[Directive - declarative, data driven and conversational 31](#_Toc438478957)

[Creating a Directive 32](#_Toc438478958)

[Bibliography 38](#_Toc438478959)

# Chapter 1. Prerequisite

## HTML

HyperText Markup Language, commonly referred to as HTML, is the standard markup language used to create web pages. Web browsers can read HTML files and render them into visible or audible web pages. HTML describes the structure of a website semantically along with cues for presentation, making it a markup language, rather than a programming language.

HTML elements form the building blocks of all websites. HTML allows images and objects to be embedded and can be used to create interactive forms. It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes and other items. (HTML, 2015)

## CSS

Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language. Although most often used to set the visual style of web pages and user interfaces written in HTML and XHTML, the language can be applied to any XML document, including plain XML, SVG and XUL, and is applicable to rendering in speech, or on other media. (Cascading Style Sheets, 2015)

## JavaScript

JavaScript is a high-level, dynamic, untyped, and interpreted programming language. Alongside HTML and CSS, it is one of the three essential technologies of World Wide Web content production; the majority of websites employ it and it is supported by all modern web browsers without plug-ins. JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. JavaScript is also used in environments that are not web-based, such as PDF documents, site-specific browsers, and desktop widgets. (JavaScript, 2015)

The secret is that the JavaScript engine makes two passes over code when it comes into scope. On the first pass it initializes variables and on the second pass it executes code. Variables are scoped by functions in JavaScript and they’re either global or local.Global variables are accessible everywhere, and local variables are only accessible where they are declared. The only block that defines scope for a variable in JavaScript is a function. JavaScript variables are declared following the var keyword. A variable can contain any type of data: arrays, integers, floats, strings, and so on. The variable type isn’t specified, so JavaScript is considered a loosely typed language. Even after a value is assigned to a variable, the type of value can be changed by assigning a value with a different type, so it’s also considered a dynamic language. (Michael S. Mikowski, 2014)

## jQuery

jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML. jQuery is the most popular JavaScript library in use today, with installation on 65% of the top 10 million highest-trafficked sites on the Web. jQuery is free, open-source software licensed under the MIT License. jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications. jQuery also provides capabilities for developers to create plug-ins on top of the JavaScript library. This enables developers to create abstractions for low-level interaction and animation, advanced effects and high-level, theme-able widgets. The modular approach to the jQuery library allows the creation of powerful dynamic web pages and web applications. (jQuery, 2015)

## S.P.A. Theory

An SPA delivers a desktop application in the browser. The result is a highly responsive experience that surprises and delights users instead of using and annoying them. An SPA is an application delivered to the browser that doesn’t reload the page during use. We can think of an SPA as a fat client that’s loaded from a web server. (Michael S. Mikowski, 2014)

There are various techniques available that enable the browser to retain a single page even when the application requires server communication.

* JavaScript frameworks such as AngularJS, Backbone.js, Ember.js, Meteor, or React.

AngularJS is a fully client-side framework. AngularJS's templating is based on bidirectional UI data binding. Data-binding is an automatic way of updating the view whenever the model changes, as well as updating the model whenever the view changes. The HTML template is compiled in the browser. The compilation step creates pure HTML, which the browser re-renders into the live view. The step is repeated for subsequent page views. In traditional server-side HTML programming, concepts such as controller and model interact within a server process to produce new HTML views. In the AngularJS framework, the controller and model state are maintained within the client browser. Therefore new pages are generated without any interaction with a server.

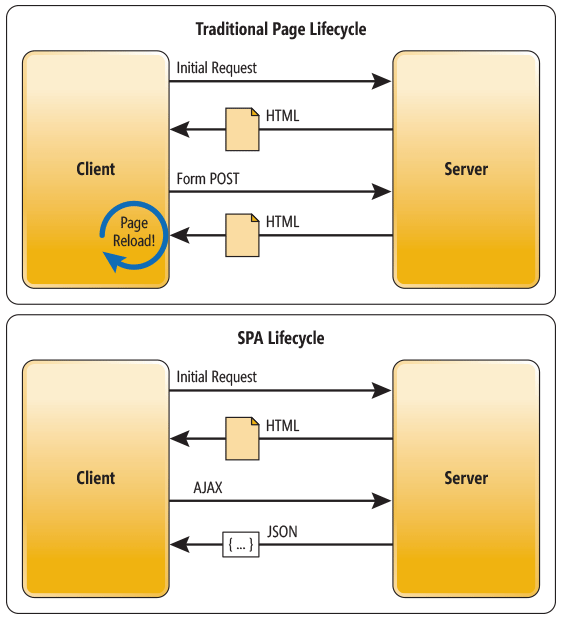
* AJAX

The most prominent technique currently being used is Ajax. Predominantly using the XMLHttpRequest object from JavaScript, other AJAX approaches include using IFRAME or script HTML elements. Popular libraries like jQuery, which normalize AJAX behavior across browsers from different manufacturers, have further popularized the AJAX technique.

* Websockets

WebSockets are a bidirectional stateful real-time client-server communication technology part of the HTML5 specification, superior to AJAX in terms of performance and simplicity.

(Single-page application, 2015)



[The Traditional Page Lifecycle vs. the SPA Lifecycle](https://msdn.microsoft.com/en-us/magazine/dn463786.aspx)

# Chapter2. The AngularJS Philosophy

AngularJS is a superheroic JavaScript MVC framework for the Web. And the best part? It’s all done in pure JavaScript and HTML. No need to learn another new programming or templating language (though you do have to understand the MVC).

## What Is MVC (Model-View-Controller)?

* a way to separate logical units and concerns when developing large applications.

The MVC architectural pattern divides an application into three distinct, modular parts:

* The **model** is the driving force of the application. This is generally the data behind the application, usually fetched from the server. Any UI with data that the user sees is derived from the model, or a subset of the model.
* The **view** is the UI that the user sees and interacts with. It is dynamic, and generated based on the current model of the application.
* The **controller** is the business logic and presentation layer, which peforms actions such as fetching data, and makes decisions such as how to present the model, which parts of it to display, etc.

The model is the data, the view is the UI, and the controller is the business logic.

## Core beliefs

### Data-driven (via data-binding)

We don’t have to waste time funneling data back and forth between the UI and the JavaScript in an AngularJS application. We just bind to the data in our HTML and AngularJS takes care of getting its value into the UI. Not only that, but it also takes care of updating the UI whenever the data changes. The **two-way data-binding** ensures that our controller and the UI share the same model, so that updates to one (either from the UI or in our code) update the other automatically.

### Declarative

AngularJS instead promotes a declarative paradigm, where you declare right in your HTML what it is you are trying to accomplish. This is done through something that AngularJS calls directives. Directives basically extend the vocabulary of HTML to teach it new tricks. We let AngularJS figure out how to accomplish what we want it to do, whether it is creating tabs or datepickers.

### Separate your concerns

There is the actual data that you want to display to the user, or get the user to enter through your application. This is the **model** in an AngularJS project, which is mostly pure data, and represented using JSON objects.

Then there is the user interface or the final rendered HTML that the user sees and interacts with, which displays the data to the user. This is the **view**.

Finally, there is the actual business logic and the code that fetches the data, decides which part of the model to show to the user, how to handle validation, and so on —core logic specific to your application. This is the **controller** for an AngularJS application.

AngularJS is not completely MVC; the controller will never have a direct reference to the view. This is great because it keeps the controller independent of the view, and also allows us to easily test the controller without needing to instantiate a DOM.

### Dependency Injection

Is the concept of asking for the dependencies of a particular controller or service, instead of instantiating them inline via the new operator or calling a function explicitly. Some other part of your code becomes responsible (in this case, the injector) for figuring out how to create those dependencies and provide them when asked for.

Is used across all of its parts, from controllers and services to modules and tests. It allows you to easily write modular, reusable code so that you can use it cleanly and simply as needed.

### Extensible

We can develop robust and complex directives that integrate with third-party libraries like jQueryUI and BootStrap, to name a few, to create a language that is specific to our needs. AngularJS has a great core set of directives for us to get started, and an API that allows us to do everything AngularJS does and more. Our imagination is really the only limit for creating declarative, reusable components.

### Test first, test again, keep testing

Every bit and piece of AngularJS is designed to be testable, from its controllers, services, and directives to its views and routes. Between Dependency Injection and the controller being independent of references to the view, the JS code that we write in an AngularJS application can easily be tested. Because we get the same Dependency Injection system in our tests as in our production code, we can easily instantiate any service without worrying about its dependencies.

# Chapter 3. Starting out with AngularJS

## Bootstrapping AngularJS

This is done through the **ng-app** directive. This is the first and most important directive that AngularJS has, which denotes the section of HTML that AngularJS controls. Putting it on the <html> tag tells AngularJS to control the entire HTML application. We could also put it on the <body> or any other element on the page.

Any element that is a child of that will be handled with AngularJS and be annotated with directives, and anything outside would not be processed.

**Only one** AngularJS application can be auto-bootstrapped per HTML document. The first ngApp found in the document will be used to define the root element to auto-bootstrap as an application. To run multiple applications in an HTML document you must manually bootstrap them **using angular.bootstrap** instead. AngularJS applications cannot be nested within each other. (ngApp, 2015)

## Modules

Modules are AngularJS’s way of packaging relevant code under a single name. An AngularJS module has two parts to it:

* A module can define its own controllers, services, factories, and directives. These are functions and code that can be accessed throughout the module.
* The module can also depend on other modules as dependencies, which are defined when the module is instantiated. What this means is that AngularJS will go and find the module with that particular name, and ensure that any functions, controllers, services, etc. defined in that module are made available to all the code defined in this module.

In addition to being a **container** for related JavaScript, the module is also what AngularJS uses to bootstrap an application. The ng-app directive takes an optional argument, which is the name of the module to load during bootstrapping.

## Using ng-bind Versus Double Curlies

There is no functional difference between the two; both take the value from the controller and display it in the UI. Both of them also keep it data-bound and up to date, so if the value underneath changes, the UI will change automatically.

The advantage ng-bind has over the double-curly notation is that it takes AngularJS time to bootstrap and execute before it can find and replace all the double curly braces from the HTML. That means, for a portion of a second while the browser starts, you might see flashing double curly braces in the UI before AngularJS has the chance to kick in and replace them. This is only for the very first page load, and not on views loaded after the first load. You will not have that issue with ng-bind.

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

## First Controller

The common responsibilities of a controller in an AngularJS application include:

* Fetching the right data from the server for the current UI.
* Deciding which parts of that data to show to the user.
* Presentation logic, such as how to display elements, which parts of the UI to show, how to style them, etc.
* User interactions, such as what happens when a user clicks something or how a text input should be validated.

An AngularJS controller is almost always directly linked to a view or HTML. We will never have a controller that is not used in the UI (that kind of business logic goes into services). It acts as the gateway between our model, which is the data that drives our application, and the view, which is what the user sees and interacts with.

We also introduce a new directive, **ng-controller.** This is used to tell AngularJS to go instantiate an instance of the controller with the given name, and attach it to the DOM element. This directive allows us to associate an instance of a controller with a UI element.

The HTML connects parts of the DOM to controllers, functions, and variables, and not the other way around. This is one of the core principles of AngularJS at play here. An AngularJS application is a data-driven app.

Anything that the user needs to see, or the HTML needs to use, needs to be defined on this. Anything that the HTML does not directly access should not be put on this, but should rather be saved as local variables in the controller’s scope.

AngularJS creates scopes or context for various elements in the DOM to ensure that there is **no global state** and each element accesses only what is relevant to it. These scopes have a **parent-child relation** by default, which allows children scopes to access functions and controllers from a parent scope.

### $scope Versus controllerAs Syntax

If you used AngularJS prior to 1.2, you might have expected the $scope variable to be injected into the controller. In AngularJS 1.2 and later, there is a new syntax, the controllerAs syntax, which allows us to define the variables on the controller instance using the **this** keyword, and refer to them through the controller from the HTML.

The advantage of this over the earlier syntax is that it makes it explicit in the HTML which variable or function is provided by which controller and which instance of the controller. It becomes immediately obvious because the controller instance is present in the HTML.

Also, as good practice, we avoid referring to the **this** keyword inside the controller, preferring to use a proxy **self** variable, which points to this.

## How AngularJS is working behind the scenes

1. The HTML is loaded. This triggers requests for all the **scripts** that are a part of it.
2. After the entire document has been loaded, AngularJS kicks in and looks for the **ng-app** directive.
3. When it finds the ng-app directive, it looks for and loads the **module** that is specified and attaches it to the element.
4. AngularJS then **traverses** the children DOM elements of the root element with the ng-app and starts looking for directives and bind statements.
5. Each time it hits an **ng-controller** or an **ng-repeat** directive, it creates what we call a **scope** in AngularJS. A scope is the context for that element. The scope dictates what each DOM element has access to in terms of functions, variables, and the like.
6. AngularJS then adds **watchers** and **listeners** on the variables that the HTML accesses, and **keeps track** of the current value of each of them. When that value changes, AngularJS updates the UI immediately.
7. Instead of polling or some other mechanism to check if the data has changed, AngularJS **optimizes** and checks for updates to the UI only on certain events, which can cause a change in the data or the model underneath.

## More directives

There are two directives in AngularJS that deal with hiding and showing HTML elements: ng-show and ng-hide. They inspect a variable and, depending on the truthiness of its value, show or hide elements in the UI, respectively. **AngularJS treats true, nonempty strings, nonzero numbers, and nonnull JS objects as truthy.**

The ng-if removes or recreates a portion of the DOM tree based on an {expression}. If the expression assigned to ngIf evaluates to a false value then the element is removed from the DOM, otherwise a clone of the element is reinserted into the DOM.

The ng-class directive is used to selectively apply and remove CSS classes from elements. It can take strings or objects as values. If it is a **string**, it simply applies the CSS classes directly. If it is an **object**, AngularJS takes a look at each key of the object, and depending on whether the value for that key is true or false, applies or removes the CSS class.

Notice also that ng-bind, ng-show, and most of these directives can directly refer to a variable on the controller or call a function to get the value.

The ng-click directive evaluates any expression passed to it when the button is clicked.

### AngularJS $watch() , $digest() and $apply()

The AngularJS $scope functions $watch(), $digest() and $apply() are some of the central functions in AngularJS. Understanding $watch(), $digest() and $apply() is essential in order to understand AngularJS.

When you create a data binding from somewhere in your view to a variable on the $scope object, AngularJS creates a "watch" internally. A watch means that AngularJS watches changes in the variable on the $scope object. The framework is "watching" the variable. Watches are created using the $scope.$watch() function.

At key points in your application AngularJS calls the $scope.$digest() function. This function iterates through all watches and checks if any of the watched variables have changed. If a watched variable has changed, a corresponding listener function is called. The listener function does whatever work it needs to do, for instance changing an HTML text to reflect the new value of the watched variable. Thus, the $digest() function is what triggers the data binding to update.

Most of the time AngularJS will call the $scope.$watch() and $scope.$digest() functions for you, but in some situations you may have to call them yourself. Therefore it is really good to know how they work. (AngularJS $watch() , $digest() and $apply(), 2015)

The ng-repeat directive is one of the most versatile directives in AngularJS, and can be used for a whole variety of situations and requirements. It can be used to show an array of elements in the HTML or to show all the keys and values of an object, sorted in a case-sensitive, alphabetic order.

Helper Variables in ng-repeat:

* **$first**, **$middle**, and **$last** are Boolean values that tell us whether that particular element is the first, between the first and last, or the last element in the array or object.
* **$index** gives us the index or position of the item in the array.
* **$odd** and **$even** tell us if the item is in an index that is odd or even (we could use this for conditional styling of elements, or other conditions we might have in our application).

## Forms

### One/two-way data-binding

We saw the **ng-bind** directive, or its equivalent double-**curly {{ }}** notation, which allowed us to take the data from our controllers and display it in the UI. That gives us our one-way data-binding.

Forms are a staple of web applications, and AngularJS provides the **ng-model** directive for us to deal with inputs and two-way data-binding.

When we need to update the UI, we just update the model fields in the controller. When we need to get the latest and greatest value, we just grab it from the controller. Again, this is the AngularJS way.

### Working with Forms

A form submit event can be triggered in multiple ways: clicking the Submit button, or hitting Enter on a text field. The **ng-submit** gets triggered on all those events, whereas the **ng-click** will only be triggered when the user clicks the button.

When you use **ng-model**, AngularJS automatically creates the objects and keys necessary in the chain to instantiate a data-binding connection. The first letter typed causes the user object to be created, and the value to be assigned to the correct field in it.

When designing your forms and deciding which fields to bind the ng-model to, you should always consider what format you need the data in.

### Form Validation and States

When you use forms (and **give them names**), AngularJS creates a FormController that holds the current state of the form as well as some helper methods. You can access the FormController for a form using the form’s name.

Things that are exposed as the **state** and kept up to date with data-binding are:

* **$error** This field on the form houses all the individual fields and the errors on each form element. We will talk more about this in the following section.
* **$valid** Tells whether an item is currently valid based on the rules you placed.
* **$invalid** Tells whether an item is currently invalid based on the rules you placed.
* **$pristine** True if the form/input has not been used yet.
* **$dirty** True if the form/input has been used.
* **$touched** True if the input has been blurred.

Each of the states mentioned (except $error) are Booleans and can be used to conditionally hide, show, disable, or enable HTML elements in the UI. As the user types or modifies the form, the values are updated as long as you are leveraging ngmodel and the form name.

Built-in **validations** that AngularJS offers:

* **required** this ensures that the field is required, and the field is marked invalid until it is

filled out.

* **ng-required** Unlike required, which marks a field as always required, the ng-required directive allows us to conditionally mark an input field as required based on a Boolean condition in the controller.
* **ng-minlength** We can set the minimum length of the value in the input field with this directive.
* **ng-maxlength** We can set the maximum length of the value in the input field with this directive.
* **ng-pattern** The validity of an input field can be checked against the regular expression pattern specified as part of this directive.
* **type="email"** Text input with built-in email validation.
* **type="number"** Text input with number validation. Can also have additional attributes for min and max values of the number itself.
* **type="date"** If the browser supports it, shows an HTML datepicker. Otherwise, defaults to a text input. The ngmodel that this binds to will be a date object. This expects the date to be in yyyy-mm-dd format (e.g., 2009-10-24).
* **type="url"** Text input with URL validation.

Need to show the user that a field is required? Then when the user starts typing, show the minimum length, and then finally show a message when he exceeds the maximum length. All these kinds of conditional messages can be shown with the AngularJS validators.

AngularJS takes the name of the validator (number, maxlength, pattern, etc.) and depending on whether or not that particular validator has been satisfied, adds the ng-valid-validator\_name or ng-invalid-validator\_name class, respectively.

### Nested Forms with ng-form

AngularJS provides an **ng-form** directive, which acts similar to form but allows nesting, so that we can accomplish the requirement of grouping related form fields under sections.

Now we can have substate within our form, evaluate quickly if each section is valid, and leverage the same binding and form states that we have looked at so far. A quick highlight of the features:

* A subform using the ng-form directive. We can give this a name to identify and grab the state of the subform.
* The state of the subform can be accessed directly (childForm.$invalid) or through the parent form (myForm.profile.$invalid).
* Individual elements of the form can be accessed as normal (childForm.firstName.$error.required).
* Subforms and nested forms still affect the outer form (the myForm.$invalid can be true because of the use of the required tags).

You could have subforms and groupings that have their own way of checking and deciding validity, and ng-form allows you to model that grouping in your HTML.

### ngMessages

The ngMessages directive listens on a key/value collection which is set on the ngMessages attribute. Since the ngModel directive exposes an $error object, this error object can be used with ngMessages to **display control error messages** in an easier way than with just regular angular template directives.

## Working with Filters

AngularJS filters are used to process data and format values to present to the user. They are applied on expressions in our HTML, or directly on data in our controllers and services.

* currency - formats a given number as currency with the commas, decimals,and currency symbol added as needed. The filter takes an optional currency symbol as the second argument; if none exists, it takes the default symbol for the current browser.
* number - takes a number and converts it to a human-readable string with comma separation. The number filter also takes an optional decimal size that tells it how many digits to keep after the decimal point.
* lowercase - A very simple string filter that takes any string and converts all the characters to lowercase.
* uppercase - A very simple string filter that takes any string and converts all the characters to uppercase.
* json – is a great tool for debugging, or for any time we need to display the contents of a JSON object or an array in the UI. It takes a JSON object or array (or even primitives) and displays it as a string in the UI.
* date – is a customizable and powerful filter that takes a date object or a long timestamp and displays it as a human-readable string in the UI. It can take a user-defined format or one of the built-in short, medium, or long formats.
* orderBy - allows us to take an array and order it by a predicate expression (or a series of predicate expressons). It also takes a second optional Boolean argument, which decides whether or not the sorted array is reversed.

Any filter, whether built-in or our own, **can be injected** into any service or controller by affixing the word “Filter” at the end of the name of the filter, and asking it to be injected.

## Services.Factories.Providers

Services are used to encapsulate functionality that you want to reuse in an application but that don’t fit neatly into the Model-View-Controller pattern. Services are commonly used to implement **cross-cutting concerns**, which is a catchall term for any functionality that is affected by more than one component or affects more than one component. Typical examples are logging, security, and networking. They are not part of the model (unless your business is logging, security, or networking), they don’t belong to the controllers because they don’t respond to user interaction or perform operations on the model, and they are not part of the view or a directive because they don’t present the model to the user.

Hold internal state as local variables inside the service. This is important because in a Single Page Application where controllers can get created and destroyed, the service can act as an application-level store.

|  |  |  |
| --- | --- | --- |
| Why | When | |
| Services allow you to package up reusable functionality so that it can be used across the application. Modules allow you to package up reusable functionality so that it can be used across multiple applications. | Create a service when functionality doesn’t fit into one of the other MVC building blocks and is a cross-cutting concern. Create a module to package functionality so that it can be used in multiple applications. | |
| Problem | | Solution |
| Create a service by defining an object | | Use the Module.factory method. |
| Create a service by defining a constructor. | | Use the Module.service method. |
| Create a service that can be configured via a provider. | | Use the Module.provider method. |

The AngularJS Module defines three methods for defining services: **factory**, **service**, and **provider**. The result of using these methods is the same—a service object that provides functionality that can be used throughout the AngularJS application—but the way that the service object is created and managed by each method is slightly different.

**The functional way using the factory method, the OO style using the service method, and the configurable version using the provider method.**

### Using the Factory Method

The simplest way to create a service is to use the Module.factory method, passing as arguments the name of the service and a factory function that returns the service object.

The object returned by the factory function is the service object and will be used by AngularJS whenever the service is requested. The factory function is called only once because the object it creates and returns is used whenever the service is required within the application. A common error is to assume that each consumer of the service will receive a different service object and assume that variables like counters will be modified by only one AngularJS component.

Be careful not to reuse the name of a service. If you do, your service will replace the existing one; services objects are singletons and any changes you make once the application has started will affect all of the components that are consuming the service—something that often causes unexpected behaviors.

You should use module.factory() to define your services if:

* You follow a functional style of programming
* You prefer to return functions and objects

We **return an object**, which becomes the **API** for the service. We also declare some **local** , these are **private** to the service, and cannot be accessed directly.

Benefits:

* The consumers of the service have no insight into or dependency on its implementation.
* The functionality of an existing service can be modified or disabled.
* The service functionality can be isolated and tested separately from the rest of the application.

**A factory is an injectable function.** A factory is a lot like a service in the sense that it is a singleton and dependencies can be specified in the function. The difference between a factory and a service is that a factory injects a plain function so AngularJS will call the function and a service injects a constructor. A constructor creates a new object so new is called on a service and with a factory you can let the function return anything you want. As you will see later on, a factory is a provider with only a $get method.

### Using the Service Method

When AngularJS needs to satisfy a dependency for a service defined by the factory method, it simply uses the object returned by the factory function, but for a service defined with the **service method**, AngularJS uses the object returned by the factory function as a **constructor** and uses the JavaScript **new** keyword to create the service object.

The service definition function is now **a JavaScript class function**. It doesn’t return anything. The service defines the public API by defining methods on its instance (using the **this** keyword). **Private state** for the service is still defined as **local variables** inside the function definition. AngularJS will perform new ItemService() (with possible dependencies injected in) and then return that instance to all functions that depend on that service.

**A service is an injectable constructor.** If you want you can specify the dependencies that you need in the function. A service is a singleton and will only be created once by AngularJS. Services are a great way for communicating between controllers like sharing data.

### Using the Provider Method

The Module.provider method allows you to take more control over the way that a service object is created or configured.

AngularJS makes the provider object available for dependency injection, using the name of the service combined with the word Provider. The most common way to obtain and use the provider object is in a function passed to the Module.config method, which will be executed when AngularJS has loaded all of the modules in the application.

The config function executes before the AngularJS app executes. Note that the provider does not use the same notation as factory and service. It doesn’t take an array as the second argument because **providers cannot have dependencies on other services**.

A provider is the most sophisticated method of all the providers. It allows you to have a complex creation function and configuration options. **A provider is actually a configurable factory. The provider accepts an object or a constructor.** [exemple](http://blog.xebia.com/differences-between-providers-in-angularjs/)

### Global Object Services

* **$anchorScroll** - Scrolls the browser window to a specified anchor
* **$document** - Provides a jqLite object that contains the DOM window.document object
* **$interval** - Provides an enhanced wrapper around the window.setInterval function
* **$location** - Provides access to the URL
* **$log** - Provides a wrapper around the console object
* **$timeout** -Provides an enhanced wrapper around the window.setITimeout function
* **$window** - Provides a reference to the DOM window object

These functions work in the same way, in that they defer the execution of a function for a specified period of time. The difference is that the **$timeout** service delays and executes the function only once, whereas **$interval** does so periodically.

### Exceptions

The **$exceptionHandler** service deals only with uncaught exceptions. You can catch an exception using a JavaScript try...catch block, and it will not be handled by the service. By default, it simply writes details of exceptions to the JavaScript console and allows the application to continue running (if that’s possible).

You can replace the default behavior with something much more complex, but I recommend caution. Error handling code needs to be bullet-proof because if it contains bugs, then you won’t see the real problems in the application. The simplest error handling is generally best.

### Working with Dangerous Data

The types of attack are endless, but one common thread is injecting malicious content into the application through forms, either so it will be displayed back to the attacker or so it will be presented to other users. AngularJS has some nice built-in support for mitigating the risk of this kind of attack.

* **$sce** Removes dangerous elements and attributes from HTML.
* **$sanitize** Replaces dangerous characters in HTML strings with their escaped counterparts.

AngularJS uses a feature called ***strict contextual escaping*** *(SCE)* that prevents unsafe values from being expressedthrough data bindings. This feature is enabled by default.

The ng-bind-html directive, which allows you to specify that a data value is trusted and should be displayed without being escaped. The ng-bind-html directive depends on the ngSanitize module.

There are some—incredibly rare—circumstances under which you may need to display potentially dangerous content without escaping or sanitizing it. You can declare content to be trustworthy by using the **$sce** service. The $sce service object defines the trustAsHtml method, which returns a value that will be displayed with the SCE process being applied.

### Services for Ajax and Promises

The **$http** service is used to make and process Ajax requests, which are standard HTTP requests that are performed asynchronously. AngularJS uses a JavaScript pattern called **promises** to **represent the result from an asynchronous operation**, such as an Ajax request. [.then() or .success()](http://www.peterbe.com/plog/promises-with-$http)

The $http service is a core Angular service that facilitates communication with the remote HTTP servers via the browser's [XMLHttpRequest](https://developer.mozilla.org/en/xmlhttprequest) object or via [JSONP](http://en.wikipedia.org/wiki/JSONP). For unit testing applications that use $http service, see [$httpBackend mock](https://docs.angularjs.org/api/ngMock/service/$httpBackend). For a higher level of abstraction, please check out the [$resource](https://docs.angularjs.org/api/ngResource/service/$resource) service ( A factory which creates a resource object that lets you interact with [RESTful](http://en.wikipedia.org/wiki/Representational_State_Transfer) server-side data sources).

The $http API is based on the [deferred/promise APIs](https://docs.angularjs.org/api/ng/service/$q) exposed by the **$q** service (A service that helps you run functions asynchronously, and use their return values (or exceptions) when they are done processing).

A **promise** is an object that defines methods that you can use to register functions that will be invoked when the operation is complete. There are two objects required for a promise: a **promise** object, which is used to **receive** notifications about the future outcome, and a **deferred** object, which is used to **send** the notifications.

AngularJS provides the **$q** service for obtaining and managing promises. You should expose the promise object only to other parts of the application and keep the deferred object out of reach of other components, which would otherwise be able to resolve or reject the promise unexpectedly.

Promises represent a **single instance** of an activity, and once they are resolved or rejected, promises cannot be used again.

A **deferred** represents the result of an asynchronic operation. It exposes an interface that can be used for signaling the state and the result of the operation it represents. It also provides a way to get the associated promise instance.

A new instance of deferred is constructed by calling $q.defer(). The purpose of the deferred object is to expose the associated Promise instance as well as APIs that can be used for signaling the successful or unsuccessful completion, as well as the status of the task.

Methods

* **resolve(value)** – resolves the derived promise with the value. If the value is a rejection constructed via $q.reject, the promise will be rejected instead.
* **reject(reason)** – rejects the derived promise with the reason. This is equivalent to resolving it with a rejection constructed via$q.reject.
* **notify(value)** - provides updates on the status of the promise's execution. This may be called multiple times before the promise is either resolved or rejected.

Properties

* **promise** – {Promise} – promise object associated with this deferred.

**While deferred has methods for changing the state of an operation, a promise exposes only methods needed to handle and figure out the state**, but not methods that can change the state. This is why in a function, returning a promise and not a deferred is a good practice. This prevents from external code to interfere the progress or the state of an operation. [example](http://www.webdeveasy.com/javascript-promises-and-angularjs-q-service/)

A new promise instance is created when a deferred instance is created and can be retrieved by calling deferred.promise. The purpose of the promise object is to allow for interested parties to get access to the result of the deferred task when it completes.

Methods

* **then(successCallback, errorCallback, notifyCallback)** – regardless of when the promise was or will be resolved or rejected,then calls one of the success or error callbacks asynchronously as soon as the result is available. The callbacks are called with a single argument: the result or rejection reason. Additionally, the notify callback may be called zero or more times to provide a progress indication, before the promise is resolved or rejected.

This method *returns a new promise* which is resolved or rejected via the return value of the successCallback, errorCallback(unless that value is a promise, in which case it is resolved with the value which is resolved in that promise using [promise chaining](http://www.html5rocks.com/en/tutorials/es6/promises/#toc-promises-queues)). It also notifies via the return value of the notifyCallback method. The promise cannot be resolved or rejected from the notifyCallback method.

* catch(errorCallback) – shorthand for promise.then(null, errorCallback)
* finally(callback, notifyCallback) – allows you to observe either the fulfillment or rejection of a promise, but to do so without modifying the final value. This is useful to release resources or do some clean-up that needs to be done whether the promise was rejected or resolved.

The **$q.all()** method takes either an object or an array of promises and **waits for all of them to resolve() or one of them to reject()**and then executes the provided callback function. The values returned from the resolve function are provided depending on the way you give the promises to all(). [example](http://www.martin-brennan.com/using-q-all-to-resolve-multiple-promises/)

The only time you should ever use deferreds is when you don't already have a promise available (or in a few other special situations). And even then, there are preferred ways to create promises these days. [So when should deferred be used?](https://github.com/petkaantonov/bluebird/wiki/Promise-anti-patterns#the-deferred-anti-pattern)

### Services for Views

At the heart of the functionality provided by the **$route** service is a set of mappings between URLs and view file names, known as URL routes or just routes. When the value returned by the **$location.path** method matches one of the mappings, the corresponding view file will be loaded and displayed. The mappings are defined using the provider for the $route service, **$routeProvider**.

Routing works when the application changes the URL, but it doesn’t work if the user changes it; the browser takes any URL that the user enters as being a literal request for a file and tries to request the corresponding content from the server.

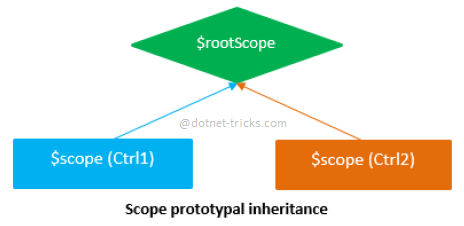
There are two kinds of route parameters: **conservative** and **eager**. A conservative route parameter will match one segment, and an eager one will match as many segments as possible.

### Managing Injection

The **$injector** service is responsible for determining the dependencies that a function declares and resolving those dependencies. I can obtain the service objects that I need through the $injector.get method, which takes the name of a service and returns the service object.

The arguments to the invoke method are the function that will invoked, the value for this, and an object whose properties correspond to the function arguments that are not service dependencies.

### Understanding AngularJS $rootScope and $scope



A **$scope** is a JavaScript object which is used for communication between controller and view. Basically, $scope binds a view (DOM element) to the viewmodel and functions defined in a controller.

The **$rootScope** is the top-most scope. An app can have **only one** $rootScope which will be shared among all the components of an app. Hence it acts like a global variable. All other $scopes are children of the $rootScope. [example](http://www.dotnet-tricks.com/Tutorial/angularjs/UVDE100914-Understanding-AngularJS-$rootScope-and-$scope.html)

## Routing Using ngRoute

Routing in AngularJS is declarative, so all routes are defined in a single configuration section where we can specify what the route is and what AngularJS needs to do when that route is encountered.

First of all, when we talk about routing in a Single-Page Application, we are not talking standard URLs, but what we call **hashbang URLs**. This is because the browser treats URLs with hashes differently than URLs without.

When the hash fragment changes, the JavaScript responds and loads only the relevant data and HTML instead of reloading the entire HTML. This makes the application faster and snappier because less data is fetched from the server. It takes care of the browser history, so you can actually use back and forward buttons in your browser to navigate within the application.

Mark which section of the page AngularJS should change when the route changes using the **ng-view** directive in the HTML. Define our routes in the **config** section using the **$routeProvider** service. $routeProvider allows us to define our routes in one place using the when() function

### Routing Options

The $routeProvider.when function takes a URL or URL regular expression as the first argument, and the route configuration object as the second. The syntax is as follows:

$routeProvider.when(**url**, {

**template**: string,

**templateUrl**: string,

**controller**: string, function or array,

**controllerAs**: string,

**resolve**: object<key, function>

});

At a conceptual level, **resolves** are a way of executing and finishing asynchronous tasks before a particular route is loaded. This is a great way to check if the user is logged in and has authorization and permissions.

Because of the resolve function, **AngularJS ensures that the route does not load until all the resolve functions are finished executing**. If there are multiple resolve keys that make asynchronous calls, AngularJS executes all of them in parallel and waits for all of them to finish executing before loading the page. If any of the resolves encounter an error or any of the promises returned are rejected (is a failure), AngularJS doesn’t load the route.

AngularJS still loads and caches the template if any of the resolves fail, but the controller associated with the route isn’t loaded and the HTML doesn’t make it into the ng-view. Each key can directly be injected into the controller by adding it as a dependency. This is over and above any AngularJS service dependency we might have.

### Events

**$routeChangeStart** - Broadcasted before a route change. At this point the route services starts resolving all of the dependencies needed for the route change to occur. Once all of the dependencies are resolved $routeChangeSuccess is fired.

**$routeChangeSuccess** - Broadcasted after a route change has happened successfully. The resolve dependencies are now available in the current.locals property.

**$routeChangeError** - Broadcasted if any of the resolve promises are rejected.

**$routeUpdate** - The reloadOnSearch property has been set to false, and we are reusing the same instance of the Controller.

### Using the $routeParams Service

A controller and a route should be able to independently bootstrap themselves. URL parameters don’t have to be parsed from the URL, but can directly be accessed from a convenient service that AngularJS provides, called **$routeParams.**

The $routeParams service is responsible for reading the URL, parsing it, and finding all these variables and making them accessible to the controller in a nice way.

### Things to Watch Out For

* Empty templates
* Resolve injection into controller
* $routeParam variable type
* One ng-view per application

## Alternatives: ui-router

We saw that we can only have one ng-view per application. The ui-router does away with that restriction. It uses the concepts of **states**, instead of routes. The ui-router provides a directive called **ui-sref** that allows us to navigate to states. We use the **$stateProvider** to define our states. We can have **multiple** named **ui-views** in our application. Each of these can respond to URL and state changes differently, which allows us to separate and modularize our code even further.

ui-router is **state-oriented**, and by default does not modify URLs. We need to specify the URL for each state individually.

There are three main ways to activate a state:

1. Call **$state.go().** High-level convenience method.
2. Click a link containing the **ui-sref** directive.
3. Navigate to the url associated with the state.

**$state** service is responsible for representing states as well as transitioning between them. It also provides interfaces to ask for current state or even states you're coming from.

### Events

**$stateChangeError** - Fired when an error occurs during transition. It's important to note that if you have any errors in your resolve functions (javascript errors, non-existent services, etc) they will not throw traditionally. You must listen for this $stateChangeError event to catch ALL errors.

**$stateChangeStart** - Fired when the state transition begins. You can use event.preventDefault() to prevent the transition from happening and then the transition promise will be rejected with a 'transition prevented' value.

**$stateChangeSuccess** - Fired once the state transition is complete.

**$stateNotFound**  - Fired when a requested state cannot be found using the provided state name during transition. The event is broadcast allowing any handlers a single chance to deal with the error (usually by lazy-loading the unfound state). A special unfoundState object is passed to the listener handler.

## Directive - declarative, data driven and conversational

Angular directives are declarative, meaning we write them into the HTML elements themselves. Declarative programming means that instead of telling an object how to behave (imperative programming), we describe what an object is. So, where in jQuery we might grab an element and apply certain properties or behaviors to it, with Angular we label that element as a type of directive, and, elsewhere, maintain code that defines what properties and behaviors make up that type of object.

Angular directives are built from the ground up with the philosophy of being data driven. The scope and attribute objects accessible to each directive form the skeleton around which the rest of a directive is built and can be monitored for changes both within the DOM as well as the rest of your JavaScript code.

Angular directives have access to all DOM events as well but $scope also provides an additional messaging system that functions only along the scope tree. The $emit and $broadcast methods serve to send messages up and down the scope tree respectively, and like DOM events, allow directives to subscribe to changes or events within other parts of the application, while still remaining modular and uncoupled from the specific logic used to implement those changes.

**$emit(name, args)** - dispatches an event name upwards through the scope hierarchy notifying the registered $rootScope, the event life cycle starts at the scope on which $emit was called.

**$broadcast(name, args)** - dispatches an event name downwards to all child scopes (and their children) notifying the registered $rootScope, the event life cycle starts at the scope on which $broadcast was called.

Directives are of two major types in AngularJS (though they can be subclassified further and further):

**Behavior modifiers** These types of directives work on existing UI and HTML snippets, and just add or modify the existing behavior of what the UI does. Examples of such directives would be ng-show (which hides or shows an existing element based on a condition).

**Reusable components** These types of directives are the more common variety, in which the directive creates a whole new HTML structure. These directives have some rendering logic (how and what should it display) and some business logic (where should it get the data, what happens when the user interacts with it) attached to it.

The main intentions of a directive are:

1. To make our intention declarative by specifying in the HTML what something is or tries to do.
2. To make something reusable so the same functionality can be achieved easily without copying and pasting the code
3. To achieve abstraction in the sense that the user of the directive doesn’t need to know or understand how something is performed, but only cares about the end result. The corollary of this is that the underlying implementation can be changed without having to change every single usage.

### Creating a Directive

Creating a directive is just like creating controllers, services, and filters in that the AngularJS module function allows us to create a directive by name. The first argument to the function is the name of the directive, and the second argument is the standard Dependency Injection array syntax, with the last element in the array being our directive function The function isn't actually the directive definition in itself, but rather a **factory**, used to create the directive. What this means is that the first time Angular needs to initialize your directive, it executes this factory function, then stores the result for all the future instances of the directive.

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

.**directive**('myAwesomeDirective', ['api', function(api) {

//Do any one-time directive initialization work here

**return function($scope, $element, $attrs)** {

//Do directive work that needs to be applied to each instance here

};

}]);

There are two ways to initialize a directive within the factory function. The first is to return our linking function, as we did earlier, and the second is to use this definition object to provide more fine-grained control over the way our directive functions.

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

.directive('myAwesomeDirective', ['api', function(api) {

//Do any one-time directive initialization work here

return {

**priority** : 10,

**terminal** : false,

**template**: '<div><h3>{{title}}</h3></div>',

**templateUrl** : 'myDirective.html',

**restrict**: 'AE',

**replace** : true,

**compile** : function (element, attributes, transclude) {},

**link** : function ($scope, $element, $attrs) {},

**scope** : true,

**controller** : function ($scope, $element, $attrs) {},

**require** : 'myAwesomeDirective',

**transclude** : true

};

}]);

**Priority** option allows us to specify in what order directives should be executed, if there are multiple directives on the same node. The default is 0, and unlike many languages, higher numbers go first here. There is no specification for what order directives of the same priority will execute in.

**Terminal** dictates whether or not directive execution should stop after the priority level. It's important to note, however, that this does not necessarily mean stop after this directive itself.

If your directive provides a custom HTML structure, you can use the **template** or **templateUrl** property to define it. Do note that the compile/link process for this directive will be suspended until the template is loaded, so if your custom HTML is minimal, it's usually more efficient to provide it inline.

The **restrict** keyword defines how someone using the directive in their code might use it. The default way of using directives is via attributes of existing elements. When we create our directive, we have control in deciding how it’s used. The letter **A** in the value for restrict specifies that the directive can be used as an attribute on existing HTML elements. The letter **E** in the value for restrict specifies that the directive can be used as a new HTML element. The letter **C** in the value for restrict specifies that the directive can be used as a class name in existing HTML elements. The letter **M** in the value for restrict specifies that the directive can be used as HTML comments. This was previously necessary for directives that needed to encompass multiple elements, like multiple rows in tables, etc.

• Class-based directives are ideal for rendering-related work (like the ng-cloak directive that hides and shows elements, or image loading directives).

• Element directives are recommended if we are creating entirely new HTML content.

• Attribute directives are usually preferred for behavior modifiers (like ng-show, ngclass, and so on).

Use of the **replace** property specifies whether the whole element should be replaced with the template, or if the template HTML should just replace the element's inner HTML. If you want to replace the original element, your template must have only one root node.

**Compile** is performing any tasks that require restructuring the DOM (and possibly adding other directives) regardless of the specific scope, and the **link** function is attaching a scope to that compiled element - it defines APIs and functions that are necessary for the directive, in addition to manipulating and working with the DOM. AngularJS executes the link function for each instance of the directive, so each instance can get its own, fully contained business logic while not affecting any other instance of the directive.

If you set a value for the compile property, Angular will ignore the link property. If you need to use both, you can return an object from the compile function, with **pre** set to your compile function, and **post** set to your link function.

angular.directive('myAwesomeDirective', function () {

return {

compile : function (tElement, tAttrs, transclude) {

**pre** : function compile ($scope, $element, $attrs) {},

**post** : function link ($scope, $element, $attrs) {}

return{

...//current pre and post text

}

}

}

});

The **scope** option can have three types of values.

If left undefined, the scope value is null, which tells Angular to give the directive the same scope as the object its attached to. This is by far the most common case,and is perfect for adding a few new values to watch.

If, however, you want to generate a new scope for your directive, there are two ways you can do so. First, simply set the scope parameter to true, which will create a new scope for the directive, but still inherit from it's parent. Secondly, if you want to isolate your directive from the rest of your application, you can create an aptly named isolate scope. This scope can be helpful in ensuring modularity and preventing accidental changes to data outside of your directive caused by shared properties or methods. To create an isolate scope, simply pass in an object hash to the scope parameter. If it's empty, no values will be inherited and your scope will be completely isolated.

If you set scope : true for multiple directives on the same node, they'll all share the same new scope. While this is usually fine, do note that only one directive on a node can request an isolate scope, and all other directives will share that scope.

In the object, we can identify what attributes are to be specified in the HTML when the directive is used, and the types of values that will be passed in to the directive. In particular, we can specify three types of values that can be passed in:

The **=** sign specifies that the value of the attribute in HTML is to be treated as a **JSON object**, which will be bound to the scope of the directive so that any changes done in the parent scope will be automatically available in the directive.

The **@** sign specifies that the value of the attribute in HTML is to be treated as a **string**, which may or may not have AngularJS binding expressions ({{ }}). The value is to be calculated and the final value is to be assigned to the directive’s scope. Any changes in the value will also be available in the directive.

The **&** sign specifies that the value of the attribute in HTML is a **function** in some controller whose reference needs to be available to the directive. Directive can then trigger the function whenever it needs to.

The **controller** function can store many of the same properties or methods that you might normally attach to the scope discussed earlier, however, if they are attached to the controller itself, they can be shared with other directives in the DOM tree. This sharing is done via the **require** property, which tells Angular to grab the instance of one directive's controller and make it available to another directive. If you need access to multiple controllers, you can also pass in an array to the require property, and likewise the fourth parameter of your link function will be an array of those controllers.

**Transclusion** (unofficial interpretation of the word is translated-inclusion) provides the ability to have an isolate scope as we discussed earlier, and still have access to the parent scope's properties for internal content. Use the standard ng-transclude directive when you want the content unaltered, and controller plus $transclude if you need to manipulate it first.

### What is a directive? What is a controller?

You might be asking at this point: What is ngModel? What is ng-model? What is NgModelController!?

ngModel is the name of the directive, and ng-model is how that directive is referred to from HTML. And finally, ngModelController is the directive controller.

The thing to understand here is that a directive itself is self-contained. To have a directive do something interesting aside from just UI (like affect state in your app), it needs to interact through controllers. Controllers are the communication channel into and out of your otherwise self-contained directive.

So, most directives have a controller and/or use a controller of some other directive. Without a controller, your directive is purely presentational because without a controller, your directive can’t affect any state or interact with any other directives.

In our case, we are using the built-in NgModelController to handle setting/saving our model value. The ng-model directive itself does nothing (well, almost nothing), it only exists for the controller NgModelController.

When we require: 'ngModel', we are really saying “give me the controller for the directive ng-model”.

A good example about two way binding in custom directives is given [here](http://radify.io/blog/understanding-ngmodelcontroller-by-example-part-1/).

NgModel – example and explanations [here](http://www.chroder.com/2014/02/01/using-ngmodelcontroller-with-custom-directives/).

## Bibliography

*AngularJS $watch() , $digest() and $apply()*. (2015, 11 13). Retrieved from Jenkov.com: http://tutorials.jenkov.com/angularjs/watch-digest-apply.html

*Cascading Style Sheets*. (2015, 11 6). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Cascading\_Style\_Sheets

*HTML*. (2015, 11 06). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/HTML

*JavaScript*. (2015, 11 6). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/JavaScript

*jQuery*. (2015, 11 6). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/JQuery

Michael S. Mikowski, J. C. (2014). *Single Page Web Applications.* Shelter Island, NY 11964: Manning.

*ngApp*. (2015, 11 13). Retrieved from docs.angularjs.org: https://docs.angularjs.org/api/ng/directive/ngApp

*Single-page application*. (2015, 11 9). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Single-page\_application