AngularJS Tutorial: A Comprehensive 10,000 Word Guide

Todd Motto
Todd Motto is a Google Developer Expert, Lead Front-End Engineer at Appsbroker, conference speaker and open source evangelist. His award-winning Javascript tool Conditionizr was .NET Magazine's Open Source Project of the Year finalist.

1 Intro to AngularJS

Angular is a client-side MVW JavaScript framework for writing compelling web applications. It's built and maintained by Google and offers a futuristic spin on the web and its upcoming features and standards.

MVW stands for Model-View-Whatever, which gives us flexibility over design patterns when developing applications. We might choose an MVC (Model-View-Controller) or MVVM (Model-View-ViewModel) approach.

This tutorial serves as an ultimate resource point to begin learning AngularJS, the concepts and APIs behind it and help you deliver fantastic web applications the modern way.

AngularJS promotes itself as a framework for enhancing HTML, it brings concepts from various programming languages, both JavaScript and server-side languages and makes HTML the dynamic language too. This means we get a fully data-driven approach to developing applications without needing to refresh Models, update the DOM and other time-consuming tasks such as browser bug fixes and inconsistencies. We focus on the data, and the data takes care of our HTML, leaving us to programming our application.

2 Engineering concepts in JavaScript frameworks

AngularJS takes a different stance on how it delivers data-binding and other engineering concepts than frameworks such as Backbone.js and Ember.js. We stick with the HTML we already know and love, letting Angular hijack it and enhances it. Angular keeps the DOM updated with any Model changes, which live in pure JavaScript Objects for data-binding purposes. Once a Model value is updated, Angular updates its Objects which become a source of truth for the application's state.

2.1 MVC and MVVM

If you're used to building static websites, you're probably familiar with manually creating HTML piece by piece, integrating your "data" and printing the same HTML over and over again. This could be columns for a grid, a navigation structure, a list of links or images and so on. In this instance, you're used to the pain of manually updating HTML for a template when one little thing changes, you've got to update all further uses of the template to keep things consistent. You've also got to stamp the same chunk of HTML for each navigation item, for example.

Take a deep breath, as with Angular we have proper separation of concerns, as well as dynamic HTML. This means our data lives in a Model, our HTML lives as a tiny template to be rendered as a View, and we use a Controller to connect the two, driving Model and View value changes. This means a navigation could be dynamically rendered from a single list element, and automatically repeated for each item in the Model. This is a very basic concept and we'll come onto templating more.

The difference between MVC and MVVM, is that MVVM is specifically targeted at user interface development. The View consists of the presentation layer, the ViewModel contains the presentation logic and the Model contains our business logic and data. MVVM was designed to make two-way data binding easier, and frameworks such as AngularJS thrive from it. We'll be focusing on an MVVM path as Angular has been leaning more towards this design in recent years.

2.2 Two way data-binding

Two way data-binding is a very simple concept which provides synchronisation between Model and View layers. Model changes propagate to the View, and View changes are instantly reflected back to the Model. This makes the Model the "single source of truth" for maintaining the applications state.

Angular use plain old JavaScript Objects for synchronising Model and View data-bindings, which makes updating both a breeze. Angular parses back to JSON and communicates best with a REST endpoint. This approach makes building front-end applications seamless as all the application state is held on the browser, not delivered in pieces from a server and state becomes lost.

The way we bind these values is through Angular expressions, which take shape as handlebar templates. We can also bind Models using an attribute called `ng-model`. Angular uses custom attributes for various APIs that feed back into the Angular core, we'll learn more about these `ng-*` prefixed attributes as we continue.

2.3 Dependency Injection (DI)

Dependency Injection is a software design pattern that deals with how components get hold of their dependencies. An injection is the passing of a dependency to a dependent Object, these dependencies are often referred to as Services.

In AngularJS, we cleverly use the arguments of a function to declare the dependencies we want, and Angular gives them to us. If we forget to pass in a dependency but reference it where we expect it, the Service will be undefined and result in a compile error inside Angular. But don't worry, Angular throws its own errors and makes them very useful to debug.

2.4 Single Page Applications (SPAs), managing state and Ajax (HTTP)

In a Single Page Application (SPA), either all necessary code (HTML, CSS and JavaScript) is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page, although modern web technologies (such as those included in HTML5) can provide the perception and navigability of separate logical pages in the application. Interaction with the single page application often involves dynamic communication with the web server behind the scenes.

In "older" applications where the server typically managed state, there were disconnections between data the user was seeing and data synchronised on the server. There was also a severe lack of application state in a model, as all data was merged in with HTML templates and very far from dynamic. Servers would render a static template, the user would fill in some information and the browser would post it back, a full page refresh would occur and the backend would update. Any state that wasn't captured was lost, and the browser then has to download all the assets again as the page refreshes.

Times have changed, and the browser is maintaining the state of the application, complex logic and client-side MVW frameworks have fast and furiously increased in popularity. It turned out these server-side practices for managing data fit really well in the browser. AngularJS (as well as other JavaScript frameworks) manage state entirely in the browser and communicate changes when we want them to via Ajax (HTTP) using GET, POST, PUT and DELETE methods, typically talking to a REST endpoint backend. The beauty of this is REST endpoints can be front-end independent, and the front-end can be back-end independent. We can serve the same endpoints to a mobile application which may have a different front-end to a web application. Being back-end independent gives us massive flexibility as to the backend, we only care about the JSON data coming back, be it from Java, .NET, PHP or any other server-side language.

2.5 Application structure

Amongst Angular's many APIs, we're given application structure in a consistent fashion, which means generally our applications are all built in a similar way and developers can quickly jump onto projects if necessary. It also creates predictable APIs and expected debugging processes which can enhance development time and rapid prototyping. Angular is structured around "testability" to be as easy to test and develop with as possible.

Let's get learning!

3 Modules

Every application in Angular is created using modules. A module can have dependencies of other modules, or be a single module all by itself. These modules act as containers for different sections of an application, making code more reusable and testable. To create a module, we hit the global `angular` Object, the framework's namespace, and use the `module` method.

3.1 Setters

Our application will always have a single app module as we can import other modules, so we'll call the main module `app` when we set it (create it).

```javascript
angular.module('app', []);
```

You'll notice the `[]` as the second argument to the `module` method, this Array contains (or will contain) all other module dependencies we wish to include in this module. Modules can have other modules as dependencies, which have modules that pull in other dependencies. For now, we'll leave it empty.

3.2 Getters

For creating Controllers, Directives, Services and other Angular features, we need to reference an existing module. It's a simple but subtle difference in syntax, in which we omit the second argument.

```javascript
angular.module('app');
angular.module('app');
```

3.3 Module practices

Modules can be stored and referenced by using a variable, but best practice is to use the chaining method provided by the AngularJS team.

Here's an example of storing the module as a variable.

```javascript
var app = angular.module('app', []);
```

From here we could reference the `app` variable to add methods to build the rest of our app, however we should stick to the previously seen chaining methods.

3.4 HTML bootstrapping

To declare where our application sits in the DOM, typically the `<html>` element, we need to bind an `ng-app` attribute with the value of our module. This tells Angular where to bootstrap our application.

```markup
<html ng-app="app">
  <head></head>
  <body></body>
</html>
```

If we're loading our JavaScript files asynchronously then we need to manually bootstrap the application using `angular.bootstrap(document.documentElement, ['app']);`.

4 Understanding $scope

One of the most common concepts you'll come across in any programming language is scope. For instance block scope and function scope. Angular has a concept of scope which sits as one of the main Objects that powers the two way data-binding cycles to maintain application state. `$scope` is a very clever Object that not only sits in our JavaScript to access data and values, it represents those values out in the DOM once Angular renders our app.

Think of `$scope` as an automated bridge between JavaScript and the DOM itself which holds our synchronised data. This allows us to template easier using handlebars syntax inside the HTML we love, and Angular will render out the associated `$scope` values. This creates a binding between JavaScript and the DOM, the `$scope` Object is really a glorified ViewModel.

We only use `$scope` inside Controllers, where we bind data from the Controller to the View.

Here's an example of how we declare some data inside the Controller.

```javascript
$scope.someValue = 'Hello';
```

To render it out in the DOM, we need to connect a Controller to some HTML and tell Angular where to bind the value.

```markup
<div ng-controller="AppCtrl">
  {{ someValue }}
</div>
```

What we're seeing above is Angular's concept of scoping, which abides by some of JavaScript's rules of lexical scoping. Outside the element that a controller is scoped to, the data is out of scope, just like a variable would be out of scope if referenced outside a scope it was available in.

We can bind any values to `$scope` that exist as types in JavaScript. This is how we transfer data from a service which talks to the server and pass it on to our View, the presentational layer.

The more Controllers and data-bindings in Angular we create, the more scopes are created. Understanding the `$scope` hierarchy is also worth noting, this is where the `$rootScope` comes in.

4.1 $rootScope

The `$rootScope` isn't all that different from `$scope`, except it is the very top level `$scope` Object from which all further scopes are created. Once Angular starts rendering your application, it creates a `$rootScope` Object, and all further bindings and logic in your application create new `$scope` Objects which then all become children of the `$rootScope`.

Generally, we don't touch the `$rootScope` that often, but we can keep it in mind for communicating between scopes with data.

5 Controllers

Before using a Controller, we need to know its purpose. An Angular Controller allows us to interact with a View and Model, it's the place where presentational logic can take place to keep the UI bindings in sync with the Model. A Controller's purpose is to drive Model and View changes, in Angular it's a meeting place between our business logic and our presentational logic.

We've already had a glimpse of Controllers above by declaring the `ng-controller` attribute to display `$scope` data. This attribute simply tells Angular where to scope and bind an instance of a Controller and make the Controller's data and methods be available in that DOM scope.

Before a Controller can be used, it needs to be created, riding off the `module` method we previously created using the best practice chaining approach.

```javascript
angular
  .module('app', [])
  .controller('MainCtrl', function () {

});
```

A Controller accepts two arguments, the first the Controller's name, to be referenced elsewhere such as the DOM, and the second a callback function. This callback shouldn't be treated as a callback, however, it's more a declaration of the Controller's body.

To avoid making Angular methods look like callbacks, and to save indenting our code lots, I tend to place my function outside of Angular's syntax and pass any functions I need into it.

```javascript
function MainCtrl () {

}

angular
  .module('app', [])
  .controller('MainCtrl', MainCtrl);
```

This makes things look a lot clearer in terms of visibility and readability, it's important not to get hooked up in Angular's syntax and just focus on writing JavaScript. We can then pass Angular the pieces we need, just like above.

You'll see I've named a function `MainCtrl`, this also helps us with stack traces when debugging, something an anonymous function can't provide us unless we name them - but it's often less clear and easily forgotten.

All further examples will assume a module has been created, so we'll use the getter syntax from now on when showing examples.

5.1 Methods and presentational logic

A Controller's purpose in the application lifecycle is to interpret business logic from a model and turn it into a presentational format. In Angular, this could be done in many ways depending on what our returned data from a particular method is.

All the Controller does is talk to a Service, and pass the data in either the same or a different format across to our View, using the `$scope` Object. Once a View is updated, the Controller logic is also updated and can be passed back to the server using a Service. Before we dive into Services, we'll create some Objects in the Controller and bind them to the `$scope` to get a feel how Controllers really work. We wouldn't actually declare business logic or or data in a Controller, but for brevity we'll add it below, we'll later fetch data from a Service.

javascript

```javascript
function MainCtrl ($scope) {
  $scope.items = [{
    name: 'Scuba Diving Kit',
    id: 7297510
  },{
    name: 'Snorkel',
    id: 0278916
  },{
    name: 'Wet Suit',
    id: 2389017
  },{
    name: 'Beach Towel',
    id: 1000983
  }];
}

angular
  .module('app')
```

Using the `$scope` Object we bind an `Array`. This `Array` is then available in the DOM, where we can pass it off to one of Angular's built-in Directives, `ng-repeat` to loop over the data and create DOM based on the template and data.

```markup
<div ng-controller="MainCtrl">
  <ul>
      <li ng-repeat="item in items">
          {{ item.name }}
      </li>
  </ul>
</div>
```

Check the live example.

5.2 New "controllerAs" syntax

Controllers are classe-like and Angular developers feel using the `$scope` Object didn't treat them very class-like. They advocated the use of the `this` keyword instead of `$scope`. The Angular team introduced this as part of the new `controllerAs` syntax, of which a Controller becomes instantiated as an instance under a variable - much like you could using the `new` keyword against a variable to create a new Object.

The first change for `controllerAs` is dropping `$scope` references for data bound inside the Controller and using `this` instead.

```javascript
function MainCtrl () {
  this.items = [{
    name: 'Scuba Diving Kit',
    id: 7297510
  },{
    name: 'Snorkel',
    id: 0278916
  },{
    name: 'Wet Suit',
    id: 2389017
  },{
    name: 'Beach Towel',
    id: 1000983
  }];
}

angular
  .module('app')
```

The next change is adding the `as` part to where we instantiate the Controller in the DOM, let's use `MainCtrl as main` to create a `main` variable.

This means any data we reference inside the Controller sits under the `main` variable, so `items` in the previous example becomes `main.items`.

```markup
<div ng-controller="MainCtrl as main">
  <ul>
      <li ng-repeat="item in main.items">
          {{ item.name }}
      </li>
  </ul>
</div>
```

This greatly helps us in identifying which Controller a property belongs to, if we were to have multiple or nested Controllers, before `controllerAs` was introduced the variables are free in the scope, whereas under a variable they belong to a particular instance, for example we could have `main.items` and `sub.items` in the same DOM scope. Previously there may have been conflicting names with variables overwriting each other.

Each `$scope` created has a `$parent` Object as scopes prototypically inherit, without `controllerAs` we would need to use `$parent` references to use any methods in the parent scope, and `$parent.$parent` for another child, and so on. This led to "magic number" style code which `controllerAs` fixes brilliantly - we just reference the variable instance.

> Further Reading:
>
> 1. Digging into Angular's "controller as" syntax
> 2. AngularJS's "controller as" and the "vm" variable
> 3. Creating AngularJS controllers with instance methods

6 Services and Factories

Services are where we would create to hold our application's Model data and business logic such as talking to the server over HTTP. There is often a lot of confusion between a Service and Factory, the main difference is the way each Object is created.

It's important to remember that all Services are application singletons, there is only one instance of a Service per injector. By convention all custom Services should follow Pascal Case like our Controllers, for example "my service" would be "MyService".

6.1 Service method

Angular gives us a `service` method for creating a new Object with, this could talk to a backend or provide utilities to handle business logic. A `service` is just a `constructor` Object that gets called with the `new` keyword, which means we'll be using the `this` keyword to bind our logic to the Service. The `service` creates a singleton Object created by a service factory.

```javascript
function UserService () {
  this.sayHello = function (name) {
    return 'Hello there ' + name;
  };
}

angular
  .module('app')
  .service('UserService', UserService);
```

We could then inject the Service into a Controller to use it.

```javascript
function MainCtrl (UserService) {
  this.sayHello = function (name) {
    UserService.sayHello(name);
  };
}

angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

A Service means we can't run any code before it as all methods are the Object instantiated. Things are different when it comes to a Factory.

6.2 Factory method

Factory methods return an `Object` or a `Function`, which means we can make usage of closures as well as returning a host Object to bind methods to. We can create private and public scope. All Factories become a Service, to we should refer to them as a Service not the pattern they are named after.

We'll recreate the `UserService` above using the `factory` method for comparison.

```javascript
function UserService () {
  var UserService = {};
  function greeting (name) {
    return 'Hello there ' + name;
  }
  UserService.sayHello = function (name) {
    return greeting(name);
  };
  return UserService;
}

angular
  .module('app')
  .factory('UserService', UserService);
```

I've split the `greeting` function out to show how we can emulate private scope through closures. We could have done something similar inside the `service` method `constructor`, but this shows what gets returned and what stays inside the Service scope. We could create private functions such as helpers that remain in the scope after the function has returned, but are still available for use by the public methods. We use Services created by the `factory` method in the exact same way inside a Controller.

```javascript
function MainCtrl (UserService) {
  this.sayHello = function (name) {
    UserService.sayHello(name);
  };
}

angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

Services are generally used for non-presentational logic, which we call the business logic layer. This tends to be things like communicating with a backend via REST endpoints over Ajax (HTTP).

> Further Reading:
>
> 1. Angular service or factory?
> 2. AngularJS: Factory vs Service vs Provider
> 3. AngularJS service vs factory – with example
> 4. Service vs provider vs factory?

7 Templating with the Angular core

Until now, we've covered more of the software engineering side of AngularJS, the principles and APIs for controlling the flow of our business logic and presentational data by using Controllers and Services. The other side to AngularJS is in its two way data-binding but we never showed how to use them in our HTML. The next step would be putting them into good practice by making use of Angular's powerful templating engine and internal core Directives.

7.1 Expressions

Angular expressions are "JavaScript-like" snippets of code that can be used amongst our templates to conditionally change the DOM, be it an element, element property (such as a class name) or text. These expressions live inside handlebar template bindings `{{ }}` that we're used to using and are evaluated against a `$scope` Object. They're basic to use in the sense that we can't use loops or conditionals like `if` and `else`, and we can't throw exceptions. They're used for small calculations only or getting the values of `$scope` properties.

When we use `{{ value }}`, this is an expression. We can add to that expression by using simple operators to "echo" different values based on those operators. Some examples are the OR `||` operator, the ternary operator `value ? true : false` and the AND `&&` operator.

Using the above operators, we can create powerful templates that respond to the data presented, which Angular will re-evaluate for us each `$digest` cycle. This means we can have rich user interaction and feedback, such as updating values as they change on a page for a user without requerying or reloads.

Let's take the following Controller:

```javascript
function MainCtrl () {
  this.items = [{
    name: 'Scuba Diving Kit',
    id: 7297510
  },{
    name: 'Snorkel',
    id: 0278916
  },{
    name: 'Wet Suit',
    id: 2389017
  },{
    name: 'Beach Towel',
    id: 1000983
  }];
}

angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

We can then retrieve the length of the Array by declaring an expression that makes use of the `length` property of an Array. This might indicate to the user how many items are in stock.

```markup
<div ng-controller="MainCtrl as main">
  {{ main.items.length }} in stock
</div>
```

Angular will render out the length property, so we'll see "4 items in stock" in the DOM.

7.2 Using Angular core Directives

After rendering an expression, those values are automatically bound inside Angular which means any updates to their source will reflect on the expression. This means if we were to remove one of the items in stock, our "4 items in stock" would automatically update to "3 items in stock" - without us doing anything. Angular gives us a lot for free, and the two way data-binding keeps all expression values up to date without us requerying or running any callbacks.

Angular gives us a whole heap of built in `ng-*` prefixed Directives, let's start with `ng-click` and bind a function to a new piece of HTML template. We're not limited to using an expression once, `$scope` is just an Object so we can reuse it as much as we like. Here I'm going to iterate over the `items` Array and show how many items are in stock.

```markup
<div ng-controller="MainCtrl as main">
  <div>
    {{ main.items.length }} items in stock
  </div>
  <ul>
      <li ng-repeat="item in main.items" ng-click="main.removeFromStock(item, $index)">
         {{ item.name }}
      </li>
  </ul>
</div>
```

If you take a close look at the above, I'm binding an `ng-click` attribute with a function inside called `main.removeFromStock()`. I'm passing in the `item`, which is the current iterated Object from `item in main.items`, and passing in `$index`. The `$index` property is extremely handy for removing items from an Array without us manually calculating the index of the element.

Now the function is setup, we can add it to the Controller, I'll keep the `items` Array brief for visibility. I pass in the `$index` and the item, and on the `removeFromStock` method

```javascript
function MainCtrl () {
  this.removeFromStock = function (item, index) {
    this.items.splice(index, 1);
  };
  this.items = [...];
}

angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

As we create methods, the `this` value may change depending on how we use it due to the function's execution context. With the Controller being a glorified `ViewModel`, I usually create a reference to the Controller using a variable `var vm = this;`, with "vm" standing for "ViewModel". This way we don't lose any lexical `this` references and avoid using `Function.prototype.bind` to keep changing context, or Angular's own `angular.bind`.

Our reworked Controller would look like this:

```javascript
function MainCtrl () {
  var vm = this;
  vm.removeFromStock = function (item, index) {
    vm.items.splice(index, 1);
  };
  vm.items = [...];
}

angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

These methods help us build up the presentational logic that handles our application's UI layer. We are missing a step, however, and that step is updated the Model. Typically before we remove an item from the `vm.items` Array, we would make a `DELETE` request to the backend and then on successful callback remove it from the Array. This way we only update the DOM upon success so the user doesn't think it's succeeded. The above gives a simple glimpse at how you can create templates with Angular, with plain HTML and sprinkles of Angular syntax.

Templates we create inside Angular, however simple, are generally encapsulated functionality that we might want to reuse elsewhere in the application or in future applications. We should always build with reuse in mind, and this is where Directives come in very nicely.

8 Directives (Core)

There are two types of Directives, the ones that power Angular's bindings, and the ones we can create ourselves. A Directive can be anything, it can either provide powerful logic to an existing specific element, or be an element itself and provide an injected template with powerful logic inside. The idea behind them is around extending HTML's capabilities, as if it were made for building compelling data-driven web applications.

Let's look over some of the most powerful Angular Directives built into the core and what they do, then we'll move onto creating our own custom Directives and the concepts behind them.

8.1 ng-repeat

We've already looked at the ng-repeat, so we'll just show a simple example to recap.

```markup
<ul>
  <li ng-repeat="item in main.items">
    {{ item }}
  </li>
</ul>
```

The ng-repeat copies an element and duplicates it whilst dynamically filling it with each Object's data in an Array, which means the amount of HTML created in your repeat mirrors the Object's it's repeated with. If an item is removed from the source Array, Angular will update the DOM automatically, adjusting all `index` values accordingly.

8.2 ng-model

When we bind a Model onto our HTML, we're tying Model data to elements. To initialise a new Model, if it doesn't exist yet, or bind an existing Model we simply bind to `ng-model`.

```markup
<input type="text" ng-model="main.message">
<p>Message: {{ main.message }}</p>
```

If the `$scope` property `main.message` holds a value, the input's `value` attribute and property are both set to that value. If `main.message` doesn't exist inside your `$scope`, Angular will just initialise it. We might pass these values to other Angular Directives, such as `ng-click`.

8.3 ng-click

The beauty of ng-click is that we don't have to manually bind event listeners to multiple elements, Angular will evaluate the expression(s) inside the `ng-click` for us and bind the relevant listeners. This makes our development much faster, imagine binding event listeners and callbacks to DOM elements manually, adding and removing them whilst the DOM is destroyed and recreated on the fly. Unlike the old `onclick="` attribute in HTML, Angular directives such as `ng-click` are scoped, and therefore are not global (which functions would need to be to be available for `onclick`).

```markup
<input type="text" ng-model="main.message">
<a href=" ng-click="main.showMessage(main.message);">Show my message</a>
```

You can see here I'm passing in `main.message` into the `main.showMessage` method, this passes a plain JavaScript Object into Angular for evaluation. This is the beauty of working with Angular, all DOM data-binding corresponds with an Object, we can just parse it, manipulate it, parse to JSON and send off to the backend. Methods such as `main.showMessage` will live inside the Controller that's powering the view, in this case I'm assuming it's `MainCtrl as main`.

8.4 ng-href/ng-src

Angular takes care of any browser quirks with dynamically setting a `href` and `src` value, typically in IE. Instead of using `href="` and `src="` inside our templates, we use `ng-href="` and `ng-src="`.

```markup
<a ng-href="{{ main.someValue }}">Go</a>
<img ng-src="{{ main.anotherValue}}" alt=">
```

8.5 ng-class

The `ng-class` Directive has a somewhat stranger syntax to other Directives, it takes on the form of an Object with properties and values. Instead of doing the traditional `elem.addClass(className)` and `elem.removeClass(className)`, Angular will add and remove classes based on the expression you provide it, along with a `className`.

```markup
<div class="notification" ng-class="{
  warning: main.response == 'error',
  ok: main.response == 'success'
}">
  {{ main.responseMsg }}
</div>
```

It's often clearer to see what's happening inside the `ng-class` Directive if we indent it like a proper Object, like above. You can see how Angular will keep evaluating `main.response` to see if the status returned is `error` or `success`, the class it will add it the Object's property name, in these cases `warning` and `ok`. If `main.response == 'error'` evalutes to `true`, Angular will add a `warning` class.

8.6 ng-show/ng-hide

Using `ng-show` and `ng-hide` are often quite common to use within Angular, it's a fantastic way to show and hide data based on a `$scope` property's value.

To show an element, we might use an `ng-click` to bind a value to, and toggle it to show and hide the element.

```markup
<a href=" ng-click="showMenu = !showMenu">Toggle menu!</a>
<ul ng-show="showMenu">
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

Depending on the semantics of the immediate content, should it be hidden or visible initially, you may choose `ng-show` or `ng-hide`. There is no other difference apart from they do the opposite of eachother. Angular adds and removes an `ng-hide` class when toggling between show and hide states, it assumes without the `ng-hide` class the element will be displayed.

8.7 ng-if

Unlike `ng-show` or `ng-hide`, which uses a class to toggle element's visibility on the page, `ng-if` actually destroys the DOM and the `$scope` it creates. If the element needs recreating due to value changes, Angular will create new `$scope` for that particular element. This also adds to improved performance, as removing elements via `ng-if` removes the `$scope`, which in turn lowers the `$$watchers` count inside Angular, speeding up further `$digest` cycles.

```markup
<div ng-if="main.userExists">
  Please log in
</div>
```

8.8 ng-switch

Think a typical `switch` case, but in the DOM or better yet, an `ng-if` on steroids. Provide Angular multiple chunks of HTML for it to swap in and out based on a single `$scope` value. This allows us to provide different HTML for different users for example,

```markup
<div ng-switch on="main.user.access">
  <div ng-switch-when="admin">
```

```
    <!-- code for admins -->
  </div>
  <div ng-switch-when="user">
    <!-- code for users -->
  </div>
  <div ng-switch-when="author">
    <!-- code for authors -->
  </div>
</div>
```

8.9 ng-bind

We're using to rendering values in the DOM using `{{ value }}` bindings, but there is another alternative called `ng-bind` which you can see the difference in syntax here.

```markup
<p>{{ main.name }}</p>
<p ng-bind="main.name"></p>
```

We would use `ng-bind` when we don't want to see any flicker whilst Angular is loading and parsing data. Angular automatically shields any content that's dynamically created with Directives or inside a dynamic view (`ng-view`). If your bindings are outside of these, we might see `{{` and `}}` text in the document, which is bad. To get around this, we use `ng-bind` which is invisible as Angular resolves the value once it's ready and removes the need for `{{ }}` bindings.

8.10 ng-view

In single page applications (SPAs), the concept is just having one page that dynamically updates. Angular delivers this to use using `ng-view` attribute on an empty `<div></div>` (or other element), which sits as a container for all dynamically injected HTML views, which are fetched via `XMLHttpRequest`.

Angular uses this `ng-view` attribute as a placeholder to then inject these different views into depending on the current route in the URL. We can configure these routes ourselves, which we'll come onto soon. In essence, we can tell Angular to inject `login.html` when the page's location (URL) says `myapp.com/#/login`, and change it for different page locations (referred to as routes). Each route can inject a custom View into `ng-view`, which makes configuring our applications a lot simpler.

8.11 Extending HTML

Angular's ideas around Directives are "extending HTML", and it does it brilliantly, creating dynamic HTML and responsive updates when Model data changes. There might come a time though where we want to create our own HTML extensions, our own Directives. Angular provides us the same API it uses for creating its own Directives.

9 Directives (Custom)

Custom Directives are possibly one of the hardest concepts and APIs to grasp in Angular as they don't adhere to any software engineering concept. They are, however, Angular's way to start using the rapidly approaching (but future) Web Components standard, which introduces Custom Elements, Shadow DOM, Templating and HTML imports. Angular Directives give us all of these and a seamless API for managing them. The easiest way to learn Directives is to break them down into their intended layers and the meaning behind them.

9.1 Custom Elements

Custom Elements are a huge thing, they provide a declarative approach to encapsulating and injecting behaviour. Custom Elements solve the problem of reusable and boilerplate code, in which we declare one element and its dependency code is injected around it.

They're not in full flux or supported by all browser vendors, so Angular created this custom implementation.

Custom Elements in Angular are less strict than the Web Components specification as Angular supports older versions of Internet Explorer which Custom Elements don't play as nicely with without manually creating references for them.

Angular provides us four ways to use its Directives, through Custom Elements, Custom Attributes, class names and comments. The last two I generally avoid as it can become confusing which is a comment and which is a class name - comments also have IE issues. The safest route for cross-browser compatibility is using a Custom Attribute. Let's explore the four options and their syntax in the following order: Element, Attribute, Class, Comment.

```markup
<my-element></my-element>
<div my-element></div>
<div class="my-element"></div>
<!-- directive: my-element -->
```

Angular Directives provide us with a `restrict` property, so we can tie down the usage to a particular implementation. By default, all Directives are set to `'EA'` which means `Element, Attribute`. The other options are `C` for class name and `M` for comment.

9.2 Shadow DOM

Shadow DOM allows DOM to be nested inside DOM, which typically means there is one main document and small pockets of other documents nested inside it. Shadow DOM offers both HTML, CSS and JavaScript scoping and encapsulation, which requires browser support itself. Instead of creating Shadow DOM, Angular injects DOM normally as if it were Shadow DOM to emulate it. In other words, it renders hidden parts of your template for you and they get injected where you instruct.

Shadow DOM allows us to specify the bare bones of the HTML, and also content to be "imported" into the Shadow DOM. This means we could put text inside a Custom Element:

```markup
<my-element>
  Hi there!
</my-element>
```

And the `Hi there!` text could be made available in the Shadow DOM. Angular has an option called `transclude`, which allows us to pull through existing content into Directive templates before being compiled and sent back to the DOM. It acts in a similar fashion to Shadow DOM, but doesn't provide the powerful scoping.

9.3 Templating and HTML Imports

Interestingly there are three different ways to use templates inside a Directive. Two look identical and the other uses a `String` to hold the template. Let's check out the `template` and `templateUrl` properties to understand what they give us.

9.3.1 template property

Template does exactly what it says, it declares the template we want to use. This is the `String` formatted template that Angular then compiles into live DOM.

An example:

```javascript
{
  template: '<div>' +
    '<ul>' +
      '<li ng-repeat="item in vm.items">' +
        '{{ item }}' +
      '</li>' +
    '</ul>' +
  '</div>'
}
```

If I'm using this setup (which can be nice as we can write JavaScript logic in between the Strings), I use `[].join('')` to make the template clearer to read and easier to indent, comma separation is also much nicer to look at.

```javascript
{
  template: [
    '<div>',
      '<ul>',
        '<li ng-repeat="item in vm.items">',
          '{{ item }}',
        '</li>',
      '</ul>',
    '</div>'
  ].join('')
}
```

9.3.2 templateUrl property

The templateUrl property allows us to point to either an external resource or a `<script>` element with our template inside.

If we specified:

```javascript
{
  templateUrl: 'items.html'
}
```

Angular would first scan the DOM for a `<script>` element with the `id` that matches, if it doesn't exist it'll attempt a `HTTP GET`.

Let's look at how the `<script>` implementation works.

```markup
<script type="text/ng-template" id="/hello.html">
  <div>
    <ul>
      <li ng-repeat="item in vm.items">
        {{ item }}
      </li>
    </ul>
  </div>
</script>
```

First, we specify the `type` as `text/ng-template`, changing the MIME type for the JavaScript engine, which makes the compiler ignore it to prevent a tonne of JavaScript errors thrown from inserting HTML. The `id` is the important piece, which pretends to be the `*.html` file, meaning we can use an inline `<script>` tag instead of making a `GET` request for each template file - a performance benefit.

The performance benefit also lives in the `template` property, whereby the template is stored as a `String`, this also means no `GET` request. Once a Directive's template has been loaded, Angular stores it internally so the template can be used by `ng-include` and `ng-view` as well.

If you're not using the inline `<script>` method, Angular won't find it and fire off `GET` request to fetch it, which it then imports the HTML and compiles into the Directive as live DOM.

All templates that are loaded by Angular, however, are loaded straight into the `$templateCache` and fetched from there for the lifetime of the application.

9.4 Directive API

Now we've learned the concepts behind custom Directives, we can begin to create our own. Let's checkout more of the API, which begins with a `return` statement which returns an Object. This is all we need to get a custom Directive up and running.

```javascript
function someDirective () {
  return {

  };
}

angular
  .module('app')
  .controller('someDirective', someDirective);
```

Let's look at some of the most commonly used Object properties for Directives. We've covered the `restrict` property, next we'll add `replace`, `controllerAs`, `controller`, `link` and `template`, this will give us an idea of how things hang together.

```javascript
function someDirective () {
  return {
    restrict: 'EA',
    replace: true,
    scope: true,
    controllerAs: 'something',
    controller: function () {

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

    },
    template: [
      '<div class="some-directive">',
        'My directive!',
      '</div>'
    ].join('')
  };
}
```

9.4.1 restrict

We've already covered the `restrict` property, but to recap - it allows you to restrict the Directive's usage for developers. If the Directive needs to be as an attribute, often when we want to bolt into an existing element and not have root template control, we can restrict to `'A'`. To restrict to just an element we use `'E'`, for comments we use `'M'` and for class names we use `'C'`.

9.4.2 replace

Very simple, it replaces the original Directive's element. If we use `<some-directive></some-directive>` and set `replace: true`, once rendered Angular will remove the original custom element declaration, leaving just the injected content.

9.4.3 scope

Allows us to inherit `$scope` from the current or parent context where the Directive is nested, or we can create isolate scope and pass specific `$scope` values in, typically done via custom attributes.

9.4.4 controllerAs

We've experimented with the `controllerAs` syntax, this is just how we declare the Controller's assignment name within a Directive. Assuming we set `controllerAs: 'something'`, any Controller references inside our template would use `something.myMethod()` for anything declared.

9.4.5 controller

Grab an existing Controller or create a new one. If `MainCtrl` already exists, we can assign it using `controller: 'MainCtrl'`, or simply use a function to create a new one. To keep encapsulation tight and easier to maintain we generally just declare a new Controller each time using `controller: function () {}`. The Controller callback should handle ViewModel changes and talk to any Services, which can be dependency injected as usual.

9.4.6 link

The link function is called after the element is compiled and injected into the DOM, which means it's the perfect place to do "post-compile" logic, as well as non-Angular logic.

As we never use DOM manipulation in a Controller, the `link` function is created as a utility for this - with the added bonus of being able to inject `$scope`, the root element of our Directive's template `$element`, and an Object called `$attrs` which contains DOM element attributes that reflect current binding state `{{ }}`. Inside `link`, we can bind event listeners, instantiate plugins, and even inject Services from Angular, making it a very flexible property.

9.5 Directive creation

Let's run through a quick example of how to create our own Directive, we'll create a simple Directive that allows us to inject an compose email component, which has To, Subject and Message fields.

We'll create a `<compose-email>` element which acts as our placeholder for Directive injection, this means we can declare ` multiple times in the DOM, and Angular will inject our template/component each time with a fresh instance. Let's start with our Directive template and get the basic template up and running:

```javascript
function composeEmail () {
  return {
    restrict: 'EA',
    replace: true,
    scope: true,
    controllerAs: 'compose',
    controller: function () {

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

    },
    template: [
      '<div class="compose-email">',
        '<input type="text" placeholder="To..." ng-model="compose.to">',
        '<input type="text" placeholder="Subject..." ng-model="compose.subject">',
        '<textarea placeholder="Message..." ng-model="compose.message"></textarea>',
      '</div>'
    ].join('')
```

We can now use the `composeEmail` Directive in multiple places without having to create the same markup each time, Angular will cleverly inject it where we declare it. It's important to remember that Angular will parse the name of your Directive, and anywhere that an uppercase character exists Angular will hyphenate it. This means that `composeEmail` will become `<compose-email></compose-email>` when used as a custom element in our HTML templates.

> Further Reading:
>
> 1. A Practical Guide to AngularJS Directives
> 2. Creating Custom AngularJS Directives

3. AngularJS Custom Directives

10 Filters (Core)

Angular filters are a way of processing data and returning a specific set of it, against some kind of logic. This could be absolutely anything, from processing a date stamp into a formatted time, to a list of names that begin with a specific letter. Let's look at some of the most common core Filters.

Filters can be used in two different ways, either in the DOM via a pipe `|` character inside our expressions, which Angular parses out for us. The second way is using the `$filter` Service, which we can dependency inject and use within our JavaScript instead of our HTML.

HTML expression syntax.

```markup
{{ filter_expression | filter : expression : comparator }}
```

JavaScript syntax.

```javascript
$filter('filter')(array, expression, comparator);
```

Usually, we'll use the HTML style expression as it's easy and concise. Let's look at a few of Angular's built-in filters.

10.1 Date filters

Working with dates can often be a time consuming and logic-heavy process, with Angular things get really easy thanks to the built-in date filter. Let's use a milliseconds time example (something like `1408466687250`) and bind it using `$scope.timeNow = new Date().getTime();`.

```markup
<p>
  Today's date: {{ timeNow | date:'dd-MM-yyyy' }}
</p>
```

The above would render out as today's date, in `dd-MM-yyyy` format, for instance `19-08-2014`.

10.2 JSON filter

The built-in JSON filter converts a JavaScript Object to a JSON String, this is really helpful for outputting Model values in the DOM during development to see values update. It's also a great help when debugging as all values will be updated. To get the best pretty printed JSON Object, wrap any JSON filters in `<pre>` tags.

```markup
<pre>
{{ myObject | json }}
</pre>
```

10.3 limitTo and orderBy

The previous Filter examples were rather basic, we pass one value in, and get one out. How does Angular handle sets of data? Using `limitTo` and `orderBy` are two fantastic Filters that have many common use cases in our applications.

Using `limitTo`, we can put a cap on the amounts of data that's presented to the View at a given time, this would generally be used inside an `ng-repeat` - Filters work here too!

```markup
<ul>
  <li ng-repeat="user in users | limitTo:10">
    {{ user.name }}
  </li>
</ul>
```

This would render out a maximum of `10` users, note how we use `| limitTo: 10` alongside the `ng-repeat`.

Using `orderBy` allows us to specify which order our Array will render based on a property inside one of the Objects. For instance, if we had a user Object:

```javascript
{
  name: 'Todd Motto',
}
```

We might want to render the list alphabetically, this is where the Filter comes in handy.

```markup
<ul>
  <li ng-repeat=" user in users | orderBy:'name' ">
```

```
    {{ user.name }}
  </li>
</ul>
```

These are just some of the basics when using Angular's internal Filters, the real power comes from using their API to create our own.

11 Filters (Custom)

We've all written Filters of some kind before, typically done inside Objects and Arrays. To make them reusable and available as part of Angular, Angular provides us with its API. This means we get two way data-binding and all the power behind our Filters without us doing a great deal. Any Filters we plug into Angular are automatically called during the `$digest` cycle.

With the Angular's built-in Filters, we covered a single value Filter, such as a date or Object, and Filters that handled data sets such as Arrays of Objects. Let's look at the differences in API on how to create our own custom Filters.

11.1 Single value Filters

Single value filters accept a single input, and returns a "filtered" output. We've seen an example previously using the `date` filter, let's look at how to create our own Filter using Angular's fantastic API, the `.filter()` method. Any Filters we create using `.filter()` are globally available in any scope, which means we can reuse them throughout the entire app.

Here's a look at an empty Filter setup which we can then build from.

```javascript
function myFilter () {
  return function () {
    // return filtered output
  };
}
angular
  .module('app')
  .filter('myFilter', myFilter);
```

The Filter API uses a function closure which gets returned and called each time Angular needs to run the Filter. Angular automatically passes the arguments we need into Filters, which makes our life a lot easier when it comes to using them - it's the exact same as Angular's built-in Filters.

Let's pass in an argument and create a basic Filter. Angular already has a "make lowercase" Filter, but it makes for a really simple example so we'll create our own too but call it something different.

```javascript
function toLowercase () {
  return function (item) {
    return item.toLowerCase();
  };
}
angular
  .module('app')
  .filter('toLowercase', toLowercase);
```

You see I pass in `item` as a local variable to the Filter, this isn't dependency injected, it's just the data Angular passes to us. Inside this Filter closure, I then `return item.toLowerCase();` which Angular sets as the updated value instead of the initial value.

Usage is the exact same as any other Filter in Angular.

```markup
<p>{{ user.name | toLowercase }}</p>
```

11.2 Data set Filters

We often need to iterate over a set of data and return a Filtered set of that data, whatever it may be. Let's take a real world example, our application might want to filter names in an address book by the first letter of their name.

Let's create a Filter that looks for users whose name starts with `'a'`.

```javascript
function namesStartingWithA () {
  return function (items) {
    return items.filter(function (item) {
      return /$a/i.test(item.name);
    });
  };
}
angular
  .module('app')
  .filter('namesStartingWithA', namesStartingWithA);
```

Usage would be inside an `ng-repeat`.

```markup
<ul>
  <li ng-repeat="item in items | namesStartingWithA">
    {{ item }}
  </li>
</ul>
```

We can also pass items into Filters, which become available as further arguments in the closure. This would allow us to pass a specific letter into a Filter function to make it dynamic. The syntax for passing in arguments is separated with a colon `:`.

```markup
<ul>
  <li ng-repeat="item in items | namesStartingWithA:something">
    {{ item }}
  </li>
</ul>
```

The `something`, be it a Model value or variable, can be passed in and accessed in the Filter closure.

```javascript
function namesStartingWithA () {
  return function (items, something) {
    // access to "items" and "something"
  };
}
angular
  .module('app')
  .filter('namesStartingWithA', namesStartingWithA);
```

11.3 Controller ($scope) Filters

We can create Filters outside of the `.filter()` method, and just pass in a function inside a Controller that acts as a Filter. This function inside the `$scope` acts as the closure we're used to.

Taking the example above, we can create `this.namesStartingWithA` as a function inside a local Controller. This means our Filter is only available inside that Controller, and not elsewhere in the app. This assumes using `controllerAs` syntax instead of using `$scope` for public methods.

```javascript
function SomeCtrl () {
  this.namesStartingWithA = function () {

  };
}
angular
  .module('app')
  .controller('SomeCtrl', SomeCtrl);
```

There's a slight difference in syntax when using it in the DOM as an Angular expression, we need to use `filter` as the first argument to our Filter.

```markup
<ul>
  <li ng-repeat="item in vm.items | filter:namesStartingWithA">
    {{ item }}
  </li>
</ul>
```

Remember, these Filters we create using functions are scoped to the Controller they were created in. Typically, we'd create all Filters using the `.filter()` method, but it's great to know what other methods we can use if the use case is there.

> Further Reading:
>
> 1. Filtering and Sorting a List
> 2. The 80/20 Guide to Writing and Using AngularJS Filters
> 3. Everything About Custom Filters in AngularJS

12 Dynamic routing with $routeProvider

Until now, we've covered how to build with Angular by understanding the concepts and APIs that help deliver those concepts. One concept we are yet to touch on is how Angular helps us deliver the Single Page Application (SPA) methodology. We can configure our application's state by using a router, which responds to changes to our applications location (the URL).

Angular used to package the router inside the core download, but realised not everybody needed it or used a third party router, and as such made it an optional drop-in module.

Inside our application's main module, we need to include is as a dependency, we'll assume the relevant JavaScript file is also available as well. The Angular module we need is called `ngRoute`.

```javascript
angular
  .module('app', [
    'ngRoute'
  ]);
```

Now the `ngRoute` module is available in our application, we can configure the routes by injecting `$routeProvider` and setting it up inside Angular's `.config()` method.

After injecting `$routeProvider`, we then get access to the `.when()` method which allows us to setup our router. Let's assume I've got an email inbox, we might want our application to hit `/inbox` and show us all our emails. The first argument we pass the `.when()` method is a String, which describes the "URL" that the application can hit, and once it does we can tell it to do something. The second argument to `.when()` is an Object, which specifies all our extras. We also get an `.otherwise()` method which will redirect people to a specific route should a route they're attempting to reach doesn't exist.

```javascript
function router ($routeProvider) {
  $routeProvider
  .when('/inbox', {})
  .otherwise({
    redirectTo: '/inbox'
  });
}

angular
  .module('app')
  .config(router);
```

Let's populate our first route to see how easy it is to configure. The first choice we need to make is typically the template we want to use with a particular view, let's assume we want the `inbox.html` template, whatever it may look like. We can use the standard `template` property we get with Directives, but this would be insane for full pages of view templates, so we'll use the `templateUrl` property and decouple our template from router config.

```javascript
$routeProvider
.when('/inbox', {
  templateUrl: 'views/inbox.html'
})
.otherwise({
  redirectTo: '/inbox'
});
```

All views need a Controller, Angular provides us a `controller` property and `controllerAs` to alias it.

```javascript
$routeProvider
.when('/inbox', {
  templateUrl: 'views/inbox.html',
  controller: 'InboxCtrl',
  controllerAs: 'inbox'
})
.otherwise({
  redirectTo: '/inbox'
});
```

This is great for setting up a simple and single route, but what about if we were to click and read an email in our pseudo email app? We need dynamic routing as all our emails and associated IDs will be different, and thankfully Angular's router provides us this out of the box. Dynamic routing is where we can pass in a particular property that is dynamic, such as the aforementioned email ID. We do this by using a colon `:` before the dynamic groups name. We can set this up to use `'/inbox/email/:id'` as the route, which might be something like `/inbox/email/173921938` in our app.

If our application then points to `/inbox/email/173921938`, Angular will load the same view template and Controller as if it pointed to `/inbox/email/902827312`, allowing us to reuse templates for core functionality.

Putting this together, we'll end up with something like the following.

```javascript
function router ($routeProvider) {
  $routeProvider
  .when('/inbox', {
    templateUrl: 'views/inbox.html',
    controller: 'InboxCtrl',
    controllerAs: 'inbox'
  })
   .when('/inbox/email/:id', {
    templateUrl: 'views/email.html',
    controller: 'EmailCtrl',
    controllerAs: 'email'
  })
  .otherwise({
    redirectTo: '/inbox'
  });
});

angular
```

This is the basis of what you need to get routes for a single page application up and running with Angular, but there is one more thing, `ng-view`. We've touched on it before, but now we need to connect the router with the `ng-view` container to let Angular know where to inject our View templates.

Typically we'll have something like this.

```markup
<div ng-view></div>
```

The `ng-view` attribute sits in the "middle" of our app, and any changes to the window's location, Angular will reference the router to see if it needs to inject a new template. This is really all there is to getting Ajax views working.

12.1 $routeParams

Angular gives us a `$routeParams` service, that automatically parses the current URI to retrieve a set of parameters and maps them back to us as an Object.

From the above example with our dynamic email view, we have `/inbox/email/:id` as the specified route. Angular will hand us the dynamic `:id` part back to us when the route contains a value. For example `/inbox/email/20999851`. This would generally be a specific ID for us to be able to hit a REST backend and bring back the JSON for that specific ID, in this case it'd be an email.

The access the `$routeParams` Object, simply inject it like any other dependency and fire it off to the backend, a simple example inside a Controller to show how to pass `$routeParams` around using the properties - with a dummy `EmailService`.

```javascript
function EmailCtrl ($routeParams, EmailService) {
  // $routeParams { id: 20999851 }
  EmailService
  .get($routeParams.id) // pass the Object in
  .success(function (response) {})
  .error(function (reason) {});
}
angular
  .module('app')
  .('EmailCtrl', EmailCtrl);
```

> Further Reading:

> 1. AngularJS Routing And Views Tutorial With Example
> 2. $routeProvider api
> 3. 'Resolve' in AngularJS Routes, Explained as Story

13 Form Validation

Created to supersede HTML5 validation, Angular's built-in validation is superb for creating forms that respond to user input much like Model changes do. It does a few powerful things, from checking Model changes against binding rules, to manipulating the DOM to provide user feedback.

To get our form validation setup, we need a form with a `name` attribute to namespace the form scope.

```markup
<form name="myForm"></form>
```

Angular will recognise this as soon as the page has rendered and check things as the user fills them out, constantly watching the state and any rules we give inputs, such as `required` attributes to ensure users fill them in.

13.1 HTML5 features

In HTML5 we've seen the addition of the `pattern` attribute, which allows the browser to validate against a custom Regular Expression (RegExp), this is just one example of how Angular ties these new features into its framework. Angular also rebuilt the `required` checking and tied it into the framework under a Directive called `ng-required` which is also continuously being evaluated against Model changes. Let's look at how some of these features are expressed by Angular form states.

13.2 $pristine

Once your page has rendered, Angular will declare your form `$pristine`. This means that the user hasn't touched it yet. This can be really useful for making sure we don't display any validation messages before the user has even typed anything. Angular will add an `ng-pristine` class to elements that are pristine.

13.4 $dirty

`$dirty` forms are pretty much the opposite of `$pristine`, they become `$dirty` once a user has interacted with it, which at this point Angular would remove any `ng-pristine` classes and replace them with `ng-dirty`. Our form cannot return to `$pristine` unless the page is refreshed, if it has been touched it will remain `$dirty`.

13.5 $valid

Alongside `$pristine` and `$dirty` values, each input inside our form will also be validated against being `$valid`. This means that if we have an `ng-required` attribute and the user enters information, Angular will automatically make the input `$valid`, adding an `ng-valid` class name to the element.

13.6 $invalid

The opposite of `$valid` comes the `$invalid` flag. By default our forms are always `$invalid`, which means we get an `ng-invalid` class name added to all inputs on page render. These states can change back and forth as the user types (and maybe removes) information.

13.7 Model based validation

There are use cases where we might want to disable or enable input fields or even buttons for the user if they've not filled out the correct portions of a form. This might be for performance reasons or validation reasons, but we want total control of the data flow and let our form respond accordingly.

A simple example to toggle the enabled/disabled state of a `<button>` based on previous Model data. If a user hasn't entered their name, they can't update it.

```markup
<input type="text" ng-model="user.name" placeholder="Enter your name">
<button ng-disabled="!user.name.length">
  Update name
</button>
```

If the `user.name.length` is truthy, the button becomes enabled. This state will be constantly watched as the user interacts with the form.

> Further Reading:
>
> 1. AngularJS Form Validation
> 2. On the Bleeding Edge: Advanced AngularJS Form Validation */

14 Server communication with $http and $resource

We've just had a peek at what server communication might look like with Angular, and what Angular does automatically for us. I've specifically left our own Ajax techniques until last so we can actually see how much Angular does for us without us setting up Ajax calls to respond to new route changes, template injection and much more.

Inside Angular, we are presented with two fantastic APIs called `$http` and `$resource`. These are superb high level services that help us communicate with the server seamlessly, they also kick `$digest` cycles into running to keep our data-binding values fresh.

14.1 $http

If you've used jQuery's `$.ajax` method then you're going to be right at home. Angular keeps `$http` very light, minimal and flexible which makes it a great option for making calls to the backend.

The `$http` method either comes as a function or Object, depending on how you'd like to use it. We can either pass in a configuration Object, such as `$http({...})` or use available shorthand methods, such as `$http.get(...)`. Angular's `$http` method is based on the deferred/promise APIs exposed by the built-in `$q` service, a Promise API.

I favour the simplicity of the shorthand methods usually so we'll use these for demonstrations. Here's a simple `HTTP GET`.

```javascript
$http.get('/url')
.success(function (data, status, headers, config) {

})
```

```javascript
.error(function (data, status, headers, config) {

});
```

Success and error callbacks are invoked asynchronously, and Angular also passes us `data, status, headers, config` back in the response. We don't usually need all of them, so we could always aim for something a little simpler:

```javascript
$http.get('/url')
.success(function (response) {

})
.error(function (reason) {

});
```

I like to adopt a more native Promise like pattern using the `.then()` method, which Angular supports:

```javascript
$http.get('/url')
.then(function (response) {
  // success
}, function (reason) {
  // error
});
```

Unlike Ajax libraries such as jQuery's `$.ajax`, Angular wraps its `$http` calls inside a `$scope.$apply()`, which forces a digest cycle and our bindings will update.

14.2 $resource

Alongside `$http` we can optionally include the `ngResource` module. This module gives us an API called `$resource`, and is extremely useful for CRUD based operations, you can even use it alongside `$http` depending on your use case. `$resource` creates an Object that lets you interact with RESTful server-side data sources.

```javascript
function MovieService ($resource) {
  return $resource('/api/movies/:id', { id: '@_id' },
    {
      update: {
        method: 'PUT'
      }
    }
  );
}
angular
  .module('app')
  .factory('MovieService', MovieService);
```

We could dependency inject this simple Factory called `Movies` into our Controllers to obtain the data we need.

```javascript
function MovieCtrl (MovieService) {
  var movies = new MovieService();
  // have something update a movie
  movies.update(/* some data */);
}
angular
  .module('app')
  .controller('MovieCtrl', MovieCtrl);
```

> Further Reading:
>
> 1. Using The $http Service In AngularJS To Make AJAX Requests
> 2. Canceling $http Requests in AngularJS
> 3. Interceptors in AngularJS and Useful Examples
> 4. $http

Tagged under

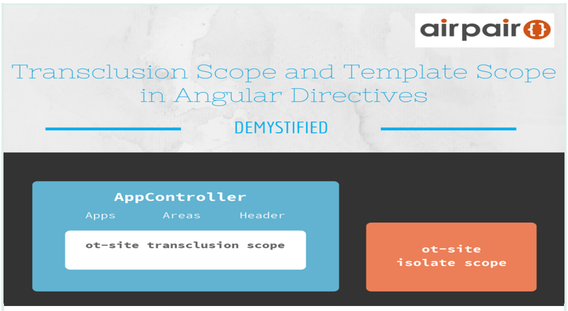
angularjs

Similar posts ALL POSTS

4 x

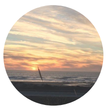
Animating elements moving between AngularJS lists

Michal Charemza


airpair
Transclusion Scope and Template Scope in Angular Directives
DEMYSTIFIED
AppController
Apps
Areas
Header
ot-site transclusion scope
ot-site isolate scope

12 x

Transclusion and Template Scope in Angular Directives Demystified



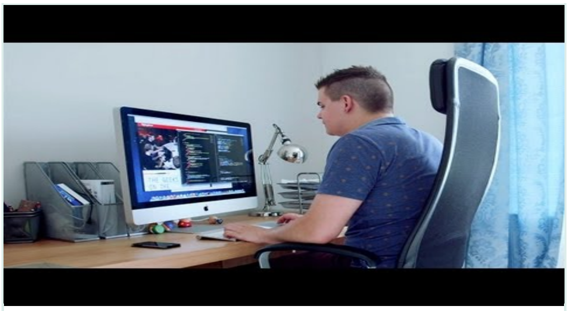
Kara Erickson

16 x

The Definitive Ionic Starter Guide

Jeremy Wilken

Other posts



HTML5 Expert, JavaScript and AngularJS Front-End Master - Todd Motto

Todd Motto