

Ultimate guide to learning AngularJS in one day

🔗 [Edit this page on GitHub \(/github.com/toddmotto/toddmotto.github.io/blob/master/posts/2013-10-02-ultimate-guide-to-learning-angular-js-in-one-day.md\)](#)

October 02, 2013

Mastered this? Try my [Opinionated AngularJS styleguide for teams \(/toddmotto.com/opinionated-angular-js-styleguide-for-teams\)](#)



Todd Motto

Director of Web Development @
[Mozio \(https://mozio.com/#/?utm_source=toddmotto.com\)](https://mozio.com/#/?utm_source=toddmotto.com)
JavaScript, AngularJS. Conference speaker.
Developer Expert
[@google \(/twitter.com/google\)](https://twitter.com/google)

Follow @toddmotto

[Posts \(/\)](#) | [About \(/about\)](#) | [Speaking \(/speaking\)](#)

[RSS \(/toddmotto.com/feed.xml\)](https://toddmotto.com/feed.xml)

[GitHub \(/github.com/toddmotto\)](https://github.com/toddmotto)



http://airpair.me/toddmotto?utm_source=expert&utm_medium=homepage&utm_content=button&utm_campaign=airpairme

For live [AngularJS help \(http://www.airpair.com/angularjs\)](http://www.airpair.com/angularjs) on AirPair

What is AngularJS?

Angular is a client-side MVC/MVVM framework built in JavaScript, essential for modern single page web applications (and even websites). This post is a full end to end crash course from my experiences, advice and best practices I've picked up from using it.

Terminology

Angular has a initial short learning curve, you'll find it's up and down after mastering the basics. It's mainly getting to grips with the terminology and "thinking MVC". MVC stands for *Model-View-Controller*. Here are the higher level and essential APIs that Angular comes with, and some terminology.

MVC

You've probably heard of MVC, used in many programming languages as a means of structuring/architecting applications/software. Here's a quick breakdown of meanings:

- **Model:** the data structure behind a specific piece of the application, usually ported in JSON. Read up on JSON before getting started with Angular as it's essential for communicating with your server and view. For instance a group of *User IDs* could have the following model:

```
{
  "users" : [{
    "name": "Joe Bloggs",
    "id": "82047392"
  }, {
    "name": "John Doe",
    "id": "65198013"
  }]
}
```

You'll then grab this information either from the server via an XHR (XMLHttpRequest), in jQuery you'll know this as the *\$.ajax* method, and Angular wraps this in *\$http*, or it'll be written into your code whilst the page is parsing (from a datastore/database). You can then push updates to your model and send it back.

- **View:** The view is simple, it's your HTML and/or rendered output. Using an MVC framework, you'll pull down Model data which updates your View and displays the relevant data in your HTML.
- **Controller:** Do what they say on the tin, they control things. But what things? Data. Controllers are your direct access from the *server* to the *view*, the middle man, so you can update data on the fly via comms with the server and the client.

Setting up an AngularJS project (bare essentials)

First, we need to actually setup the essentials to an Angular project. There are certain things to note before we begin, which generally consist of an *ng-app* declaration to define your app, a *Controller* to talk to your view, and some DOM binding and inclusion of Angular. Here are the bare essentials:

Some HTML with *ng-** declarations:

```
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <!-- controller logic -->
  </div>
</div>
```

An Angular Module and Controller:

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



<http://carbonads.com>

```
var myApp = angular.module('myApp', []),
```

```
myApp.controller('MainCtrl', ['$scope', function ($scope) {  
    // Controller magic  
}]);
```

Before jumping in, we need to create an *Angular module* which all our logic will bolt onto. There are many ways to declare modules, and you can chain all of your logic like this (I don't like this method):

```
angular.module('myApp', [])  
.controller('MainCtrl', ['$scope', function ($scope) {...}])  
.controller('NavCtrl', ['$scope', function ($scope) {...}])  
.controller('UserCtrl', ['$scope', function ($scope) {...}]);
```

Setting up a global Module has proved to be the best for Angular projects I've worked on. The lack of semi-colons and accidental closing of the 'chain' proved counter-productive and frequented unnecessary compiling errors. Go for this:

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

Each new file I create simply grabs the *myApp* namespace and automatically bolts itself into the application. Yes, I'm creating new files for each Controller, Directive, Factory and everything else (you'll thank me for this). Concatenate and minify them all and push the single script file into the DOM (using something like Grunt/Gulp).

Controllers

Now you've grasped the concept of MVC and a basic setup, let's check out Angular's implementation on how you can get going with Controllers.

Taking the example from above, we can take a baby step into pushing some data into the DOM from a controller. Angular uses a templating-style *{{ handlebars }}* syntax for talking to your HTML. Your HTML should (ideally) contain no physical text or hard coded values to make the most of Angular. Here's an example of pushing a simple String into the DOM:

```
<div ng-app="myApp">  
  <div ng-controller="MainCtrl">  
    {{ text }}  
  </div>  
</div>
```

```
var myApp = angular.module('myApp', []);  
  
myApp.controller('MainCtrl', ['$scope', function ($scope) {  
  
    $scope.text = 'Hello, Angular fanatic';  
  
}]);
```

And the live output:

ResultJavaScriptHTML

Edit in JSFiddle



The key rule concept here is the *\$scope* concept, which you'll tie to all your functions inside a specific controller. The *\$scope* refers to the current element/area in the DOM (no, not the same as *this*), and encapsulates a very clever scoping capability that keeps data and logic completely scoped inside elements. It brings JavaScript public/private scoping to the DOM, which is fantastic.

The *\$scope* concept may seem scary at first, but it's your way in to the DOM from the server (and static data from the client). It's the bridge between the two worlds, and it's what makes AngularJS so powerful.

if you have that too)! The demo gives you a basic idea of how you can 'push' data to the DOM.

Let's look at a more representative data structure that we've hypothetically retrieved from the server to display a user's login details. For now I'll use static data; I'll show you how to fetch dynamic JSON data later.

First we'll setup the JavaScript:

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

myApp.controller('UserCtrl', ['$scope', function ($scope) {

    // Let's namespace the user details
    // Also great for DOM visual aids too
    $scope.user = {};
    $scope.user.details = {
        "username": "Todd Motto",
        "id": "89101112"
    };

}]);
```

Then port it over to DOM to display this data:

```
<div ng-app="myApp">
  <div ng-controller="UserCtrl">
    <p class="username">Welcome, {{ user.details.username }}</p>
    <p class="id">User ID: {{ user.details.id }}</p>
  </div>
</div>
```

Output:



It's important to remember that Controllers are for *data* only, and creating functions (event functions too!) that talk to the server and push/pull JSON data. No DOM manipulation should be done here, so put your jQuery toolkit away. Directives are for DOM manipulation, and that's up next.

Protip: throughout the Angular documentation (at the time of writing this) their examples show this usage to create Controllers:

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

function MainCtrl ($scope) {
    //...
};
```

... DON'T do this. This exposes all your functions to the global scope and doesn't keep them tied in very well with your app. This also means that you can't minify your code or run tests very easily. Don't pollute the global namespace and keep your controllers *inside* your app.

Directives

A directive (**[checkout my post on Directives from existing scripts/plugins \(http://toddmotto.com/creating-an-angularjs-directive-from-one-of-your-existing-plugins-scripts\)](http://toddmotto.com/creating-an-angularjs-directive-from-one-of-your-existing-plugins-scripts)**) in it's simplest form is a small piece of templated HTML, preferably used multiple times throughout an application where needed. It's an easy way to inject DOM into your application with no effort at all, or perform custom DOM interactions. Directives are not simple at all, however, there is an incredible learning curve to fully conquering them, but this next phase will let you hit the ground running.

So what are directives useful for? A lot of things, including DOM components, for example tabs or navigation elements - really depends on what your app makes use of in the UI. Let me put it this way, if you've toyed with *ng-show* or *ng-hide*, those are directives (though they don't inject DOM).

For this exercise, I'm going to keep it really simple and create a custom type of button (called *customButton*) that injects some markup that I hate to keep typing out. There are various ways to define Directives in the DOM, these could look like so:

```
<!-- 1: as an attribute declaration -->
<a custom-button>Click me</a>

<!-- 2: as a custom element -->
<custom-button>Click me</custom-button>

<!-- 3: as a class (used for old IE compat) -->
<a class="custom-button">Click me</a>

<!-- 4: as a comment (not good for this demo, however) -->
<!-- directive: custom-button -->
```

I prefer using them as an attribute, custom elements are coming in the future of HTML5 under the Web Components, but Angular report these quite buggy in some older browsers.

Now you know how to declare where Directives are used/injected, let's create this custom button. Again, I'll hook into my global namespace *myApp* for the application, this is a directive in it's simplest form:

```
myApp.directive('customButton', function () {
  return {
    link: function (scope, element, attrs) {
      // DOM manipulation/events here!
    }
  };
});
```

I define my directive using the *.directive()* method, and pass in the directive's name 'customButton'. When you capitalise a letter in a Directive's name, it's use case is then split via a hyphen in the DOM (as above).

A directive simply returns itself via an Object and takes a number of parameters. The most important for me to master first are, *restrict*, *replace*, *transclude*, *template* and *templateUrl*, and of course the *link* property. Let's add those others in:

```
myApp.directive('customButton', function () {
  return {
    restrict: 'A',
    replace: true,
    transclude: true,
    template: '<a href="" class="myawesomebutton" ng-transclude>' +
      '<i class="icon-ok-sign"></i>' +
      '</a>',
    link: function (scope, element, attrs) {
      // DOM manipulation/events here!
    }
  };
});
```

Output:



Make sure you *Inspect Element* to see the additional markup injected. Yes, I know, there is no icon included as I never included Font Awesome, but you see how it works. Now for the Directive properties explanations:

- *restrict*: This goes back to usage, how do we restrict the element's usage? If you're using a project that needs legacy IE support, you'll probably need attribute/class declarations. Restricting as 'A' means you restrict it as an *Attribute*. 'E' for *Element*, 'C' for *Class* and 'M' for *Comment*. These have a default as 'EA'. Yes, you can restrict to multiple use cases.

- **replace**: This replaces the markup in the DOM that defines the directive, used in the example, you'll notice how initial DOM is replaced with the Directive's template.
- **transclude**: Put simply, using transclude allows for existing DOM content to be copied into the directive. You'll see the words 'Click me' have 'moved' into the Directive once rendered.
- **template**: A template (as above) allows you to declare markup to be injected. It's a good idea to use this for tiny pieces of HTML only. Injected templates are all compiled through Angular, which means you can declare the handlebar template tags in them too for binding.
- **templateUrl**: Similar to a template, but kept in it's own file *or* `<script>` tag. You can do this to specify a template URL, which you'll want to use for manageable chunks of HTML that require being kept in their own file, just specify the path and filename, preferably kept inside their own *templates* directory:

```
myApp.directive('customButton', function () {
  return {
    templateUrl: 'templates/customButton.html'
    // directive stuff...
  };
});
```

And inside your file (filename isn't sensitive at all):

```
<!-- inside customButton.html -->
<a href="" class="myawesomebutton" ng-transclude>
  <i class="icon-ok-sign"></i>
</a>
```

What's really good about doing this, is the browser will actually *cache* the HTML file, bravo! The other alternative which isn't cached is declaring a template inside `<script>` tags:

```
<script type="text/ng-template" id="customButton.html">
<a href="" class="myawesomebutton" ng-transclude>
  <i class="icon-ok-sign"></i>
</a>
</script>
```

You'll tell Angular that it's an *ng-template* and give it the ID. Angular will look for the *ng-template* or the **.html* file, so whichever you prefer using. I prefer creating **.html* files as they're very easy to manage, boost performance and keep the DOM very clean, you could end up with 1 or 100 directives, you want to be able to navigate through them easily.

Services

Services are often a confusing point. From experience and research, they're more a stylistic design pattern rather than providing *much* functional difference. After digging into the Angular source, they look to run through the same compiler and they share a lot of functionality. From my research, you should use Services for *singletons*, and Factories for more complex functions such as Object Literals and more complicated use cases.

Here's an example Service that multiplies two numbers:

```
myApp.service('Math', function () {
  this.multiply = function (x, y) {
    return x * y;
  };
});
```

You would then use this inside a Controller like so:

```
myApp.controller('MainCtrl', ['$scope', function ($scope) {
  var a = 12;
  var b = 24;

  // outputs 288
  var result = Math.multiply(a, b);
}]);
```

Yes, multiplication is very easy and doesn't need a Service, but you get the gist.

When you create a Service (or Factory) you'll need to use dependency injection to tell Angular it needs to grab hold of your new Service - otherwise you'll get a compile error and your Controller will break. You may have noticed the *function (\$scope)* part inside the Controller declaration by now, and this is simple dependency injection. Feed it the code! You'll also notice `['$scope']` before the *function (\$scope)* too, I'll come onto this later. Here's how to use dependency injection to tell Angular you need your service:

```
// Pass in Math
myApp.controller('MainCtrl', ['$scope', 'Math', function ($scope, Math) {
    var a = 12;
    var b = 24;

    // outputs 288
    var result = Math.multiply(a, b);
}]);
```

Factories

Coming from Services to Factories should be simple now, we could create an Object Literal inside a Factory or simply provide some more in-depth methods:

```
myApp.factory('Server', ['$http', function ($http) {
    return {
        get: function(url) {
            return $http.get(url);
        },
        post: function(url) {
            return $http.post(url);
        },
    };
}]);
```

Here I'm creating a custom wrapper for Angular's XHR's. After dependency injection into a Controller, the usage is simple:

```
myApp.controller('MainCtrl', ['$scope', 'Server', function ($scope, Server) {
    var jsonGet = 'http://myserver/getURL';
    var jsonPost = 'http://myserver/postURL';
    Server.get(jsonGet);
    Server.post(jsonPost);
}]);
```

If you wanted to Poll the server for changes, you could then setup a *Server.poll(jsonPoll)*, or perhaps if you're using a Socket you could setup *Server.socket(jsonSocket)*. It opens up doors to modularising code as well as creating tools for you to use and keep code inside the Controllers to a complete minimum.

Filters

Filters are used in conjunction with arrays of data and also outside of loops. If you're looping through data and want to filter out specific things, you're in the right place, you can also use Filters for filtering what a user types inside an <input> field for example. There are a few ways to use Filters, inside Controllers or as a defined method. Here's the method usage, which you can use globally:

```
myApp.filter('reverse', function () {
    return function (input, uppercase) {
        var out = '';
        for (var i = 0; i < input.length; i++) {
            out = input.charAt(i) + out;
        }
        if (uppercase) {
            out = out.toUpperCase();
        }
        return out;
    }
});

// Controller included to supply data
myApp.controller('MainCtrl', ['$scope', function ($scope) {
    $scope.greeting = 'Todd Motto';
}]);
```

DOM usage:

```
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <p>No filter: {{ greeting }}</p>
    <p>Reverse: {{ greeting | reverse }}</p>
  </div>
</div>
```

Output:

Result JavaScript HTML Edit in JSFiddle



You can see that we passed the greeting data to a filter via the pipe (|) character, applying the reverse filter to the greeting data.

And it's usage inside an *ng-repeat*:

```
<ul>
  <li ng-repeat="number in myNumbers | filter:oddNumbers">{{ number }}</li>
</ul>
```

And here's a real quick example of a Filter *inside* a Controller:

```
myApp.controller('MainCtrl', ['$scope', function ($scope) {

    $scope.numbers = [10, 25, 35, 45, 60, 80, 100];

    $scope.lowerBound = 42;

    // Does the Filters
    $scope.greaterThanNum = function (item) {
        return item > $scope.lowerBound;
    };

}]);
```

And it's usage inside an *ng-repeat*:

```
<li ng-repeat="number in numbers | filter:greaterThanNum">
  {{ number }}
</li>
```

Output:

Result JavaScript HTML Edit in JSFiddle



That's the main bulk behind AngularJS and it's APIs, this only only diving into the shallow end of the waters, but this is more than enough to get you building your own Angular application.

Two-way data-binding

When I first heard about two-way data-binding, I wasn't really sure what it was. Two-way data-binding is best described as a full-circle of synchronised data: update the *Model* and it updates the *View*, update the *View* and it updates the *Model*. This means that data is kept in sync without making any fuss. If I bind an *ng-model* to an `<input>` and start typing, this creates (or updates an existing) model at the same time.

Here I create the `<input>` and bind a Model called 'myModel', I can then use the curly handlebars syntax to reflect this model and it's updates in the view all at once:

```
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <input type="text" ng-model="myModel" placeholder="Start typing..." />
    <p>My model data: {{ myModel }}</p>
  </div>
</div>
```

```
myApp.controller('MainCtrl', ['$scope', function ($scope) {
  // Capture the model data
  // and/or initialise it with an existing string
  $scope.myModel = '';
}]);
```

Output:

ResultJavaScriptHTML

Edit in JSFiddle



XHR/Ajax/\$http calls and binding JSON

You've got the idea when it comes to pushing basic data to the `$scope` now, and a rough idea of how the models and two-way data-binding works, so now it's time to emulate some real XHR calls to a server. For websites, this isn't essential unless you have specific Ajax requirements, this is mainly focused on grabbing data for a web application.

When you're developing locally, you're possibly using something like Java, ASP .NET, PHP or something else to run a local server. Whether you're contacting a local database or actually using the server as an API to communicate to another resource, this is much the same setup.

Enter 'dollar http'. Your best friend from now on. The `$http` method is a nice Angular wrapper for accessing data from the server, and so easy to use you could do it blindfolded. Here's a simple example of a 'GET' request, which (you guessed it) gets data from the server. It's syntax is very jQuery-like so transitioning is a breeze:

```
myApp.controller('MainCtrl', ['$scope', '$http', function ($scope, $http) {
  $http({
    method: 'GET',
    url: '//localhost:9000/someUrl'
  });
}]);
```

Angular then returns something called a *promise*, which is a much more efficient and readable way of handling callbacks. Promises are chained onto the function they're initiated from using dot notation `.myPromise()`. As expected, we've got error and success handlers:

```
myApp.controller('MainCtrl', ['$scope', function ($scope) {
  $http({
    method: 'GET',
    url: '//localhost:9000/someUrl'
  })
  .success(function (data, status, headers, config) {
    // successful data retrieval
  })
  .error(function (data, status, headers, config) {
    // error handling
  });
}]);
```



```
//
.error(function (data, status, headers, config) {
  // something went wrong :(
});
});
```

Very nice and readable. This is where we merge the View and the server by binding a Model or updating Model data to the DOM. Let's assume a setup and push a username to the DOM, via an Ajax call.

Ideally, we should setup and design our JSON first, which will affect how we bind our data. Let's keep it simple, this is what a backend guy will setup as an API feed down to your application, you'll be expecting the following:

```
{
  "user": {
    "name": "Todd Motto",
    "id": "80138731"
  }
}
```

This means we'll get an Object returned back from the server (under an alias we'll call 'data' [you can see *data* is passed into our promise handlers]), and have to hook into the *data.user* property. Inside the *data.user* prop, we have *name* and *id*. Accessing those is easy, we'll need to look for *data.user.name* which will give us 'Todd Motto'. Now let's fetch it!

The JavaScript (check inline annotations for what's going on here):

```
myApp.controller('UserCtrl', ['$scope', '$http', function ($scope, $http) {

  // create a user Object
  $scope.user = {};

  // Initiate a model as an empty string
  $scope.user.username = '';

  // We want to make a call and get
  // the person's username
  $http({
    method: 'GET',
    url: '//localhost:9000/someUrlForGettingUsername'
  })
  .success(function (data, status, headers, config) {
    // See here, we are now assigning this username
    // to our existing model!
    $scope.user.username = data.user.name;
  })
  .error(function (data, status, headers, config) {
    // something went wrong :(
  });
}]);
```

And now in the DOM, we can just do this:

```
<div ng-controller="UserCtrl">
  <p>{{ user.username }}</p>
</div>
```

This will now print the username. Now we'll take this even further with understanding declarative data-binding which is where things get really exciting.

Declarative data-binding

Angular's philosophy is creating dynamic HTML that's rich in functionality and does a hell of a lot of work seamlessly that you would never expect on the client-side of the web. This is exactly what they've delivered.

Let's imagine we've just made an Ajax request to get a list of emails and their Subject line, date they were sent and want to render them in the DOM. This is where jaws drop at the power of Angular. First I'll need to setup a Controller for Emails:

```
myApp.controller('EmailsCtrl', ['$scope', function ($scope) {

  // create a emails Object
  $scope.emails = {};
```

```
// pretend data we just got back from the server
// this is an ARRAY of OBJECTS
$scope.emails.messages = [{
  "from": "Steve Jobs",
  "subject": "I think I'm holding my phone wrong :/",
  "sent": "2013-10-01T08:05:59Z"
}, {
  "from": "Ellie Goulding",
  "subject": "I've got Starry Eyes, lulz",
  "sent": "2013-09-21T19:45:00Z"
}, {
  "from": "Michael Stipe",
  "subject": "Everybody hurts, sometimes.",
  "sent": "2013-09-12T11:38:30Z"
}, {
  "from": "Jeremy Clarkson",
  "subject": "Think I've found the best car... In the world",
  "sent": "2013-09-03T13:15:11Z"
}
];

});
```

Now we need to plug it into our HTML. This is where we'll use declarative bindings to *declare* what the application will do - to create our first piece of dynamic HTML. We're going to be using Angular's built-in *ng-repeat* directive, which will iterate over data and render an output with absolutely no callbacks or state changes, it's all for free:

```
<ul>
  <li ng-repeat="message in emails.messages">
    <p>From: {{ message.from }}</p>
    <p>Subject: {{ message.subject }}</p>
    <p>{{ message.sent | date:'MMM d, y h:mm:ss a' }}</p>
  </li>
</ul>
```

Output:



I've also snuck in a *date filter* in there too so you can see how to render UTC dates.

Dig into Angular's suite of *ng-** directives to unleash the full power of declarative bindings, this shows you how to join the dots from server to Model to View and render the data.

Scope functions

As a continuation from declarative-binding, scope functions are the next level up in creating an application with some functionality. Here's a basic function to *delete* one of our emails in our data:

```
myApp.controller('MainCtrl', ['$scope', function ($scope) {

  $scope.deleteEmail = function (index) {
    $scope.emails.messages.splice(index, 1)
  };

}]);
```

Pro tip: It's important to think about deleting *data* from the Model. You're not deleting elements or anything

actual DOM related, Angular is an MVC framework and will handle all this for you with it's two-way binding and callback-free world, you just need to setup your code intelligently to let it respond to your data!

Binding functions to the scope are also run through *ng-** Directives, this time it's an *ng-click* Directive:

```
<a ng-click="deleteEmail($index)">Delete email</a>
```

This is way different to inline click handlers, for many reasons. This will be covered soon. You'll see I'm also passing in the *\$index*, Angular knows which item you're deleting (how much code and logic does that save you!).

Output (delete some emails!):

ResultJavaScriptHTML

Edit in JSFiddle



电脑管家提醒您
当前网站危险状况未知

Declarative DOM methods

Now we'll move onto *DOM Methods*, these are also Directives and simulate functionality in the DOM which you'd normally end up writing even more script logic for. A great example of this would be a simple toggling navigation. Using *ng-show* and a simple *ng-click* setup, we can create a flawless toggling nav:

```
<a href="" ng-click="toggle = !toggle">Toggle nav</a>
<ul ng-show="toggle">
  <li>Link 1</li>
  <li>Link 2</li>
  <li>Link 3</li>
</ul>
```

This is where we enter MVVM, you'll notice there is no Controller being introduced here, we'll come onto MVVM soon.

Output (get toggling!):

ResultJavaScriptHTML

Edit in JSFiddle



电脑管家提醒您
当前网站危险状况未知

Expressions

One of my favourite parts of Angular, what you'd usually use JavaScript for and write a lot of repetitive code.

Have you ever done this?

```
elem.onclick = function (data) {
  if (data.length === 0) {
    otherElem.innerHTML = 'No data';
  } else {
    otherElem.innerHTML = 'My data';
  }
};
```

This would potentially be a callback from a *GET* request, and you'll alter the DOM based on the data's state. Angular gives you this for free too, and you'll be able to do it inline without writing any JavaScript!

```
<p>{{ data.length > 0 && 'My data' || 'No data' }}</p>
```

This will just update itself dynamically without callbacks as your application polls/fetches data. If there's no data, it'll tell you - if there's data, it'll say. There are so many use cases for this and Angular handles it all automatically via two-way binding magic.

Output:



Dynamic views and routing

The philosophy behind single-page web applications (and also websites!). You have a header, footer, sidebar, and the content in the middle magically injects new content based on your URL.

Angular makes this setup a breeze to configure what we'd call *dynamic views*. Dynamic views inject specific Views based on the URL, through the *\$routeProvider*. A simple setup:

```
myApp.config(['$routeProvider', function ($routeProvider) {

    /**
     * $routeProvider
     */
    $routeProvider
    .when('/', {
        templateUrl: 'views/main.html'
    })
    .otherwise({
        redirectTo: '/'
    });

}]);
```

You'll see that 'when' the URL is '/' (i.e. the root of the site), you'll want to inject *main.html*. It's a good idea to call your initial View *main.html* and not *index.html* as you'll already have an *index.html* page for your single-page setup (a mouthful, yes). Adding more Views based on the URL is so simple:

```
myApp.config(['$routeProvider', function ($routeProvider) {

    /**
     * $routeProvider
     */
    $routeProvider
    .when('/', {
        templateUrl: 'views/main.html'
    })
    .when('/emails', {
        templateUrl: 'views/emails.html'
    })
    .otherwise({
        redirectTo: '/'
    });

}]);
```

```
//
    .otherwise({
        redirectTo: '/'
    });
}));
```

We could then have *emails.html* simply loaded with our HTML which generates our email list. You end up creating a very sophisticated application with little effort at this point.

There is a lot more to the *\$routeProvider* service which is well worth reading up on, but this'll get the ball rolling for you. There are things such as *\$http* interceptors that'll fire events when an Ajax call is in progress, things we could show some spinners for whilst we load in fresh data.

Global static data

Gmail handles a lot of it's initial data by writing the JSON into the page (right-click View Source). If you want to instantly set data in your page, it'll quicken up rendering time and Angular will hit the ground running.

When I develop our apps, Java tags are placed in the DOM and when rendered, the data is sent down from the backend. [I've zero experience with Java so you'll get a made up declaration from me below, you could use any language though on the server.] Here's how to write JSON into your page and then pass it into a Controller for immediate binding use:

```
<!-- inside index.html (bottom of page ofc) -->
<script>
window.globalData = {};
globalData.emails = <javaTagHereToGenerateMessages>;
</script>
```

My made up Java tag will then render the data whilst the page parses and Angular will render your emails instantly. Just feed it your data inside a Controller:

```
myApp.controller('EmailsCtrl', ['$scope', function ($scope) {

    $scope.emails = {};

    // Assign the initial data!
    $scope.emails.messages = globalData.emails;

}]]);
```

Minification

I'll talk a little about minifying your Angular code. You'll probably have experimented a little at this point and perhaps ran your code through a minifier - and perhaps encountered an error!

Minifying your AngularJS code is simple, you need to specify your dependency injected content in an array before the function:

```
myApp.controller('MainCtrl',
['$scope', 'Dependency', 'Service', 'Factory',
function ($scope, Dependency, Service, Factory) {

    // code

}]]);
```

Once minified:

```
myApp.controller('MainCtrl',
['$scope', 'Dependency', 'Service', 'Factory',
function (a,b,c,d) {

    // a = $scope
    // b = Dependency
    // c = Service
    // d = Factory

    // $scope alias usage
    a.someFunction = function () {...};

}]]);
```

Just remember to keep the injectors in the order they appear, or you'll probably cause you and your team a headache.

Differences in MVC and MVVM

We're closing up on the gigantic AngularJS post, and I'm going to briefly cover the differences in MVC/MVVM which Angular prides itself on:

- **MVC**: talks to a Controller, Model-View-Controller
- **MVVM**: encapsulates declarative data-binding that technically talks to itself. Model-View-View-Model. Model talks to the View, and the View can talk to the Model. Angular's two way data-binding allows it to keep itself in sync without you doing anything. It also allows you to write logic without a Controller!

A quick example of this, you could create an *ng-repeat* without a Controller providing data:

```
<li ng-repeat="number in [1,2,3,4,5,6,7,8,9]">
  {{ number }}
</li>
```

For quick testing this is okay, but I'd always recommend a Controller when you tidy things up.

Output:



HTML5 Web Components

You'll probably have caught it earlier, but AngularJS allows you to create custom elements:

```
<myCustomElement></myCustomElement>
```

This is actually bringing the web inline with the future of HTML5. HTML5 introduces Web Components and the `<template>` element, Angular lets us use this today. Web Components comprises of custom elements complete with dynamic JavaScript injection for View population that's incredibly exciting - we can do this stuff now with Angular! They're thinking ahead and future proofing upcoming web practices - hats off.

Scope comments

Scope comments I think are a really nice addition to my workflow, instead of declaring chunks of my HTML with comments like this:

```
<!-- header -->
<header>
  Stuff.
</header>
<!-- /header -->
```

When introducing Angular, let's think about Views and Scopes, not the DOM! Scopes are in fact *scoped*, meaning unless you deliberately share the data between Controllers, your data is scoped and inaccessible elsewhere. I find laying out my scoped areas to be a real helper:

```
<!-- scope: MainCtrl -->
<div class="content" ng-controller="MainCtrl">

</div>
<!-- /scope: MainCtrl -->
```

Debugging AngularJS

There's an awesome Chrome Extension that the guys at Google recommend for developing and debugging

with Angular, it's called Batarang and you can grab it [here](https://chrome.google.com/webstore/detail/angularjs-batarang/ighdmehidhipcmcojgiloacoafjmpfk) (<https://chrome.google.com/webstore/detail/angularjs-batarang/ighdmehidhipcmcojgiloacoafjmpfk>).

Happy coding.

Further reading

- **Structuring Angular Controllers** (<http://toddmotto.com/rethinking-angular-js-controllers>)
- **All about custom Angular filters** (<http://toddmotto.com/everything-about-custom-filters-in-angular-js>)
- **Controller as syntax** (<http://toddmotto.com/digging-into-angulars-controller-as-syntax>)
- **\$scope and \$rootScope event system \$emit, \$broadcast and \$on** (<http://toddmotto.com/all-about-angulars-emit-broadcast-on-publish-subscribing>)
- **Modular IIFE modules** (<http://toddmotto.com/minimal-angular-module-syntax-approach-using-an-iife>)
- This post, translated in **French!** (<http://vfsvp.fr/article/apprendre-angular-en-un-jour-le-guide-ultime>)
- This post, on **SpeakerDeck in slides** (<https://speakerdeck.com/toddmotto/angularjs-in-one-day>)
- **Code your own Directive** (<http://toddmotto.com/creating-an-angularjs-directive-from-one-of-your-existing-plugins-scripts>) from a custom script or plugin

Tweet 1,175 Like 617 +1 711

134 Comments Todd Motto

 dongguangming ▾

 Recommend 30  Share

Sort by Newest ▾



Join the discussion...

Matti Bragge • 3 months ago

Pretty nice article all-in-all, introducing the core concepts of Angular. But I'm afraid the "Differences in MVC and MVVM" chapter is very misleading if not outright wrong.

MVVM has three components (just like MVC), a View, a Model, and a ViewModel. The View uses two way data binding against the ViewModel, which again communicates with and gets updated by the Model. This architectural pattern was mainly invented to allow a UI/UX designer to work on the View-side, and an application developer on the Model-side.

What you described was just having a View and a Model, and using two-way data binding between them. And then you put out an example with no separation of concerns, and everything in the view.

^ | ▾ • Reply • Share ▾

Nissim Levy • 4 months ago

This is not a tutorial for someone who knows nothing about AngularJS. For example, very early in the tutorial the author mentions ng-show and ng-hide as though the learner has already used them. Also, directives are introduced before the author mentions a word about ng-model

^ | ▾ • Reply • Share ▾

vishwesh jha • 4 months ago

Awesome stuff mate keep posting

^ | ▾ • Reply • Share ▾

hrust4 • 4 months ago

Great article!! Quickly explained the basics of what it is to use AngularJS

^ | ▾ • Reply • Share ▾

ibbatta • 4 months ago

Love this guide.

It was very helpful for my initialization to angular

^ | ▾ • Reply • Share ▾

Declan Magee • 5 months ago

Awesome article for any beginner into Angular before the deep dive!

^ | ▾ • Reply • Share ▾

Samuel Mburu • 7 months ago

Awesome tutorial, gives a more grounded pragmatic view of what Angular has to offer and how to get started with Angular.

Keep it up!

Samuel

^ | v • Reply • Share ›

Mark Racette • 8 months ago

Thanks for the wonderful tutorial!

^ | v • Reply • Share ›

Antonio Pol • 9 months ago

Hi Todd, I am trying to make a app, and i have two template structure:

- Login template that has a login form

- Platform template that i use route-ui or route of AngularJS for nesting views

But, What is the best way to mix two structures templates in the same angular.module app?

Thanks

^ | v • Reply • Share ›

dxmaestro • 10 months ago

Hi man.Thanks for the time you took to write this up. I wish the guys at AngularJs would do the same and actually help us understand Angular.

^ | v • Reply • Share ›

Ty Cahill • 10 months ago

Fantastic guide! I've been advocating AngularJS for SPA development for a couple years, and I keep sending people to this guide for a clear and step-by-step explanation of Angular. Todd, thanks for taking the time to put this together!

^ | v • Reply • Share ›

RENISH B • a year ago

Marvelous Article.. very clear!

^ | v • Reply • Share ›

Xbr Mpqueeug • a year ago

In filters section if i check how the console is doing like this:

```
console.log(item);
```

```
console.log(item > $scope.lowerBound);
```

I see that is going trough all the items twice, why is that happening?

PS: second question: just by doing if(uppercase) angularjs knows that you are referring to "item"? How does angular do that?

thank you so much for this great tutorial!!

^ | v • Reply • Share ›

Samuel Mburu ➔ Xbr Mpqueeug • 7 months ago

Xbr,

I too had the same question regarding the uppercase. Turns out that you don't actually need that if (uppercase) block since Angular provides you with a built in uppercase filter: <https://docs.angularjs.org/api...>

I removed the if(uppercase) and it works look here: <http://jsfiddle.net/c89b0zvd/>

Turns out you can also chain filters together via the pipe "|"

Best of luck

Samuel

^ | v • Reply • Share ›

Bård • a year ago

Great write-up. Thanks for sharing!

^ | v • Reply • Share ›

Bruno Finger • a year ago

I would like to ask a question, and forgive me if it sounds rude, but when you say "frequented unnecessary compiling errors", do you really mean compiling? I mean, is there something really being compiled behind the scenes by the framework? I am just starting with Angular, so I'm not really sure when asking this.

^ | v • Reply • Share ›

Prashant Palikhe ➔ Bruno Finger • a year ago

I think it was a mistake from his part. I read it "parsing" error.

^ | v • Reply • Share ›

CowsRule • a year ago

I wonder what drugs the developer was on who decided to call it a "promise", considering the word "callback" has been in use since the dinosaurs.

^ | v • Reply • Share ›

Cricket La Chica ➔ CowsRule • a year ago

Promises and callbacks are actually two entirely different things! :-)

<http://stackoverflow.com/quest...>

4 ^ | v • Reply • Share ›

Andrei Duma ➔ CowsRule • a year ago

"Promise" was not invented with Angular. There are promises in functional languages like Racket (Scheme). They allow for delayed execution.

1 ^ | v • Reply • Share ›

Todd Motto Mod ➔ Andrei Duma • a year ago

And native promises in JavaScript now, new Promise(fn).then(success, error);

2 ^ | v • Reply • Share ›

Anjan • a year ago

Excellent, Awesome and much better than official documentations, Thank you :)

2 ^ | v • Reply • Share ›

ivp • a year ago

Great job!

^ | v • Reply • Share ›

jacob • a year ago

its really ultimate (y) Thank you :)

1 ^ | v • Reply • Share ›

sergio • a year ago

is angularjs good to use with a mobile application using cordova/phonegap?

^ | v • Reply • Share ›

Luis Alberto Romero Calderon ➔ sergio • a year ago

you may test this framework <http://ionicframework.com/>

^ | v • Reply • Share ›

Andrei Duma ➔ sergio • a year ago

I'd like to know that as well.

^ | v • Reply • Share ›

Todd Motto Mod ➔ Andrei Duma • a year ago

Depends what you mean by "good". Certainly do-able and we've done it before with great success.

^ | v • Reply • Share ›

Evan Hobbs • a year ago

Just awesome. Thanks!

^ | v • Reply • Share ›

basecss • a year ago

I think this is an one of the best Angular article and guide.

1 ^ | v • Reply • Share ›

Цэрэнбат Улаанхүү • a year ago

Thank you, good tutorial

^ | v • Reply • Share ›

Karl Keefer • a year ago

Thanks for writing this. Very helpful and clearly written.

^ | v • Reply • Share ›

monkeyboy • a year ago

Fantastic post, the official docs are dry and dusty, they should take a leaf out of your book

^ | v • Reply • Share ›

Amit Bajaj • a year ago

Thank you

^ | v • Reply • Share ›

Ritesh • a year ago

Thank you...this article/blog was very helpful to me.

^ | v • Reply • Share ›

MAK • a year ago

Thanks a lot Tedd, for this simple and to the point post about Angularjs. You made it very easy and simple to understand.

1 ^ | v • Reply • Share ›

AndrewGoel • a year ago

Great Article

^ | v • Reply • Share ›

Kevin Lin • a year ago

Thanks for the great guide! Very developer-friendly indeed.

^ | v • Reply • Share ›

Julia • a year ago

Really helpful article. Thanks!

^ | v • Reply • Share ›

Kunal Gandhi • a year ago

Awesome Post Buddy.. I am really thankful to you for this Article

^ | v • Reply • Share ›

Harry Green • a year ago

Excellent overview tutorial; I've recommended it to others.

^ | v • Reply • Share ›

Reptar • a year ago

Great Post!!! Very talented writer/educator. Thanks a lot for sharing your wealth of knowledge!

^ | v • Reply • Share ›

James Klein • a year ago

Very helpful for beginners and intermediates. Thanks!

^ | v • Reply • Share ›

Jonathan San Juan • a year ago

thanks for the great guide! it was the easiest to digest angular tutorial I found (for someone new to html and javascript)

5 ^ | v • Reply • Share ›

metric152 • a year ago

This was a fantastic intro to angular. I've wanted to check it out for a long time. Just enough to start digging into it.

3 ^ | v • Reply • Share ›

Lyndon • a year ago

Much thanks for taking the time to write this. It really helped!

1 ^ | v • Reply • Share ›

Alex B • a year ago

This helped a lot in understanding the basics of Angular.js, thanks!

^ | v • Reply • Share ›

Frédéric Roy • a year ago

Great article. Thanks!

1 ^ | v • Reply • Share ›

dilesh • a year ago

I have designed a small application using bootstrap and angularjs, I want to make it a app to use in android phone, How can I do this, Can any one help me with these queries ? (In short, how to make a native app/hybrid app of this) ?

7 ^ | v • Reply • Share ›

anonymous ➔ dilesh • 7 months ago

You can use ionic framework ? :)

^ | v • Reply • Share ›

[Load more comments](#)

ALSO ON TODD MOTTO

Methods to determine if an Object has a given property

9 comments • a year ago



Todd Motto — LOL @ me for that. It was a last min addition and I was all Boolean'd out. Will update this.

WHAT'S THIS?

AngularJS one-time binding syntax


17 comments • 8 months ago



Raf — im on angular 1.3.15 but ::vm.users part does not work. {{ ::field}} seems ok, but whenever i add it in the ng-repeat, it doesn't ...

A better way to \$scope, angular.extend, no more “vm = this”


45 comments • 3 months ago



Pascal Precht — Hey Todd! While I understand that using `angular.extend()` is a bit nicer because of not typing `this` all the time, ...

All about Angular’s \$emit, \$broadcast, \$on publish/subscribing

59 comments • a year ago



Olawajuwaju Doyin — This is simply great. Thanx bro