**AngularJS**

**Lesson 1. Your first app**

AngularJS is a JavaScript web framework aimed to make web apps simple to build and easy to maintain.

We'll start by building a simple AngularJS app. After making this app, we'll generalize a few steps that can be used to build more complex apps. By the end of this course, you'll be able to use this sequence of steps to jumpstart your own AngularJS apps.

1. Let's get started by making a simple AngularJS app. We'll explain each step in the next exercise.

In **app.js**, type in the contents exactly as you see here:

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

1. Open up **index.html**. Modify the <body> tag so it looks like this:

<body ng-app="myApp">

1. Open up **js/controllers/MainController.js**. Type in the contents exactly as you see here:

app.controller('MainController', ['$scope', function($scope) {

$scope.title = 'Top Sellers in Books';

}]);

1. Go to **index.html**. Modify the <div class="main"> tag so it looks like this:

<div class="main" ng-controller="MainController">

1. In **index.html** inside <div class="main">, modify the <h1>element so it looks like this:

<h1>{{ title }}</h1>

View the AngularJS app in the browser by visiting[http://localhost:8000](http://localhost:8000/). The "Top Sellers in Books" content appears as the heading of the page.

Awesome! You built an AngularJS app. How does it work?

1. In **app.js**, we created a new *module* named MyApp. A *module*contains the different components of an AngularJS app.
2. Then, in **index.html** we added <body ng-app="myApp">. The ng-appis called a *directive*. It tells AngularJS that the MyApp module will live within the <body> element, termed the application's *scope*. In other words, we used the ng-app directive to define the application scope.
3. In **MainController.js** we created a new *controller* namedMainController. A *controller* manages the app's data. Here we use the property title to store a string, and attach it to $scope.
4. Then, in **index.html**, we added<div class="main" ng-controller="MainController">. Like ng-app,ng-controller is a *directive* that defines the controller scope. This means that properties attached to $scope in MainController become available to use within <div class="main">.
5. Inside <div class="main"> we accessed $scope.title using {{ title }}. This is called an *expression*. Expressions are used to display values on the page.
6. The value of title showed up when we viewed the app in the browser.

**6.** Both the controller MainController and the view **index.html** have access to $scope. This means we can use $scope to communicate between the controller and the view. In the controller, change the value of title to your own string.

**7.** Likewise, any new properties attached to $scope will become available to use in the view. In the controller, attach promo to$scope, and set its value to your own string.

**8.** In the view under the <h1> element, add an <h2> element and use an expression to display promo on the page.

So far this is our typical workflow when making an AngularJS app:

1. Create a module, and use ng-app in the view to define the application scope.
2. Create a controller, and use ng-controller in the view to define the controller scope.
3. Add data to $scope in the controller so they can be displayed with expressions in the view.

**9.** Let's add more data to the controller and display them in the view. In the controller, attach another property to $scope named product. Set it equal to an object with the following properties:

{

name: 'The Book of Trees',

price: 19

}

**10.** Then, in **index.html** inside <p class="title">, access the product's name with product.name and display it using an expression.

**11.** In <p class="price">, access and display the product's price.

Well done! In the controller, you used an object to group together related data about a product. Then in the view, you used dot notation to display the values.

**12.** Currently the product price shows up as a number. It would be better to format it as a currency. Rather than change the data in the controller, let's use an AngularJS *filter* to format the data in the view.

In **index.html** in <p class="price">, change the expression to look like this:

{{ product.price | currency }}

We'll explain how this works in the next exercise.

Great! The product price changed from a number to a formatted currency. How does it work?

1. AngularJS gets the value of product.price.
2. It sends this number into the currency filter. The pipe symbol (|) takes the output on the left and "pipes" it to the right.
3. The filter outputs a formatted currency with the dollar sign and the correct decimal places.

In this way, filters help to separate the content in the controller from its presentation in the view.

**13.** AngularJS comes with [a few more built-in filters](https://docs.angularjs.org/api/ng/filter). Let's use two more.

In **MainController.js** inside $scope.product, add a third property named pubdate:

pubdate: new Date('2014', '03', '08')

**14.** In **index.html** inside <p class="date">, display the product'spubdate.

**15.** Format the product's pubdate by piping it to the date filter.

**16.** Format the product's name by piping it to the uppercase filter.

Let's do a quick review:

* A **module** contains the different components of an AngularJS app
* A **controller** manages the app's data
* An **expression** displays values on the page
* A **filter** formats the value of an expression

**17.** Let's add more data to the controller and display them in the view.

In the controller, delete the $scope.product object.

**18.** Attach a new property to $scope named products. Set it equal to an array of objects. Type in the contents exactly as you see here:

[

{

name: 'The Book of Trees',

price: 19,

pubdate: new Date('2014', '03', '08'),

cover: 'img/the-book-of-trees.jpg'

},

{

name: 'Program or be Programmed',

price: 8,

pubdate: new Date('2013', '08', '01'),

cover: 'img/program-or-be-programmed.jpg'

}

]

**19.** In the view inside <div class="main">, delete the<div class="col-md-6"> element.

Replace it with this HTML. Type in the contents exactly as you see here:

<div ng-repeat="product in products" class="col-md-6">

<div class="thumbnail">

<img src="img/the-book-of-trees.jpg">

<p class="title">{{ product.name }}</p>

<p class="price">{{ product.price | currency }}</p>

<p class="date">{{ product.pubdate | date }}</p>

</div>

</div>

You'll see that both products have the same cover image. Let's fix this bug in the next exercise. Click Next to continue.

Well done! You got both books in $scope.products to show up in the view. How does it work?

1. In the controller, we used products to store an array containing two objects.
2. Then in the view, we added <div ng-repeat="product in products">. Like ng-app and ng-controller, the ng-repeat is a directive. It loops through an array and displays each element. Here, the ng-repeatrepeats all the HTML inside <div class="col-md-6"> for each element in the products array.

In this way, ng-repeat shows both products in the $scope.products array. Instead of writing the same HTML twice as before, we just use ng-repeat to generate the HTML twice.

**20.** The problem now is that both products have the same image. Let's fix this.

In the view inside <div class="col-md-6">, replace

<img src="img/the-book-of-trees.jpg">

with

<img ng-src="{{ product.cover }}">

The ng-src is a directive that sets the <img> element's src to a property in the controller.

We've used a few directives so far - ng-app, ng-controller, ng-repeat, and ng-src. What can we generalize about directives?

Directives bind behavior to HTML elements. When the app runs, AngularJS walks through each HTML element looking for directives. When it finds one, AngularJS triggers that behavior (like attaching a scope or looping through an array).

**21.** In the controller, add two of your favorite books to the$scope.products array.

The view will update as the ng-repeat loops the new elements in the $scope.products array.

So far we've made a static AngularJS app by adding properties in the controller and displaying them in the view. AngularJS is a framework for building dynamic web apps, so let's start to make this app interactive.

**22.** In the controller in the $scope.products array, add a new property named likes to each element. Set all likes properties to 0.

**23.** In the view under <p class="date">, type in a rating element:

<div class="rating">

<p class="likes">+ </p>

</div>

Inside <p class="likes">, display a product's likes using an expression.

**24.** Back in the controller after $scope.products, attach a new property to $scope named plusOne. Set it equal to function. Type in the contents exactly as you see here:

function(index) {

$scope.products[index].likes += 1;

};

**25.** In the view modify <p class="likes"> to look like this:

<p class="likes" ng-click="plusOne($index)">

View the AngularJS app in the browser. Click on the +0 in each product tile.

Great! Each time you click on the number of likes, the number goes up. How does it work?

1. The ng-click is a directive. When <p class="likes"> is clicked,ng-click tells AngularJS to run the plusOne() function in the controller.
2. The plusOne() function gets the index of the product that was clicked, and then adds one to that product's likes property.

Notice that the plusOne() doesn't interact with the view at all; it just updates the controller. Any change made to the controller shows up in the view.

**26.** In the controller in the $scope.products array, add a new property named dislikes to each element. Set dislikes to 0.

**27.** In the view under <p class="likes">, add an element for<p class="dislikes">. Then display a product's dislikes using an expression.

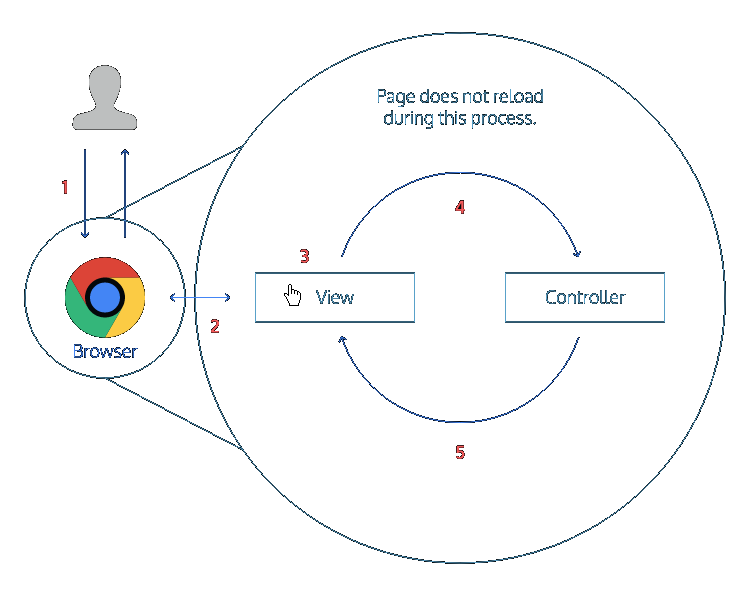
**28.** Back in the controller after $scope.products, attach a new property to $scope named minusOne. Set it equal to a function that adds one to a product's dislikes property. Adapt the code from theplusOne() function to do this.

**29.** In the view, use ng-click to trigger the minusOne() function when<p class="dislikes"> is clicked.

View the AngularJS app in the browser. Click on the -0 in each product tile.

Congratulations! You built an AngularJS app from scratch. What can we generalize so far?

1. A user visits the AngularJS app.
2. The *view* presents the app's data through the use of *expressions*, *filters*, and *directives*. Directives bind new behavior HTML elements.
3. A user clicks an element in the view. If the element has a directive, AngularJS runs the function.
4. The function in the *controller* updates the state of the data.
5. The view automatically changes and displays the updated data. The page doesn't need to reload at any point.



**Lesson 2. Directives**

Here's an AngularJS app for a mobile app store:

* In the controller **MainController.js**, there are three objects $scope.move,$scope.shutterbugg, and $scope.gameboard that each contain info about an app, like its title and price.
* In the view **index.html** in the .main section, each app is displayed inside a .card div.

But looking at the view, the same code is written over and over again to display each app. This is repetitive and error-prone. Let's fix this.

**1.** In the new file **js/directives/appInfo.js**, type in this code:

app.directive('appInfo', function() {

return {

restrict: 'E',

scope: { info: '=' },

templateUrl: 'js/directives/appInfo.html'

};

});

**2.** Include this new JavaScript file in **index.html** in line 48 as a <script> element.

**3.** In the new file **js/directives/appInfo.html**. Type in this HTML to display an app's info:

<img class="icon" ng-src="{{ info.icon }}">

<h2 class="title">{{ info.title }}</h2>

<p class="developer">{{ info.developer }}</p>

<p class="price">{{ info.price | currency }}</p>

**4.** In **index.html**, replace the contents of the first .card div with the new <app-info>element:

<div class="card">

<app-info info="move"></app-info>

</div>

**5.** Do the same for the second and third .card divs. Replace their contents with<app-info info="shutterbugg"></app-info> and<app-info info="gameboard"></app-info>

View the AngularJS app in the browser by typing [http://localhost:8000](http://localhost:8000/).

What did we just do? We wrote code to teach the browser a new HTML element <app-info>, and used it in the view to display each app's details.

First in **js/directives/appInfo.js**, we made a new *directive*. We used app.directive to create a new directive named'appInfo'. It returns an object with three options:

1. restrict specifies how the directive will be used in the view. The 'E' means it will be used as a new HTML element.
2. scope specifies that we will pass information into this directive through an attribute named info. The = tells the directive to look for an attribute named info in the <app-info> element, like this:

<app-info info="shutterbugg"></app-info>

The data in info becomes available to use in the template given by templateURL.

1. templateUrl specifies the HTML to use in order to display the data in scope.info. Here we use the HTML in**js/directives/appInfo.html**.

Looking at **js/directives/appInfo.html**, we define the HTML to display details about an app, like its title and price. We use expressions and filters to display the data.

Then in **index.html** we use the new directive as the HTML element <app-info>. We pass in objects from the controller's scope ($scope.shutterbugg) into the <app-info> element's info attribute so that it displays.

Why is creating your own directives useful?

1. **Readability.** Directives let you write expressive HTML. Looking at **index.html** you can understand the app's behavior just by reading the HTML.
2. **Reusability.** Directives let you create self-contained units of functionality. We could easily plug in this directive into another AngularJS app and avoid writing a lot of repetitive HTML.

**6.** In the controller, there is a new property $scope.forecast. Display this property in the view by creating another <div class="card"> element containing an <app-info> element.

We know that AngularJS comes with a few built-in directives like ng-repeat and ng-click.

We've seen that AngularJS makes it possible to create your own custom directives, such as<app-info>.

We can use Angular's built-in directives together with custom directives to create more readable apps.

For reference, here's how to use ng-repeat:

<div ng-repeat="product in products">

<img ng-src="{{ product.cover }}">

<p class="title">{{ product.name }}</p>

</div>

**7.** In the controller, create a new property $scope.apps. Set it equal to an array of objects:

[

{

icon: 'img/move.jpg',

title: 'MOVE',

developer: 'MOVE, Inc.',

price: 0.99

},

{

icon: 'img/shutterbugg.jpg',

title: 'Shutterbugg',

developer: 'Chico Dusty',

price: 2.99

}

]

**8.** Add two more objects to the array describing your favorite apps. Make sure to define the four properties for each app.

**9.** In the view, use ng-repeat to loop through $scope.apps and display each element. To do this, add ng-repeat to a <div class="card">, and then use the custom directive <app-info>to display each element.

Directives are a core feature of AngularJS. So far we've used custom directives to simply display static content, but they can do more than this. It's possible to bake interactivity into directives.

Let's start creating another directive that reacts to a user's click.

**10.** In the new file **js/directives/installApp.js**, create a new directive named installApp. Refer to the appInfo directive for an example:

* use app.directive to create a new directive named installApp
* use the restrict option to create a new Element
* set the scope option to an empty object {}
* use the templateUrl option to tell this directive to use the**js/directives/installApp.html** file

**11.** Include this new JavaScript file in **index.html** as a <script> element.

**12.** In the installApp directive, add a fourth option named link, and type in the following function:

function(scope, element, attrs) {

scope.buttonText = "Install",

scope.installed = false,

scope.download = function() {

element.toggleClass('btn-active');

if(scope.installed) {

scope.buttonText = "Install";

scope.installed = false;

} else {

scope.buttonText = "Uninstall";

scope.installed = true;

}

}

}

We used app.directive to create a new directive named 'installApp'.

* The directive contains the three options restrict, scope, and templateUrl that we saw before in the'appInfo' directive.
* It also contains a fourth option link. The link is used to create interactive directives that respond to user actions.

The link function takes three inputs:

1. scope refers to the directive's scope. Any new properties attached to $scope will become available to use in the directive's template.
2. element refers to the directive's HTML element.
3. attrs contains the element's attributes.

Inside the link function, there are two properties buttonText and installed, and the function download(). We'll use these in the directive's template next.

**13.** Next, write the directive's template:

In the new file **js/directives/installApp.html**. Type in the following HTML:

<button class="btn btn-active" ng-click="download()">

{{ buttonText }}

</button>

The template uses Angular's built-in ng-click directive. When the button is clicked, ng-click will tell AngularJS to run the download() function in the directive.

The download() function uses the scope.installed property to check if an app is installed. When an app is installed,download() does three things:

* toggles the .btn-active class
* changes the button text to "Uninstall"
* changes scope.installed to true

**14.** Finally, use the new directive in the view:

In **index.html**, add the new <install-app> element inside the .card div under the <app-info> element.

**15.** View the AngularJS app in the browser by typing [http://localhost:8000](http://localhost:8000/).

Well done! An "Install" button now shows up under each app. When you click the button, it changes to an "Uninstall" button. When you click it again, it changes back to an "Install" button.

What can we generalize so far?

* Directives are a powerful way to create self-contained, interactive components. Unlike jQuery which adds interactivity as a layer on top of HTML, AngularJS treats interactivity as a native component of HTML.

**Lesson 3. Services**

So far we've made AngularJS apps by adding data to a controller, and then displaying it in a view.

But what happens when the data contains hundreds of items, or if it's constantly changing like weather or financial data? Hardcoding data into a controller won't work anymore.

A better solution is to read the live data from a server. We can do this by creating a *service*.

**1.** In the browser frame on the right, visit <https://s3.amazonaws.com/codecademy-content/courses/ltp4/forecast-api/forecast.json>. It's a JSON object containing acity\_name and an array days containing weather data for the next five days.

**2.** Create a service named forecast that fetches the weather data from the server. In the new file **js/services/forecast.js**. Type in this code exactly as you see here:

app.factory('forecast', ['$http', function($http) {

return $http.get('https://s3.amazonaws.com/codecademy-content/courses/ltp4/forecast-api/forecast.json')

.success(function(data) {

return data;

})

.error(function(err) {

return err;

});

}]);

**3.** Include **js/services/forecast.js** in **index.html** in line 62 as a new <script> element.

**4.** In the controller, modify MainController by passing in the forecast service, like this:

app.controller('MainController', ['$scope', 'forecast', function($scope, forecast) {

// ...

}]);

**5.** Inside MainController, use the forecast service to save the weather data into$scope.fiveDay, like this:

forecast.success(function(data) {

$scope.fiveDay = data;

});

Well done! The city\_name now shows up in the view. How does it work?

1. First in **js/services/forecast.js**, we made a new *service*. We used app.factory to create a new service named forecast
2. The forecast service needs to use AngularJS's built-in $http to fetch JSON from the server. Therefore, we add $http to the forecast service as a dependency, like this:

['$http', function($http) {

// ...

}]

Now $http is available to use inside forecast.

1. Then, inside forecast, we use $http to construct an HTTP GET request for the weather data. If the request succeeds, the weather data is returned; otherwise the error info is returned.
2. Next in the controller, we used the forecast service to fetch data from the server. First we addedforecast into MainController as a dependency so that it's available to use. Then within the controller we used forecast to asynchronously fetch the weather data from the server and store it into $scope.fiveDay
3. As before, any properties attached to $scope become available to use in the view. This means in**index.html**, we can display the city\_name using an expression as done before.

**6.** In the browser, visit <https://s3.amazonaws.com/codecademy-content/courses/ltp4/forecast-api/forecast.json>. Looking at the format of the data in the days array, each day has four pieces of data - datetime, icon, high, and low.

**7.** Visit [http://localhost:8000](http://localhost:8000/) to view the AngularJS app. Let's finish the view so that it displays the weather for each day.

Notice in the view, we're using ng-repeat to loop through each item in the days array. Inside this loop, display a day's four pieces of data.

Remember to use ng-src to display an image.

**8.** Use the date filter to format the datetime.

Why are services useful? Instead of filling the controller with code to fetch weather data from a server, it's better to move this independent logic into a service so that it can be reused by other parts of the app.

What can we generalize so far?

* Directives are a way to make standalone UI components, like <app-info>
* Services are a way to make standalone communication logic, like forecast which fetches weather data from a server

**Lesson 4. Routing**

So far we've made AngularJS apps that display data in a single view **index.html**.

But what happens when the app grows and needs to display more information? Stuffing more code to a single view will quickly make things messy.

A better solution is to use multiple templates that display different pieces of data based on the URL that the user is visiting. We can do this with Angular's *application routes*.

**1.** In **index.html** under the .header section, type in the code exactly as you see here:

<div ng-view></div>

**2.** View the AngularJS app in the browser by typing <http://localhost:8000/>.

**3.** In **app.js** under the angular.module, type in this code:

app.config(function ($routeProvider) {

$routeProvider

.when('/', {

controller: 'HomeController',

templateUrl: 'views/home.html'

})

.otherwise({

redirectTo: '/'

});

});

Great! A gallery of images shows up. How does it work?

1. In **app.js** inside the app.config() method, we use Angular's $routeProvider to define the application routes.
2. We used .when() to map the URL / to to the controller HomeController and the templatehome.html. The HomeController uses the service **js/services/photos.js** to fetch the array of all photos from <https://s3.amazonaws.com/codecademy-content/courses/ltp4/photos-api/photos.json>and stores it into $scope.photos. The home.html uses ng-repeat to loop through each item in thephotos array and display each photo.
3. Otherwise if a user accidentally visits a URL other than /, we just redirect to / using .otherwise().
4. Now when a user visits /, a view will be constructed by injecting home.html into the<div ng-view></div> in **index.html**.

**4.** In **app.js** between the .when() and .otherwise(), add another .when() that maps the URL/photos/:id to the controller PhotoController and the template **views/photo.html**.

**5.** Visit <http://localhost:8000/> and click on a photo. A detail page about that photo should appear.

What did we just do?

1. In **app.js**, we mapped a URL to PhotoController and photo.html. We added a variable part named id to the URL, like this: /photos/:id.
2. In **PhotoController**, we used Angular's $routeParams to retrieve id from the URL by using $routeParams.id. Notice that we injected both $routeParams and the photosservice into the PhotoController dependency array to make them available to use inside the controller.
3. Then to fetch an individual photo, we first used the photos service to fetch the array of photos from the server, and then used $routeParams.id to access the specific photo by its index.
4. As before, any properties attached to $scope become available to use in the view. This means in **photo.html**, we can display the photo's detail using expressions as done before.

Notice that when you click on links, the app doesn't do a full reload. Only the part of the view specified by <div ng-view></div> changes.

**6.** Finish the template so that it displays a photo's details.

First, in the browser, visit <https://s3.amazonaws.com/codecademy-content/courses/ltp4/photos-api/photos.json>. Looking at the format of the data in the array, each photo has six pieces of data - title, author, url, pubdate, upvotes, andviews.

**7.** In the template **photo.html**, display a photo's remaining five pieces of data. Then visit<http://localhost:8000/> in the browser and click on a photo. A detail page about that photo should appear.

**8.** Use the number filter to format the views and upvotes.

**9.** Use the date filter to format the pubdate.

**10.** View the result in the browser. The photo's views, upvotes, and date should now be formatted.

Why are routes useful? Instead of filling a single view with more code than needed, routes let us map URLs to self-contained controllers and templates. Furthermore, now that the app has URLs, users can easily bookmark and share the app's pages.

What can we generalize so far?

* Directives are a way to make standalone UI components, like <app-info>
* Services are a way to make standalone communication logic, likeforecast which fetches weather data from a server
* Routes are a way to manage apps containing more views.