

# AngularJS

Dr. Patrick Young, Stanford University  
CS193C

In this handout, we'll learn some of the basics of using AngularJS version 1. There is an Angular 2, but it uses TypeScript (which is a typed language based on JavaScript) and requires a lot more setup work. We'll take a look at both TypeScript and Angular 2 in class, but for your homework, you'll just experiment with the simpler Angular 1.

When reading through this, keep in mind what I've talked about in lecture. Working with a *framework* you are not in control anymore. You need to work within the framework.

You can download the Angular 1 code at this address:

<https://code.angularjs.org/1.6.5/angular.js>

## Creating an AngularJS Webpage

To start out with, we'll begin with a standard HTML5 starter file and will modify it with a few additions:

```
<!DOCTYPE html>
<html ng-app>
<head>
<meta charset="UTF-8" />
<title></title>
<script src="angular.js"></script>
</head>
<body>
</body>
</html>
```

Here we've added the ng-app attribute to our HTML file. This tells angular that we have a Angular application. This is called an Angular Directive. AngularJS adds new syntax called directives giving instructions to the web browser. These come in different forms, but the addition of new attributes is a technique used throughout AngularJS.<sup>1</sup>

In addition I've added a script tag instructing my webpage to download the angular code.

**Note:** For brevities sake, most of the remaining examples in this handout will start with the body tag. When working with actual files, you should include everything starting with the DOCTYPE. Don't forget to include the ng-app directive and don't forget to load the angular.js code!

---

<sup>1</sup> AngularJS's new HTML attribute directives will not validate. If you want to validate these in HTML 5 you can add a "data-" in front of the name of the directive for example changing ng-app to data-ng-app. Angular will still recognize these, but the HTML5 validator will no longer complain.

## Angular Data and Angular Bindings

Angular applications store data and allow presentation of that data. The angular data is typically stored on a controller which is part of the MVC Model View Controller paradigm we talked about in class. Before we get to that though we can simply store data associated with the application itself. This will allow us to experiment with AngularJS presentation of data in a still relatively simple environment.

We can add data using the ng-init command. You can add an ng-init command to any of your tags and the code in the ng-init should execute before AngularJS begins its regular work. Let's add an ng-init command to our body tag, so our body tag now looks like this:

```
<body ng-init="univ='Stanford'; mascot='Cardinal';">
```

Pay particular attention to the mix of single and double quotes. Note that the ng-init attribute's value is one big long double-quoted string that contains smaller single-quoted strings in it.

Now, let's add something to actually output the values we've told Angular about:

```
<body ng-init="univ='Stanford'; mascot='Cardinal';">  
  I go to {{univ}}. Our mascot is {{mascot}}.  
</body>
```

If you load this in the browser you should see:

I go to Stanford. Our mascot is Cardinal.

What happened? The sections in the body which contain double curly braces are called AngularJS bindings. They contain Angular expressions. These expressions are automatically executed by Angular and replaced by whatever their values are. In this case {{univ}} was replaced by the value we assigned to univ. {{mascot}} acted similarly.

**Warning:** Angular values are actually properties on angular objects and are not global variables (or properties of the window object). If you try this:

```
<body ng-init="univ='Stanford'; mascot='Cardinal';">  
  I go to {{univ}}. Our mascot is {{mascot}}.  
  <script>  
    console.log(univ);  ///// BAD DOES NOT WORK  
  </script>  
</body>
```

It won't work. univ is not a regular variable. Similarly if you get rid of the ng-init and try something like this it won't work either:

```
<script>  
var univ='Stanford';  ///// BAD DOES NOT WORK  
var mascot='Cardinal';  
</script>  
I go to {{univ}}. Our mascot is {{mascot}}.  
</body>
```

The Angular bindings such as {{univ}} are looking for Angular values not regular variables.

As I mentioned above, the Angular bindings contain Angular expressions. As the name implies you can put more complex expressions than simply variables in them. For example if we do this:

```
<body ng-init="univ='Stanford'; mascot='Cardinal';">
  {{univ + ' ' + mascot}}.
</body>
```

It will output:

Stanford Cardinal

## Working with Lists

When working with lists of information, AngularJS provides a very handy ng-repeat directive. When this directive is placed on an HTML element, that element is duplicated once for every item in an array. Here's an example:

```
<body ng-init="dorms=['FloMo', 'Wilbur', 'Stern', 'Lag'];">
<h1>Stanford Dorms</h1>
<ul>
  <li ng-repeat="dorm in dorms">{{dorm}}</li>
</ul>
</body>
```

This creates the output:

### Stanford Dorms

- FloMo
- Wilbur
- Stern
- Lag

Because the <li> contains an ng-repeat directive it is duplicated with each copy having a different value for the dorm property.

Please note that while the ng-repeat is very commonly used to create lists, it can be used in any tag. We could use:

```
<p ng-repeat="dorm in dorms">{{dorm}} is a Stanford Dorm</p>
```

or even:

```
<div ng-repeat="dorm in dorms">
  <h2>{{dorm}}</h2>
  <p>{{dorm}} is a Stanford Dorm</p>
</div>
```

We could create even more interesting combinations if the items in our array were complex objects instead of simply strings. Let's try that:<sup>2</sup>

---

<sup>2</sup> I'm afraid I couldn't find the actual dorm capacities online, so these are only approximations.

```

<body ng-init="dorms=[{name: 'FloMo',capacity: 470},
                      {name: 'Wilbur', capacity: 700},
                      {name: 'Stern', capacity: 600},
                      {name: 'Lag', capacity: 400}];">
  <h1>Stanford Dorms</h1>
  <div ng-repeat="dorm in dorms">
    <h2>{{dorm.name}}</h2>
    <p>{{dorm.name}} houses {{dorm.capacity}} students.</p>
  </div>

```

This creates one section for each of the dorms. Here's a sample output:

## FloMo

FloMo houses 470 students.

## Linking Data with Controls

So far we've seen how we can display Angular's data on the webpage. Actually this can be a bidirectional relationship. In this example we create a text field on our webpage and use the `ng-model` directive to create a relationship between the text field and an Angular data property. Here's our HTML:

```

<body>
  <input type="text" ng-model="example" /> {{example}}
</body>

```

So what does it do? The `ng-model="example"` tells Angular that this text field should be bound to the Angular `example` property. The `{{example}}` is, of course, an Angular expression telling the system to replace the `{{example}}` text with the actual value of the `example` property. This means that the text outside the field will match whatever is typed into it, for example:

Go Cardinal

or:

CS is Fun!!!

This works with other controls as well. If we use

```

<body>
  <input type="checkbox" ng-model="example" /> {{example}} <br />
  <select ng-model="chosen">
    <option>Greater</option>
    <option>Equal</option>
    <option>Smaller</option>
  </select>
  {{chosen}}
</body>

```

This creates outputs like this:

☒ true  
 Smaller

## Filtering Outputs

AngularJS filters can be used to determine what items get used in an ng-repeat. Let's take a look at an example:

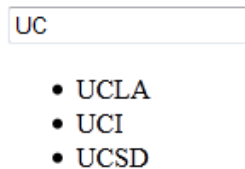
```
<body ng-init="universities=['Stanford','UCLA','UCI','UCSD','Cal'];">

<input type="text" ng-model="searchString" />

<ul>
<li ng-repeat="univ in universities | filter: searchString">{{univ}}</li>
</ul>
</body>
```

What does this do?

The text field creates an angular property named searchString. The ng-repeat would normally list all the universities in the array. However, we've added on the | filter: searchString. This tells angular to pass each university through a filter comparing it to the searchString. If the searchString is not a substring of the university, it does not pass through the filter. Here's a screenshot showing the webpage in action:



A screenshot of a web application. At the top, there is a text input field containing the text "UC". Below the input field, there is a bulleted list of university names: "UCLA", "UCI", and "UCSD".

Neither Stanford nor Cal contain a "UC" in them, so neither go through the filter.<sup>3</sup>

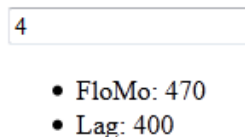
If we have complex objects, we can still filter through them. We can use a single search string like this:

```
<body ng-init="dorms=[{name: 'FloMo', capacity: 470},
                      {name: 'Wilbur', capacity: 700 },
                      {name: 'Stern', capacity: 600},
                      {name: 'Lag', capacity: 400}];">

<input type="text" ng-model="searchString" />

<ul>
<li ng-repeat="dorm in dorms | filter: searchString">
  {{dorm.name}}: {{dorm.capacity}}</li>
</ul>
</body>
```

This filters items in any of the fields:



A screenshot of a web application. At the top, there is a text input field containing the text "4". Below the input field, there is a bulleted list of dorm names and their capacities: "FloMo: 470" and "Lag: 400".

---

<sup>3</sup> What's happening under the hood is that all items in the original array are passed through the filter, which creates a new array. The new array is used for the ng-repeat action.

Here we've picked up the 4s in the capacity field, but if one of the dorm names had had a 4 in it, that would have passed through the filter as well.

For more complex situations, we can actually have the ng-models create properties on a single object like this:

```
<body ng-init="dorms=[{name: 'FloMo', capacity: 470},
                        {name: 'Wilbur', capacity: 700 },
                        {name: 'Stern', capacity: 600},
                        {name: 'Lag', capacity: 400}];">

Name: <input type="text" ng-model="searchitem.name" /> <br />
Capacity: <input type="text" ng-model="searchitem.capacity" />

<ul>
<li ng-repeat="dorm in dorms | filter: searchitem">{{dorm.name}}:
{{dorm.capacity}}</li>
</ul>
</body>
```

Now we have two search fields, each contributing a property to a larger search item. Now we can search either on name or capacity (or both):

Name: <input type="text" value="r"/>	Name: <input type="text"/>	Name: <input type="text" value="f"/>
Capacity: <input type="text"/>	Capacity: <input type="text" value="4"/>	Capacity: <input type="text" value="4"/>
<ul style="list-style-type: none"> <li>• Wilbur: 700</li> <li>• Stern: 600</li> </ul>	<ul style="list-style-type: none"> <li>• FloMo: 470</li> <li>• Lag: 400</li> </ul>	<ul style="list-style-type: none"> <li>• FloMo: 470</li> </ul>

## Working with a Controller

In a real application, the data would be separated from the presentation of the data in true Model/View/Controller fashion. To do this in AngularJS, we create a separate controller function. This function gets initialized when our file loads up. Typically this is done in a separate file. Here's what our new HTML file will look like:

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
<meta charset="UTF-8" />
<title></title>
<script src="angular.js"></script>
<script src="controller-basic.js"></script>
</head>
<body>
<div ng-controller="myController">
Name: <input type="text" ng-model="searchitem.name" /> <br />
Capacity: <input type="text" ng-model="searchitem.capacity" />

<ul>
<li ng-repeat="dorm in dorms | filter: searchitem">{{dorm.name}}:
{{dorm.capacity}}</li>
</ul>
</div>
</body>
</html>
```

This looks very similar to what we had before except:

1. We've given the ng-app attribute in the <body> tag an explicit name.
2. We've added a new script tag to download our controller file.
3. We've removed the ng-init which was initializing our data. It's the controller's job to initialize the data.
4. I've added a new div with an ng-controller attribute. This could have gone on the body, but in a real application there could potentially be several controllers, this emphasizes that.

Here's the actual controller file:

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

myApp.controller('myController', function($scope) {
    $scope.dorms=[{name: 'FloMo', capacity: 470},
                  {name: 'Wilbur', capacity: 700},
                  {name: 'Stern', capacity: 600},
                  {name: 'Lag', capacity: 400}];
});
```

We create an angular module with the name matching the one we used in the HTML file's <body> tag. We then add a controller to that module. We give the controller a name that matches the one we used in our HTML file and the controller function itself sets up the data.

We specify a special \$scope parameter (AngularJS uses the \$ to denote special values, don't use these in your own names). Any properties assigned to \$scope will be retrieved as variables in AngularJS expressions. The \$scope works very similarly to the Window object in standard browser based JavaScript. Remember in standard JavaScript everything is scoped to the Window and all variables are actually properties of the window. In AngularJS the variables are actually properties of the \$scope.

Let's take a look at an example with two controllers. Here's our HTML file (I've removed the search fields to keep the example smaller):

```

<!DOCTYPE html>
<html ng-app="myApp">
<head>
<meta charset="UTF-8" />
<title></title>
<script src="angular.js"></script>
<script src="controller-two.js"></script>
</head>
<body>
<div ng-controller="myController">
<ul>
<li ng-repeat="dorm in dorms">{{dorm.name}}:
{{dorm.capacity}}</li>
</ul>
</div>
<div ng-controller="uciController">
<ul>
<li ng-repeat="dorm in dorms">{{dorm.name}}:
{{dorm.capacity}}</li>
</ul>
</div>
</body>
</html>

```

We now have two separate controllers, each corresponding to a different div in the HTML file. The first div uses my original controller and stores information about Stanford dormitories. The second div uses a new controller and stores information about UC Irvine dormitories. Here's what the controller file looks like:

```

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

myApp.controller('myController', function ($scope) {
    $scope.dorms=[{name: 'FloMo',capacity: 470},
                  {name: 'Wilbur', capacity: 700},
                  {name: 'Stern', capacity: 600},
                  {name: 'Lag', capacity: 400}];
});

myApp.controller('uciController', function ($scope) {
    $scope.dorms=[{name: 'Middle Earth',capacity: 1500},
                  {name: 'Mesa Court', capacity: 1200}];
});

```

Notice that although both controllers use the exact same property name – dorms – they have different scopes. So the upper div ends up displaying dorms from Stanford and the lower div displays dorms from UC Irvine.

While my examples show the data hard coded, in a real application, the controller would likely load at least some of the data from an actual database.