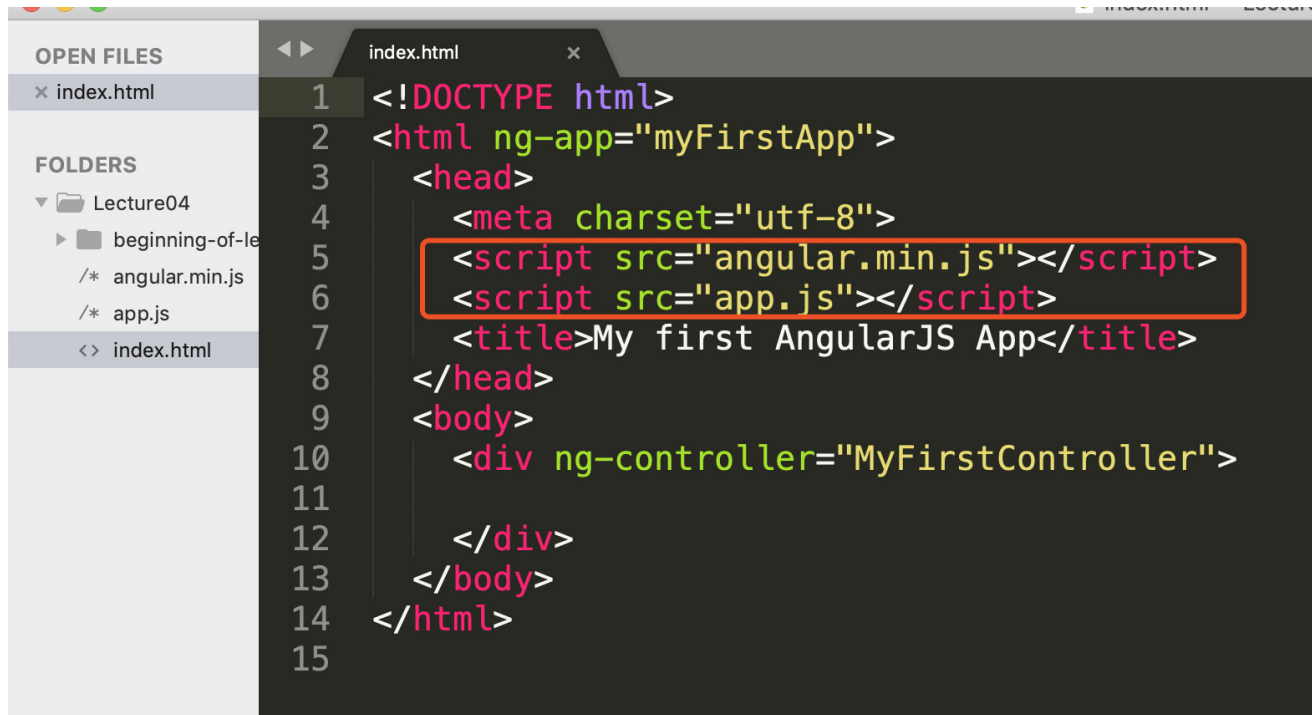


AngularJS Installation & Basics

To get the `angular.min.js`, go to the angularjs website and download it.



```
1 <!DOCTYPE html>
2 <html ng-app="myFirstApp">
3   <head>
4     <meta charset="utf-8">
5     <script src="angular.min.js"></script>
6     <script src="app.js"></script>
7     <title>My first AngularJS App</title>
8   </head>
9   <body>
10    <div ng-controller="MyFirstController">
11
12    </div>
13  </body>
14 </html>
15
```

And obviously, we want our `app.js` to follow `angular.min.js`, because this is HTML, and browsers interpret these lines sequentially. So since we're going to be using AngularJS inside of our `app.js`, we want that to be processed first and then be available to use inside `app.js`.

We want to make sure that no local variables bleed into the global scope. Once we've defined our IIFE, we're ready to start our AngularJS application.

And the very first thing we want to do is define our main app, the thing that's going to be responsible for some chunk of HTML in our `index.html`. So the thing that AngularJS exposes on the global scope is really just one particular object, and that object is called `angular`.

And the first thing that this module method takes is the name of our application. And the second is a list of dependencies that our module or our application is going to need, in the form of an array.

We can now go ahead and hook up this `myFirstApp` into our `index.html`. There's a special attribute that we could place really anywhere we want, but usually we want to put it on the very outer tag that you want the Angular application to be responsible for. And the attribute is called `ng-app`, which stands for Angular application. And we want to name it exactly the same way we named it in our `app.js`, which is `myFirstApp`. So this module has now been bound, this AngularJS app has been bound to our HTML (because we put the attribute in the `html` tag).

```

1 <!DOCTYPE html>
2 <html ng-app="myFirstApp">
3   <head>
4     <meta charset="utf-8">
5     <script src="angular.min.js"></script>
6     <script src="app.js"></script>
7     <title>My first AngularJS App</title>
8   </head>
9   <body>
10    <div ng-controller="
11      MyFirstController">
12
13    </div>
14  </body>
15 </html>

```

Remove the semicolon and since module function returns module instance, we can just go ahead and chain the next call to `.controller`. And .controller function is really the way we define the view model of our view, our view being `index.html`. It takes the name of our view model, or the name of our controller, and we'll call it `MyFirstController`. And it takes a function that defines the functionality for that particular controller.

Lecture 5

And now what we're going to do is we're going to use a special object that AngularJS provides for us in order to share data between our view model and our view. And that special object is called \$scope. Whenever, by the way, you see inside AngularJS **a dollar sign** in front of some variable name, it means that this is something that is reserved for Angular, this is something that Angular provides. There's nothing really stopping you from naming your own variables with the dollar sign, but the convention is don't do that.

```

5
6 .controller('MyFirstController', function ($scope) {
7   $scope.name = "Yaakov";
8   $scope.sayHello = function () {
9     return "Hello Coursera!";
10  };
11 });

```

Now this `name` and `sayHello` are actually now sitting on the scope, and this scope is available inside my `ng-controller` or anything below it or inside of it that this `ng-controller` is in charge of.

```
<h1>My first AngularJS App</h1>
<div ng-controller="MyFirstController">
  <input type="text" ng-model="name">
  Inside my input is: {{name}}
</div>
```

And I'll use a special Angular attribute called **ng-model** to tell it that I want the value of this input attribute to always be equal to something on my scope, something called name.

Lecture 7: What's Behind the "Magic": Custom HTML Attributes

It is not a standard HTML attribute, and the browser, the software that runs the browser tries to interpret every one of these tags and every one of these attributes. And it doesn't really know what to do with this greeting attribute, and therefore just ignores it. However, there are methods and functions in JavaScript to let us get at the actual attribute and at the value.

Previous Knowledge

IIFE

An **IIFE** (Immediately Invoked Function Expression) is a [JavaScript function](#) that runs as soon as it is defined.

```
(function () {
  statements
})();
```

It is a design pattern which is also known as a [Self-Executing Anonymous Function](#) and contains two major parts:

1. The first is the anonymous function with lexical scope enclosed within the [Grouping Operator](#) `()`. This prevents accessing variables within the IIFE idiom as well as polluting the global scope.
2. The second part creates the immediately invoked function expression `()` through which the JavaScript engine will directly interpret the function.

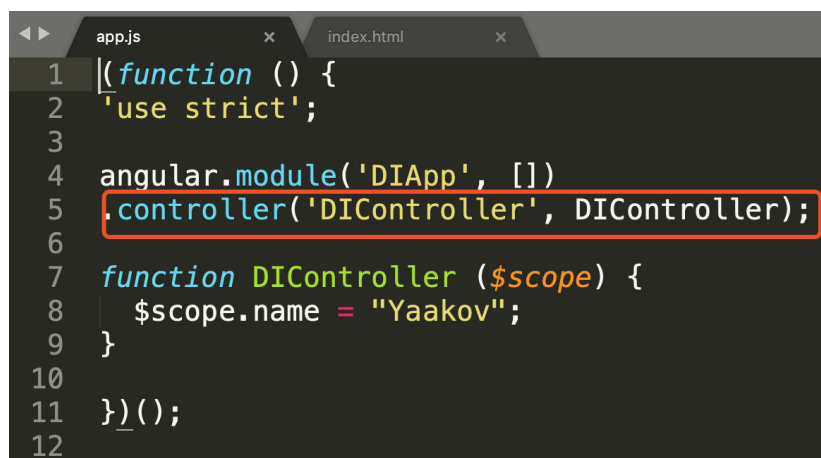
Assigning the IIFE to a variable stores the function's return value, not the function definition itself.

```
var result = (function () {  
    var name = "Barry";  
    return name;  
})();  
// Immediately creates the output:  
result; // "Barry"
```

Dependency Injection

Dependency injection is yet another design pattern. We've seen model, view, ViewModel, and now there's another called dependency injection. So, it's another design pattern that implements something called inversion of control (IoC) for resolving dependencies. (imagine shopping cart and check out using credit cards from different banks)

Client gets called with the dependency by some system, in our case, the 'system' is AngularJS. Client is not responsible for instantiating the dependency.



```
1 | (function () {  
2 |   'use strict';  
3 |  
4 |   angular.module('DIApp', [])  
5 |     .controller('DIController', DIController);  
6 |  
7 |   function DIController ($scope) {  
8 |     $scope.name = "Yaakov";  
9 |   }  
10 |  
11 |   _();  
12 | }
```

We are creating a DIController, and what I'm passing next is the value of the function.

Pretty much anything in Angular with a **dollar sign** in front of it, not only does it belong to Angular, but it's also referred to as a service. So from now on, when I refer to \$scope, I'm really going to just refer to the scope service.

What Angular does is it can actually go ahead and parse out these names and find a matching service and then go ahead and instantiate those services and then call DIController with those instantiations. The service that does all this in Angular is called \$inject.

```

7  function DIController ($scope,
8                             $filter,
9                             $injector) {
10     $scope.name = "Yaakov";
11
12     // pretty much anything in Angular with a dollar
13     $scope.upper = function () {
14         var upCase = $filter('uppercase');
15         $scope.name = upCase($scope.name);
16     };
17
18     console.log($injector.annotate(DIController));
19 }
20

```

You'll see that our `$injector.annotate` gave us an array of the argument names to the function `DIController`. And this is exactly what Angular is using internally in order to figure out where to inject which services.

Lecture 10: Protecting Dependency Injection from Minification

Why we minify?

Minification is the process of removing all unnecessary characters from source code. And what's here, italicized here is without changing the functionality of the source code. So, it should be clear that minification is going to be the process that's going to make our code completely unreadable, yet at the same time, 100% functional as before.

[Javascript-minifier.com](http://javascript-minifier.com)

while directly use the minified code will cause error

```

app.js
1  (function () {
2      'use strict';
3
4      angular.module('DIApp', [])
5          .controller('DIController', DIController);
6
7      function DIController ($scope, $filter) {
8          $scope.name = "Yaakov";
9

```

The second argument right now is a function value. But the truth is, it doesn't have to be. It can be an array, and it's an array of strings with the last element in the array, be the function that is responsible to be the controller function.

```

4  angular.module('DIApp', [])
5  .controller('DIController', ['$scope', '$filter', DIController]);
6

```

we've told Angular is that scope and filter in this order is going to be appearing as the arguments into this DIController function, and this solves the above issue, as now, these strings are protected from minification because these are string literals. Similarly, this name Yaakov right here is a string literal, and it'll obviously will never get minified because it's real data.

Another way to solve this, more elegant: attach \$inject property to the function object

```
4 angular.module('DIApp', [])
5 .controller('DIController', DIController);
6
7
8 DIController.$inject = ['$scope', '$filter'];
9 function DIController ($scope, $filter) {
10     $scope.name = "Yaakov";
11
12     $scope.upper = function () {
13         var upCase = $filter('uppercase');
14         $scope.name = upCase($scope.name);
15     };
16 }
```

Angular is going to look for this function, is going to also look and check to see whether or not we have a \$inject property on the function that is supposed to be the controller. And if it finds it, it will use that array as guidance to see which service to inject into which argument and the controller function itself.

Lecture 11, Part 1: Expressions and Interpolation

Expression: {{ exp }}

Something that evaluates to some value, processed by AngularJS & roughly similar to the result of `eval(some_js)`. Executed in the context of \$scope and has access to properties of \$scope. Doesn't throw errors if a type error or reference error. Control flow functions (like if else) are not allowed. Accept a filter or a filter chain to format the output.

