

# Deep dive into Angular.js directives



Jesús Rodríguez Rodríguez

By the author of [angular-tips.com](http://angular-tips.com)



# Deep dive into Angular.js directives

Jesús Rodríguez Rodríguez

This book is for sale at <http://leanpub.com/angularjsdirectives>

This version was published on 2015-05-07



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 Jesús Rodríguez Rodríguez

# **Tweet This Book!**

Please help Jesús Rodríguez Rodríguez by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#angularjs](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#angularjs>

# Contents

Preface . . . . .	i
<b>I Unit test primer . . . . .</b>	<b>1</b>
1. What is unit testing and why should I care? . . . . .	2
2. Introduction to Jasmine . . . . .	3
3. Spies on Jasmine . . . . .	14
4. Jasmine's custom matchers . . . . .	19
5. Testing Angular.js . . . . .	22
<b>II Directive Definition Object - Part 1 . . . . .</b>	<b>26</b>
6. Introduction . . . . .	27
7. Template . . . . .	28
8. Restrict . . . . .	33
9. Link function . . . . .	38
10.Scope . . . . .	41
10.1 Same Scope . . . . .	42
10.2 New Scope . . . . .	43
10.3 Isolated Scope . . . . .	48
11.Transclusion . . . . .	63
11.1 transclude: true . . . . .	64
11.2 Transcluding by hand . . . . .	67
11.3 Tranclusion and its scope . . . . .	70
11.4 Transclude 'element' . . . . .	75

## CONTENTS

<b>III Directives Lifecycle</b>	<b>85</b>
12. Templates revisited	86
13. Compile	87
14. Controller	92
14.1 Controller As	101
14.2 Swapping controllers	107
15. Pre and post link	111
<b>IV Directive Definition Object - Part 2</b>	<b>114</b>
16. Priority	115
17. Terminal	124
18. Require	129
19. Multi Element	140
<b>V Advanced usages</b>	<b>146</b>
20. Observing attributes	147
21. Optional attributes	153
22. Manual \$compile	162
23. Dynamic templates	168
24. Wrapping a jQuery plugin	174
<b>Appendix A - Using ngModelController</b>	<b>180</b>
Local validation	181
Remote validation	185
Formatters and Parsers	189

# Preface

Are you an Angular.js developer who wishes to learn more about directives? You came to the right place.

Directive is what makes Angular the best Javascript framework out there, but it is also the most complicated concept of it. Directives are complex, there are infinite use cases for them and you can easily find yourself deep into different options, and not being sure of which one you need to apply to succeed.

So if directives seemed magical to you and you need to learn how they work to be able to create your own, keep reading :).

In this book, you will learn:

- How directives are built.
- How to use all the different options on the `directive` definition object.
- How to apply some tips and tricks to make your directives more flexible.
- How to combine our directives with `ng-model` to create outstanding directives.
- How to create custom validations for our use case.
- How to wrap jQuery plugins into directives.

And much more.

## What this book covers

- *Unit testing primer*: The book code is fully tested and to do that, I'll explain how testing works on Javascript and Angular.
- *Directive definition object*: All the different properties on the **DDO** are covered in this book.
- *Directive lifecycle*: Different steps our directive does to be fully processed.
- *Advanced usages and patterns*: There are some common patterns and advanced usages that we want to know about.
- *Using ng-model*: `ng-model` is a vital part in some directives so we need to integrate it into our directives.

## What you need for this book

To try these recipes you only need a web browser and a text editor.

You can use any workflow of your choice to work with this book, but you could also use any online tool like Plunker<sup>1</sup>. Here I provide you with two templates I made:

- [Trying out a directive](#)<sup>2</sup>
- [Testing a directive](#)<sup>3</sup>

The first one is used to see the directive working, without the tests. The second one is used to test the directives.

You can use Plunker for the entire book, but if any example needs something different, I will provide more details.

## How to read the book

I recommend you to read this book from cover to cover, I try to link the knowledge from one chapter to another, but you can certainly jump to the most interesting part.

## Who this book is for

This book is meant for people who know the basics of Angular but willing to learn all about directives. Even if you already wrote your own custom directives, you can learn some advanced tips & tricks to make your directives better.

## Example code

Every example will contain a proper link to Plunker.

## Questions and feedback

Please, leave your feedback as an issue [here](#)<sup>4</sup>

Also, you can contact me via IRC in the channel #angular.js of the freenode server (I am Foxandxss). On the other hand, you can go to that channel for general help, there are a lot of helpful people around there.

## About the cover picture

The cover picture was taken from Bryce Edwards at [flicker](#)<sup>5</sup>

---

<sup>1</sup><http://plnkr.co>

<sup>2</sup><http://plnkr.co/edit/tpl:Gm3LwKi9QJluREnXGruj>

<sup>3</sup><http://plnkr.co/edit/tpl:2EfD65Eg9SivfQ6fceRl>

<sup>4</sup><https://github.com/angular-tips/deep-angular-issues>

<sup>5</sup>[https://www.flickr.com/photos/bryce\\_edwards/3127639851](https://www.flickr.com/photos/bryce_edwards/3127639851)

# I Unit test primer

In this part, we are going to learn the basics of unit testing on Javascript.



# 1. What is unit testing and why should I care?

Unit tests are a bunch of Javascript files that we create to make sure that every part of our application (and in our case, a directive) works as it is expected to work. That means that we need to write hundred of lines of code to assert that our code does what is supposed to do.

## Isn't that a waste of time?

The boss is always telling us that we need to be faster and hundred of lines don't sound like *fast*. Au contraire, that bunch of code will save us **HOURS**. Don't believe me? I have proof.

## Extra code

How many times did you end with code that is not used? Maybe we added some extra loops that are not needed or some function to do something and then realize that we are not using it. When we code our modules before any test, we don't actually know what we are going to need or if our algorithm is going to support any kind of input (that could lead to those extra loops). More code means more stuff to maintain, which also means, more money.

## Bad API design

Maybe we need to create a new service to do something, and then we start writing functions to do the work and we make some of them public to define the service's API. Good, that is the idea, isn't it? Some time later we get complaints about our really poor API, which, well, is not as intuitive as we expected. This category also includes those API functions that are not really needed (which is also *extra code*).

## Refactor

What happens when we want to refactor our code? We are in big trouble. Even when we decide not to break the API, maybe that internal change is not working properly in some edge cases where it worked in the past. That will break the application for some people and they won't be happy at all (and that kind of bugs are normally a pain in the ass).

## Will it work?

That is the end goal and probably the biggest time waster of anything you have to do in your application. Something as simple as a *calendar*, involves some math and some magic numbers to make it work. We really need to be sure it works. How? We open a certain date, we manually check with our OS calendar to see if it matches. We repeat that for some random dates (old ones, future ones). Then we change something in our service and well, we need to check the dates again to assert that nothing is broken. Repeat that 20 times for a normal service development.

## 2. Introduction to Jasmine

Ok, you convinced me that maybe I was wrong about not doing unit testing. But how can it help with those problems? What if we see a really simple example? (General example, not Angular related and it will be in a overly slow peace to make the point).

Let's say I want an object which will be able to do some basic maths (Addition, Subtraction, Multiplication, Division) and your first thought would be to start writing a constructor with some prototype functions to do some math. We will end doing something like that, but what we are going to do, is to test it first. Test it first? Why? Bear with me.

(If you want to follow this, I have a [plunker](#)<sup>1</sup> for you to work.)

### Calculator example

Our object should be able to add 5 and 3 and get 8. Let's test that:

```
1 describe('Calculator', function() {
2     var calc;
3
4     beforeEach(function() {
5         calc = new Calculator();
6     });
7
8     describe('Addition', function() {
9         it('should be able to add 5 and 3 to return 8', function() {
10             var result = calc.addition(5, 3);
11             expect(result).toBe(8);
12         });
13     });
14 });
```

If we put that on a spec file and run it we get:

---

<sup>1</sup><http://plnkr.co/edit/tpl:BwELtfQGfM9ODbyuj9RG?p=catalogue>

**1 spec, 1 failure**

raise ex

**Spec List | Failures****Calculator Addition should be able to sum 5 and 3 to return 8**

ReferenceError: Calculator is not defined

```
ReferenceError: Calculator is not defined
    at Object.<anonymous> (http://run.plnkr.co/4Ss8NXCEd41FxJtD/appSpec.js:5:12)
    at attemptSync (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1510:12)
    at QueueRunner.run (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1498:9)
    at QueueRunner.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1485:10)
    at Spec.Env.queueRunnerFactory (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:518:35)
    at Spec.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:306:10)
    at Object.<anonymous> (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1708:37)
    at attemptAsync (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1520:12)
    at QueueRunner.run (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1496:16)
    at QueueRunner.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1485:10)
```

TypeError: Cannot read property 'addition' of undefined

```
TypeError: Cannot read property 'addition' of undefined
    at Object.<anonymous> (http://run.plnkr.co/4Ss8NXCEd41FxJtD/appSpec.js:10:18)
    at attemptSync (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1510:12)
    at QueueRunner.run (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1498:9)
    at QueueRunner.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1485:10)
```

**This first test fails**

It says that it can't create a new `Calculator` and it is not able to do that addition (surprise!). Well, we have no code. Before continuing, I am going to explain how Jasmine tests work.

Jasmine is like writing English. It is something easy to read and understand (which is way cool). Jasmine spec files are normally wrapped on a `describe` block which receives a string to define what are we describing. They are used to group tests. We can see how we have another `describe` block which is nested in the previous one with the `addition` as parameter. See how we are grouping the tests?

What we need to do to write tests is to use the `it` function. It receives the name of the test and a callback function that will contain the test itself. In this case it is testing if what we get from the `addition` function is the correct value.

See how easy it is to make a test. We use the `expect` function where we pass our result and then the `toBe` jasmine function which receives the expected value. Read with me: `expect result to be 8`.

What about that `beforeEach`? There is an important concept in unit testing. Every test, AKA every `it` function, should be called with a fresh state. That means that if in one `it` we save something, the next `it` won't see it. In our case, if we create a new calculator on our `it`, the next one won't have it, so we need to create one for each test. That is not DRY, isn't it? That is what `beforeEach` is for, it

is handy way of preparing each test. In this case, we can read: before each test, create a new calculator, yay, just what we needed.

On the other hand, are you starting to see what we are getting here so far? **API design**. By using our object before we coded it, we are using the API as we would like to use it. That is a much much better way to define our API.

Let's make that test pass:

```
1 function Calculator() {  
2 }  
3  
4 Calculator.prototype.addition = function(num1, num2) {  
5   return 5 + 3;  
6 };
```

Does it pass?

1 spec, 0 failures

Calculator

Addition

should be able to sum 5 and 3 to return 8

It works!

Yes it does! This is an example of no **extra code**. We coded the minimum necessary to make it work, and well, that is what we need at this point.

Of course, we are not finished yet with our tests. We want to know if we can add 7 and 0. We test it on a new it function:

```
1 describe('addition', function() {  
2   // earlier test hidden  
3   it('should be able to add a number with 0', function() {  
4     var result = calc.addition(7, 0);  
5     expect(result).toBe(7);  
6   });  
7 });
```

**2 specs, 1 failure**

raise e

Spec List | Failures

**Calculator Addition should be able to sum a number with 0**

Expected 8 to be 7.

```
Error: Expected 8 to be 7.
    at stack (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1293:17)
    at buildExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1270:14)
    at Spec.Env.expectationResultFactory (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:484:18)
    at Spec.addExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:260:46)
    at Expectation.addExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:442:21)
    at Expectation.toBe (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1209:12)
    at Object.<anonymous> (http://run.plnkr.co/4Ss8NXCEd41FxJtD/appSpec.js:15:22)
    at attemptSync (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1510:12)
    at QueueRunner.run (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1498:9)
    at QueueRunner.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1485:10)
```

Ohh, it fails again

Well, that fails, and we know why. For the sake of learning we are going to do an extra step to fix it:

```
1 Calculator.prototype.addition = function(num1, num2) {
2   return 7 + 0;
3 };
```

**2 specs, 1 failure**

raise e

Spec List | Failures

**Calculator Addition should be able to sum a number with 0**

Expected 8 to be 7.

```
Error: Expected 8 to be 7.
    at stack (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1293:17)
    at buildExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1270:14)
    at Spec.Env.expectationResultFactory (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:484:18)
    at Spec.addExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:260:46)
    at Expectation.addExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:442:21)
    at Expectation.toBe (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1209:12)
    at Object.<anonymous> (http://run.plnkr.co/4Ss8NXCEd41FxJtD/appSpec.js:15:22)
    at attemptSync (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1510:12)
    at QueueRunner.run (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1498:9)
    at QueueRunner.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1485:10)
```

Uops...

Ups, we broke the last test. That is wonderful. That solves our **Will it work?** problem. We can immediately see that we broke our code when we modified our function to pass the new test.

Let's fix it for once:

```
1 Calculator.prototype.addition = function(num1, num2) {  
2   return num1 + num2;  
3 };
```

2 specs, 0 failures

Calculator

Addition

should be able to sum 5 and 3 to return 8  
should be able to sum a number with 0

Yes, finally!

Uh, finally. Now we have a proper addition method with just the needed code to make it work, no extra params either. We can add some more tests (to the addition describe):

```
1 it('should be able to add a negative number with a positive result', function() {  
2   var result = calc.addition(7, -3);  
3   expect(result).toBe(4);  
4 });  
5  
6 it('should be able to add a negative number with a negative result', function() {  
7   var result = calc.addition(-20, 7);  
8   expect(result).toBe(-13);  
9 });
```

**4 specs, 0 failures**

raise ex

**Calculator****Addition**

should be able to sum 5 and 3 to return 8  
should be able to sum a number with 0  
should be able to sum a negative number with a positive result  
should be able to sum a negative number with a negative result

Uh, it works

Uh, it works without any extra code! Better for us. Let's do the division:

```
1 describe('division', function() {  
2   it('should be able to do a exact division', function() {  
3     var result = calc.division(20, 2);  
4     expect(result).toBe(10);  
5   });  
6 });
```

**5 specs, 1 failure**

rai

**Spec List | Failures****Calculator division should be able to do a exact division**

TypeError: undefined is not a function

```
TypeError: undefined is not a function  
    at Object.<anonymous> (http://run.plnkr.co/4Ss8NXCEd41FxJtD/appSpec.js:31:25)  
    at attemptSync (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1510:12)  
    at QueueRunner.run (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1498:9)  
    at QueueRunner.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1485:10)  
    at Spec.Env.queueRunnerFactory (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:518:35)  
    at Spec.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:306:10)  
    at Object.<anonymous> (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1708:37)  
    at attemptAsync (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1520:12)  
    at QueueRunner.run (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1496:16)  
    at QueueRunner.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1485:10)
```

Ah, that was expected

We see it fails, it doesn't have that division method.

```
1 Calculator.prototype.division = function(num1, num2) {  
2   return num1 / num2;  
3 };
```

## 5 specs, 0 failures

### Calculator

#### Addition

- should be able to sum 5 and 3 to return 8
- should be able to sum a number with 0
- should be able to sum a negative number with a positive result
- should be able to sum a negative number with a negative result

#### division

- should be able to do a exact division

This time you won't trick me

We are smart enough to make a proper first version which actually passes.

Now, for non exact divisions, we want to round the result, we don't want any decimals.

```
1 it('returns a rounded result for a non exact division', function() {  
2   var result = calc.division(20, 3);  
3   expect(result).toBe(7);  
4 });
```



**6 specs, 1 failure**

raise ex

Spec List | Failures

**Calculator division returns a rounded result for a non exact division**

Expected 6.666666666666667 to be 7.

```
Error: Expected 6.666666666666667 to be 7.
    at stack (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1293:17)
    at buildExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1270:14)
    at Spec.Env.expectationResultFactory (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:484:18)
    at Spec.addExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:260:46)
    at Expectation.addExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:442:21)
    at Expectation.toBe (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1209:12)
    at Object.<anonymous> (http://run.plnkr.co/4Ss8NXCEd41FxJtD/appSpec.js:37:22)
    at attemptSync (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1510:12)
    at QueueRunner.run (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1498:9)
    at QueueRunner.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1485:10)
```

That was also expected

Our current implementation is not rounding the result at all. Let's fix that:

```
1 Calculator.prototype.division = function(num1, num2) {
2   return Math.round(num1 / num2);
3 };
```

**6 specs, 0 failures**

Calculator

Addition

- should be able to sum 5 and 3 to return 8
- should be able to sum a number with 0
- should be able to sum a negative number with a positive result
- should be able to sum a negative number with a negative result

division

- should be able to do a exact division
- returns a rounded result for a non exact division

Ok, nothing is broken

This time we didn't break our last implementation, that is something :P.

What about throwing an exception if we divide something by 0? Sure:

```

1 it('should throw an exception if we divide by 0', function() {
2   expect(function() {
3     calc.division(5, 0);
4   }).toThrow(new Error('Calculator does not allow division by 0'));
5 });

```

7 specs, 1 failure

raise ex

Spec List | Failures

Calculator division should throw an exception if we divide by 0

Expected function to throw an exception.

```

Error: Expected function to throw an exception.
    at stack (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1293:17)
    at buildExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1270:14)
    at Spec.Env.expectationResultFactory (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:484:18)
    at Spec.addExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:260:46)
    at Expectation.addExpectationResult (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:442:21)
    at Expectation.toThrow (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1209:12)
    at Object.<anonymous> (http://run.plnkr.co/4Ss8NXCEd41FxJtD/appSpec.js:43:10)
    at attemptSync (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1510:12)
    at QueueRunner.run (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1498:9)
    at QueueRunner.execute (http://cdn.jsdelivr.net/jasmine/2.0.0/jasmine.js:1485:10)

```

It is not throwing anything

This test is a little bit different. Instead of passing a variable to expect, we are passing a function. We expect that call to end on an exception, so saving the result as we previously did won't work (we expect to never return that result but throw an exception). We also use the `toThrow` function on Jasmine.

It fails, we are not throwing any exception yet. Let's fix that:

```

1 Calculator.prototype.division = function(num1, num2) {
2   if (num2 === 0) {
3     throw new Error('Calculator does not allow division by 0');
4   }
5   return Math.round(num1 / num2);
6 };

```

## 7 specs, 0 failures

### Calculator

#### Addition

- should be able to sum 5 and 3 to return 8
- should be able to sum a number with 0
- should be able to sum a negative number with a positive result
- should be able to sum a negative number with a negative result

#### division

- should be able to do a exact division
- returns a rounded result for a non exact division
- should throw an exception if we divide by 0

Now it throws!

Well, we just need to check if num2 is 0 to throw the exception.

With that, we finished our division method... Wait a second... those parameter names suck. I agree, let's change them:

```
1 Calculator.prototype.division = function(dividend, divisor) {  
2   if (divisor === 0) {  
3     throw new Error('Calculator does not allow division by 0');  
4   }  
5   return Math.round(dividend / divisor);  
6 };
```

## 7 specs, 0 failures

### Calculator

#### Addition

- should be able to sum 5 and 3 to return 8
- should be able to sum a number with 0
- should be able to sum a negative number with a positive result
- should be able to sum a negative number with a negative result

#### division

- should be able to do a exact division
- returns a rounded result for a non exact division
- should throw an exception if we divide by 0

Still working

Uh, we did a **refactor** and we didn't break anything.

I will leave the other two calculator functions as an exercise.

## Summary

Even when it is a really really simple example, we already saw how we can address those problems I described earlier:

Our calculator doesn't have any **extra code**, because we coded just what we needed to make our calculator work. Its **API design** is good enough, that is because we used it as we would like to use it in the real world. **Will it work?** Sure, I have a bunch of tests that proves that. What about **refactor**? Go ahead, if the tests still pass, then you're doing good.

Maybe you won't notice it with this example, but with proper tests, you will save a lot of hours of maintaining **extra code** and dealing with **API design** which hopefully won't end up breaking changes; on the contrary, you will **refactor** code without fear and of course you will be sure that your code **will work**.

Testing is your friend, and with little effort on it, it will save us real pain.

## 3. Spies on Jasmine

Before we deep into Angular lands, I want to talk about spies. No, not that kind of spies.

When you are doing unit testing, you don't want to leave your SUT (subject under test) domain. If you're testing a directive in Angular and it injects 3 services, you don't care about those services, you only want to test that directive but also make sure it uses the services as intended.

That means that if your directive calls a service when doing something, you only want to know that the service was called, but without actually using the service. Uhm, I am confused, how can I test that I called a service without using it? And what if I need what it was supposed to return to the directive? And and and...

Jasmine has a concept called spies. You can spy functions and then you will be able to assert a couple of things about them. You can check if it was called, what parameters it had, if it returned something or even how many times it was called!

Spies are highly useful when writing tests, so I am going to explain how to use the most common of them here.

### Spies demo

For the tutorial sake, we are going to use a little snippet I created earlier: (You can follow along using this [plunk<sup>1</sup>](http://plnkr.co/edit/tpl:BwELtfQGfM9ODbyuj9RG))

```
1  //This is the service we don't care about
2  function RestService() {
3  }
4
5  RestService.prototype.init = function() {
6    // Some init stuff
7  };
8
9  RestService.prototype.getAll = function() {
10   return []; // Return elements
11 };
12
13 RestService.prototype.update = function(item) {
14   console.log("Updating the item");
```

---

<sup>1</sup><http://plnkr.co/edit/tpl:BwELtfQGfM9ODbyuj9RG>

```
15 };
16
17 // This is our SUT (Subject under test)
18 function Post(rest) {
19     this.rest = rest;
20     rest.init();
21 }
22
23 Post.prototype.retrieve = function() {
24     this.posts = this.rest.getAll();
25 };
26
27 Post.prototype.accept = function(item, callback) {
28     this.rest.update(item);
29     if (callback) {
30         callback();
31     }
32 };
```

We have here our SUT, which is a Post constructor. It uses a RestService to fetch its stuff. Our Post will delegate all the rest work to the RestService, which will be initialized when we create a new Post object. Let's start testing it step by step:

```
1 describe('Posts', function() {
2     var rest, post;
3
4     beforeEach(function() {
5         rest = new RestService();
6         post = new Post(rest);
7     });
8 });
```

Nothing new here. Since we are going to need both instances in every test, we put the initialization on a beforeEach, so we will have a new instance every time.

Upon Post creation, we initialize the RestService. We want to test that, how can we do that?:

```
1 it('will initialize the rest service upon creation', function() {
2   spyOn(rest, 'init');
3   post = new Post(rest);
4   expect(rest.init).toHaveBeenCalled();
5 });
```

We want to make sure that `init` on `rest` is being called when we create a new `Post` object. For that purpose we use the jasmine `spyOn` function. The first parameter is the object we want to put the spy in, and the second parameter is a string which represents the function to spy. In this case we want to spy the function `'init'` on the `rest` object. Then we just need to create a new `Post` object that will call that `init` function. The final part is to assert that `rest.init` has been called. Easy right? Something important here is that when you spy a function, the real function is never called. So here `rest.init` doesn't actually run.

Wait a second, we already created the new `Post` object in the `beforeEach`! That is true, but here is an important concept: You can't call a function, then spy it and assert it. Since on the `beforeEach` we already called the `init` function, the spy won't work, that is why we recreate a new object for the test sake. You can also create the spy on the `beforeEach`, but since we just need it once, we do it on the `it`.

Our next test is about retrieving the data from the `RestService`. Since it doesn't return anything, what we could do is to make sure that the `getAll` function has been called and also that our `this.posts` contains what `getAll` returned. Wait a second. Does the spy return stuff? Sure!

```
1 it('will receive the list of posts from the rest service', function() {
2   var posts = [
3     {
4       title: 'Foo',
5       body: 'Foo post'
6     },
7     {
8       title: 'Bar',
9       body: 'Bar post'
10    }
11  ];
12
13  spyOn(rest, 'getAll').and.returnValue(posts);
14  post.retrieve();
15  expect(rest.getAll).toHaveBeenCalled();
16  expect(post.posts).toBe(posts);
17 });
```

We create an array of fake posts and then we call `spyOn` on the `getAll` function. We can make an spy return something when called using `and.returnValue`, which is called with our fake posts.

Having our spy in place, all what we need to do is to call `post.retrieve` and then assert that `getAll` was called, and that also `post.posts` contains our fake posts that were returned by `getAll`.

We now want to test our `accept` function. We know that it will send a post to be updated on the `RestService`, so we need to be sure that the post was sent as a parameter.

```
1 it('can accept a post to update it', function() {
2   var postToAccept = {
3     title: 'Title',
4     body: 'Post'
5   };
6   spyOn(rest, 'update').and.callThrough();
7   post.accept(postToAccept);
8   expect(rest.update).toHaveBeenCalledWith(postToAccept);
9 });
```

We start with the fake post we want to update and then we create the spy. What is that `and.callThrough`? Well, as you know, the real function is never called, but if you really need to spy it and also call it, you can do it with `and.callThrough`. To see this working, you can check how `rest.update` logs a message on the dev console.

**Note:** There is no real reason to use `and.callThrough` here, I did it to show how it works.

After that, we just call `post.accept` passing our fake post and then assert that `rest.update` was indeed called with fake post as a parameter.

If you remember, we had the option to pass a callback to the `accept` function. Let's test that:

```
1 it('can receive a callback upon accept', function() {
2   var fn = jasmine.createSpy();
3   var postToAccept = {
4     title: 'Title',
5     body: 'Post'
6   };
7   post.accept(postToAccept, fn);
8   expect(fn).toHaveBeenCalled();
9 });
```

See how we created a spy this time. We needed a function to pass to the `accept` function as a callback method, so we could perfectly create an object with a function and spy it as we used to, but we can create a spied function from scratch using `createSpy`. After that, we pass it to `accept` and as always, we assert that it was called.

In Angular spies are heavily used on any kind of unit test. When we are testing something, we don't care about the injected dependencies, we just need to make sure that they were used properly.



Here you can find the [plunker<sup>2</sup>](#) with the final solution.

## Summary

Unit testing without spies is impossible. The good thing about unit testing is that we are testing a component in isolation and thanks to spies, we can do that.

Here we learnt the different kinds of spies that Jasmine comes with.

We can make simple spies with `spyOn`, make them return values with `and.returnValue`, call the original function with `and.callThrough` or even create new spied functions on the fly with `createSpy`.

---

<sup>2</sup><http://plnkr.co/edit/473hGDlrLKYy9vKSPg4Z?p=preview>

## 4. Jasmine's custom matchers

Jasmine has a bunch of matchers: `toBe(...)`, `toEqual(...)`, etc. They are really nice but sometimes they don't describe what we really need to assert. Imagine we have the need to check if a number is even, what can we do?

```
1 expect(42 % 2 === 0).toBe(true);
```

That certainly works, but it is not readable. In Jasmine, the goal is to be able to read the expectations like if they were plain English. "Expect 42 modulus 2 to equal 0 to be true" well, a bit hard to read, forces you to do some maths and think about the meaning. What about: "Expect 42 to be even", isn't that better? We can code it like:

```
1 expect(42).toBeEven();
```

That is definitely easier to read. The issue is that there is no built-in matcher called `toBeEven`. Luckily for us, Jasmine allows us to create custom matchers. Let's create one:

```
1 describe('Even numbers', function() {  
2   beforeEach(function() {  
3     jasmine.addMatchers({  
4  
5     });  
6   });  
7 });
```

Our matcher should be declared inside a `describe` block and ideally inside a `beforeEach` (that will allow all the tests inside that describe to use the custom matcher). To add new matchers, we call `jasmine.addMatchers` passing an object with all our matches. Inside it, we put:

```
1  toBeEven: function(util, customEqualityTesters) {  
2    return {  
3      compare: function(actual, expected) {  
4  
5      }  
6    };  
7  }
```

A custom matcher is a function which receives an `util` parameter which is a set of utility functions for matchers to use and a `customEqualityTesters` parameter which needs to be passed in if `util.equals` is ever called. The matchers should return an object with a `compare` function. The `compare` function receives the `actual` value and the `expected` value. For example:

```
1  expect(foo).toBe(10);  
2  expect(20).toFoo();
```

In the first case, `foo` is the `actual` and `10` the `expected`. In the second case, `20` is the `actual` and there is no `expected` value.

Back to our example, the `actual` is the number we want to evaluate and since our comparison is static, we don't need to pass any `expected` value.

Next, `compare` needs an object called `result` which will contain a `pass` boolean to indicate if our expectation passed or not:

```
1  var result = {  
2    pass: (actual % 2) === 0  
3  };
```

Optionally, we can define a `message` property on our `result` object:

```
1  if (!result.pass) {  
2    result.message = 'Expected ' + actual + ' to be even';  
3  }
```

Now if our number is not even, we get a nice error message. Lastly, we need to return the `result`. The final result is:

```
1 describe('Even numbers', function() {
2   beforeEach(function() {
3     jasmine.addMatchers({
4       toBeEven: function(util, customEqualityTesters) {
5         return {
6           compare: function(actual, expected) {
7             var result = {
8               pass: (actual % 2) === 0
9             };
10
11             if (!result.pass) {
12               result.message = 'Expected ' + actual + ' to be even';
13             }
14
15             return result;
16           }
17         };
18       }
19     });
20   });
21 });
```

We can add a few tests to test our matcher:

```
1 it('matches even numbers', function() {
2   expect(42).toBeEven();
3 });
4
5 it('can also negates the matcher', function() {
6   expect(43).not.toBeEven();
7 });
```

Check it [here](http://plnkr.co/edit/iARSeuly1bD7fXSQAEj6?p=preview)<sup>1</sup>

## Summary

If you want to create a nice test suite for your application / library, custom matchers are really nice. If you're creating a calendar directive, you can create matchers like: `toHas31Days`, `toBeLeapYear`, etc. That allows you to have smaller and very expressive tests.

---

<sup>1</sup><http://plnkr.co/edit/iARSeuly1bD7fXSQAEj6?p=preview>

## 5. Testing Angular.js

You will be surprised to see that testing Angular components is not much different from what we did in our previous tests.

The new bits we need to learn is how to deal with Angular modules and its dependency injection. They don't work as we are used to, they need some manual work but luckily for us, angular-mocks has everything we need.

Let's see the skeleton of a service test:

```
1 describe('type: name', function() {
2     var $scope, myService, $location
3
4     beforeEach(module('app'));
5
6     beforeEach(inject(function($rootScope, _myService_, _$location_) {
7         $scope = $rootScope.$new();
8         myService = _myService_;
9         $location = _$location_;
10    }));
11
12    it('should work', function() {
13        // Do something
14        // Expect something
15    });
16 });
```

It should be very familiar now, but there are some Angular bits. First of all, *my* convention is to put what we are testing and the name, for example: controller: foo. Then we need to load all the modules involved; if it is just app, we just need to load that one. Since we need it before each test, we use the module function (from angular-mocks) to load that app module for us.

The other different part is how we inject stuff. To inject we need to use the inject function (from angular-mocks) which will receive a function with all the stuff we need to inject. What about those underscores? Angular ignores them, so we can use that to be able to have our local variables with the original service name.

The rest is not new. Once we have our modules loaded and our services injected and saved in local variables, we just need to do some testings. Let's move to the next section to see a real example :).

## Testing directives

Directives is the hardest component to test on Angular, but since this book is all about directives, we need to learn the basics of how to test them.

To test directives we are going to use jQuery<sup>1</sup>, but only for tests.

### To jQuery or not to jQuery

There is a little controversy with the directive tests. Some people prefer to test their directives using end to end tests, but others prefer unit tests. I prefer the latter and there jQuery becomes essential.

Let's test this simple directive:

```
1 angular.module('app').directive('myDir', function() {
2   return {
3     scope: {
4       value: "=argument"
5     },
6     template: '<div class="dir">{{value}}</div>'
7   }
8 });
```

And a skeleton of how to test it:

```
1 describe("directive: my-dir", function() {
2   var element, scope;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function($rootScope, $compile) {
7     scope = $rootScope.$new();
8
9     element = angular.element('<div my-dir argument="foo"></div>');
10
11     scope.foo = "something";
12
13     $compile(element)(scope);
14     scope.$digest();
```

---

<sup>1</sup>Even when it is not required at all, it is really handy to be able to select specific element nodes.

```

15     }));
16
17     it ("should ...", function() {
18         // Test here
19     });
20
21     it ("and should ...", function() {
22         // Test here
23     });
24 });

```

We are testing `my-dir`, which is a directive that will output what we pass to it via the `argument` attribute.

To test the directive we need to create an element with it. In this case we are testing `my-dir`; thus we create an element containing the directive, that also includes the `argument` attribute that the directive needs. Since the `argument` attribute receives a Javascript primitive, we need to create that primitive in our scope. That is as easy as creating it in the scope we created.

Ok, we have our element and a scope (we can imagine that this scope is a controller scope) and what we need to do is to compile our element. We use `$compile` for that job. It receives an element and returns a `link` function that we should call with our scope. The last step is to execute a `$digest`. Why should we need to do that? Because when we assign some stuff to our scope (like what we did with `foo`) we need to ask Angular.js to digest it. If we don't do that, the `argument` attribute will be undefined.



Don't worry if this is hard to grasp at first glance. `$compile` will be properly explained in its own chapter in this book.

What is exactly that compiled element? It is our element with all the directives compiled. For example, our:

```

1 <div my-dir argument="foo"></div>

```

Gets compiled to:

```

1 <div my-dir="" argument="foo" class="ng-scope ng-isolate-scope">
2   <div class="dir ng-binding">{{value}}</div>
3 </div>

```

That is our element with our directive compiled.



When testing, we are going to do by hand some things that Angular.js normally does automatically.

So, before every test, we have our compiled element with the directive, our scope working... What remains is our tests. Directives have a lot of different use cases, so two directives could have different type of tests. Testing the output, or the relation with other directives, the scope...

Our `my-dir` directive will output what we passed to the `argument` attribute inside a `div` with the `dir` class. We need to be sure that our compiled element contains in our case “something”:

```
1 it ("should contain 'something' on it", function() {  
2   var text = element.find('.dir').text();  
3   expect(text).toBe('something');  
4 });
```

This test is really simple and that is thanks to jQuery. We just need to find the element with the `dir` class and grab its text. Then we expect it to be “something”.



jQuery on tests could lead to false positives. It’s not common at all, but there is the possibility. On `ui-bootstrap` with hundred of tests, that never happened.

Check here the complete code: [plunker](#)<sup>2</sup>

## Summary

Testing Angular is not hard at all, it is much like plain Javascript code with just some extra bits. Testing directives is indeed more complicated because we need to create our element, compile it and bind it to a scope but at the end of the day, the same principles apply to all directives.

Here in this book we are going to find dozens of tests, so when you finish the book, directive testing will hold no secrets for you.



If you are interested in unit testing and want more information about how to test the other Angular components, check my [blog](#)<sup>3</sup>

---

<sup>2</sup><http://plnkr.co/edit/k3NVn7XsTTi1cNxSWGIM?p=preview>

<sup>3</sup><http://angular-tips.com/blog/2014/02/introduction-to-unit-test-toc/>



# II Directive Definition Object - Part 1

Directives have a lot of different options: `template`, `scope`, `link`, `controller`, `transclusion`... all of those are part of the `directive definition object`. In this part, we are going to learn how they work and when to use them.

## 6. Introduction

The star feature of Angular are the directives. They teach our HTML new tricks from a simple element that shows a custom message to a full blown grid system. It is also the most complex part of Angular because you can practically do anything with them.

Let's see how do we create a directive:

```
1 angular.module('app').directive('myDirective', function() {
2   return {
3     template: '<div>{{message}}</div>',
4     link: function(scope, element, attrs) {
5       scope.message = 'Hello, World';
6     }
7   };
8 });
```

We are going to explain how `template` and `link` works later, but for now, you can see that a directive is basically a function (where you can inject your stuff) which returns an object. That object is the directive definition object.

This directive just creates a new `message` property on the scope and print it out inside a `<div>`. How do we use it? By default we can say for now that a directive is going to be used as an attribute (more on that later). Having that in mind, we can do:

```
1 <div my-directive></div>
```

There are two things to notice here: First, we are using it inside a `div` tag. That is not a requirement, you can use it on any other tag. Second we are using `snake-case` on the HTML to use our directive, but when we created it, we used `camelCase`. What is that all about? In Angular we create our directives using `camelCase` but in our HTML, you have to use `snake-case`. Later in this book you will see more cases where this rules also needs to be applied.

There is not much more behind how a directive should looks like. In the course of the book you will learn all the different properties of the DDO as well of a bunch of advanced use cases.

Here is the [plunker](http://plunker.co/edit/waFUgZHvk4DOnBhdoUoJ?p=preview)<sup>1</sup> with that example.

---

<sup>1</sup><http://plunker.co/edit/waFUgZHvk4DOnBhdoUoJ?p=preview>

## 7. Template

Let's begin with the first property of the DDO in this book: `template` / `templateUrl`.

A directive can do a variety of different things, and one of those is the output of some HTML. Right, how can we achieve that?

Let's say I want a directive to print some [bootstrap's jumbotron](http://getbootstrap.com/components/#jumbotron)<sup>1</sup> on the screen with a custom header and message. How can we achieve that? Easy:

```
1 angular.module('app').directive('jumbotron', function() {
2   return {
3     template: '<div class="jumbotron">' +
4               '<h1>{{header}}</h1>' +
5               '<p>{{message}}</p>' +
6               '</div>'
7   };
8 });
```



This is not the best example of this directive, but with the knowledge we gathered so far, there isn't a better way to do it. Stay tuned for the next chapters :)

So, there is a `template` property on the directive where we can put some HTML that is going to be shown when the directive is used. I see that there are a `{{header}}` and a `{{message}}`. Where do they come from? I don't see anything on the directive defining those properties on any `$scope`.

By default, a directive uses the same `$scope` available on the current element, for example, if we use this directive like:

```
1 <body ng-controller="MainCtrl">
2   <div jumbotron></div>
3 </body>
```

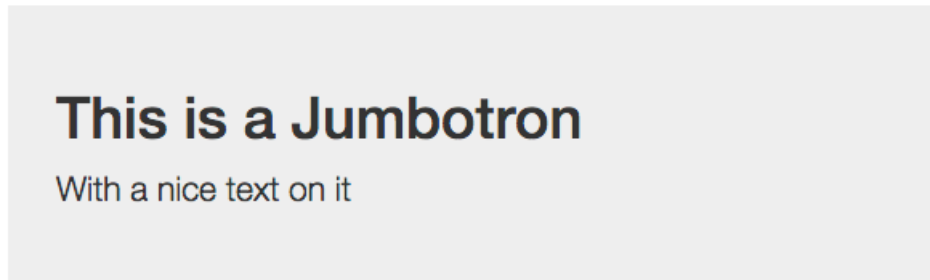
Since the `div` that contains our `jumbotron` has the `MainCtrl` `$scope`, the `jumbotron` directive will use that exact `$scope` for itself. That means that if that `MainCtrl` has a `header` and `message` properties, the `jumbotron` will use them to output that template.

---

<sup>1</sup><http://getbootstrap.com/components/#jumbotron>

```
1 angular.module('app').controller('MainCtrl', function($scope) {  
2   $scope.header = "This is a Jumbotron";  
3   $scope.message = "With a nice text on it";  
4 });
```

Now if we look in our HTML, we can see our jumbotron working as we expect:



Our own jumbotron directive!

Yay, we can feel the power! See it working [here](#)<sup>2</sup>



## Can we put other directives inside the template property?

Yes! We can put any directive inside the template property (built-in or custom). We will see a couple of examples of that in the rest of the book.

## The tests

Now that we made it, what about testing it? In this case we just need to make sure that the template contains the right info. Let's start preparing the test:

```
1 describe('Directive: jumbotron', function() {  
2   var element;  
3  
4   beforeEach(module('app'));  
5  
6   beforeEach(inject(function($compile, $rootScope) {  
7     var scope = $rootScope.$new();  
8  
9     scope.header = 'Hello, Jumbotron';
```

---

<sup>2</sup><http://plnkr.co/edit/wjLHKGKXg3bFmYQVdNgc?p=preview>

```
10     scope.message = 'Testing a jumbotron';
11
12     element = angular.element('<div jumbotron></div>');
13
14     $compile(element)(scope);
15
16     scope.$digest();
17     }));
```

Nothing new here. We created a new child scope, assigned the properties we expect on our directive, created the element with the directive, compiled it and finally we do a `$digest` so all the bindings are processed and the DOM gets updated with our directive.



There is no real need to create a new scope, we can use `$rootScope` directly on this test. Personally, I prefer to use a new scope, it is just an extra line of code but is easier to read.

For the test themselves, I ended with:

```
1  it('should contain a header on the template', function() {
2      var header = element.find('.jumbotron > h1');
3      expect(header.text()).toBe('Hello, Jumbotron');
4  });
5
6  it('should contain a message on the template', function() {
7      var message = element.find('.jumbotron > p');
8      expect(message.text()).toBe('Testing a jumbotron');
9  });
```

Nothing fancy. We leverage all the tests to the jQuery power. All we need is to select the proper DOM element and check if its text is what we expect. Simple directive, simple tests.

Check the test [plunker<sup>3</sup>](http://plunker.co/edit/5MqTnTqQGtOsbWCoYmC?p=preview).

Jokes aside, our directive is good (for the knowledge we have so far) but I feel like that `template` property can get out of hand pretty quickly, can't we move it to a different file? Of course we can!

We can create an HTML file on our project and then move our template there. Let's call it `jumbotron.html`:

---

<sup>3</sup><http://plunker.co/edit/5MqTnTqQGtOsbWCoYmC?p=preview>

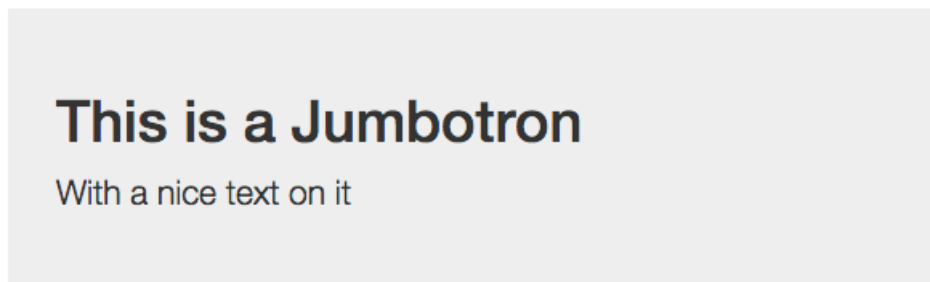
```
1 <div class="jumbotron">
2   <h1>{{header}}</h1>
3   <p>{{message}}</p>
4 </div>
```

Same HTML but now it is easier to read. Then we need to update our directive:

```
1 angular.module('app').directive('jumbotron', function() {
2   return {
3     templateUrl: 'jumbotron.html'
4   };
5 });
```

There we have `templateUrl`. Works much the same as `template` but instead of using the string directly as a template, uses the string as a path to fetch the template. For our demo, we put the template on the root folder, but we can put them into a `/templates` directory and then use it like: `templateUrl: '/templates/jumbotron.html'`

We can see that nothing changed in our final result:



Updated code, same result

Check the [plunker](#)<sup>4</sup>

Now you're wondering about how we test the new change. We can't actually test our directive in the plunker as we used to. Why? Angular loads the template using a GET request and that is not something we want to do on a unit test. On the real world, if you use Karma as a test runner, you can import the templates in the same way we load our modules, something like:

```
1 beforeEach(module('jumbotron.html'));
```

Karma will put the template into the `$templateCache` on the fly so there won't be any need to do a GET. Problem solved.

---

<sup>4</sup><http://plnkr.co/edit/U8aPeBtbWBCg2ws62eRq?p=preview>

On my real world thought, I put all the templates into `$templateCache` in advance, so when I load my Javascript files into the tests, the templates are already there, so I don't need to do that trick.



We can use functions instead of strings on our `template / templateUrl`, but we want to learn more first about directives.

## Summary

With both `template` and `templateUrl` we can abstract a certain piece of DOM and insert it where needed. When we learn more about directives, we can add some extra behaviour to make our templates a bit more dynamic.

## 8. Restrict

We have seen how to create attribute directives with a template, but what if I want to use the directive as an element? Well, there are 4 ways of using a directive:

- Attribute, like: `<div foo></div>`
- Element, like: `<foo></foo>`
- Class, like: `<div class="foo"></div>`
- Comment, like: `<!-- directive: foo -->`

You can use any of those in your applications but in the real world, I didn't saw any comment directive and even class one is pretty rare to find. The good part about Angular directives is that they make the intentions for our HTML very clear. So, something like:

```
1 <jumbotron></jumbotron>
```

It is really really clear of what's going on. That element is a jumbotron and there is no space for doubts.

```
1 <div jumbotron></div>
```

This example is clear as well, not as clear as the element one, but you can certainly know what's going.

```
1 <div class="jumbotron"></div>
```

Did bootstrap updated the jumbotron class to add more functionality to it? Ah no, that is a directive. It is not that clear. It has its use cases, but for normal directive usage, it is not a common choice.

```
1 <!-- directive: jumbotron -->
```



On the element kind of directives, **always** close the tag, don't use the short tag form.

This one seems like someone added a TODO to add a jumbotron there but never did. Comment directives were commonly used to overcome a DOM limitation with directives that spanned multiple elements. Today Angular supports those kind of directives, so comment directives are not used anymore.

Leaving all this fuss behind, how can we use it? Check our earlier jumbotron directive:



```
1 angular.module('app').directive('myJumbotron', function() {
2   return {
3     restrict: 'ACE',
4     template: '<div class="jumbotron">' +
5               '<h1>{{header}}</h1>' +
6               '<p>{{message}}</p>' +
7               '</div>'
8   };
9 });
```

There is a new `restrict` property of the DDO which accepts a string containing some capital letters, there are 4 available letters:

- **A**: For attribute directives.
- **E**: For element directives.
- **C**: For class directives.
- **M**: For comment directives.

You can combine depending on how you want to use your directive. In my last example I put ACE so it can be used as an attribute, class, element.



Don't restrict your directives with all the options at the same time, it is better to restrict it only to what makes sense on its use case.

Now you can use your directive as an element as well!

**This is a Jumbotron**

With a nice text on it

uhhh, element directives

Out of curiosity, I changed the directive name and it is because now you can use it as a class and there is already an element inside it with a jumbotron class (the bootstrap one), that would cause an infinite loop. Also I am using the template version so we can test it easy on a Plunker.

See it working [here](http://plnkr.co/edit/cfPu0mkPQ0rTyXwIYpK0?p=preview)<sup>1</sup>

---

<sup>1</sup><http://plnkr.co/edit/cfPu0mkPQ0rTyXwIYpK0?p=preview>

## The tests

On the testing side, the tests themselves doesn't change (the output is the same) but how you create the directive for testing, changes.

Let's see how can we do that. First prepare the tests:

```
1 describe('Directive: jumbotron', function() {
2   var $compile, scope, element;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function(_$compile_, $rootScope) {
7     scope = $rootScope.$new();
8     $compile = _$compile_;
9
10    scope.header = 'Hello, Jumbotron';
11    scope.message = 'Testing a jumbotron';
12  }));
13 });
```

We are saving a reference to our scope and \$compile function. Since our directive can be used in different ways, we need to create it multiple times. Let's see how:

```
1 describe('as an attribute', function() {
2   beforeEach(function() {
3     element = $compile('<div my-jumbotron></div>')(scope);
4     scope.$digest();
5   });
6
7   it('should contain a header on the template', function() {
8     var header = element.find('.jumbotron > h1');
9     expect(header.text()).toBe('Hello, Jumbotron');
10  });
11
12  it('should contain a message on the template', function() {
13    var message = element.find('.jumbotron > p');
14    expect(message.text()).toBe('Testing a jumbotron');
15  });
16 });
17
18 describe('as an element', function() {
19   beforeEach(function() {
```

```
20     element = $compile('<my-jumbotron></my-jumbotron>')(scope);
21     scope.$digest();
22 });
23
24 it('should contain a header on the template', function() {
25     var header = element.find('.jumbotron > h1');
26     expect(header.text()).toBe('Hello, Jumbotron');
27 });
28
29 it('should contain a message on the template', function() {
30     var message = element.find('.jumbotron > p');
31     expect(message.text()).toBe('Testing a jumbotron');
32 });
33 });
34
35 describe('as a class', function() {
36     beforeEach(function() {
37         element = $compile('<div class="my-jumbotron"></div>')(scope);
38         scope.$digest();
39     });
40
41     it('should contain a header on the template', function() {
42         var header = element.find('.jumbotron > h1');
43         expect(header.text()).toBe('Hello, Jumbotron');
44     });
45
46     it('should contain a message on the template', function() {
47         var message = element.find('.jumbotron > p');
48         expect(message.text()).toBe('Testing a jumbotron');
49     });
50 });
```

A little bit verbose (this is a book after all) but here we are. We created 3 inner describe, each one for a concrete restrict type. We just need to \$compile an element with it and then run our tests. Notice that the tests themselves doesn't change. As I said, what changes here in the tests is how the directive is created and not the way we test them.

Ok, but should we do that on the Real Life ? No. We may not test something that is not part of our application. I mean, even when the directive is our, we have to assume that the restrict options is already tested on the Angular core and that it works as we expect. In this case what our directive does is to output a template with some text. That is what we need to test, what makes our directive special. Saying that, the correct test is the one we did on the last chapter, even when this one has more restrict options, we only need to use one restrict option on our tests. Still for the sake of learning, I

showed you how to `$compile` different kind of directives.

See the test working [here](#)<sup>2</sup>

Last, but not least, our `jumbotron` directive was working correctly on the last chapter as an attribute. Is there any default restriction when the property is not there? Yes! By default in `1.2.x` all directives are restricted by `attribute`, in `1.3.x` they are restricted both by `attribute` and `element`.

## Summary

Not all directives make sense to work as an element nor all directives are meant to be used as an attribute. Before coding your directive, ask yourself... Is this a directive that will **transform an element** into something different? If so, restrict it to `element` or maybe it is a directive that add **behaviour** to an element? Then restrict it to an attribute. If you said yes to both questions, you can restrict the directive to both (which is the default). Class directives are not that used because the end user would expect an element or attribute to be a directive and a class exist just for CSS purposes.

---

<sup>2</sup><http://plnkr.co/edit/2ol4afTBO7qFK4wOhzRI?p=preview>

## 9. Link function

If there is any property of the DDO which is virtually in any directive and also the most useful, it is the `link` function one.

So we have directives with their own template, we have directives that can be used as an element or attribute. Good, but what about putting our own logic? So far our directives were a little bit static.

There is a property on the DDO called `link` which is like:

```
1 link: function(scope, element, attrs) {  
2   // Our logic here  
3 }
```

For now we can say that this `link` function is what gets called when a directive is compiled from the DOM (which is actually true, but it is more involved as we are going to see later on this book). There we can do practically anything that makes sense within a directive.

Let's talk about the parameters, one by one, but first something important: The parameters there are not injections, they are normal and fixed parameters, in fact you can do something like:

```
1 link: function(a, b, c) {  
2   // Our logic here  
3 }
```

And for what Angular cares, it will work the same. Even though, the convention is what I put earlier.

At this stage of the book, we can't make much use of this `link` function, so for now, I am going to explain what can we do with each parameter and then on the following chapters, with some extra knowledge on our backs, we can make a proper use of it.

The `scope` parameter by default is the same scope of the element where the directive lives, so for example in:

```
1 <div ng-controller="foo">  
2   <my-directive></my-directive>  
3 </div>
```

The scope will be the `$scope` from `foo`. That means that I can set something in the controller like:

```
1 angular.module('app').controller('foo', function($scope) {  
2     $scope.something = "Hello";  
3 });
```

And be able to access it on the link function:

```
1 link: function(scope, element, attrs) {  
2     console.log(scope.something);  
3 }
```

Of course you can do it the other way down.

The `element` is the element where the directive lives. So if we have a directive called `foo` on an element like this:

```
1 <div foo>  
2     <p>Something else</p>  
3 </div>
```

In this case, the `element` parameter will be that entire element, including the `<p>` inside it. If the directive has a template, for example:

```
1 template: '<div>Something</div>'
```

And you use it like:

```
1 <div foo></div>
```

The element will be:

```
1 <div foo=""><div>Something</div></div>
```

There are two important things to say about `element`. First, it is already a `jqLite` object, so you can directly execute some functions on it like `addClass` or `children`. That also means that if you are loading `jQuery` before `Angular`, the `element` will be already a `jQuery` object, so you don't need to do:

```
1 link: function(scope, element, attrs) {  
2   // WRONG  
3   $(element).foo();  
4 };
```

Second, the `element` we got here is the **FINAL** DOM. That means that our element will be decorated with classes and stuff that Angular uses to do the bindings and thing like that. This can mean nothing to you, but in a later chapter it will make sense. Also we are going to see what can we do with this element, but first we need to learn a couple of things more.

The third parameter is `attrs`. It is basically an object which contains the element attributes on it. Having a `foo` directive in an element like:

```
1 <div foo bar="hello" baz="{{name}}"></div>
```

if you log out the `attrs` parameter you will get, among other things:

```
1 {  
2   foo: "",  
3   bar: "hello",  
4   baz: "Fox"  
5 }
```

Let's talk about what we see here:

- **foo**: An empty attribute like `foo` gets an empty string as a value.
- **bar**: We passed a string to it, which gets assigned as the value.
- **baz**: `baz` contains an interpolated value and since the `link` function is the last step on the directive lifecycle (more on that in a later chapter), it will contain the final value and not the interpolation string.

That is basically the main idea behind the `link` function. I know I said later chapter a lot here, but without explaining this a little bit now, we can't move on with the book.

## Summary

The `link` function is where we put our directive logic, where we can manipulate the DOM, work with our directive's scope, and even do some cool advanced stuff that we will see soon.

# 10. Scope

Having a directive use the same scope as the element where it lives is not the only option we have, in fact we have three different options we are going to cover on this lengthy chapter.



## 10.1 Same Scope

There are some types of directives that create a new scope (as we are going to see on the next section) and when we use one of those directives on an element, all the others directives on that element (with exceptions) and child elements will have access to that scope.

So imagine we have a `foo` directive. By default all our directives uses the scope as the element where the directive lives on, so if we have something like:

```
1 <div ng-controller="MyCtrl">
2   <div foo></div>
3 </div>
```

Since `ng-controller` creates a new scope, his `div` and all it children elements will have access to that scope, including our `foo` directive.

We can do even something like:

```
1 <foo ng-controller="MyCtrl">
2 </foo>
```

Again, since `ng-controller` creates a new scope, the element where it lives, receives that new scope, in other words, our `foo` directive will use that scope.

## Summary

Our directives by default will take the scope available on the element, created there or created on a parent element.

## 10.2 New Scope

So, new scopes, how can we implement them on our directives?

Let's imagine an example where I have some values on the controller and I want a directive with a new scope to see how both scopes behave.

Imagine this controller:

```
1 angular.module('app').controller('MainCtrl', function($scope) {  
2     $scope.name = 'John';  
3 });
```

And this directive:

```
1 angular.module('app').directive('myDirective', function() {  
2     return {  
3         scope: true,  
4         templateUrl: 'template.html',  
5         link: function(scope, elements, attrs) {  
6             scope.city = "Chicaco";  
7  
8             scope.changeName = function() {  
9                 scope.name = 'Brad';  
10            };  
11        }  
12    };  
13 });
```

With this template:

```
1 <p>Directive name: {{name}}</p>  
2 <p>Directive city: {{city}}</p>  
3 <button ng-click="changeName()">Change directive name</button>
```

Finally we use our directive here:

```
1 <body ng-controller="MainCtrl">
2   <p>Controller name: {{ name }}</p>
3   <p>Controller city: {{ city }}</p>
4   <div my-directive></div>
5 </body>
```

So we have a controller where we define a name (**John**) and on the directive we create a city (**Chicago**) then we output the name and city of the controller and same properties but from the directive.

What should we see?

Controller name: John

Controller city:

Directive name: John

Directive city: Chicago

Change directive name

Yes, you were right

On the controller part, we see the name we set and since we didn't set any city, it is just blank. So far so good. What about the directive part? Since we didn't set any new name there, it inherits the parent's name (**John**) and shows it, we also see they city (**Chicago**) that we set on the new directive's scope.

This explain a little bit how this new scope work. What we put on the controller's scope (parent scope) is inherited by the directive's scope (child scope), but what we put on that child scope is not visible on the parent scope. To be more exact, new properties or assigning new stuff into the new scope won't change the parent scope, but modifying an object or array of the parent scope from the child, will change both. For more info, check [this](https://github.com/angular/angular.js/wiki/Understanding-Scopes)<sup>1</sup>.

So what happen if we click that button?

---

<sup>1</sup><https://github.com/angular/angular.js/wiki/Understanding-Scopes>

Controller name: John  
Controller city:  
Directive name: Brad  
Directive city: Chicago  
[Change directive name](#)

### An expected change

No surprises here. I set a new name on the directive and as I said previously, changing a property that came from the parent only changes the child scope and not the parent one. That is why we see Brad on the directive but John on the controller.

Check the demo [here](#)<sup>2</sup>

## The tests

What about testing this directive? There is nothing special about new scopes in testing, but since I plan to test every example of the book, let's take a look:

```
1 describe('Directive: myDirective', function() {
2   var scope, element;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function($rootScope, $compile) {
7     scope = $rootScope.$new();
8
9     scope.name = "Fox";
10
11     element = $compile('<div my-directive></div>')(scope);
12
13     scope.$digest();
14   }));
15
16   it('contains a default city set', function() {
17     var dirScope = element.scope();
18
19     expect(dirScope.city).toBe('Chicago');
20   });
```

---

<sup>2</sup><http://plnkr.co/edit/i0f8WS1Ctc4AOFQjOwCL?p=preview>

```
21
22 it('changes the directive name on button click', function() {
23     var dirScope = element.scope();
24
25     var button = element.find('button');
26
27     button.click();
28
29     expect(dirScope.name).toBe('Brad');
30
31     // This test is not needed, just for the learning purposes
32     expect(scope.name).toBe('Fox');
33 });
34
35 it('changes the directive name calling the scope function', function() {
36     var dirScope = element.scope();
37
38     dirScope.changeName();
39
40     expect(dirScope.name).toBe('Brad');
41 });
42 });
```

To be honest with you, my dear reader, this is a little stupid directive, but there are a couple of new things here. On the first test, we can see how we can grab the directive's scope which is basically a scope method on our directive element. With that we can simply assert that there is by default a city with Chicago on it.

On the second test, we need to trigger the `changeName` function, so we can grab the button and “click” it using the `click` function that jQuery gives us. The click will trigger the `ng-click` and that will call our `changeName` function. We assert the change and we are good to go.

There is a third test which does the same as the previous one. You don't need to do both, I just show you the different possibilities you have to test a certain feature.



Notice that the scope we are passing to the `$compile` function is the one that is going to be the parent scope of our directive.

For the sake of completion, we could test that our template does what it is supposed to do and it also reacts to the name change:

```
1  it('contains the name on the directive template', function() {
2    var nameEl = element.find('p:eq(0)');
3
4    expect(nameEl.text()).toContain('Fox');
5  });
6
7  it('updates the name after a click', function() {
8    var dirScope = element.scope();
9
10   var button = element.find('button');
11
12   button.click();
13
14   var nameEl = element.find('p:eq(0)');
15
16   expect(nameEl.text()).toContain('Brad');
17 });
18
19 it('contains the city on the directive template', function() {
20   var nameEl = element.find('p:eq(1)');
21
22   expect(nameEl.text()).toContain('Chicago');
23 });
```

We just grab the correct `p` and we assert that it contains the name or the city. We also test that when we click the button, the name on the screen also changes. On a real world directive, even when the test number doesn't hurt, the one where we click and check the DOM does the same thing as the one where we just check the scope.

See the test [here](http://plnkr.co/edit/n0Qvp0EgMPBXRWC2fZ2?p=preview)<sup>3</sup>

## Summary

There are times where we don't want to use the same exact scope as the element's one, so we are able to create our own scope which will inherit from the current scope. The downside of this approach is that the new scope created for our directive will be the one that all the children elements will have.

---

<sup>3</sup><http://plnkr.co/edit/n0Qvp0EgMPBXRWC2fZ2?p=preview>

## 10.3 Isolated Scope

In Angular we can create new scopes like this (as seen in our tests):

```
1 var scope = $rootScope.$new();
```

That will create a new scope that will be a direct child of the `$rootScope` which will also inherit all its properties.

Nothing stops me of doing:

```
1 var newScope = scope.$new();
```

Then I will have something like: `$rootScope -> scope -> newScope`.

Angular provides us with a way of creating a new child scope, which will inherit all the functionality of the `$rootScope` (like events, watchers, digest, etc) but none of its properties, said in other words, an isolated scope:

```
1 var isolatedScope = $rootScope.$new(true);
```

Passing `true` to the `$new` function gives us a child scope that won't have any property of its parent but still maintain the inheritance relationship with its parent (and children).

We can also do:

```
1 var scope = $rootScope.$new();  
2 var isolatedScope = scope.$new(true);
```

To have: `$rootScope -> scope -> isolatedScope`.

The `isolatedScope` is still a child scope of `scope` but won't have any of its properties.

An isolated scope can also have children scopes, both isolated and normal.

We can use an isolated scope on a directive, but since it doesn't come with any property on it, we need a way to pass what we need from the controller to our directive.

Angular provides us with a couple of different options to be able to do two-way databinding with our controller, one-way databinding, pass simple strings, functions... That way we can have an isolated scope but also initialize it with a couple of properties of our choice.

```
1 <my-directive data="myData" show-all="true" name="{{user.name}}" done="done()">
2 </my-directive>

1 <div my-directive data="myData" show-all="true" name="John" done="done()"></div>
```

As you see here, to pass properties to our directive we just need to put them on attributes, then in our code we can treat those attributes in the way we need.



We are going to learn more about those attributes syntax in the upcoming chapters.

To do that, instead of passing `false` or `true` to our `scope` property on the directive's DDO, we pass an object with a special syntax, for example:

```
1 scope: {
2   data: '=',
3   all: '=showAll',
4   name: '@',
5   done: '&'
6 }
```

As any object in Javascript, it is composed by key - value pairs. The key is the internal representation of the property and the value is the type. We can optionally provide the name of the attribute next to the type. For now we can resume that like:

- For **data** we want a two-way databinding (that is what `=` means) with the attribute with the same name (**data**).
- For **all** we want a two-way databinding but in this case with the attribute called **showAll**.
- With **name** we want a one-way databinding with the **name** attribute (that is what `@` is for).
- Last, for **done** we want to pass a callback function in the attribute with the same name (using `&`).



An important note here is that an attribute uses snake case (`show-all`) but in the code we use it using camel case (`showAll`).

We will end having `data`, `all`, `name` and `done` in our directive's scope. What we can do with those stuff is our next topic.





Remember when I said that you can combine directives with same and new scope on an element but with exceptions? This is the exception. When you make a directive to use isolated scope, you can't use a directive with a new scope or isolated scope in the same element. Also, that isolated scope won't be shared to the other directives on the element, which will continue using the scope from the parent element.

## Scope =

Now that we have isolated scopes in our directives, we could have the need of passing some properties of our parent \$scope to this new isolated scope.

As we saw in the previous section, we could use the = to have a two-way databinding between our two scopes, so when we change that property in any of them, they get sync.

Let's imagine I have a music page and I have a section where I want to show a list of the best songs of some music bands.

It is a good idea to abstract that into a directive so we just need to put it where we need it.

Let's think... What do we need? We need to pass the **band name** and also the **list of songs**. With that we can simply ng-repeat the list and show it.

Let's go:

```
1 angular.module('app').directive('favoriteSongs', function() {
2   return {
3     restrict: 'E',
4     scope: {
5       songs: "=",
6       band: "="
7     },
8     template: '<h3>Favorite {{ band }} songs:</h3>' +
9               '<ul class="list-group">' +
10              '<li ng-repeat="song in songs" class="list-group-item">' +
11                '<span class="fa fa-music"></span>' +
12                ' {{ song }}' +
13                '</li>' +
14              '</ul>'
15   };
16 });
```

As you can see, we expect two attributes with the name of songs and band. We are going to store their values into our isolated scope with the names of songs and band respectively. Since we used =, if we change songs or band in our controller, the isolated scope will see the changes and viceversa.

The template itself is not special, we can see there how we reference band and songs that we now have in our isolated scope.

How can we use it? First imagine this controller:

```
1 angular.controller('MainCtrl', function($scope) {  
2     $scope.songs = [  
3         'Erotomania',  
4         'Fatal tragedy',  
5         'Metropolis',  
6         'Vacant',  
7         'Anna Lee'  
8     ];  
9  
10    $scope.bandName = 'Dream Theater';
```

A list of songs and the band name.

Now the element itself:

```
1 <favorite-songs songs="songs" band="bandName"></favorite-songs>
```

So, to pass our songs list via attribute, we just need to remember... What is expecting our directive? A songs attribute. Right, we just need to create it and use the name of the property on the current scope as the attribute value.

The same thing happen with the band, in this case the property on the current scope is called bandName so we pass that as the value of the attribute.

Notice how we are passing our \$scope properties to the attributes, just like songs="songs". When using = the directive expects the attributes to receive the name of the property on the scope.

That will give the following result:

### Favorite Dream Theater songs:

🎵 Erotomania
🎵 Fatal tragedy
🎵 Metropolis
🎵 Vacant
🎵 Anna Lee

Long life to DT!

Check the demo [here](#)<sup>4</sup> where we can see that updating the controller also updates the directive's isolated scope.

## The tests

Let's move to testing. How can we test this directive? Real easy.

We start coding the basic skeleton:

```
1 describe('directive: favorite-songs', function() {
2     var element, scope;
3
4     beforeEach(module('app'));
5
6     beforeEach(inject(function($rootScope, $compile) {
7         scope = $rootScope.$new();
8
9         element = angular.element(
10             '<favorite-songs songs="songs" band="band"></favorite-songs>');
11
12         scope.songs = [
13             'Erotomania',
14             'Fatal tragedy',
15             'Metropolis',
16             'Vacant',
17             'Anna Lee'
18         ];
19
20         scope.band = 'Dream Theater';
21
22         $compile(element)(scope);
23
24         scope.$digest();
25     }));
26 });
```

At this stage of the game, you won't be surprised that much. We created a scope, we gave it a list of songs and a band name and then we compiled our directive element.

Now that we have our directive created and compiled, now we test that it works properly:

---

<sup>4</sup><http://plnkr.co/edit/SD5qdsVxfloYn1DKOYx0?p=preview>

```
1 it('should contain a header with the band name', function() {
2   var headerText = element.find('h3').text();
3   expect(headerText).toContain(scope.band);
4 });
5
6 it('should contain a <li> per song', function() {
7   var lis = element.find('li');
8
9   expect(lis.length).toBe(scope.songs.length);
10
11   for (i = 0; i < scope.songs.length - 1; i++) {
12     expect(lis.eq(i).text()).toContain(scope.songs[i]);
13   }
14 });
```

First, we check that we have a header with the band name which is easily done finding the `h3` and checking its text property.

We also need to check that there is a `li` per song. For that we checked two different things: first, we expect that the number of `li` elements are the same as the number of songs and second, we loop through all `li` to check that they contain the proper song name.

Check the [test](#)<sup>5</sup>

## Scope @

In the last section, we learnt that we can do two-way databinding between our controller and the directive's isolated scope using the `=` operator. That is nice, but what if we just need to pass a simple string? Maybe a static string I write directly on the attribute or maybe the string value of any of my controller properties.

We are not forced to do two-way databinding in our directives, we can pass simple strings to it.

Imagine you are doing some landing page where you need some `svg` circles laying around the screen and you want to create a directive to simplify a little bit those circles creation.

Ok, good idea! Let think about what attributes we need... Ok, we need an attribute for the circle `size`, one for the `stroke` color and one for the `filling` color. How are we going to use those attributes? I would say like strings. `stroke` can be a simple string like `red` or `blue` and the same for the other attributes.

Let's code it:

---

<sup>5</sup><http://plnkr.co/edit/YbsiTMLbuTSAiUANMfKL?p=preview>

```

1 angular.directive('svgCircle', function() {
2   return {
3     restrict: 'E',
4     scope: {
5       size: "@",
6       stroke: "@",
7       fill: "@"
8     },
9     templateUrl: 'circle.html',
10    link: function(scope, element, attr) {
11      var size = parseInt(scope.size, 10);
12
13      var canvasSize = size * 2.5;
14
15      scope.values = {
16        canvas: canvasSize,
17        radius: size,
18        center: canvasSize / 2
19      };
20    }
21  };
22 });

```

And the template:

```

1 <svg ng-attr-height="{{values.canvas}}"
2     ng-attr-width="{{values.canvas}}"
3     class="gray">
4   <circle ng-attr-cx="{{values.center}}"
5           ng-attr-cy="{{values.center}}"
6           ng-attr-r="{{values.radius}}"
7           stroke="{{stroke}}"
8           stroke-width="3"
9           fill="{{fill}}" />
10 </svg>

```

The first thing we notice here is that we are using @ to populate our isolated scope and that means that we are passing simple strings instead of doing two-way databinding.



The @ is a little bit more complex than “passing simple strings” but we will cover that in a later part of the book.

In the link part we do some calculations to get all the different values we need to do a circle in svg. Then we just need to assign those values to the different parts of the svg to render it.

Let me explain a little bit what are those values: The circle radius is the size we inserted, the canvas size is 2.5 times the radius and the circle center is the half of the canvas size. That way we have a 100% visible circle centered on the element.

Here is two examples of circles:

```
1 <svg-circle size="123" stroke="blue" fill="red"></svg-circle>
2 <svg-circle size="{{circle1size}}" stroke="red" fill="blue"></svg-circle>
```

For the first one, we hardcode all the different attributes: size of 123, blue stroke and red fill.

For the second one, assuming we have a `circle1size` property on our controller, it will interpolate that value and insert it into the attribute.

So if we have:

```
1 $scope.circle1size = 200;
```

in our directive, when we do the:

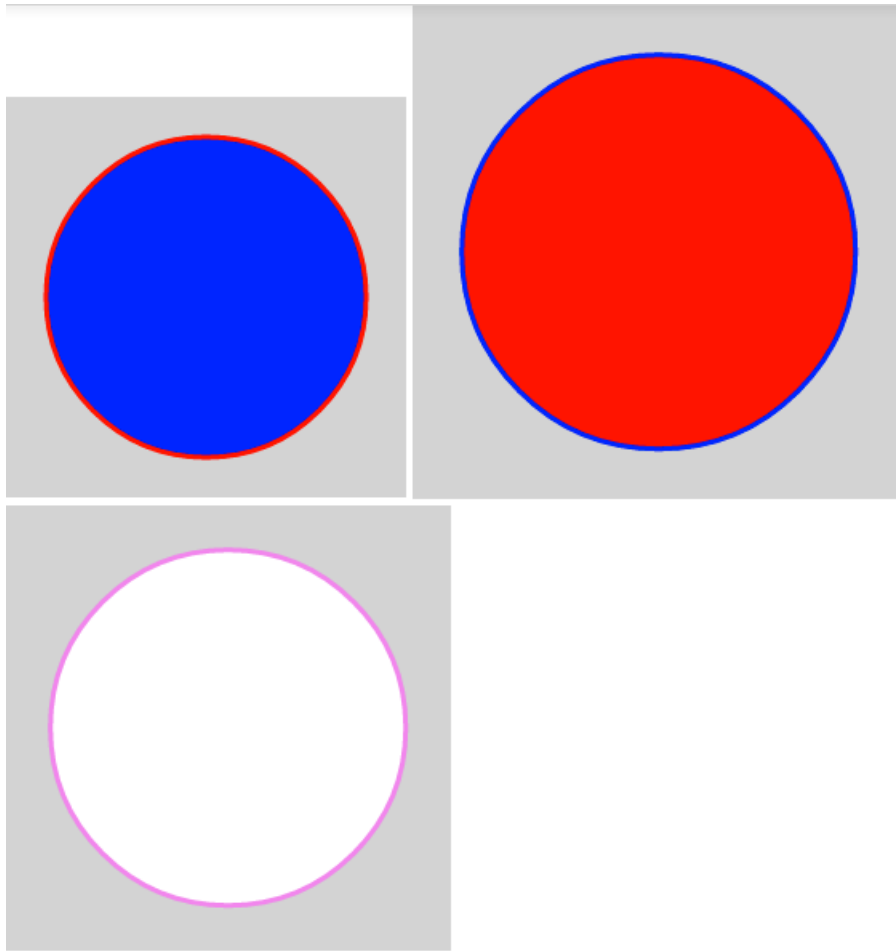
```
1 size="{{circle1size}}"
```

it is going to interpolate that property of our scope, see that it is 200 and do something like:

```
1 size="200"
```

As you noticed, that is the same thing we are doing on the directive's template, we are interpolating our calculation's values in the different svg attributes.

Said that, we can say that when a directive has attributes with @, it expect us to pass a simple string to them (like `foo`) or the name of a property on the scope (like `{{property}}`).



Three nice circles

See it [here](http://plnkr.co/edit/dHnIQbeOiz3alHclj5xM?p=preview)<sup>6</sup>

## The tests

Testing, testing, testing... As always, let's start with the basic skeleton:

---

<sup>6</sup><http://plnkr.co/edit/dHnIQbeOiz3alHclj5xM?p=preview>

```
1 describe('directive: svg-circle', function() {
2   var element, scope;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function($rootScope, $compile) {
7     scope = $rootScope.$new();
8
9     element = angular.element(
10      '<svg-circle size="100" stroke="{{stroke}}" fill="blue"></svg-circle>');
11
12     scope.stroke = "black";
13
14     $compile(element)(scope);
15     scope.$digest();
16   }));
17 });
```

To show that you can put a string or an interpolation as the attribute values, I mixed it a little bit. Nothing special apart from that.

For the test themselves we are going to test 3 different things:

That our calculation logic is correct:

```
1 it("should compute the size to create other values", function() {
2   var isolated = element.isolateScope();
3   expect(isolated.values.canvas).toBe(250);
4   expect(isolated.values.center).toBe(125);
5   expect(isolated.values.radius).toBe(100);
6 });
```

With `element.isolateScope()` we can access the isolated scope of the element (an element can only have one isolated scope). Having that, we just check our values.

Check that the svg is created correctly:

```
1 it("should contain a svg tag with proper size", function() {
2   expect(element.find('svg').attr('height')).toBe('250');
3   expect(element.find('svg').attr('width')).toBe('250');
4 });
```

And the circle created inside it, is correct as well:



```
1 it("should contain a circle with proper attributes", function() {
2   expect(element.find('circle').attr('cx')).toBe('125');
3   expect(element.find('circle').attr('cy')).toBe('125');
4   expect(element.find('circle').attr('r')).toBe('100');
5   expect(element.find('circle').attr('stroke')).toBe('black');
6   expect(element.find('circle').attr('fill')).toBe('blue');
7 });
```

With that, our directive is tested correctly.

Check the [tests](#)<sup>7</sup>

## Scope &

So far, we were able to pass properties from our scope to do two-way databinding and also pass simple strings to our isolated scope. The last piece of this puzzle are functions. What if we want to pass functions to the directive? So I can callback them when I need to.

This is a simple concept, same concept as the callbacks we use every day with Javascript. We pass a function, we call it later when we need it. Yeah, but in directives world, it is a bit complex. Let's put a simple example here to see this "twist".

Let's create a directive with a couple of inputs and a button to ask our name, then we pass back the result of that to our controller. Not a real directive to be honest, but enough to learn this :)

```
1 angular.module('app').directive('nameForm', function() {
2   return {
3     scope: {
4       callback: "&"
5     },
6     template: '<input ng-model="name" placeholder="Name" /><br>' +
7               '<input ng-model="lastname" placeholder="Last name"/><br>' +
8               '<button ng-click="onClick()">Click when finish</button>',
9     link: function(scope, element, attr) {
10      scope.onClick = function() {
11        var name = scope.name + " " + scope.lastname;
12        // Call the function here
13      };
14    }
15  };
16 });
```

---

<sup>7</sup><http://plnkr.co/edit/QdvFnMkNoRdvAzll4XwO?p=preview>

We notice that for the functions, we use the `&` symbol, and same rules applies as always, we expect an attribute called `callback` (as you know, we can use any name) and then we create a property on our isolated scope with that name.

The code shown here is not difficult. Two inputs, one for name, one for last name and also a button to trigger a `onClick` method on our scope.

So, call the function, how do we call the function? Like...

```
1 scope.callback();
```

That way? No, remember, we want to pass the parameters back, oh, okay:

```
1 scope.callback(name);
```

That way! No, not really. To really really explain what's going on, we would need an entire [book](http://teropa.info/build-your-own-angular)<sup>8</sup> so I am going to explain it simple:

Under the hood, angular will `$parse` that attribute, that will return a function that when we call it, it will end calling the original function (like a wrapper). The point here is that when we call this `scope.callback`, we are actually calling this wrapper.

Okay, so what parameters does this wrapper expect? Well, it wants an object where we map the original function parameters with our locals.

Imagine we pass a function which expect 2 parameters, `name` and `age` and after some computations we have our `scope.name` and `scope.age` in our directive. We need to create an object like this:

```
1 var locals = {  
2   name: scope.name,  
3   age: scope.age  
4 }
```

and then:

```
1 scope.ourFunction(locals);
```

Translating this new knowledge back to our directive, we now know what we need to do:

```
1 scope.callback({name: name});
```

This is like saying... For the `name` parameter, pass our local `name`.

Final directive is:

---

<sup>8</sup><http://teropa.info/build-your-own-angular>

```
1 angular.module('app').directive('nameForm', function() {
2   return {
3     scope: {
4       callback: "&"
5     },
6     template: '<input ng-model="name" placeholder="Name" /><br>' +
7               '<input ng-model="lastname" placeholder="Last name"/><br>' +
8               '<button ng-click="onClick()">Click when finish</button>',
9     link: function(scope, element, attr) {
10      scope.onClick = function() {
11        var name = scope.name + " " + scope.lastname;
12        scope.callback({name: name});
13      };
14    }
15  };
16 });
```

We use it like this:

```
1 <div name-form callback="setGreeting(name)"></div>
```

Here we are giving the function name and also the parameters we expect back, AKA we want the callback to reference the setGreeting function on our controller which will receive a name parameter.

Can you see now the whole picture? Since setGreeting expect a name parameter, we call our callback function with an object which at the end, will call setGreeting with that name parameter.

In our controller we have:

```
1 angular.module('app').controller('MainCtrl', function($scope) {
2   $scope.setGreeting = function(name) {
3     $scope.name = name;
4   };
5 });
```

Nothing special in this side.

Doctor
Who
Click when finish

Hello, Doctor Who!

Doctor? Doctor who?



We can use any name on the controller's `setGreeting` function parameter, we are not forced to use `name` here.

See it [working](#)<sup>9</sup>

## The tests

Well, I guess you're pretty curious of how can we test this after all this explanation... Well, this is going to be the easier test so far in the book.

```

1 describe("directive: name-form", function() {
2     var element, scope, $compile;
3
4     beforeEach(module('app'));
5
6     beforeEach(inject(function(_rootScope_, _$compile_) {
7         var $rootScope = _rootScope_;
8         $compile = _$compile_;
9
10        scope = $rootScope.$new();
11
12        element = angular.element('<div name-form callback="onName(name)"></div>');
13
14        scope.onName = jasmine.createSpy('onName');
15
16        $compile(element)(scope);
17    }));
18 });

```

Since we need a function in our parent scope to pass it as a callback, we can just simply use `createSpy` to create one on the fly.

For the test itself, we just need to:

---

<sup>9</sup><http://plnkr.co/edit/AQ0wAr7FKyAwyaKAX1RB?p=preview>

```
1 it ("should call the callback function on the submit button click", function() {  
2   // We simulate that we entered our name  
3   element.isolateScope().name = "Jesus";  
4   element.isolateScope().lastname = "Rodriguez";  
5  
6   element.find('button').click();  
7   expect(scope.onName).toHaveBeenCalledWith('Jesus Rodriguez');  
8 });
```

With unit test we can't directly write on the inputs (and we shouldn't!) so we simulate we wrote stuff on it, we click on the button and then we expect that the `onName` function of our scope have been called with the right parameter.

Just that!

Check it [here](#)<sup>10</sup>

## Summary

If your directive is meant to add new behaviour, it's always better to give it an isolate scope so it won't pollute the parent scope. It is also a good way to create reusable directives.

When using isolated scopes, we can pass to it different things to achieve the results we expect:

- `=`: It is using if we want to achieve two-way databinding between a property on our parent scope and a property on the directive's scope.
- `@`: If we want to pass mere strings to our directive, we can use `@` symbol.
- `&`: If what we need is to pass some callback functions, that we can trigger after something happens on our directive.

---

<sup>10</sup><http://plnkr.co/edit/EOClxK8vSUdzBM5KZKcQ?p=preview>

# 11. Transclusion

So here it comes the transclusion. We heard a lot about transclusion in the different media (forums, twitter, IRC) thinking that transclusion is something really really complex so they had to use a weird word to describe it. In reality, transclusion is a pretty easy concept and we are going to tackle it here.

So what is all this transclusion about?

Imagine you have this html:

```
1 <div my-directive>
2   <span>Hello there my friend</span>
3 </div>
```

And the directive is like:

```
1 angular.module('app').directive('myDirective', function() {
2   return {
3     template: 'This is my-directive'
4   };
5 });
```

So we know that when we put a template inside our directive, it will be placed inside the element which contains our directive (the div in our case). Good, but what happen when have some html content already inside that element? Are we going to see the template one? The one that is already there? Both? None?

This is my-directive

If you inspect the element, you can see:

```
1 <body ng-controller="MainCtrl" class="ng-scope">
2   <div my-directive="">This is my-directive</div>
3 </body>
```

So there the `<span>` was replaced with our directive's template. That can be good for some cases, but for other cases, we would love to grab that existent html and use it on our directive's template. Can we do that? Yeah, we can **transclude** it!

So basically transclusion is a way of grabbing that html and be able to work with it inside our directive. Keep reading!

## 11.1 transclude: true

Imagine we want to create a directive to show some card elements. Those cards are composed of an `<h3>` as a header as well as some content like a couple of `<p>`. We could do something like:

```
1 $scope.content = '<h3>Card header</h3><p>This is my </p><p>lovely card</p>';

1 <card body="content"></card>
```

Yeah we could, but do you like that? We shouldn't put any DOM stuff on our controller and to be honest, it is a real pain. What if instead of that, we do:

```
1 <card>
2   <h3>Card header</h3>
3   <p>This is my </p>
4   <p>lovely card</p>
5 </card>
```

Isn't that way cleaner? I would say so.

So, in this kind of cases, where you want to use that html inside the directive's element for your own purposes, you need transclusion. Right, how do we use it?

We need two things: use the `transclude` property on the directive's DDO and also specify **where** we want to transclude (insert) that html in our directive's template.

Alright, how could we specify that? There is a directive called `ng-transclude` that we can put on an element of our template and then, the transcluded html (the one we are grabbing from "outside") is going to be inserted as a child of that element. Let's see our card directive:

```
1 angular.module('app').directive('card', function() {
2   return {
3     scope: {
4       color: '@'
5     },
6     transclude: true,
7     template: '<div class="card {{color}}" ng-transclude></div>'
8   };
9 });
```

We set the `transclude` property to `true` and then on our template we created a simple `div` where we put the `ng-transclude` directive. That means that our transcluded html is going to sit **inside** that `div`.

Let's see it in action:

```
1 <card color="red">
2   <h3>Card number 1</h3>
3   <p>We like our first card</p>
4   <p>Simple but useful</p>
5 </card>
```

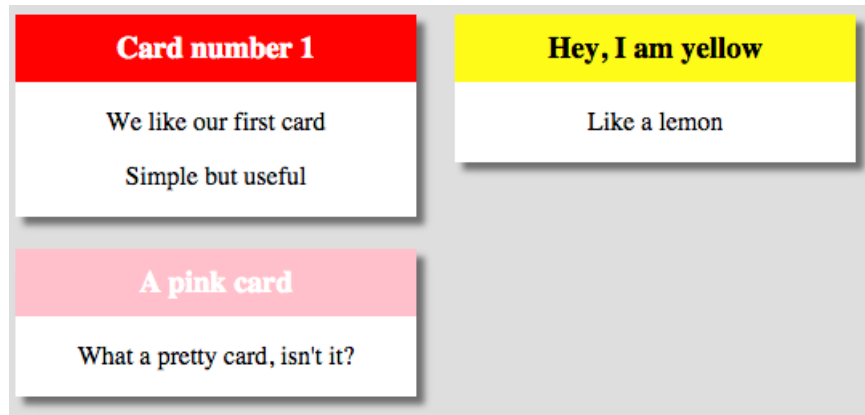
That will output an html like:

```
1 <card color="red" class="ng-isolate-scope">
2   <div class="card red" ng-transclude="">
3     <h3 class="ng-scope">Card number 1</h3>
4     <p class="ng-scope">We like our first card</p>
5     <p class="ng-scope">Simple but useful</p>
6   </div>
7 </card>
```

You see? It wasn't any complicated :)



There is another way of inserting the transcluded html into our template which we are going to see later on this book.



A couple of cards

See them [working](#)<sup>1</sup>

## The tests

Time to test it!

Let's setup our test:

---

<sup>1</sup><http://plnkr.co/edit/UDWxE01FfDHUcoDdRJgS?p=preview>



```
1 describe('directive: card', function() {
2   var element;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function($rootScope, $compile) {
7     element = angular.element(
8       '<card color="red"><h3>Hello world</h3><p>content</p></card>'
9     );
10
11     $compile(element)($rootScope);
12     $rootScope.$digest();
13   }));
14 });
```

Nothing that will surprise us now.

The tests are easy as well:

```
1 it('should contain a red card', function() {
2   var card = element.find('div.card');
3
4   expect(card.attr('class')).toContain('red');
5 });
6
7 it('should contain the header and the content', function() {
8   var header = element.find('h3').text();
9   var p = element.find('p').text();
10
11   expect(header).toBe('Hello world');
12   expect(p).toBe('content');
13 });
```

We just need to test that it sets the right color on it and also that the template contains our transcluded html.

Check the test [here](http://plnkr.co/edit/PmDuC87ncjkFgpgauWDD?p=preview)<sup>2</sup>

## Summary

With `transclude: true` we can get that DOM that lies inside our directive and insert it on the directive's template where we want.

---

<sup>2</sup><http://plnkr.co/edit/PmDuC87ncjkFgpgauWDD?p=preview>

## 11.2 Transcluding by hand

So, that `ng-transclude` directive is pretty cool, but a little bit inflexible right? What if we want a dynamic point of transclusion, put that transcluded html more than once or modify it before we transclude it? There are a **lot** of different use cases where the `ng-transclude` directive is not enough for us. What can we do in that regard? Well, we can transclude by hand.

On the next part, we will learn more about the directives lifecycle, but for now we can say that when a directive hits the `link` function, all the directives in the template are compiled (with their children) and our scope is set in place and ready. Thanks to that, we can safely do any DOM manipulation in our `link` function.

Let's review the function signature:

```
1 link: function(scope, element, attrs) {}
```

That is what we have been using so far because those are the most used parameters, but the real signature is:

```
1 link: function(scope, element, attrs, ctrl, transclude) {}
```

So apart from the parameters we already know, there are two extra parameters. The first one `ctrl` is used for some advanced stuff (which we will cover in a later chapter) and the last one is a `transclude` function.



Remember, the parameters names are not fixed, you can change them.

That `transclude` function is what we need if we want to do some advanced transclusion. Let's see its signature:

```
1 transclude([scope], function(clone, scope) {});
```

The first and optional parameter is used to attach a concrete scope on that transcluded html if we need to override it (more in the next chapter). The second parameter is a callback function, that function receives a clone of the transcluded html and its new scope. Inside that callback function we can do absolutely what we want.

As a first example, let's redo what we did on the last directive but transcluding by hand:

```
1 angular.module('app').directive('card', function() {
2   return {
3     scope: {
4       color: '@'
5     },
6     transclude: true,
7     template: '<div class="card {{color}}"></div>',
8     link: function(scope, element, attrs, noop, transclude) {
9       transclude(function(clone) {
10         element.find('div').append(clone);
11       });
12     }
13   };
14 });
```

The template is now missing the ng-transclude directive and then on the link function we are transcluding by hand. In the transclude function, we are using the callback to find the div in our template and then append there the transcluded html.

As you see, we can do what we want with our transcluded html, append it, copy it x times, etc.

See it working [here](#)<sup>3</sup>

## The tests

For the tests we can use the same we used in the last example because the result itself doesn't change:

```
1 describe('directive: card', function() {
2   var element;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function($rootScope, $compile) {
7     element = angular.element(
8       '<card color="red"><h3>Hello world</h3><p>content</p></card>'
9     );
10
11     $compile(element)($rootScope);
12     $rootScope.$digest();
13   }));
14
15   it('should contain a red card', function() {
```

---

<sup>3</sup><http://plnkr.co/edit/XLPwPZBDgilJHgJqPErg?p=preview>

```
16     var card = element.find('div.card');
17
18     expect(card.attr('class')).toContain('red');
19   });
20
21   it('should contain the header and the content', function() {
22     var header = element.find('h3').text();
23     var p = element.find('p').text();
24
25     expect(header).toBe('Hello world');
26     expect(p).toBe('content');
27   });
28
29 });
```

Check it [here](#)<sup>4</sup>

## Summary

If ng-transclude doesn't provide enough flexibility for your directive, you can always transclude by hand. That provides us with the flexibility of getting the transcluded html and insert it in a more dynamic way.

---

<sup>4</sup><http://plnkr.co/edit/aUmwM8Kf8DdIMw0cv06v?p=preview>

## 11.3 Tranclusion and its scope

We talked about automatic and manual tranclusion and we also mentioned that we can pass an scope as the first parameter of the transclusion function in case we need to override it. Override it? Why?

Let's imagine this simple controller and directive:

```
1 angular.module('app').controller('MainCtrl', function($scope) {
2     $scope.person = {
3         name: 'John Doe',
4         profession: 'Fake name'
5     };
6
7     $scope.header = 'Person';
8 });
9
10 angular.module('app').directive('person', function() {
11     return {
12         restrict: 'EA',
13         scope: {
14             header: '='
15         },
16         transclude: true,
17         template: '<div ng-transclude></div>',
18         link: function(scope, element, attrs) {
19             scope.person = {
20                 name: 'Directive Joe',
21                 profession: 'Scope guy'
22             };
23
24             scope.header = 'Directive\'s header';
25         }
26     };
27 });
```

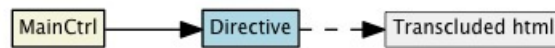
I have a controller with a person and a header on the scope, also a person directive which creates a person and modifies the header. The directive has an isolate scope, so it is not aware of the controller's person object. Let's use it:

```

1 <person header="header">
2   <h2>{{header}}</h2>
3   <p>Hello, I am {{person.name}} and,</p>
4   <p>I am a {{person.profession}}</p>
5 </person>

```

What is supposed to happen here? What should we see here? Let me think about it... We have a person directive which have a person on it called Directive Joe and also a header which says Directive's header. Then in our HTML we used the directive passing the controller's header and then we put some HTML **inside** the directive. Alright, we should see the Directive's header and also the information about Directive Joe. That is obvious since the HTML inside the directive (which is the transcluded html) is going to be transcluded into our directive. So our scopes are more or less like:



(The normal arrow is for new isolated scopes and the dashed is for new non-isolated scopes)

Let's see the result:

## Directive's header

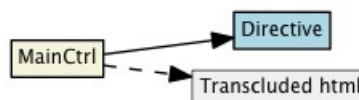
Hello, I am John Doe and,

I am a Fake name

Check it [here](#)<sup>5</sup>

So what happened here? We got mixed results... on the one hand, we have our Directive's header as expected, but on the other hand we got the controller's John Doe person object. That makes no sense at all. What's going on?

The misconception here is to think that the transcluded html has access to the isolate scope or that the transcluded html is a new child scope of it (as I showed on the diagram before). The reality is that the transcluded html is a new child scope of the **controller's** one. Yes, that is right:



(The normal arrow is for new isolated scopes and the dashed is for new non-isolated scopes)

<sup>5</sup><http://plnkr.co/edit/uiHDCulP21Mf2tvGyzFq?p=preview>



The isolate scope is never shared. Not with other directives on the same element nor children.

Having this in mind, the result makes more sense. The transcluded `html` only sees what is on the controller's scope. For the `person` object it is clear, it is showing it as is. But what about the header? It is showing the directive's one. Well, that isn't true. Since we created a two-way databinding on the header, when we changed the header on the directive, the controller's one also changed. That is why we saw Directive's header, because the controller's header was also updated.

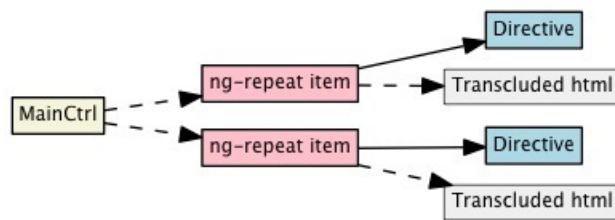
So, the transcluded `html` is a new child scope of the current scope on that DOM. In this case, the controller's scope. In the case that you put a `ng-repeat` like:

```

1 <div ng-repeat="foo in foos">
2   <person header="header">
3     <h2>{{header}}</h2>
4     <p>Hello, I am {{person.name}} and,</p>
5     <p>I am a {{person.profession}}</p>
6   </person>
7 </div>

```

The scopes would be like:



(The normal arrow is for new isolated scopes and the dashed is for new non-isolated scopes)

That is how the transclusion and its scope works by default. That doesn't mean that we can't do something to modify this behavior.

As we saw in the previous chapter, the `link` function contains a `transclude` function like:

```

1 transclude([scope], function(clone, scope) {});

```

We know that the `transclude` function will receive a clone of the transcluded `html` with its scope. With the first an optional parameter, we can override that scope. Yes, we can give any scope we want to our transcluded `html`.

So by default it is pretty much like doing:

```
1 angular.module('app').directive('person', function() {
2   return {
3     restrict: 'EA',
4     scope: {
5       header: '='
6     },
7     transclude: true,
8     link: function(scope, element, attrs, ctrl, transclude) {
9       scope.person = {
10         name: 'Directive Joe',
11         profession: 'Scope guy'
12       };
13
14       scope.header = 'Directive\'s header';
15
16       var childScope = scope.$parent.$new();
17
18       transclude(childScope, function(clone, scope) {
19         element.append(clone);
20       });
21     }
22   };
23 });
```

There we created a new child scope from the directive's parent scope (which is the controller's one) and then assigned it to be the scope of the transcluded html.

If you run it, you won't see any difference.

Check it [here](#)<sup>6</sup>

We could as well use the parent scope directly as the scope of the transcluded html. That would give the same result without creating yet another scope. Personally I prefer to create it as angular does for us.

On the other hand, you can get the behavior you expected when you started to read this chapter, that is the transcluded html using the isolate scope. I know you're smart and you figured it out, but there it is:

---

<sup>6</sup><http://plnkr.co/edit/5HZ33b7IbO8badIqrMrn?p=preview>



```
1 angular.module('app').directive('person', function() {
2   return {
3     restrict: 'EA',
4     scope: {
5       header: '='
6     },
7     transclude: true,
8     link: function(scope, element, attrs, ctrl, transclude) {
9       scope.person = {
10         name: 'Directive Joe',
11         profession: 'Scope guy'
12       };
13
14       scope.header = 'Directive\'s header';
15
16       transclude(scope, function(clone, scope) {
17         element.append(clone);
18       });
19     }
20   };
21 });
```

### Directive's header

Hello, I am Directive Joe and,

I am a Scope guy

Check it [here](#)<sup>7</sup>

Take in mind that maybe the people that will consume your directives are not aware that you're tweaking the transcluded scope, so if you use it, be sure you document it well.

## Summary

The scope on the transcluded html is not as straightforward as we thought it were. The transcluded html doesn't contain the same scope as our directive but the directive's parent scope. Luckily for us, when transcluding by hand, we can override that scope if we need to, but beware, your users couldn't expect that.

---

<sup>7</sup><http://plnkr.co/edit/VhoLVzl50h6wfx1uLegF?p=preview>

## 11.4 Transclude 'element'

Let's build a simple clone of `ng-repeat` here, just the basic features. The first version we are going to implement has a little issue, but we need to realize why that happen before we fix it.

We need a name, restrict it just for attributes and also set the transclusion to true:

```
1 angular.module('app').directive('atRepeat', function() {
2   return {
3     restrict: 'A',
4     transclude: true,
5     link: function(scope, element, attrs, ctrl, transclude) {
6
7     }
8   };
9 });
```

So far so good. What's the first thing we need to achieve inside our link function? We need to be able to parse the expression we pass to it (`item in collection`) and extract the different bits of it. For that we need a regular expression:

```
1 var expression = attrs.atRepeat;
2 var match = expression.match(/^s*(.+)s+in\s+(.*)s*$/);
```

First, we get the expression from the attribute `atRepeat` and then we check if it matches with our regex. Our regex basically does:

- `^` -> Matches the beginning of the string.
- `\s*` -> Any number of spaces.
- `(.+)s+` -> A group that matches any character (minimum 1).
- `\s+` -> A space or more.
- `in` -> the word "in" literally.
- `\s+` -> A space or more.
- `(.+)s*` -> A group that matches any character (minimum 1).
- `\s*` -> Any number of spaces.
- `$` -> Matches the end of the string.

We create two groups, the first one will match the `item` of the iteration and the second one will match the `collection`.

If we don't use the correct expression, we can leave early:

```
1  if (!match) {  
2    throw new Error("Expected expression in form of '_item_ in _collection_');  
3  }
```

Now that we are sure that we have the correct expression, let's split it:

```
1  var iterationItemExpr = match[1];  
2  var collectionExpr = match[2];
```

The first match group is the iteration item name, and the second match group is the name of the collection we iterate.

So if for example we have `person in people`, `person` is the item name we use to iterate through the collection and `people` itself is the collection.

Even when we have now access to the collection to iterate, our directive needs an internal array to maintain its own state:

```
1  var items = [];
```

Ok, what now? We need to retrieve our real collection from parsing `collectionExpr` and for every item on it we want to get a clone of our transcluded `html` and append it to our element.

We can achieve that by using `$parse` on our `collectionExpr`, but since we want to update our directive every time the collection changes, I think it is more sensible to use a `$watch` which will parse the collection for us and also fire when our collection changes.



`$parse` is a service which gets expressions like `foo` and returns a function that when invoked, matches that expression against the scope and return a value. So if we have `$scope.foo = 20`; when we run the function that `$parse` returns, we get 20.

Let's code the `$watch`:

```
1  scope.$watchCollection(collectionExpr, function(collection) {  
2  
3  });
```

The good thing about the watch is that it will run every time the collection changes but also once when it is created. That means that we can put all of our logic inside and it will also run when the directive is executed for the first time.



`$watchCollection` is a variation of `$watch` where it allows watching over a collection. It is more performant than using the third parameter of `$watch` (deep watch) but it goes less deep.

Now inside it, we need append a clone of the transcluded html for every item in the collection. Every sounds like a for, right? Let's define some variables and that for:

```
1  var i, block, childScope;
2
3  for(i = 0, l = collection.length; i < l; i++) {
4
5  }
```

If you ever used ng-repeat you will know that it creates a new child scope for every item of the collection, let's do that (inside the for):

```
1  childScope = scope.$new();
2  childScope[iterationItemExpr] = collection[i];
```

So we created here a new childScope based on the current scope of our directive and then we put inside it the current item of the collection iteration. If we follow the example we put earlier, that item we created on the childScope will be named person.

Next step is to append the transcluded html for every item (inside the for as well):

```
1  transclude(childScope, function(clone, cloneScope) {
2    element.append(clone);
3    block = {};
4    block.el = clone;
5    block.scope = cloneScope;
6    items.push(block);
7  });
```

Here, we use our childScope as the scope of the transcluded html and we append it to the element. Also we create an object which will contain the transcluded html as well as the childScope and we push it to our internal items collection.

This is our complete directive so far:

```
1 angular.module('app').directive('atRepeat', function() {
2   return {
3     restrict: 'A',
4     transclude: true,
5     link: function(scope, element, attrs, ctrl, transclude) {
6       var expression = attrs.atRepeat;
7       var match = expression.match(/^s*(.+)s+in\s+(.*)s*$/);
8
9       if (!match) {
10        throw new Error(
11          "Expected expression in form of '_item_ in _collection_'
12        );
13      }
14
15      var iterationItemExpr = match[1];
16      var collectionExpr = match[2];
17      var items = [];
18
19      scope.$watchCollection(collectionExpr, function(collection) {
20        var i, block, childScope;
21
22        for(i = 0, l = collection.length; i < l; i++) {
23          childScope = scope.$new();
24          childScope[iterationItemExpr] = collection[i];
25
26          transclude(childScope, function(clone, cloneScope) {
27            element.append(clone);
28            block = {};
29            block.el = clone;
30            block.scope = cloneScope;
31            items.push(block);
32          });
33        }
34      });
35    }
36  };
37 });
```

Let's try it with this controller:

```
1 angular.module('app').controller('MainCtrl', function($scope) {
2     $scope.pages = [
3         {
4             url: 'www.google.com',
5             name: 'Google'
6         },
7         {
8             url: 'www.angular-tips.com',
9             name: 'Angular Tips'
10        },
11        {
12            url: 'www.stackoverflow.com',
13            name: 'Stackoverflow'
14        }
15    ];
16 });
```

And this html:

```
1 <ul>
2   <li at-repeat="page in pages">
3     <a href="{{page.url}}">{{page.name}}</a>
4   </li>
5 </ul>
```

And what we get is:

- [Google Angular Tips Stackoverflow](#)

Uh, that is weird, we expect three `<li>` elements, one per item, not one `<li>` with the three items on it. What happened?

```
▼ <ul>
  ▼ <li at-repeat="page in pages">
    <a href="www.google.com" class="ng-binding ng-scope">Google</a>
    <a href="www.angular-tips.com" class="ng-binding ng-scope">Angular Tips</a>
    <a href="www.stackoverflow.com" class="ng-binding ng-scope">Stackoverflow</a>
  </li>
</ul>
```

So there is just one `<li>` with the three elements on it. How so?

Let's visit our html again:

```
1 <ul>
2   <li at-repeat="page in pages">
3     <a href="{{page.url}}">{{page.name}}</a>
4   </li>
5 </ul>
```

We have our directive on the `<li>` element and then inside it what we have is an `<a>` tag. The transcluded html is in fact that `<a>` tag so if we create one per item, we are in fact creating three `<a>` tags and where do we put it? inside the element with the directive, which is the `<li>` tag.

I understand that, but that is not the behavior we want. We want to grab the children elements but **also** the element which contains the directive. In other words, we want to grab the `<li>` and `<a>` tags and clone it as many times as needed.

Here is where transclusion: `'element'` comes. The difference of `element` compared to `true` is that `element` will also grab the element where the directive is in. As a downside, you can't use a template with `element` transclusion.

Alright, let's change our directive to use this new kind of transclusion:

```
1 transclude: 'element'
```

Now, let's revisit how we appended our transcluded html:

```
1 element.append(clone);
```

If our element itself is also part of the transcluded html, we can't append it to itself, right? In fact what we want to do is to append it to the parent element (the `<ul>`). We can do that:

```
1 element.parent().append(clone);
```

If we run now our code, we will see:

- [Google](#)
- [Angular Tips](#)
- [Stackoverflow](#)

That is quite better, and in fact:

```

▼ <ul>
  <!-- atRepeat: page in pages -->
  ▼ <li at-repeat="page in pages" class="ng-scope">
    <a href="www.google.com" class="ng-binding">Google</a>
  </li>
  ▼ <li at-repeat="page in pages" class="ng-scope">
    <a href="www.angular-tips.com" class="ng-binding">Angular Tips</a>
  </li>
  ▼ <li at-repeat="page in pages" class="ng-scope">
    <a href="www.stackoverflow.com" class="ng-binding">Stackoverflow</a>
  </li>
</ul>

```

We got it right this time!

Now, if you remember, we have an internal `items` collection. What for? Well, if we push one new item to our collection, our directive will run and will append a new transclusion html per item **without** removing the old ones! So, before we did the transclusion, we do:

```

1  if (items.length > 0) {
2    for (i = 0, l = items.length; i < l; i++) {
3      items[i].el.remove();
4      items[i].scope.$destroy();
5    }
6    items = [];
7  }

```

Then when our collection changes, we loop through our internal `items` collection to remove every item element and destroy its scope. That will fix the issue.

Final result:

```

1  angular.module('app').directive('atRepeat', function() {
2    return {
3      restrict: 'A',
4      transclude: 'element',
5      link: function(scope, element, attrs, ctrl, transclude) {
6        var expression = attrs.atRepeat;
7        var match = expression.match(/^s*(.+)\s+in\s+(.*)\s*$$/);
8
9        if (!match) {
10         throw new Error(
11           "Expected expression in form of '_item_ in _collection_'
12         );
13       }
14
15       var iterationItemExpr = match[1];
16       var collectionExpr = match[2];

```



```

17     var items = [];
18
19     scope.$watchCollection(collectionExpr, function(collection) {
20         var i, block, childScope;
21
22         // Delete all items first
23         if (items.length > 0) {
24             for (i = 0, l = items.length; i < l; i++) {
25                 items[i].el.remove();
26                 items[i].scope.$destroy();
27             }
28             items = [];
29         }
30
31         for(i = 0, l = collection.length; i < l; i++) {
32             childScope = scope.$new();
33             childScope[iterationItemExpr] = collection[i];
34
35             transclude(childScope, function(clone, cloneScope) {
36                 element.parent().append(clone);
37                 block = {};
38                 block.el = clone;
39                 block.scope = cloneScope;
40                 items.push(block);
41             });
42         }
43     });
44 }
45 };
46 });

```

You can see it working [here](#)<sup>8</sup>



NOTE: This directive is not meant for real usage, ng-repeat has a bunch of performance improvements that we don't have here.

## The tests

Now the testing! Thankfully, they are way easier.

Let's create the basic skeleton:

---

<sup>8</sup><http://plnkr.co/edit/0lsXHRLhygIyDyLNq7xP?p=preview>

```
1 describe('directive: atRepeat', function() {
2   var parentScope, liEls, element, $compile;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function($rootScope, _$compile_) {
7     parentScope = $rootScope;
8
9     $compile = _$compile_;
10
11     parentScope.numbers = [1, 2, 3, 4, 5];
12     element = angular.element(
13       '<ul><li at-repeat="number in numbers">Number: {{number}}</li></ul>'
14     );
15
16     $compile(element)(parentScope);
17     parentScope.$digest();
18
19     liEls = element.find('li');
20   }));
21 });
```

We created a list of numbers on our scope, compiled an element with our directive and also we can see how I grab all our `<li>` items and store it on a `liEls` variable. I do this last bit because I need to do that in every test, and a `beforeEach` is the right place for that.

Right, first test:

```
1 it('creates one li per number (5)', function() {
2   expect(liEls.length).toBe(5);
3 });
```

If we have 5 numbers, we expect 5 `<li>` elements.

```
1 it('have the numbers in the correct order', function() {
2   expect(liEls.eq(0).text()).toBe('Number: 1');
3   expect(liEls.eq(4).text()).toBe('Number: 5');
4 });
```

They need to be in order, so the first `<li>` will be the one with the number 1 and the last `<li>` the one with the number 5.

```
1 it('contains a new scope for every item', function() {
2   var scopeLi0 = liEls.eq(0).scope();
3   var scopeLi1 = liEls.eq(1).scope();
4   expect(scopeLi0.$parent).toBe(parentScope);
5   expect(scopeLi0).not.toBe(scopeLi1);
6 });
```

Every item should have their own scope, so we verify that they are not the parent scope and that every item have their own scope.

```
1 it('updates when a new element is added to the collection', function() {
2   expect(liEls.length).toBe(5);
3
4   parentScope.numbers.push(6);
5   parentScope.$digest();
6
7   liEls = element.find('li');
8   expect(liEls.length).toBe(6);
9 });
```

We have 5 items on it, if we push a new item to our collection, we now have 6 `<li>` items.

```
1 it('throws an exception if the repeat syntax is not correct', function() {
2   element = angular.element(
3     '<ul><li at-repeat="number on numbers">Number: {{number}}</li></ul>'
4   );
5
6   expect(function() {
7     $compile(element)(parentScope);
8   }).toThrow();
9 });
```

Lastly, we verify that a wrong usage of our directive throws an exception.

Check the tests [here](#)<sup>9</sup>

## Summary

In the cases where we need to also transclude the element with out directive, we can use `transclude: element`. `ng-repeat` is a brilliant example of doing that because it needs to clone not only what's inside of the element but also the element itself.

---

<sup>9</sup><http://plnkr.co/edit/ou7RI5zkoj9IcmHNI2UU?p=preview>

# III Directives Lifecycle

The directives have their own lifecycle or in other words, they go through a serie of steps to initialize every part of it. In this part, we are going to dig into the more important steps.

## 12. Templates revisited

When Angular starts processing an HTML file, it will evaluate every element, attribute, class and comment to see if they match a directive, and once all directives has been collected, it will start the lifecycle for every each of them.

One of the first steps of the lifecycle is dealing with the `template` / `templateUrl`. If it has a `template`, it will just override the directive's element content with the template. If it has a `templateUrl` angular will request it and when it arrives, it will override the directive's element content as usual and it will continue with the next step on the lifecycle.

We already know how to use `template` and `templateUrl` and since we now know more about directives, we can dig a little bit more on this topic.

Both `template` and `templateUrl` accepts a function instead of a string:

```
1 template: function(tElement, tAttrs) {}
```

and:

```
1 templateUrl: function(tElement, tAttrs) {}
```

The parameters are:

- **tElement**: The original element where our directive lives.
- **tAttrs**: The attributes on the element.

We have to keep in mind, that at this stage of the lifecycle, we haven't created a scope yet, so if we have any interpolation in our element, they won't be resolved yet.



The `t` that prefix our parameters is a convention and means `template`.

Within the `template` function we can do anything we want (with the scope limitation) and finally return a string which will represent our template.

On the counterpart, `templateUrl` needs to return a string which represents the URL of the template to fetch.

### Summary

Templates is one of the first step that occurs on the lifecycle. We can use them on their simple form or maybe using a function.

## 13. Compile

After we are done with the template, the compiler will check if our directive has a function called `compile` (which is part of the DDO) and if so, it will execute it. It is like:

```
1 compile: function(tElement, tAttrs) {  
2  
3 }
```

The parameters are the same exact ones that the ones from `template` / `templateUrl` but in this case they will have the changes we made on `template`.

Let's see an example:

```
1 angular.module('app').directive('foo', function() {  
2   return {  
3     scope: {  
4       myAttr: '@'  
5     },  
6     transclude: true,  
7     template: '<div>Hello, {{myAttr}} here</div><span ng-transclude></span>',  
8     compile: function(tElement, tAttrs) {  
9       console.log(tElement.prop('outerHTML'));  
10      console.log(tAttrs.myAttr);  
11    }  
12  };  
13 });
```

```
1 <div foo my-attr="{{something}}">  
2   <p>Something to transclude</p>  
3 </div>
```

Here we have a simple directive with a template and transclusion. Works perfectly and since we created a `compile` function, angular will call it when it starts applying the directive to the node.

What's the status of the directive at this point of the lifecycle? And what can we do (and not do) here?

Let's start with the first question: As I said before, at this point Angular has not created a scope yet, so there is no data binding and the transclusion is not even done. In fact if we check the console, in the first log we see:

```

1 <div foo="" my-attr="{{something}}">
2   <div>Hello, {{myAttr}} here</div>
3   <span ng-transclude=""></span>
4 </div>

```

As I said, we see {{myAttr}} instead of the real value of something and the transclusion is not done yet.

If we see the second log, we see:

```

1 {{something}}

```

Here we have the “raw” element, no interpolation being done.

So what can't we do here? For starters, we can't interpolate that attribute nor we can resolve that unfinished transclusion and that is fine because that is meant for a future step on the lifecycle.

Alright, so what can we do then? Here we can do some DOM manipulation before our directive moves to the next step and the databinding is setup and the transclusion is done. In a future chapter we will see that we can't simply add new DOM inside the link function because the databinding is already set up so the new added DOM won't have its directives processed. But don't worry, we will look into an easy solution for that :)

Back to compile, at this stage of the lifecycle, the databinding is yet to be set up, so we can safely add new DOM with directives and they will be processed as well when our directive reaches the next step.

To prove that, let's change the directive a bit:

```

1 angular.module('app').directive('foo', function() {
2   return {
3     scope: {
4       myAttr: '@'
5     },
6     transclude: true,
7     template: '<div>Hello, {{myAttr}} here</div>',
8     compile: function(tElement, tAttrs) {
9       tElement.append('<span ng-transclude></span>');
10    }
11  };
12 });

```

Here we removed the element with the transclusion from the template and we appended it by hand on the compile function. As I said earlier, we can add here elements with directives because the

step where the databinding is set up is yet to be reached. This change for itself has no point, but we can potentially do some calculations and append this element on a certain way based on those calculations.

Another interesting thing about `compile` is its returning value. What does it returns? `compile` returns a function and that function is the `link` function:

```
1 compile: function(tElement, tAttrs) {  
2   // Do stuff here  
3   return function(scope, element, attrs) {  
4     // Our link function  
5   };  
6 }
```

In fact, the typical:

```
1 link: function(scope, element, attrs) {  
2  
3 }
```

Is a [syntactic sugar](http://en.wikipedia.org/wiki/Syntactic_sugar)<sup>1</sup> for the previous snippet. Said that, we can't use this the `link` function in this way if we have a `compile` function in our directive, we have to return it from `compile`.

Now, what are the use cases of having a `compile` function? In the past, when angular performance wasn't that great, we used to write a lot of code on the `compile` function before the databinding is set up, because there you could write stuff that were more performant than if you wrote them in the `link` function. Back to present, we don't have that need anymore. We can achieve the same performance in the `link` function, so the `compile` function is not that useful anymore.

There is at least one use case where the `compile` function is useful. Imagine we have a directive where we force our users to use it on a certain way, maybe the expression we pass into it is special, or we have a few optional attributes but some of them only work if another attribute is also there. In those cases you can check all those conditions and if any of them doesn't match, we can throw an exception or we simply stop the directive execution. If we know that our directive is that picky, we wouldn't like to run its entire lifecycle to just stop it.

Uhm, in fact, we don't have to imagine it. Remember our `atRepeat` we created in a previous chapter? That is what I call a picky directive. If the expression we pass to it doesn't match our regular expression, we just simply throw an exception. We can check this on the `compile` function, before any databinding is being done, because if that doesn't pass, we don't need angular to continue processing our directive.

Check how can we rewrite this:

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Syntactic\\_sugar](http://en.wikipedia.org/wiki/Syntactic_sugar)



```

1  compile: function(tElement, tAttrs) {
2      var expression = tAttrs.atRepeat;
3      var match = expression.match(/^s*(.)\s+in\s+(.*)\s*$/);
4
5      if (!match) {
6          throw new Error("Expected expression in form of '_item_ in _collection_');
7      }
8
9      var iterationItemExpr = match[1];
10     var collectionExpr = match[2];
11     var items = [];
12
13     return function(scope, element, attrs, ctrl, transclude) {
14         scope.$watchCollection(collectionExpr, function(collection) {
15             var i, block, childScope;
16
17             // Delete all items first
18             if (items.length > 0) {
19                 for (i = 0, l = items.length; i < l; i++) {
20                     items[i].el.remove();
21                     items[i].scope.$destroy();
22                 }
23                 items = [];
24             }
25
26             for(i = 0, l = collection.length; i < l; i++) {
27                 childScope = scope.$new();
28                 childScope[iterationItemExpr] = collection[i];
29
30                 transclude(childScope, function(clone, cloneScope) {
31                     element.parent().append(clone);
32                     block = {};
33                     block.el = clone;
34                     block.scope = cloneScope;
35                     items.push(block);
36                 });
37             }
38         });
39     };
40 }

```

We moved the regexp matching and initialization into `compile` and we left the `$watchCollection` for the `link` function. There is no real need to move the initialization there, but I think it is more

readable that way. Now it is easier to see that we are checking some expression and then, extracting it.

Check it [here](#)<sup>2</sup>

The tests for this directive doesn't change. We didn't change the behaviour, we only refactored it a bit.

## Summary

We can create a `compile` function in our directives which will be called eventually on our directive. That function will receive the original element and attributes before any databinding is set up.

Also, our `compile` function will return our well known `link` function.

---

<sup>2</sup><http://plnkr.co/edit/fLB9t4mpPywOjaTQ7YgA?p=preview>

# 14. Controller

Half of the lifecycle is done at this point and it is now when angular will attach the scope on our directive and set up the databinding. It will use the same scope of the element, create a new one or even an isolated one.

Once we have our scope at hand, angular will check if our directive has a controller defined. A Controller? Yes, you can use a controller in your directives and for now, we can say that it allows us to do the same things we do with our link function (we will talk more about when to use one or the other later on this chapter).

There are two ways of defining a controller in our directive:

```
1 angular.module('app').directive('foo', function() {
2   return {
3     controller: function() {
4       // Something in here
5     }
6   };
7 });
```

or:

```
1 angular.module('app')
2
3   .controller('FooCtrl', function() {
4     // Something in here
5   })
6
7   .directive('foo', function() {
8     return {
9       controller: 'FooCtrl'
10    };
11  });
```

So you can directly create a controller within the directive or you can create one outside and then reference it on the directive (the name of the controller doesn't matter). If possible, stick with latter option, that will allow you to test the controller in isolation without having to load the directive.

On the other hand, Angular will be kind enough to create some objects for us:

- **\$scope**: It is the scope we set up earlier, the one that our directive will use.
- **\$element**: The element where our directive lives, but in this case after the databinding and transclusion is set up.
- **\$attrs**: The attributes on our element, also after databinding is set up.
- **\$transclude**: A reference to the transclude function.

Having this in mind, we can do something like:

```
1 angular.module('app')
2   .controller('FooCtrl', function($scope, $element, $attrs, $transclude) {
3
4   });
```

Sounds familiar to you? Sounds like any controller we already use on our applications! Also the parameters are the same ones that the ones on the `link` function. The only difference is that the controller ones are injected so you can't change their name but you can change the order. Also, you can inject more stuff into the controller.



It is now when you realize **why** you can't inject `$scope` in a service and that is because `$scope` is not a service, it is an object that angular creates for us to use in a controller.

Allright, we are controller experts now, let's go with the example, shall we?

This time, let's create a stopwatch using our controller. The stopwatch will contain a timer, a registry of laps and a bunch of buttons to work with it. We also decided that we want to call back the app's controller when we stop the stopwatch with the laps. Let's see:

```
1 angular.module('app')
2   .directive('stopwatch', function() {
3     return {
4       scope: {
5         onStop: '&'
6       },
7       templateUrl: 'stopwatch.html',
8       controller: 'StopwatchController'
9     };
10  });
```

There is our callback function, the template and our controller. Let's see the template first:

```
1 <p class="current">{{ currentTime | date:'mm:ss' }}</p>
2
3 <div class="laps">{{ laps }}</div>
4
5 <button ng-disabled="running" ng-click="start()">Start</button>
6 <button ng-disabled="!running" ng-click="lap()">Lap</button>
7 <button ng-disabled="!running" ng-click="stop()">Stop</button>
8 <button ng-disabled="running" ng-click="reset()">Reset</button>
```

Nothing too important here. Let's define the controller:

```
1 angular.module('app')
2   .controller('StopwatchController', function($scope, $interval) {
3     var interval;
4
5     $scope.running = false;
6     $scope.laps = [];
7     $scope.currentTime = 0;
8   });
```

We need here access to the directive's scope (isolated scope) and also to the `$interval` service. The first thing we need to do is to define some variables: One for the interval, an empty array for laps, the current time on the timer and also a boolean to indicate that the stopwatch is currently stopped.

```
1 $scope.start = function() {
2   $scope.running = true;
3
4   interval = $interval(function() {
5     $scope.currentTime += 1000;
6   }, 1000);
7 };
```

The first method we will implement is the one that runs our timer and an `$interval` works pretty well for that use case. We also set that our stopwatch is currently running.

```
1 $scope.stop = function() {  
2   $scope.running = false;  
3   $interval.cancel(interval);  
4   $scope.laps.push(addLap($scope.currentTime));  
5   $scope.currentTime = 0;  
6  
7   if ($scope.onStop) {  
8     $scope.onStop({laps: $scope.laps});  
9   }  
10 };
```

To stop the timer, we just need to cancel the interval we created. We also set the state of the stopwatch to be not running, we add the current time as a lap and if we set the callback, we will run it passing our laps as we learnt on a previous chapter.

```
1 $scope.lap = function() {  
2   $scope.laps.push(addLap($scope.currentTime));  
3   $scope.currentTime = 0;  
4 };  
5  
6 $scope.reset = function() {  
7   $scope.laps = [];  
8 };
```

Every time we hit the lap button, we just simply add a lap and we restart the timer. To reset, we just need to put a new empty array for laps.

```
1 function addLap(time) {  
2   return Math.floor(time / 1000);  
3 }
```

Finally the little helper we created to add the laps.

00:07

[4,8]

Check it [here](#)<sup>1</sup>

---

<sup>1</sup><http://plnkr.co/edit/i2hiXc9k81q3agA00zK0?p=preview>

## The tests

Let's test this out:

```
1 describe('Directive: stopwatch', function() {
2
3   beforeEach(module('app'));
4
5   describe('directive', function() {
6     var element;
7
8     beforeEach(inject(function($compile, $rootScope) {
9       element = angular.element('<stopwatch></stopwatch>');
10
11       $compile(element)($rootScope);
12       $rootScope.$digest();
13     }));
14
15     it('should contain a formatted current time', function() {
16       var current = element.find('.current');
17       expect(current.text()).toBe('00:00');
18     });
19
20     it('should contain an empty list of laps', function() {
21       var laps = element.find('.laps');
22       expect(laps.text()).toBe('[]');
23     });
24
25     it('should contain 4 buttons', function() {
26       var buttons = element.find('button');
27       expect(buttons.length).toBe(4);
28     });
29   });
30
31 });
```

First, we are going to test the directive part of it and since the directive itself doesn't have the functionality, we just need to check that our template is rendering what we expect.

Now it is time to test the controller, something that we never did on this book, so let's go step by step this time:

(Insert this describe below the describe of the directive but inside of the main one)

```
1 describe('controller', function() {
2     var $scope, $interval;
3
4     beforeEach(inject(function($rootScope, $controller, _$interval_) {
5         $scope = $rootScope.$new();
6         $interval = _$interval_;
7
8         $controller('StopwatchController', {$scope: $scope});
9     }));
10 });
```

To test a controller, we inject the `$controller` service which is what angular uses internally to load our controllers. `$controller` receives the name of the controller to instantiate and optionally an object with some of the injections. Wait, some? Yes, imagine we have a controller like:

```
1 angular.module('app')
2   .controller('MyController', function(foo, bar, baz) {
3
4   });
```

Then when testing that controller, you decide that you want to mock `foo` and `baz` but not `bar`. In that case you can do:

```
1 $controller('MyController', {foo: {}, baz: aMockBaz});
```

It will instantiate that controller but for `foo` it will pass an empty object and for `baz` it will pass a reference to something called `aMockBaz`.

Back to our controller. Since we need a reference of its scope, we created one outside and then we pass it to `$controller` so if we modify it inside the controller, our `$scope` “from outside” will also be updated (same reference). We also injected `$interval` on our tests but we didn’t pass it to `$controller` as we did with `$scope`. Can we simply pass it? Yes, nothing will change, but since `$interval` is a normal service, it is a singleton, that means that the one that `StopwatchController` will receive and the one we have already is the same one.

The good part here is that we don’t need to run any directive’s lifecycle to get the scope, we just need to provide one scope, any scope.

Now that our controller is instantiated, we can start creating tests.



```
1  it('starts a time on start', function() {
2    $scope.start();
3
4    $interval.flush(50 * 1000); // 50 seconds
5
6    expect($scope.currentTime).toBe(50000);
7  });
```

When testing a code that runs an `$interval`, we can use the method `flush` to simulate that our interval did run for a certain amount of time. In this test, we start the timer, we simulate that 50 seconds have passed and then we assert that the current time is well, 50 seconds.

```
1  it('is able to add laps', function() {
2    $scope.start();
3
4    // We let pass 30 seconds
5    $interval.flush(30 * 1000);
6
7    $scope.lap();
8
9    expect($scope.laps).toEqual([30]);
10
11    // We let pass another 15 seconds
12    $interval.flush(15 * 1000);
13
14    $scope.lap();
15
16    expect($scope.laps).toEqual([30, 15]);
17  });
```

To tests the laps, we start the timer, we simulate 30 seconds and we call `$scope.lap` and then verify our laps array. We did it again to verify that new laps are always added.

```
1 it('stops the timer', function() {
2   $scope.start();
3
4   spyOn($interval, 'cancel');
5
6   $scope.stop();
7
8   expect($interval.cancel).toHaveBeenCalled();
9 });
```

We spy on the `cancel` method of the `$interval` to verify that it has been called when we stop the timer.

```
1 it('adds a lap on stop', function() {
2   $scope.start();
3
4   $interval.flush(10 * 1000);
5
6   $scope.stop();
7
8   expect($scope.laps).toEqual([10]);
9 });
```

Also on stop, we add a new lap.

```
1 it('calls back the controller on stop', function() {
2   $scope.onStop = jasmine.createSpy();
3
4   $scope.stop();
5
6   expect($scope.onStop).toHaveBeenCalledWith({laps: $scope.laps});
7 });
```

Since we are not using the directive here, we can't pass our callback as a parameter to test it, so we can simply create a dummy `onStop` function on our scope and then verify that it gets called.

```
1  it('can reset the laps', function() {
2    $scope.start();
3
4    $interval.flush(10 * 1000);
5
6    $scope.stop();
7
8    $scope.reset();
9
10   expect($scope.laps).toEqual([]);
11 });
```

We can add some laps in here and then verify that our laps array gets reseted correctly.

Check the [tests](#)<sup>2</sup>

## Summary

We can define controllers for our directives and we can do anything we want on those controllers. Also, we can define them outside of the directive so we can test them easily. We just need to grab a reference to its `$scope` and then use it to assert what you need!

If you are wondering... When to use a controller and when to use the link function? The Angular core team recommends you to use the `link` function where possible and leaving the controller to when you need to share functionality. Share functionality? That is a subject for the next part of the book.

---

<sup>2</sup><http://plnkr.co/edit/TXl0tSs65XuEoJL1KkfV?p=preview>

## 14.1 Controller As

I liked the controller example from before, but what if I am a Controller As user? Angular has you covered. Let's rewrite the previous demo to use Controller As:

```
1 angular.module('app')
2   .directive('stopwatch', function() {
3     return {
4       scope: {
5         onStop: '&'
6       },
7       templateUrl: 'stopwatch.html',
8       controller: 'StopwatchController',
9       controllerAs: 'ctrl'
10    };
11  });
```

There is a property on the DDO named `controllerAs` which we can use for that purpose! As an alternative, you can simply do:

```
1 controller: 'StopwatchController as ctrl'
```

Both ways are correct.

Let's see the template:

```
1 <p class="current">{{ ctrl.currentTime | date:'mm:ss' }}</p>
2
3 <div class="laps">{{ ctrl.laps }}</div>
4
5 <button ng-disabled="ctrl.running" ng-click="ctrl.start()">Start</button>
6 <button ng-disabled="!ctrl.running" ng-click="ctrl.lap()">Lap</button>
7 <button ng-disabled="!ctrl.running" ng-click="ctrl.stop()">Stop</button>
8 <button ng-disabled="ctrl.running" ng-click="ctrl.reset()">Reset</button>
```

The same one as before, but using Controller As.

For the controller itself:

```
1 angular.module('app')
2
3 .controller('StopwatchController', function($interval) {
4     var ctrl = this;
5
6     var interval;
7
8     ctrl.running = false;
9     ctrl.laps = [];
10    ctrl.currentTime = 0;
11
12    ctrl.lap = function() {
13        ctrl.laps.push(addLap(ctrl.currentTime));
14        ctrl.currentTime = 0;
15    };
16
17    ctrl.reset = function() {
18        ctrl.laps = [];
19    };
20
21    ctrl.start = function() {
22        ctrl.running = true;
23
24        interval = $interval(function() {
25            ctrl.currentTime += 1000;
26        }, 1000);
27    };
28
29    ctrl.stop = function() {
30        ctrl.running = false;
31        $interval.cancel(interval);
32        ctrl.laps.push(addLap(ctrl.currentTime));
33        ctrl.currentTime = 0;
34
35        if (ctrl.onStop) {
36            ctrl.onStop({laps: ctrl.laps});
37        }
38    };
39
40    function addLap(time) {
41        return Math.floor(time / 1000);
42    }
```

```
43    });
```

The implementation is the same one, we just needed to remove `$scope` and start using this everywhere. Actually I assigned `ctrl` as this because this can be something different in some parts.

There is something important here... What happens with our callback (`onStop`) ? That is going to be created on the isolated scope, but we are not using it with `Controller As`. Should we inject `$scope` then? No.

There is another property on the DDO called `bindToController` and if you set it to `true`, angular will move `onStop` to the controller instance for us.

```
1  angular.module('app')
2    .directive('stopwatch', function() {
3      return {
4        scope: {
5          onStop: '&'
6        },
7        templateUrl: 'stopwatch.html',
8        controller: 'StopwatchController',
9        controllerAs: 'ctrl',
10       bindToController: true
11     };
12   });
```

00:07

[4,8]

Start Lap Stop Reset

Check it [here](http://plnkr.co/edit/y3YfX1sYO1tXR21MtOm?p=preview)<sup>3</sup>

## The tests

The tests for the controller here are a bit different, let see them:

---

<sup>3</sup><http://plnkr.co/edit/y3YfX1sYO1tXR21MtOm?p=preview>

```
1 describe('Directive: stopwatch', function() {
2
3   beforeEach(module('app'));
4
5   describe('controller', function() {
6     var ctrl, $interval;
7
8     beforeEach(inject(function($controller, _$interval_) {
9       $interval = _$interval_;
10
11       ctrl = $controller('StopwatchController');
12     }));
13
14     it('starts a time on start', function() {
15       ctrl.start();
16
17       $interval.flush(50 * 1000); // 50 seconds
18
19       expect(ctrl.currentTime).toBe(50000);
20     });
21
22     it('is able to add laps', function() {
23       ctrl.start();
24
25       // We let pass 30 seconds
26       $interval.flush(30 * 1000);
27
28       ctrl.lap();
29
30       expect(ctrl.laps).toEqual([30]);
31
32       // We let pass another 15 seconds
33       $interval.flush(15 * 1000);
34
35       ctrl.lap();
36
37       expect(ctrl.laps).toEqual([30, 15]);
38     });
39
40     it('stops the timer', function() {
41       ctrl.start();
42
```

```
43     spyOn($interval, 'cancel');
44
45     ctrl.stop();
46
47     expect($interval.cancel).toHaveBeenCalled();
48 });
49
50 it('adds a lap on stop', function() {
51     ctrl.start();
52
53     $interval.flush(10 * 1000);
54
55     ctrl.stop();
56
57     expect(ctrl.laps).toEqual([10]);
58 });
59
60 it('calls back the controller on stop', function() {
61     ctrl.onStop = jasmine.createSpy();
62
63     ctrl.stop();
64
65     expect(ctrl.onStop).toHaveBeenCalledWith({laps: ctrl.laps});
66 });
67
68 it('can reset the laps', function() {
69     ctrl.start();
70
71     $interval.flush(10 * 1000);
72
73     ctrl.stop();
74
75     ctrl.reset();
76
77     expect(ctrl.laps).toEqual([]);
78 });
79 });
80 });
```

`$controller` will instantiate and return an instance and we can use that instance to do our tests. So where we used `$scope` before, we just need to use that `ctrl` variable.



Check the [tests](#)<sup>4</sup>

## Summary

If you prefer to use `Controller As` over `$scope`, you can simply tell your directive to do that. Also if we have an isolated scope, we can tell angular to move all those properties to the controller's instance using `bindToController`.

---

<sup>4</sup><http://plnkr.co/edit/UHQbSP8EOKFLpLhSvH94?p=preview>

## 14.2 Swapping controllers

Imagine we have a directive where we have a couple of different implementations of the same idea. Can we put those different implementations in different controllers? In fact, yes, you can.

Instead of doing something like:

```
1 controller: function ($scope) { ... }
```

or:

```
1 controller: 'MyController'
```

You can do:

```
1 controller: '@'
```

If we put the controller property as @, we can specify what controller to use. If we have a person directive and we want to specify its controller, we can do:

```
1 <div person="AController"></div>
```

So AController will be the one used in the person directive in this case. We can also choose what attribute will be the one used to specify the controller, so we could do:

```
1 controller: '@',  
2 name: 'ctrl'
```

So the last example could be rewritten into:

```
1 <div person ctrl="AController"></div>
```



The controller name needs to be a string and never a property on the scope, that will never work. That means, no dynamic controllers.

I know you are smart enough and you already figured out that what we are doing here is no different than doing:

```
1 <div ng-controller="MainController"></div>
```

Right? In fact it doesn't. All the controllers we have been using since day 0 with angular are nothing more than a directive with a controller which explains where the `$scope` comes from.

```
1 angular.module('ng')
2   .directive('ngController', function() {
3     return {
4       restrict: 'A',
5       scope: true,
6       controller: '@',
7       priority: 500
8     };
9   });
```

That is the original code of the `ngController` directive. Ignore the priority for now (which will be explained in the next part of the book) and we can see that the directive is quite easy. It creates a new scope, it is restricted by attribute and it allows us to define what controller to use.

For the sake of practicing, let's create our own `ngController` but in this case instead of creating a new scope, let's create an isolated scope.

```
1 angular.module('app')
2   .directive('atController', function() {
3     return {
4       restrict: 'A',
5       scope: {},
6       controller: '@'
7     };
8   });
```

We can use it like we use `ngController`:

```
1 <div at-controller="MainController"></div>
```

The difference is that if we define this element nested inside another element, the new scope created won't inherit from the parent scope.

It could be useful to do stuff like:

```
1 <div ng-include="'foo.html'" at-controller="FooController"></div>
```

This way you can add partials to your app and the controller won't inherit the existing stuff.

Check it [here](#)<sup>5</sup>

## The tests

For the tests, we are going to test what makes this directive different than the original. Let's start with the skeleton:

```
1 describe('Directive: at-controller', function() {
2   var $controllerProvider, element;
3
4   beforeEach(module('app', function(_$controllerProvider_) {
5     $controllerProvider = _$controllerProvider_;
6   }));
7
8   beforeEach(inject(function($rootScope, $compile) {
9     var MainCtrl = function($scope) {
10       $scope.main = 'This is main ctrl';
11     };
12
13     var ChildCtrl = function($scope) {
14     };
15
16     $controllerProvider.register('MainCtrl', MainCtrl);
17     $controllerProvider.register('ChildCtrl', ChildCtrl);
18
19     element = angular.element('<div at-controller="MainCtrl">' +
20                               '<div id="child" at-controller="ChildCtrl">' +
21                               '{{main || "nothing inherited"}}' +
22                               '</div>' +
23                               '</div>');
24     $compile(element)($rootScope);
25     $rootScope.$digest();
26   }));
27 });
```

This time we not only need to load controllers, we also need to create them. Let's go step by step:

---

<sup>5</sup><http://plnkr.co/edit/ivBcL36ZUuiDmEuXwoFs?p=info>

The first thing we need is the “Controller creator” and that is the `$controllerProvider`. We can’t simply inject it using the `inject` function because it can’t simply inject providers. What can we do to inject a provider in our tests? The `module` function we were using all the time to load our modules has a second parameter which is a function where we can inject providers.

Ok, we can inject `$controllerProvider` from there and assign it to a local variable. What’s next? Next we need to create two controller so we just need to create two constructor functions and then use the `.register` method on `$controllerProvider` to register them as controllers.

Last, we create our dummy element with our directive and we compile it. For this test, we nested two `atController` directives to assert that the `ChildCtrl` is not inheriting from the parent.

```
1 it('should not allow inheritance', function() {  
2   var childDiv = element.find('#child');  
3   expect(childDiv.text()).toBe('nothing inherited');  
4 });
```

So we grab the `#child` div and we expect that its text is `nothing inherited` instead of `This is main ctrl`.

Check the [test](#)<sup>6</sup>

## Summary

If we have a directive where we have different implementations based on some condition, we can use `controller: '@'` to specify what implementation we can use at any moment. This could be useful if we have coded some type of dynamic templates.

---

<sup>6</sup><http://plnkr.co/edit/PmSS7UCisr47p8S1z9nU?p=preview>

# 15. Pre and post link

Wait, pre and post link? I thought that link was the next step... You weren't wrong, it is.

What you already know as link function is actually called post link function. Why is called post ?

Let me explain: Imagine we have 3 directives, foo, bar and baz like this:

```
1 angular.module('app')
2   .directive('foo', function() {
3     return {
4       template: '<div bar></div>',
5       compile: function() {
6         console.log('Foo compile');
7         return function() {
8           console.log('Foo link');
9         };
10      }
11    };
12  })
13
14  .directive('bar', function() {
15    return {
16      template: '<div baz></div>',
17      compile: function() {
18        console.log('Bar compile');
19        return function() {
20          console.log('Bar link');
21        };
22      }
23    };
24  })
25
26  .directive('baz', function() {
27    return {
28      compile: function() {
29        console.log('Baz compile');
30        return function() {
31          console.log('Baz link');
```

```
32     };  
33   }  
34   };  
35 })
```

What could you expect here? Something like:

```
1  Foo compile  
2  Bar compile  
3  Baz compile  
4  Foo link  
5  Bar link  
6  Baz link
```

Wrong! It is actually:

```
1  Foo compile  
2  Bar compile  
3  Baz compile  
4  Baz link  
5  Bar link  
6  Foo link
```

Angular will execute each directive compile (if any) in order, each controller (if any) in order and when it reaches the link function, it will start processing recursively that link function starting from the last one until it reaches the first directive link function. The goal behind this is that when Foo's link function is executed, all the children directives are completely processed. That allows us to do DOM manipulation there. Now you understand why it is called post link because it gets executed after all the linking has been done.

So, is there a pre link? Sure, how can we define it? We can:

```
1  compile: function() {  
2    return {  
3      pre: function() {  
4  
5      },  
6      post: function() {  
7  
8      }  
9    }  
10 }
```

Or:

```
1 link: {  
2   pre: function() {  
3  
4   },  
5   post: function() {  
6  
7   }  
8 }
```

The latter approach is the most common one if you don't need a compile function.

pre link receives the same arguments as post link except the transcludeFn function. The main difference here is that when pre link is executed, the child directives on our template are not linked yet so we can't do any DOM manipulation in here. There are some directives that can't be modified after being linked so you can configure them before its linking occurs and pre link is a good spot.

If we log our directives using both pre and post link, we get:

```
1 Foo compile  
2 Bar compile  
3 Baz compile  
4 Foo prelink  
5 Bar prelink  
6 Baz prelink  
7 Baz postlink  
8 Bar postlink  
9 Foo postlink
```

## Summary

What we were using all along is called post link function, and it is executed after all our children directives are completely processed (linked). We can safely do DOM manipulation there, but we can't do it on the pre link functions because if we change something in there, the compiler will fail to locate the correct elements for linking.



## **IV Directive Definition Object - Part 2**

Now that we have a good grasp on how directives work and how the directives do their lifecycle, it is time to continue with the rest of the DDO.

# 16. Priority

Having an element with a couple of directives, which one runs first? By default the directives on an element are ordered by alphabetical order.

Normally you won't care about which one runs first, but for me, there is two different use cases. The first one is when a directive modifies something (e.g. the element) and you want another directive to react different depending on that modification. In this case you can't rely on that alphabetical order, you need some kind of priority. Ok, but first a bit of theory.

## Changing directives priorities

Let's create two dummy directives:

```
1  angular.module('app')
2    .directive('foo', function() {
3      return {
4        compile: function() {
5          console.log('foo compile')
6
7          return {
8            pre: function() {
9              console.log('foo pre');
10             },
11            post: function() {
12              console.log('foo post')
13            }
14          }
15        }
16      };
17    })
18
19    .directive('bar', function() {
20      return {
21
22        compile: function() {
23          console.log('bar compile')
24
25          return {
```

```
26     pre: function() {
27         console.log('bar pre');
28     },
29     post: function() {
30         console.log('bar post')
31     }
32 }
33 }
34 };
35 })
```

And we use them:

```
1 <div foo bar></div>
```

What should we see? bar should run first (alphabetical order):

```
1 bar compile
2 foo compile
3 bar pre
4 foo pre
5 foo post
6 bar post
```

And that is correct, bar lifecycle runs before foo one and then when post link comes, it goes in the other way. By default a directive has a priority of 0 and we can change it:

```
1 angular.module('app')
2   .directive('foo', function() {
3     return {
4       priority: 1,
5       compile: function() {
6         console.log('foo compile')
7       }
8     }
9     return {
10      pre: function() {
11        console.log('foo pre');
12      },
13      post: function() {
14        console.log('foo post')
15      }
16    }
17  });
```

```
15     }
16   }
17   };
18 });
```

Since foo has more priority than bar, we now get:

```
1  foo compile
2  bar compile
3  foo pre
4  bar pre
5  bar post
6  foo post
```

Now foo runs before than bar and again the post-link goes the other way.

So we can say that the directive with more priority runs first and the directive with less priority is the one who runs its post-link first. Also, we can use negative priorities.

Let's put a real example now. Imagine you want to create directives for bootstrap buttons, so you have a btn directive, a btn-primary directive... And you need them to run in a certain order because you can't put a button as primary if there is no button.

Let's code it:

```
1  angular.module('plunker', [])
2    .directive('btn', function() {
3      return {
4        priority: -1,
5        link: function(scope, element, attrs) {
6          element.addClass('btn');
7        }
8      };
9    })
10
11   .directive('btnPrimary', function() {
12     return {
13       link: function(scope, element, attrs) {
14         if (element.hasClass('btn')) {
15           element.addClass('btn-primary');
16         } else {
17           throw new Error('You need to apply this directive with a btn');
18         }
19       }
20     };
21   })
```

```
19     }
20   };
21 });
```

Our `btn` directive has a negative priority so its `link` function will run before any other directive on the element (if we assume that the other directives has a priority 0 or more). With `btn` running first, we add a class to our element named `btn`. Then we just need to create other directives for the different types of buttons. These directives will have the default priority (0) so their `link` function will run after `btn`'s one. In our `btn-primary` directive, we check if the `btn` class exists and if so we add the `btn-primary` class and if not, we throw an exception.

So if we do something like:

```
1 <a btn btn-primary href="http://angular-tips.com">Angular Tips</a>
```

And assuming that we loaded `bootstrap.css`, we get:



Get it [here](#)<sup>1</sup>

## The tests

Tests are dead simple for this example, but here they are:

```
1 describe('Directive: bootstrap buttons', function() {
2   var $rootScope, $compiler;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function(_$rootScope_, _$compile_) {
7     $rootScope = _$rootScope_;
8     $compile = _$compile_;
9   }));
10
11  describe('btn', function() {
12    it('applies a btn class', function() {
13      var el = angular.element('<div btn></div>');
14      $compile(el)($rootScope);
15    });
16  });
17 });
```

---

<sup>1</sup><http://plnkr.co/edit/qi2dG099pR2gRb2NZz94?p=preview>

```

16     expect(el.hasClass('btn')).toBe(true);
17   });
18 });
19
20 describe('btn-primary', function() {
21   it('applies a btn-primary class if btn exist', function() {
22     var el = angular.element('<div btn btn-primary></div>');
23     $compile(el)($rootScope);
24
25     expect(el.hasClass('btn-primary')).toBe(true);
26   });
27
28   it('throws an error if there is no btn', function() {
29     var el = angular.element('<div btn-primary></div>');
30
31     expect(function() {
32       $compile(el)($rootScope);
33     }).toThrow();
34   });
35 });
36 });

```

I put both directives' test on the same file. For btn we simply check if it adds the btn class. For btn-primary we check that the btn-primary class is added if the btn class is present and we also check that if the btn class doesn't exist, we throw an exception.

Play with the tests [here](#)<sup>2</sup>

## Playing with transclude: 'element'

A directive with transclude: 'element' plays differently with priority. Let's see an example:

Using the at-repeat we created and updated through the book, let's code this:

```

1 <ul>
2   <li at-repeat="page in pages">
3     <a href="{ {page.url} }">{ {page.name} } </a>
4   </li>
5 </ul>

```

And now the controller:

---

<sup>2</sup><http://plnkr.co/edit/Lps0EB7GOMsDkUiEcMM8?p=preview>

```
1 angular.module('app')
2   .controller('MainCtrl', function($scope) {
3     $scope.pages = [
4       {
5         url: 'www.google.com',
6         name: 'Google'
7       },
8       {
9         url: 'www.angular-tips.com',
10        name: 'Angular Tips'
11      },
12      {
13        url: 'www.stackoverflow.com',
14        name: 'Stackoverflow'
15      }
16    ];
17  });
```

When we click a link, we get redirected to a different page. That worked for us on a previous example, but now we want to “hijack” that click event and instead of a redirect, we just want to log the click. For the sake of the example, instead of using `ng-click` we are going to create a dummy directive here:

```
1 angular.module('app')
2   .directive('myClick', function() {
3     return {
4       link: function(scope, element, attrs) {
5         element.on('click', function(e) {
6           e.preventDefault();
7           console.log('You are going to visit', scope.page.name);
8         });
9       }
10    };
11  });
```

And we use it on the `<li>` element:

```
1 <ul>
2   <li my-click at-repeat="page in pages">
3     <a href="{{page.url}}">{{page.name}}</a>
4   </li>
5 </ul>
```

As you learned, if we use it on the same element as the `at-repeat` and since it creates a new scope, our `my-click` directive will have access to the `page` object. Alright, we click on an item and the `click` event of our directive will run preventing the redirection and logging. Are we completely sure that it works? [Try it<sup>3</sup>](#). Uhm, it doesn't and I checked the code a couple of times and I don't see anything weird.

Let's debug it a bit (this is why I chose a directive, to be able to debug easily). Is it even loaded? Let's log the `link` function:

```
1 link: function(scope, element, attrs) {
2   console.log('I am here');
3   element.on('click', function(e) {
4     e.preventDefault();
5     console.log('You are going to visit', scope.page.name);
6   });
7 }
```

I can see that log so definitely the directive is being processed by Angular. Why is our click event not working? Let's go step by step:

- Angular see both `my-click` and `at-repeat` on the HTML.
- After being ordered, `at-repeat` will run first.
- Since `at-repeat` has `transclude: 'element'`, the element itself is going to be replaced with a comment.
- It runs `at-repeat` compile. There is no compile on `my-click`.
- There is no `pre-link` on any of those.
- Next step is `post-link` which will execute `my-click`'s one (hence the log we see). Also it will bind the `click` event to the comment node.
- Lastly it will execute `at-repeat`'s `post-link` which will create a new element for each item on the collection.

So what happens here? We instantiated `my-click` and then `at-repeat` did its magic. What we would need here is: let `at-repeat` do his magic and then apply `my-click` after that. Does that sound like priorities? Yes it does. The idea is not exactly as we learned on the previous section though. If we use

---

<sup>3</sup><http://plnkr.co/edit/JXGoBtpDchSkaPpkh94h?p=preview>



transclude: 'element' in one of our directives, we need to use priority as well. How so? First, we would like to run our at-repeat first (so no other directive runs on the dummy comment node). On the other hand, when angular sees a directive with transclude: 'element' and a priority, it will set a flag with that priority so when angular processes the other directives, and before doing anything, it checks the priority flag and if the current directive that is processing has less priority, angular will skip it.

So setting a priority on our at-repeat will prevent other directives with less priority on the element from running and then, when we use the transcludeFn angular will then process them.

Let's do it:

```
1 angular.module('app')
2   .directive('atRepeat', function() {
3     return {
4       restrict: 'A',
5       transclude: 'element',
6       priority: 1000,
7       compile: function(tElement, tAttrs) {
8         var expression = tAttrs.atRepeat;
9         var match = expression.match(/^s*(.+)s+in\s+(.*)s*$/);
10
11         if (!match) {
12           throw new Error(
13             "Expected expression in form of '_item_ in _collection_'
14           );
15         }
16
17         var iterationItemExpr = match[1];
18         var collectionExpr = match[2];
19         var items = [];
20
21         return function(scope, element, attrs, ctrl, transclude) {
22           scope.$watchCollection(collectionExpr, function(collection) {
23             var i, block, childScope;
24
25             // Delete all items first
26             if (items.length > 0) {
27               for (i = 0, l = items.length; i < l; i++) {
28                 items[i].el.remove();
29                 items[i].scope.$destroy();
30               }
31               items = [];
32             }
```

```

33
34     for(i = 0, l = collection.length; i < l; i++) {
35         childScope = scope.$new();
36         childScope[iterationItemExpr] = collection[i];
37
38         transclude(childScope, function(clone, cloneScope) {
39             element.parent().append(clone);
40             block = {};
41             block.el = clone;
42             block.scope = cloneScope;
43             items.push(block);
44         });
45     }
46     });
47 };
48 }
49 };
50 });

```

We set a priority of 1000 (which is the priority that the original `ng-repeat` has). Why 1000? We leave a big margin between the default 0 and 1000 so the other directives on the element can play with priorities when they are processed later.

If we try again, we can see how our `myClick` directive is logged three times (3 items in our collection) and if we click on an URL, we are “hijacking” it correctly.

Check it [here](#)<sup>4</sup>

## Summary

If we ever need to run a directive before / after another directive, we can’t simply rely on the alphabetical order. There is where priorities comes in. The default priority is 0 but we can change it to negative and positive values.

On the other hand, if you’re playing with `transclude: 'element'`, make sure you set a priority on the directive so the internal workings of that type of transclusion does its job. If you’re curious enough, check the core directives and you will see how a bunch of them have priorities set so they can play with `ng-repeat` and `ng-include` correctly.

---

<sup>4</sup><http://plnkr.co/edit/lAdO9omMHksPbd9pSWir?p=preview>

# 17. Terminal

So as we learnt in the previous chapter, angular will grab all the directives on the element and then sort them by priority. That is really handy in some use cases, but is there a way to stop processing directives when a concrete priority is met? Sure.

There is another property on the DDO called `terminal`. Terminal will make angular to stop processing any further directive but there are rules and exceptions. If a directive with priority 500 is set as terminal, angular will still process any other directive with priority 500 and then stop. That also includes processing directives on children element or directives on the directive's template.

Examples:

```
1 angular.module('app')
2   .directive('foo', function() {
3     return {
4
5     }
6   })
7
8   .directive('bar', function() {
9     return {
10       terminal: true
11     }
12   });
```

```
1 <div bar foo></div>
```

bar is terminal, will foo run? Yes, they have the same priority so angular will run both.

```
1 angular.module('app')
2   .directive('foo', function() {
3     return {
4       priority: 1
5     }
6   })
7
8   .directive('bar', function() {
9     return {
10      terminal: true
11    }
12  });
```

```
1 <div bar foo></div>
```

Here? Same, foo has more priority so it runs first and then bar.

```
1 angular.module('app')
2   .directive('foo', function() {
3     return {
4
5     }
6   })
7
8   .directive('bar', function() {
9     return {
10      terminal: true,
11      priority: 1
12    }
13  });
```

```
1 <div bar foo></div>
```

Here just bar, it has more priority than foo and it is also terminal.

```
1 angular.module('app')
2   .directive('foo', function() {
3     return {
4       priority: 1
5     }
6   })
7
8   .directive('bar', function() {
9     return {
10      terminal: true
11    }
12  });
```

```
1 <div bar>
2   <div foo></div>
3 </div>
```

foo won't run. Having more priority only affects the same element and foo is not in the same but a children one and as we said, priority also stop children element to stop processing directives.

```
1 angular.module('app')
2   .directive('foo', function() {
3     return {
4
5     }
6   })
7
8   .directive('bar', function() {
9     return {
10      template: '<div foo></div>'
11      terminal: true
12    }
13  });
```

```
1 <div bar></div>
```

terminal also stop template's directives to stop processing.

As an example, if you have a blog and you decide one day that you want to write an article about Angular and also make it interactive, you would need to make that post an Angular app itself. Ok, that sounds nice, what's the problem? You can't simply put any code snippet on the article. Uhm? How so? Since the article itself is an Angular app, it will treat those snippets as app code instead of simply snippets.

Let's create a directive to tell angular: Hey, ignore all this markup, don't process it.

```
1 angular.module('app', [])
2   .directive('dontBind', function() {
3     return {
4       priority: 1001,
5       terminal: true
6     };
7   });
```

With priority 1001 and terminal set to true, we will prevent any directive with priority 1000 or less to be processed.



Angular doesn't have any built-in directive with more than priority 1000.

So if we use this directive like:

```
1 <p dont-bind>What I put in here won't be processed by {{angular}}</p>
```

We will see:

What I put in here won't be processed by {{angular}}

Check it [here](#)<sup>1</sup>



Angular has this directive built-in under the name: ngNonBindable.

## The tests

Testing this is almost as easy as the directive itself:

---

<sup>1</sup><http://plnkr.co/edit/7MNoBkDLRusdTciTJoeB?p=preview>

```
1 describe('Directive: dontBind', function() {
2   beforeEach(module('app'));
3
4   it('won\'t bind anything on the element',
5     inject(function($rootScope, $compile) {
6       var element = angular.element(
7         '<div dont-bind foo="{{a}}">This is {{b}}</div>'
8       );
9       $rootScope.a = 'a';
10      $rootScope.b = 'b';
11
12      $compile(element)($rootScope);
13      $rootScope.$digest();
14
15      expect(element.attr('foo')).toEqual('{{a}}');
16      expect(element.text()).toContain('{{b}}');
17    }));
18 });
```

We create an element and we check that both the attribute `foo` and the interpolation inside is not being processed by Angular.

Check the tests [here](#)<sup>2</sup>

## Summary

If we ever need to stop processing directives in a certain point, we can use `terminal` for that. Just be sure you know all the rules and exceptions stated in this chapter.

---

<sup>2</sup><http://plnkr.co/edit/zs8QMrL1bpap3bOhZWOX?p=preview>

# 18. Require

A directive works really good by its own, but what if we need multiple directives to achieve some goal? For example, if we need to build a menu, we would like to do something like:

```
1 <menu>
2   <menu-item>Menu 1</menu-item>
3   <menu-item>Menu 2</menu-item>
4   <menu-item>Menu 3</menu-item>
5 </menu>
```

Or an accordion:

```
1 <accordion>
2   <accordion-group>Foo</accordion-group>
3   <accordion-group>Bar</accordion-group>
4   <accordion-group>Baz</accordion-group>
5 </accordion>
```

Sure, it looks good on my HTML, but is there a real need? Can't I use something like:

```
1 <div>
2   <accordion>Foo</accordion>
3   <accordion>Bar</accordion>
4   <accordion>Baz</accordion>
5 </div>
```

Let's think for a second about the "main" feature of an accordion... What happens when you click on a group's header? The other groups collapse. How can you make the other accordion groups to collapse? They are not related to each other so there is no way to tell the others to collapse (other than using pub/sub which is not a good idea).

In this case, you can create a parent directive that will be able to manage all the groups for you.

Let's explain the technical details first. In Angular you can require a directive in your own directive. Wait what? What do you mean with "You can require a directive"? Let's put an example:



```
1 angular.module('app')
2   .directive('foo', function() {
3     return {
4       template: '<div>foo</div>',
5       controller: function($scope) {
6         // stuff
7       },
8       link: function(scope) {
9         // more stuff
10      }
11    };
12  })
13
14  .directive('bar', function() {
15    return {
16      require: 'foo',
17      // more properties
18    };
19  });
```

In this example, the directive `bar` is requiring the directive `foo`, but what is `bar` requiring exactly? Everything? Template? Link? Well, when you require a directive, what you get, is the instance of that directive's controller. In other words, now `bar` has access to the controller of `foo`.

How do `bar` access to `foo`'s controller? Through the `link` function:

```
1 link: function(scope, element, attrs, ctrl) {
2   // you can access the required controller from here
3 }
```

The fourth parameter is the controller we required, in our case, it is the `foo` controller.

There is something important to understand here. What we get, is the instance of the controller, that means that we can only access to what we put on `this`. in the controller. For example:

```
1 controller: function($scope) {
2   this.foo = 'foo';
3   $scope.something = 'something';
4 }
```

In this were the `foo` controller and we log it on `bar`, we would get:

```
1 { foo: 'foo' }
```

Just foo and not something.

This is everything we need to know about “what do we require”. Let’s talk about the different combinations of require:

- **require: ‘foo’** -> This require a foo directive to live in the same element as our directive.
- **require: ‘^foo’** -> This require a foo directive to live in the same element or a parent element (it will start checking on the same element).
- **require: ‘^^foo’** -> This require a foo directive on a parent element.
- **require: ‘?foo’** -> This require an **optional** foo directive on the same element.
- **require: ‘?^foo’** -> This require an **optional** foo directive on the same element or a parent element.
- **require: ‘?^^foo’** -> This require an **optional** foo directive on a parent element.



You can do `?^foo` or `^?foo`. Order doesn’t matter.

We can also require more than one directive:

- **require: [‘foo’, ‘^bar’, ‘^^baz’]** -> Any number, different combinations.

When you require more than one directive, the `ctrl` parameter becomes an array. Here is a common convention:

```
1 require: ['foo', 'bar'],
2 link: function(scope, element, attrs, ctrls) {
3     var fooCtrl = ctrls[0],
4         barCtrl = ctrls[1];
5
6     // Do stuff
7 }
```

Lastly, when we have a controller in our directive, you can access it as well:

```
1 controller: function() {},
2 link: function(scope, element, attrs, ctrl) {
3   // ctrl here is our own controller
4 }
```

That is really handy. There is a problem though, if we require another directive, we lose this shortcut to our own controller. We can fix that like:

```
1 angular.module('app')
2   .directive('bar', function() {
3     return {
4       controller: function() {},
5       require: ['bar', 'foo'],
6       link: function(scope, element, attrs, ctrls) {
7         var barCtrl = ctrls[0],
8             fooCtrl = ctrls[1];
9       }
10    };
11  });
```

If we need to require a different directive, we can require our directive to get access to its controller.

For the example, we are going to build that accordion from earlier. Let's define what should it do and how could we structure it:

To create an accordion, we need a bunch of accordion-groups elements which will contain a header and some content. Since we need to close the other groups when we click one, we need those groups to be aware of each other. To achieve that, we need an accordion element wrapping all those groups.

Ok, so far we need an accordion directive and an accordion-group directive. Since we need the accordion directive as a parent to manage the groups, we need to expose some interface to the groups and to do that, we need a controller on the accordion.

Let's start coding the controller:

```
1 angular.module('accordion')
2   .controller('AccordionController', function() {
3     this.groups = [];
4   });
```

When you need a parent directive (accordion) to be able to control its children (accordion-group), you need that parent directive to have a reference of every children. The best reference we can get is the scope of the children. Ok, let's create a method to register new children:

```
1  this.addGroup = function(groupScope) {
2    this.groups.push(groupScope);
3  };
```

This is a method that the children will use to register themselves. Since we can add / remove groups dynamically, we need a way to remove a group from the list. How? We can listen to the `$destroy` event of a scope. Let's modify the `addGroup` method we just added:

```
1  this.addGroup = function(groupScope) {
2    this.groups.push(groupScope);
3
4    groupScope.$on('$destroy', function() {
5      removeGroup(groupScope);
6    });
7  };
```

When we add a new group, we add a listener for the `$destroy` event so we can delete it when needed. Let's code the `removeGroup` function:

```
1  function removeGroup(group) {
2    var index = that.groups.indexOf(group);
3    if (index !== -1) {
4      that.groups.splice(index, 1);
5    }
6  }
```

This function will simply remove the group from the list. Notice that it is not under `this`. so it is private for the children, that is what we need.

Lastly, all this adding / removing should be useful for something right? Yes! We need a way to close all groups except the clicked one:

```
1  this.closeOthers = function(openGroup) {
2    angular.forEach(this.groups, function(group) {
3      if (group !== openGroup) {
4        group.isOpen = false;
5      }
6    });
7  };
```

We simply need to iterate through all our groups and set isOpen to false in all of them except the one we clicked.

Think in all the possibilities we have here! We can access the scope of any children so we can control what they do from the parent, and any children can make the parent do anything for it, even involving other children.

Our controller is done. Let's code the directive:

```
1 angular.module('accordion')
2 .directive('atAccordion', function() {
3   return {
4     controller: 'AccordionController',
5     transclude: true,
6     templateUrl: 'accordion.html'
7   };
8 });
```

```
1 <div class="panel-group" ng-transclude></div>
```

The directive is really simple, it just includes the controller we just created. The transclusion + template is not really necessary, but we need to style our accordion following bootstrap rules so we need all our groups inside a div with the panel-group class.

Next, the accordion-group directive:

```
1 angular.module('accordion')
2 .directive('atAccordionGroup', function() {
3   return {
4     require: '^atAccordion',
5     transclude: true,
6     templateUrl: 'accordion-group.html',
7     scope: {
8       heading: '@'
9     },
10    link: function(scope, element, attrs, accordionCtrl) {
11      accordionCtrl.addGroup(scope);
12
13      scope.isOpen = false;
14
15      scope.$watch('isOpen', function(value) {
16        if (value) {
17          accordionCtrl.closeOthers(scope);
```

```

18         }
19     });
20 }
21 };
22 });

```

```

1 <div class="panel panel-default">
2   <div class="panel-heading">
3     <h4 class="panel-title">
4       <a href class="accordion-toggle"
5         ng-click="isOpen = !isOpen">{{heading}}</a>
6     </h4>
7   </div>
8   <div class="panel-collapse" ng-if="isOpen">
9     <div class="panel-body" ng-transclude></div> </div>
10 </div>

```

To make it work, we put `require: '^atAccordion'`, in other words, we require an `atAccordion` directive to live a parent element. The interesting part goes to the `link` function. When we register a new `atAccordionGroup` we call the `atAccordion`'s controller and we register ourselves and when `isOpen` changes, we trigger the `closeOthers` on the controller.

You need to code 3 components to make an accordion to work, but at the end it is really simple. We click a header, that triggers the `$watch` and that will tell the controller to close the others groups. We can add more stuff to this accordion, like an optional `closeOthers` or being able to bind `isOpen` from outside, but since both needs to be optional, we need to wait to the next part of the book.

Let's try it:

```

1 <at-accordion>
2   <at-accordion-group heading="Heading 1">
3     Content of group 1
4   </at-accordion-group>
5   <at-accordion-group heading="{{custom}}">
6     Content of group 2
7   </at-accordion-group>
8   <at-accordion-group
9     heading="{{ 'Custom ' + item }}" ng-repeat="item in items">
10     Content of custom {{item}}
11   </at-accordion-group>
12 </at-accordion>

```

Notice how we used `ng-repeat` to create a dynamic groups.

Heading 1
<u>Custom Heading</u>
Content of group 2
Custom 1
Custom 2
Custom 3

See it working [here](#)<sup>1</sup>

## The tests

Thanks to having a separate controller, testing this is really easy. Let's start with the controllers tests:

```
1 describe('Directive: atAccordion', function() {
2
3   beforeEach(module('accordion'));
4
5   describe('controller', function() {
6     var ctrl, $rootScope;
7
8     beforeEach(inject(function(_$rootScope_, $controller) {
9       $rootScope = _$rootScope_;
10      ctrl = $controller('AccordionController');
11    }));
12
13    it('can add new group items', function() {
14      var group1 = $rootScope.$new();
15      var group2 = $rootScope.$new();
16      ctrl.addGroup(group1);
17      ctrl.addGroup(group2);
18      expect(ctrl.groups.length).toBe(2);
19      expect(ctrl.groups[0]).toBe(group1);
20      expect(ctrl.groups[1]).toBe(group2);
21    });
22  });
23 })
```

---

<sup>1</sup><http://plnkr.co/edit/p1uHCbsm6VtffO5knHB0?p=preview>

```
21     });
22
23     it('closes the rest of groups when you open one', function() {
24         var group1 = $rootScope.$new();
25         var group2 = $rootScope.$new();
26         var group3 = $rootScope.$new();
27
28         ctrl.addGroup(group1);
29         ctrl.addGroup(group2);
30         ctrl.addGroup(group3);
31
32         group1.isOpen = group2.isOpen = group3.isOpen = true;
33
34         ctrl.closeOthers(group2);
35
36         expect(group1.isOpen).toBe(false);
37         expect(group2.isOpen).toBe(true);
38         expect(group3.isOpen).toBe(false);
39     });
40
41     it('removes a group if you delete it', function() {
42         var group1 = $rootScope.$new();
43         var group2 = $rootScope.$new();
44         ctrl.addGroup(group1);
45         ctrl.addGroup(group2);
46
47         group1.$destroy();
48         expect(ctrl.groups.length).toBe(1);
49     });
50 });
51 });
```

We don't care about parent directives, children directives... We just need scopes. For the first test, we create two dummy groups (again, just scopes), we add them and we assert that they are registered successfully. The second test adds a bunch of groups and we assert that `closeOthers` close all groups but the one we send. Third test asserts that groups are deleted from our list.

Let's test the `atAccordionGroup` directive (put the code inside the parent `describe` block):



```
1 describe('atAccordionGroup', function() {
2   var element, $rootScope;
3
4   beforeEach(inject(function(_$rootScope_, $compile) {
5     $rootScope = _$rootScope_;
6     element = angular.element(
7       '<at-accordion>' +
8       '<at-accordion-group heading="Header 1">' +
9       'Content 1' +
10      '</at-accordion-group>' +
11      '<at-accordion-group heading="Header 2">' +
12      'Content 2' +
13      '</at-accordion-group>' +
14      '</at-accordion>'
15    );
16
17    $compile(element)($rootScope);
18    $rootScope.$digest();
19  }));
20
21  it('creates one header per group', function() {
22    var headers = element.find('.panel-heading');
23
24    var header1 = headers.eq(0).find('.accordion-toggle').text();
25    var header2 = headers.eq(1).find('.accordion-toggle').text();
26
27    expect(headers.length).toBe(2);
28    expect(header1).toBe('Header 1');
29    expect(header2).toBe('Header 2');
30  });
31
32  it('toggles groups on click', function() {
33    var headers = element.find('.panel-heading');
34    var firstHeader = headers.eq(0).find('a');
35    var secondHeader = headers.eq(1).find('a');
36
37    firstHeader.click();
38
39    expect(firstHeader.scope().isOpen).toBe(true);
40    expect(secondHeader.scope().isOpen).toBe(false);
41
42    secondHeader.click();
```

```
43
44     expect(firstHeader.scope().isOpen).toBe(false);
45     expect(secondHeader.scope().isOpen).toBe(true);
46   });
47 });
```

We create a dummy element with a bunch of groups and for the first test, we expect the groups to be there in the DOM with their headers. For the second test, we simulate a click on a group to verify that the others groups are no longer opened.

There is no need to test `atAccordion`.

Check the tests [here](#)<sup>2</sup>

## Summary

When you need two related directives to communicate, you will find `require` to be really helpful. Having a controller as a middle ground between parent and children is really handy. There you can define methods and properties that can be used from both sides.

---

<sup>2</sup><http://plnkr.co/edit/WaopDBidXXdGuXonplj?p=preview>

## 19. Multi Element

Directives spans one element so if we put a directive on a `div` element, it will only affect that `div` element and its children. That is usually fine for most directives, but some others directives are more useful if they could span more elements. Let's put an example:

```
1 <p>
2   This is paragraph one.
3 </p>
4 <p>
5   Here I put more stuff I want to hide.
6 </p>
```

I have this two `p` elements and I want to show them only under certain circumstances, what can I do? Maybe:

```
1 <p ng-show="foo">
2   This is paragraph one.
3 </p>
4 <p ng-show="foo">
5   Here I put more stuff I want to hide.
6 </p>
```

That would work, but repeating code is never good. Alright, what about:

```
1 <div ng-show="foo">
2   <p>
3     This is paragraph one.
4   </p>
5   <p>
6     Here I put more stuff I want to hide.
7   </p>
8 </div>
```

That also works, but your designer will blame you if you start adding extra elements on the template. Angular fixed this issue for us. Let's code our own `ng-show` and then we make it work with multiple elements:

```

1 angular.module('app')
2   .directive('atShow', function($animate) {
3     return {
4       restrict: 'A',
5       link: function(scope, element, attrs) {
6         scope.$watch(attrs.atShow, function atShowWatch(value) {
7           element[value ? 'removeClass' : 'addClass']('ng-hide');
8         });
9       }
10    };
11  });

```

Yes, is that simple. What we need to do is to watch the attribute `atShow` for changes. When it is false, we call `element.removeClass` to remove the `ng-hide` class, on the other hand, if it is true, we call `element.addClass` to add that class.



Angular injects a bit of css in our application to hide the elements with the `ng-hide` class.

In fact, we could add animation support for this directive, change the `$watch` like:

```

1 scope.$watch(attrs.atShow, function atShowWatch(value) {
2   $animate[value ? 'removeClass' : 'addClass'](element, 'ng-hide');
3 });

```

This allow animation when adding / removing the class. Also, don't forget to inject `$animate` on the directive.

Try it:

```

1 <p at-show="aFalse">
2   This won't show
3 </p>

```

And if you have `aFalse` in your controller's `$scope` set to false, it won't show.

Good, how can we make this support multiple elements? Easy as pie:

```
1 angular.module('app')
2   .directive('atShow', function($animate) {
3     return {
4       restrict: 'A',
5       multiElement: true,
6       link: function(scope, element, attrs) {
7         scope.$watch(attrs.atShow, function atShowWatch(value) {
8           $animate[value ? 'removeClass' : 'addClass'](element, 'ng-hide');
9         });
10      }
11    };
12  });
```

One line, just that. With `multiElement` we tell angular that our directive can potentially span more than one element. How does this work on the html? To use multi element directives, we append `-start` on the first element and `-end` on the last element. So our first example would be like:

```
1 <p at-show-start="aFalse">
2   This is paragraph one.
3 </p>
4 <p at-show-end>
5   Here I put more stuff I want to hide.
6 </p>
```

We can have as many elements as we want between those two, when angular finds a `-start` directive, it will grab all the elements in between until it finds the `-end` one. If it doesn't exist, Angular will throw an exception.

Try it [here](http://plnkr.co/edit/hEvYEOx5kV2SGHs4YGHc?p=preview)<sup>1</sup>

## The tests

Testing this directive is easy, but we will make this easier by creating some custom jasmine's matchers:

---

<sup>1</sup><http://plnkr.co/edit/hEvYEOx5kV2SGHs4YGHc?p=preview>

```
1  function isNgElementHidden(element) {
2    var hidden = true;
3    angular.forEach(angular.element(element), function(element) {
4      if ((' ' + (element.getAttribute('class') || '') + ' ')
5          .indexOf(' ng-hide ') === -1) {
6        hidden = false;
7      }
8    });
9    return hidden;
10 }
11
12 var customMatchers = {
13   toBeShown: function(util, customEqualityTesters) {
14     return {
15       compare: function(actual, expected) {
16         var result = {
17           pass: util.equals(
18             !isNgElementHidden(actual),
19             true,
20             customEqualityTesters
21           )
22         };
23
24         if (!result.pass) {
25           result.message = 'The element is not being shown';
26         }
27
28         return result;
29       }
30     };
31   },
32
33   toBeHidden: function(util, customEqualityTesters) {
34     return {
35       compare: function(actual, expected) {
36         var result = {
37           pass: util.equals(
38             isNgElementHidden(actual),
39             true,
40             customEqualityTesters
41           )
42         };
43       }
44     };
45   }
46 };
```

```
43
44     if (!result.pass) {
45         result.message = 'The element is not being hidden';
46     }
47
48     return result;
49 }
50 };
51 }
52 };
```

First we created a helper function which will return a boolean whether an element is hidden or not. We also created two matchers, one to check if an element is shown and one to check if it is hidden.

With that, we can start the fun:

```
1 describe('Directive: atShow', function() {
2     var scope, $compile;
3
4     beforeEach(module('app'));
5
6     beforeEach(inject(function(_$compile_, $rootScope) {
7         scope = $rootScope.$new();
8         $compile = _$compile_;
9     }));
10
11     beforeEach(function() {
12         jasmine.addMatchers(customMatchers);
13     });
14 });
```

As always, load our module, inject some needed stuff and in this case, load those matchers as well. For the tests, just two:

```
1  it('should show and hide an element', function() {
2    var element = angular.element('<div ng-show="exp"></div>');
3    $compile(element)(scope);
4    scope.$digest();
5    expect(element).toBeHidden();
6    scope.exp = true;
7    scope.$digest();
8    expect(element).toBeShown();
9  });
10
11 it('should make hidden element visible', function() {
12   var element =
13     angular.element('<div ng-class="ng-hide" ng-show="exp"></div>');
14   $compile(element)(scope);
15   scope.$digest();
16   expect(element).toBeHidden();
17   scope.exp = true;
18   scope.$digest();
19   expect(element).toBeShown();
20 });
```

On the first one, we verify that the element is hidden on a undefined parameter, and shown when we set it to true. On the second one, if we have an element with ng-hide on it, ng-show with a truthy value will show it again.

I love custom matchers, they make the tests a breeze.

If you're brave enough, you can implement ngHide by yourself, it is not any hard!

Check the tests [here](#)<sup>2</sup>

## Summary

Making a directive work with multiple elements is a breeze, we just need to set multiElement to true and the element will magically accept a -start and -end suffix to work with multiple elements. We don't have to upset our designers anymore!

---

<sup>2</sup><http://plnkr.co/edit/MYVPZ8XJ5kkSAUPHUTMi?p=preview>



# V Advanced usages

With the DDO under your belt, it is time to learn some advanced use cases with directives.

## 20. Observing attributes

On a previous chapter, we learnt that the link function's third parameter is an object which contains all our attributes. In fact we can do stuff like:

```
1 <directive foo="foo"></directive>
```

And then:

```
1 link: function(scope, element, attrs) {  
2   console.log(attrs.foo); // "foo"  
3 }
```

We can also interpolate a value from the controller:

```
1 <directive foo="{{something}}"></directive>
```

```
1 link: function(scope, element, attrs) {  
2   var foo = $interpolate(attrs.foo)(scope);  
3   console.log(foo); // value of "something" on the controller  
4 }
```

We can pass a simple string to the attribute or interpolate a value coming from the controller's scope.



We will learn more about `$interpolate` and more advanced attributes values on the next chapter.

But now I am wondering... What happens if something in this latter case, changes? By default, nothing, but if we interpolate the value again, we will grab its new value. Could we be notified when something changes? So we don't need to manually check it. Sure, we can do that.

That third parameter, which we usually call `attrs` contains several methods, and one of those methods is called `$observe`. `$observe` is much like `$watch` but with some differences. Let's see its usage first:

```

1 link: function(scope, element, attrs) {
2   attrs.$observe('foo', function(newFoo) {
3     console.log(newFoo);
4   });
5 }

```

`$observe` receives the attribute name and a callback function. The callback function is going to be called every time the `foo` attribute changes, that means that when we change something in our controller, that `$observe` function will fire.

Like `$watch`, the `$observe`'s callback function will be fired once when registered (in concrete, after the next `$digest`) but unlike `$watch`, the observed attribute won't be checked in every `$digest` cycle but only when the attribute changes.

When we talk about this, the '@' on the isolate scope comes to our mind and in fact, this is how it is implemented:

```

1 case '@':
2   attrs.$observe(attrName, function(value) {
3     destination[scopeName] = value;
4   });
5   attrs.$$observers[attrName].$$scope = scope;
6   if (attrs[attrName]) {
7     destination[scopeName] = $interpolate(attrs[attrName])(scope);
8   }
9   break;

```

As you can see, Angular is internally creating an `$observe` for our '@' attribute, and it is updating the directive's scope with the new value. It is also assigning the value immediately to the scope and that is because the `$observe` method won't run until the next `$digest`.

Do you remember the directive we coded back on the chapter 10? The one that draw canvas circles. Let's revisit it again:

```

1 angular.directive('svgCircle', function() {
2   return {
3     restrict: 'E',
4     scope: {
5       size: "@",
6       stroke: "@",
7       fill: "@"
8     },
9     templateUrl: 'circle.html',

```

```

10     link: function(scope, element, attrs) {
11         var size = parseInt(scope.size, 10);
12
13         var canvasSize = size * 2.5;
14
15         scope.values = {
16             canvas: canvasSize,
17             radius: size,
18             center: canvasSize / 2
19         };
20     }
21 };
22 });

```

And the template:

```

1  <svg ng-attr-height="{{values.canvas}}"
2      ng-attr-width="{{values.canvas}}"
3      class="gray">
4      <circle ng-attr-cx="{{values.center}}"
5              ng-attr-cy="{{values.center}}"
6              ng-attr-r="{{values.radius}}"
7              stroke="{{stroke}}"
8              stroke-width="3"
9              fill="{{fill}}" />
10 </svg>

```

Let's observe the size attribute so we can dynamically change the size of our circles. The first thing we need to do is to wrap the algorithm in a function:

```

1  link: function(scope, element, attrs) {
2      var calculateValues = function(size) {
3          var canvasSize = size * 2.5;
4
5          scope.values = {
6              canvas: canvasSize,
7              radius: size,
8              center: canvasSize / 2
9          };
10     };
11 }

```

Now, we just need to observe the size attribute and call this method passing the new size:

```
1 link: function(scope, element, attrs) {
2   var calculateValues = function(size) {
3     var canvasSize = size * 2.5;
4
5     scope.values = {
6       canvas: canvasSize,
7       radius: size,
8       center: canvasSize / 2
9     };
10  };
11
12  attrs.$observe('size', function(newSize) {
13    calculateValues(parseInt(newSize, 10));
14  });
15 }
```

The `$observe`'s callback function will run once at the beginning, which will make the circle appear and then if the size changes, it will recompute the circle's values and change it.

As an extra, you can remove the size key on the scope DDO object:

```
1 scope: {
2   stroke: "@",
3   fill: "@"
4 }
```

This is not really needed and you can say that having it there helps “making your intentions clear”, so it is more up to you.

Check the directive [here](#)<sup>1</sup>

## The tests

Create a new plunker for the tests, grab the tests from the previous example and modify the `beforeEach` block to use this one:

---

<sup>1</sup><http://plnkr.co/edit/Mptx5luG7s29xTaAxJSQ?p=preview>

```
1 beforeEach(inject(function($rootScope, $compile) {
2   scope = $rootScope.$new();
3
4   element =
5     '<svg-circle size="{{size}}" stroke="black" fill="blue"></svg-circle>';
6
7   scope.size = 100;
8
9   element = $compile(element)(scope);
10  scope.$digest();
11 }));
```

Then for the tests themselves, we still need them (we need to be sure that the old behavior is still working). You could wrap them into a describe block like:

```
1 describe('with the first given value', function() {
2   // The three old tests moved here
3 });
```

For the new tests, let's create a new describe block with the tests inside:

```
1 describe('when changing the initial value to a different one', function() {
2
3   beforeEach(function() {
4     scope.size = 160;
5     scope.$digest();
6   });
7
8   it("should compute the size to create other values", function() {
9     var isolated = element.isolateScope();
10    expect(isolated.values.canvas).toBe(400);
11    expect(isolated.values.center).toBe(200);
12    expect(isolated.values.radius).toBe(160);
13  });
14
15  it("should contain a svg tag with proper size", function() {
16    expect(element.find('svg').attr('height')).toBe('400');
17    expect(element.find('svg').attr('width')).toBe('400');
18  });
19
20  it("should contain a circle with proper attributes", function() {
21    expect(element.find('circle').attr('cx')).toBe('200');
```

```
22     expect(element.find('circle').attr('cy')).toBe('200');
23     expect(element.find('circle').attr('r')).toBe('160');
24     expect(element.find('circle').attr('stroke')).toBe('black');
25     expect(element.find('circle').attr('fill')).toBe('blue');
26   });
27 });
```

Before each test, we are going to set a new size, then we assert that the algorithm computes the new values and that our circle is updated with its new values.

Check the tests [here](#)<sup>2</sup>

## Summary

When we interpolate a value from a controller's scope, we can also be aware when it changes and then run some logic based on that. `$observe` is much like `$watch`, but unlike the latter, `$observe` doesn't run on every `$digest` cycle but only when the attribute changes.

---

<sup>2</sup><http://plnkr.co/edit/tirhLwFEXLKSzukbsW1q?p=preview>

# 21. Optional attributes

When designing directives, we also need to think about what attributes we need to pass information to our directive. So far, we assumed that all the attributes were required. In the real world, that is not always the case.

Optional attributes are straightforward to implement, but there are different types of parameters which have their own set of rules.

Let's start with `=`. This one has a built-in support for optional attributes:

```
1 scope: {  
2   foo: '=?'  
3 }
```

If we mark it with a `?`, we make it optional. Making it optional allows Angular to skip all the logic behind `=` (which involves creating a `$watch`).

For `&` we have the same built-in support:

```
1 scope: {  
2   callback: '&?'  
3 }
```

In this case, Angular will skip the parsing of the attribute if it is not given. In any case (optional callback or not) I like to guard my code like:

```
1 if (scope.callback) {  
2   scope.callback();  
3 }
```

For `@` on the other hand, depending of what we expect on the attribute, we need to code different solutions.

Let's code an example. Imagine we want a directive to create fancy texts where we can have a `<div>` in pink or even in blue. This directive will have three optional `@` attributes:

- **foreground:** We use this attribute to define the font color of the div.
- **value:** Instead of using transclusion, we are going to pass the text in an attribute.



- **enabled:** We can enable or disable the font color.

This directive is over complicated but we need to see the different ways we have for optional @ attributes.

When using optional @ attributes, we can provide some default values. To do that, we can create a constant service:

```
1 angular.module('app')
2   .constant('fancyTextConfig', {
3     foreground: 'blue',
4     value: 'Default text',
5     enabled: 'true'
6   });
```

Here we defined that we want by default a blue “Default text”. Now let’s create our directive skeleton:

```
1 angular.module('app')
2   .directive('fancyText', function(fancyTextConfig, $interpolate) {
3     return {
4       restrict: 'E',
5       scope: {
6       },
7       template: '<div>{{value}}</div>',
8       link: function(scope, element, attrs) {
9
10      }
11    };
12  });
```

Let’s talk about the foreground. How do we expect people using this attribute? As a string, something along the lines: red, blue or even #BADA55. When we expect our attribute to use strings, we use \$interpolate that attribute like this:

```
1 var foreground = angular.isDefined(attrs.foreground) ?
2     $interpolate(attrs.foreground)(scope.$parent) :
3     fancyTextConfig.foreground;
```

If the attribute is defined, we interpolate its value, if not, we grab the default one. \$interpolate receives what we want to interpolate and returns the interpolation function which we call passing the parent scope. Why the parent scope? Because we need to compute that interpolation using the correct scope, and the attributes has the controller’s scope, not the isolated scope. Thanks to this we can do things like:

```

1 <fancy-text foreground="red"></fancy-text>
2 <fancy-text foreground="#000000"></fancy-text>
3 <fancy-text foreground="{somethingInScope}"></fancy-text>

```

What about the `enabled` attribute? This one is not exactly what we expect from `@`, but it is an optional attribute after all. Two things to consider for this one. First, we expect users to use this attribute in a more dynamic way, so instead of forcing them to use interpolation, we can `$eval` the attribute directly. On the other hand, we want to allow users to be able to put `true` or `false` and treat that as a boolean.

```

1 var enabled = angular.isDefined(attrs.enabled) ?
2     scope.$parent.$eval(attrs.enabled) :
3     fancyTextConfig.enabled;

```

So instead of using `$interpolate` we are using `$eval` which will evaluate the attribute against the desired scope. This allows us to use the attribute like this:

```

1 <fancy-text enabled="false"></fancy-text>
2 <fancy-text enabled="somethingInScope"></fancy-text>

```

So the rule of thumb here is: If we expect mostly strings on the attribute, use `$interpolate`, if on the other hand we expect boolean values, use `$eval`.



The reason to use `$eval` on booleans is that `$interpolate` returns a string and we don't need `"true"` but `true`.

For `value` we will do things differently. We want it to behave as a normal `@` attribute but still make it optional. First let's add it to the scope object:

```

1 scope: {
2   value: '@'
3 }

```

Then, we just need to check if there is a value and if not, we put the default one:

```

1 if (angular.isUndefined(attrs.value)) {
2   attrs.value = fancyTextConfig.value;
3 }

```

Why set the value on `attrs` instead of the scope? Because on the next `$digest`, that scope will be overwritten with the value of the attribute, AKA nothing.

You can use it like:



```

17
18     var enabled = angular.isDefined(attrs.enabled) ?
19         scope.$parent.$eval(attrs.enabled) :
20         fancyTextConfig.enabled;
21
22     if (enabled) {
23         var css = {
24             'color': foreground
25         };
26         element.css(css);
27     }
28 }
29 };
30 });

```

Pink message

Not fancy at all

Hello readers

Change last text:

Try it [here](#)<sup>1</sup>

## The tests

The tests for this directive are kinda long, but we want to make sure that all the different options works for us.

```

1 describe('directive: fancy-text', function() {
2     var element, scope, $compile;
3
4     beforeEach(module('app'));
5
6     beforeEach(inject(function($rootScope, _$compile_) {
7         scope = $rootScope.$new();
8         $compile = _$compile_;
9     }));
10 });

```

That works as a skeleton, now we want a describe blog per attribute. Let's start with the foreground:

<sup>1</sup><http://plnkr.co/edit/0xOGkqC3bO8Hc8GLyYgc?p=preview>

```
1 describe('foreground', function() {
2   it('should show a blue text', function() {
3     element = angular.element('<fancy-text></fancy-text>');
4     $compile(element)(scope);
5     scope.$digest();
6
7     expect(element.css('color')).toBe('blue');
8   });
9
10  it('should be able to change the text color passing a string', function() {
11    element = angular.element('<fancy-text foreground="red"></fancy-text>');
12    $compile(element)(scope);
13    scope.$digest();
14
15    expect(element.css('color')).toBe('red');
16  });
17
18  it('should change the color passing an interpolated string', function() {
19    element = angular.element(
20      '<fancy-text foreground="{{foreground}}"></fancy-text>');
21    scope.foreground = "green";
22    $compile(element)(scope);
23    scope.$digest();
24
25    expect(element.css('color')).toBe('green');
26  });
27
28  it('should not change if the scope changes', function() {
29    element = angular.element(
30      '<fancy-text foreground="{{foreground}}"></fancy-text>');
31    scope.foreground = "green";
32    $compile(element)(scope);
33    scope.$digest();
34
35    expect(element.css('color')).toBe('green');
36
37    scope.foreground = "red";
38    scope.$digest();
39    expect(element.css('color')).toBe('green');
40  });
41 });
```

To test all the cases, we need to create a new element per test. We do the same for enabled:

```
1 describe('enabled', function() {
2   it('should enable the custom font color by default', function() {
3     element = angular.element('<fancy-text></fancy-text>');
4     $compile(element)(scope);
5     scope.$digest();
6
7     expect(element.css('color')).toBe('blue');
8   });
9
10  it('should disable the custom font color if enabled is false', function() {
11    element = angular.element('<fancy-text enabled="false"></fancy-text>');
12    $compile(element)(scope);
13    scope.$digest();
14
15    expect(element.css('color')).toBe('');
16  });
17
18  it('should be able to a bool from the scope', function() {
19    element = angular.element(
20      '<fancy-text enabled="enabled"></fancy-text>');
21    scope.enabled = false;
22    $compile(element)(scope);
23    scope.$digest();
24
25    expect(element.css('color')).toBe('');
26  });
27
28  it('should not change if the scope changes', function() {
29    element = angular.element(
30      '<fancy-text enabled="enabled"></fancy-text>');
31    scope.enabled = false;
32    $compile(element)(scope);
33    scope.$digest();
34
35    expect(element.css('color')).toBe('');
36
37    scope.enabled = true;
38    scope.$digest();
39    expect(element.css('color')).toBe('');
40  });
41 });
```

And finally for value:

```
1 describe('value', function() {
2   it('should say `Default text` if no value is provided', function() {
3     element = angular.element('<fancy-text></fancy-text>');
4     $compile(element)(scope);
5     scope.$digest();
6
7     expect(element.text()).toBe('Default text');
8   });
9
10  it('should accept a value attribute', function() {
11    element = angular.element(
12      '<fancy-text value="Hello readers"></fancy-text>');
13    $compile(element)(scope);
14    scope.$digest();
15
16    expect(element.text()).toBe('Hello readers');
17  });
18
19  it('should accept an interpolation as value', function() {
20    element = angular.element('<fancy-text value="{{value}}"></fancy-text>');
21    scope.value = "Hello readers";
22    $compile(element)(scope);
23    scope.$digest();
24
25    expect(element.text()).toBe('Hello readers');
26  });
27
28  it('should change if the scope changes', function() {
29    element = angular.element('<fancy-text value="{{value}}"></fancy-text>');
30    scope.value = "Hello readers";
31    $compile(element)(scope);
32    scope.$digest();
33
34    expect(element.text()).toBe('Hello readers');
35
36    scope.value = "Hello again";
37    scope.$digest();
38
39    expect(element.text()).toBe('Hello again');
40  });
41 });
```

Easy tests but there are a lot of them.

Check them [here](#)<sup>2</sup>

## Summary

Optional attributes are practically the bread and butter of directives. They are pretty common in a configurable directive. They are not hard to implement, but the weird API of directives forces you to think in how it is going to be used instead of what you want to achieve.

---

<sup>2</sup><http://plnkr.co/edit/AzXQU5Ul8k0cFyVFWLLA?p=preview>



## 22. Manual \$compile

Back on the lifecycle chapters, we learnt that once we hit the `link` function, all the changes made to the directive's element that involves other directives, would need a manual compilation. This means that if we do something like:

```
1 link: function(scope, element, attrs) {
2   element.append('<div ng-click="foo()">Click here</div>');
3 }
```

It won't work. You can see the actual `Click here` message but the `ng-click` won't do anything. The rule is simple: Every piece of HTML containing directives needs to be compiled by angular. Angular does it for us in some cases (as explained on the lifecycle part) but this one is not the case.

To compile it we need to use the `$compile` service. At this stage of the game, where we used it on every test of the book, it won't be a surprise.

As a recap, `$compile` receives the HTML we want to compile and returns a link function that we should call with a scope. That scope will be the one used by those directives inside the HTML.

Once we have the HTML compiled, we can do with it anything we need. Append, replace, etc.

```
1 link: function(scope, element, attrs) {
2   element.append($compile('<div ng-click="foo()">Click here</div>')(scope));
3 }
```

Now it will work as expected.

For this chapter, we are going to create a directive to show authentication forms. We can choose between `register` and `login` forms. For the sake of learning, we are going to implement this directive twice. The first implementation, done in this chapter, will work as expected but it will be a bit inflexible and in the next chapter, we are going to reimplement it.

Let's start coding the basic skeleton:

```
1 angular.module('app')
2   .directive('authForm', function() {
3     return {
4       restrict: 'E',
5       scope: {
6       },
7       link: function(scope, element, attrs) {
8       }
9     };
10  });
```

What will be the interface of this directive? We need to be able to specify the type of the form we want, where to store the form's result and a optional callback method for the form's submit.

```
1 scope: {
2   type: '@',
3   model: '=',
4   onSubmit: '&'
5 }
```

Now the turn of the logic inside the `link` function. First, we need the templates for the different forms:

```
1 var templates = {};
2
3 templates.register =
4   '<h2>Register form</h2>' +
5   '<form>' +
6     '<input ng-model="model.name" placeholder="username">' +
7     '<input ng-model="model.pass" placeholder="password">' +
8     '<input ng-model="model.pass2" placeholder="Confirm password">' +
9     '<button ng-click="submit()">Submit</button>' +
10  '</form>';
11
12 templates.login =
13   '<h2>Login form</h2>' +
14   '<form>' +
15     '<input ng-model="model.name" placeholder="username">' +
16     '<input ng-model="model.pass" placeholder="password">' +
17     '<button ng-click="submit()">Submit</button>' +
18   '</form>';
```

Hardcoding templates is a pain and those are basic templates... If we start adding some bootstrap styles or additional properties to the elements, this can get hairy pretty soon. We will fix this on the next chapter.



We can move the templates to a service so we can add more of them without changing the directive. Still, we are going to fix this in the next chapter.

Now the idea is simple: We read the type of the form we want, we grab its template, we compile it and we append it to the directive's element. Also, it would be good if we provide some feedback in case that the requested type has no template available:

```
1 var formType = templates[scope.type];
2
3 if (!formType) {
4   formType = '<h2>There is no template for the current type.</h2>';
5 }
6
7 var el = $compile(formType)(scope);
8
9 element.append(el);
```

It isn't that hard, right? We use `$compile` on the template we need so when we append it to the element, all the angular directives in it will work as expected. Oh, and don't forget to inject `$compile` in the directive.

Alright, the last bit we need in here is the callback method:

```
1 scope.submit = function() {
2   if (scope.onSubmit) {
3     scope.onSubmit({model: scope.model});
4   }
5 };
```

Nice, we finished it. Let's use it:

```
1 <auth-form type="register" model="user" on-submit="onSubmit(model)">
2
3 </auth-form>
```

## Register form

username	password	Confirm password	Submit
----------	----------	------------------	--------

Bit ugly, but hard to style with string concatenation.

Not bad, but we can and we will do better on the next chapter.

Check the code [here](#)<sup>1</sup>

## The tests

To test this directive, we want to make sure that the right template is shown in all the different cases.

Let's start with the basic skeleton:

```

1 describe('Directive: authForm', function() {
2   var $compile, $rootScope;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function(_$compile_, _rootScope_) {
7     $compile = _$compile_;
8     $rootScope = _rootScope_;
9   }));
10 });
```

First, we will test that requesting the register form, gives what we expect:

```

1 describe('register', function() {
2   var element;
3
4   beforeEach(function() {
5     element = angular.element(
6       '<auth-form type="register" model="foo" on-submit="onSubmit(model)">' +
7       '</auth-form>');
8     $compile(element)($rootScope);
9   });
10
11   it('shows a register form for type register', function() {
12     var header = element.find('h2');
```

<sup>1</sup><http://plnkr.co/edit/9J5MOjd14wd1ffinzZUS?p=preview>

```
13     expect(header.text()).toEqual('Register form');
14
15     var inputs = element.find('input');
16     expect(inputs.length).toBe(3);
17 });
18
19 it('calls back the controller on submit', function() {
20     var submitButton = element.find('button');
21
22     $rootScope.onSubmit = jasmine.createSpy();
23
24     submitButton.click();
25
26     expect($rootScope.onSubmit).toHaveBeenCalled();
27 });
28 });
```

Here we compile our directive requesting the register form and then we make sure that the one shown is the one we need and also that the submit button works as intended.

For the login form, we do the same thing:

```
1 describe('login', function() {
2     var element;
3
4     beforeEach(function() {
5         element = angular.element(
6             '<auth-form type="login" model="foo" on-submit="onSubmit(model)">' +
7             '</auth-form>');
8         $compile(element)($rootScope);
9     });
10
11     it('shows a register form for type register', function() {
12         var header = element.find('h2');
13         expect(header.text()).toEqual('Login form');
14
15         var inputs = element.find('input');
16         expect(inputs.length).toBe(2);
17     });
18
19     it('calls back the controller on submit', function() {
20         var submitButton = element.find('button');
21
22     });
23 });
```

```
22     $rootScope.onSubmit = jasmine.createSpy();
23
24     submitButton.click();
25
26     expect($rootScope.onSubmit).toHaveBeenCalled();
27   });
28 });
```

Finally, we also need to be sure that our directive notifies the user when a template is not found:

```
1 describe('no template', function() {
2   it('should render an error if the template is not found', function() {
3     var element = angular.element('<auth-form type="foo"></auth-form>');
4     $compile(element)($rootScope);
5
6     var header = element.find('h2');
7     expect(header.text())
8       .toEqual('There is no template for the current type.');
```

And that is it!

Check it [here](#)<sup>2</sup>

## Summary

When we need to manipulate the DOM inside our directives and those changes involves other directives, we need to manually compile those changes to let Angular process the directives. For that, all we need to do is to use the `$compile` service which will take care of that for us.

---

<sup>2</sup><http://plnkr.co/edit/8JJ40PzNc9pdMjqghZuo?p=info>

## 23. Dynamic templates

On the last chapter, we learnt all about `$compile` and we made an `authForm` directive to show different types of forms. The directive was good enough, but it had several shortcomings, for example, not being able to style our templates correctly or having to hardcode them inside the directive itself.

We can get rid of those shortcomings easily, but how? Dynamic templates were possible since day 1, but it was quite a manual work. We needed to check if our `$templateCache` contained the template and if not, `$http` get it. Since angular 1.3.x we have a new service called `$templateRequest` which is able to do all of that for us:

```
1 $templateRequest('foo.html').then(function(result) {
2   // "result" is our template
3 }, function() {
4   // Template not found, sorry.
5 });
```

That is what I call an interesting service. Having this in mind, we can extract our templates outside the directive:

login.html

---

```
1 <h2>Login form</h2>
2 <form name="register">
3   <div class="form-group">
4     <label>Email</label>
5     <input name="email" ng-model="model.email"
6       placeholder="Email" class="form-control" />
7   </div>
8   <div class="form-group">
9     <label>Password</label>
10    <input name="password" ng-model="model.password"
11      type="password"
12      placeholder="Password" class="form-control" />
13  </div>
14  <button type="submit" ng-click="submit()" class="btn btn-default">
15    Submit
16  </button>
17 </form>
```

---

## register.html

---

```

1 <h2>Register form</h2>
2 <form name="register">
3   <div class="form-group">
4     <label>Email</label>
5     <input name="email" ng-model="model.email"
6       placeholder="Email" class="form-control" />
7   </div>
8   <div class="form-group">
9     <label>Password</label>
10    <input name="password" ng-model="model.password"
11      type="password"
12      placeholder="Password" class="form-control" />
13  </div>
14  <div class="form-group">
15    <label>Password Confirmation</label>
16    <input name="password" ng-model="model.passwordConfirmation"
17      type="password"
18      placeholder="Password" class="form-control" />
19  </div>
20  <button type="submit" ng-click="submit()" class="btn btn-default">
21    Submit
22  </button>
23 </form>

```

---

Ohh, now they are styled properly with bootstrap, yay.

Now with our templates outside the directive, let's revisit it:

```

1 angular.module('app')
2   .directive('authForm', function($compile, $templateRequest) {
3     return {
4       restrict: 'E',
5       scope: {
6         type: '@',
7         model: '=',
8         onSubmit: '&'
9       },
10      link: function(scope, element, attrs) {
11
12      }

```



```
13     };  
14   });
```

The basic skeleton is pretty much the same, we just injected `$templateRequest`. What should we do on the `link` function? Well, we need to grab the template type, request it using the new service and append it to our element:

```
1  var template = scope.type + '.html';  
2  
3  $templateRequest(template, true).then(function(result) {  
4    element.append($compile(result)(scope));  
5  });
```

Easy as pie.



The second argument on `$templateRequest` is set to `true` so it won't throw an exception if the template is not found.

Uhm, I remember that we were able to notify the user when a template wasn't found. Oh right, we just need to append another function to `$templateRequest`:

```
1  $templateRequest(template, true).then(function(result) {  
2    element.append($compile(result)(scope));  
3  }, function() {  
4    element.append('<h2>There is no template for the current type.</h2>');  
5  });
```

Now, if a template is not found, we will append a message into the element.

Last, we add the submit handler like we did before:

```
1  scope.submit = function() {  
2    if (scope.onSubmit) {  
3      scope.onSubmit({model: scope.model});  
4    }  
5  };
```

The directive is now much much simpler and supports an infinite number of different templates.

Its usage didn't change:

```
1 <auth-form type="login" model="user" on-submit="onSubmit(model)">
2
3 </auth-form>
```

## Login form

Email

Password

Ahh much better with styling.

Check it [here](#)<sup>1</sup>

## The tests

For the tests we are going to use a custom [plunker template](#)<sup>2</sup> with our templates cached (plunker makes a bit harder the testing with templates).

The tests are pretty much the same with some exceptions:

```
1 describe('Directive: authForm', function() {
2   var $compile, $rootScope;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function(_$compile_, _$rootScope_) {
7     $compile = _$compile_;
8     $rootScope = _$rootScope_;
9   }));
10 });
```

Skeleton didn't change, let's see the describe blocks:

---

<sup>1</sup><http://plnkr.co/edit/TZmLh7AVeF5sDZfrppjf?p=preview>

<sup>2</sup><http://plnkr.co/edit/2nY1uOyBvypJja4YbN8J?p=info>

```
1 describe('register', function() {
2   var element;
3
4   beforeEach(function() {
5     element = angular.element(
6       '<auth-form type="register" model="foo" on-submit="onSubmit(model)">' +
7       '</auth-form>');
8     $compile(element)($rootScope);
9     $rootScope.$digest();
10  });
11
12  it('shows a register form for type register', function() {
13    var header = element.find('h2');
14    expect(header.text()).toEqual('Register form');
15
16    var inputs = element.find('input');
17    expect(inputs.length).toBe(3);
18  });
19
20  it('calls back the controller on submit', function() {
21    var submitButton = element.find('button');
22
23    $rootScope.onSubmit = jasmine.createSpy();
24
25    submitButton.click();
26
27    expect($rootScope.onSubmit).toHaveBeenCalled();
28  });
29 });
```

We are not forced to test both register and login in this directive, we don't have those templates built-in so we just need to make sure that all this "dynamic templating" works. So here I reused the register template tests we had on the previous chapter. We had to add a `$digest` on the `beforeEach` to "run" the promises of `$templateRequest`.

We will skip the login template tests (they exist on the demo plunker tho) and test what happen when a template is not found:

```
1 describe('no template', function() {
2   it('should render an error if the template is not found',
3     inject(function($httpBackend) {
4
5       $httpBackend.expectGET('foo.html').respond(404);
6       var element = angular.element('<auth-form type="foo"></auth-form>');
7       $compile(element)($rootScope);
8       $httpBackend.flush();
9
10      var header = element.find('h2');
11      expect(header.text())
12        .toEqual('There is no template for the current type.');
```

Since `$templateRequest` will use `$http` to request the template, we need to tell `$httpBackend` that there will be a GET to grab `foo.html` and also to “make” the request, we need to flush the `$httpBackend`. So the idea is like:

- Expect a GET on `foo.html`.
- Create and compile the directive.
- Let the directive make the request.
- Simulate the completion of the request using `flush`.
- Test that everything is in place.

Check the tests [here](#)<sup>3</sup>

## Summary

Requesting templates dynamically inside a directive to be able to modify its template is easily done with `$templateRequest`. It will do the hard work of finding the template for us, so we just need to provide it with a path and it will do its magic.

---

<sup>3</sup><http://plnkr.co/edit/WldiNDq1xjrRt0MO93fD?p=preview>

## 24. Wrapping a jQuery plugin

It is well known that Angular developers tries to avoid jQuery at all costs. Even while that is true, there are times where we need to use a plugin and there is no rewrite done by the community and we don't have the time to do it ourselves.

jQuery plugins are normally executed on a jQuery element like:

```
1 $('#element').plugin();
```

We need a way to do that in Angular. Before trying to copy & paste that into your code, remember one of the most important rules of Angular: All the DOM manipulation should be done on a directive.

Luckily for us, the `link` function of a directive receives the element where the directives sit and even more, if we have jQuery loaded (and we should if we want to use a plugin) that element is in fact a jQuery element. That allows us to do:

```
1 link: function(scope, element, attrs) {  
2     element.plugin();  
3 }
```

Isn't that wonderful?

Let's wrap the tooltip plugin from `bootstrap.js`.



There is a rewrite of `bootstrap.js` at [ui-bootstrap](https://github.com/angular-ui/bootstrap)<sup>1</sup>.

All we need to do is to load `bootstrap.js` and jQuery into our app and create a directive:

---

<sup>1</sup><https://github.com/angular-ui/bootstrap>

```
1 angular.module('app')
2   .directive('tooltip', function() {
3     return {
4       restrict: 'A',
5       link: function(scope, element, attrs) {
6         element.tooltip();
7       }
8     };
9   });
```

And that is it!



jQuery needs to be loaded before Angular or it won't work.

That simple? Yes, we can use it like:

```
1 <button tooltip title="foo" data-placement="right">Hardcoded</button>
```

The tooltip plugin needs two attributes:

- **title:** The content of the tooltip
- **data-placement:** The position of the tooltip.

We created a button where we attach our directive (to run the plugin itself) and also those two attributes.

Keep in mind that those attributes are not directives nor something our directive will use, but attributes that the tooltip will read from the element to work.

On the other hand, the tooltip plugin accepts an options object. There you can define its title, placement, delay, etc.

Let's update our directive to accept those options:

```
1 angular.module('app')
2   .directive('tooltip', function() {
3     return {
4       restrict: 'A',
5       link: function(scope, element, attrs) {
6         var options = scope.$eval(attrs.options);
7         element.tooltip(options);
8       }
9     };
10  });
```



To learn more about the tooltip options, read the [official docs](#)<sup>2</sup>

Here we `$eval` the `options` attribute to get the options object and then we pass it to the tooltip plugin.

Thanks to that, we can do:

```
1 <button tooltip options="options">Custom options</button>
```

Where options is something like:

```
1 $scope.options = {
2   title: 'Custom one!',
3   placement: 'bottom'
4 };
```

Hardcoded

Custom options

Custom one!

Check it [here](#)<sup>3</sup>

---

<sup>2</sup><http://getbootstrap.com/javascript/#tooltips-options>

<sup>3</sup><http://plnkr.co/edit/idHpFOzg0pIDq8w3vWSk?p=preview>

## The tests

Testing this wrappers is not that complicated, we just need to assert that what the plugin does, is present in our screen. In this case, we need to see that when we put the mouse on top of the element, we see the tooltip.



The plugin should be already tested by its team, so we don't need to re-test it.

Let's start with the skeleton as always:

```
1 describe('Directive: tooltip', function() {
2     var element, button, $compile, $document, $rootScope;
3
4     beforeEach(module('app'));
5
6     beforeEach(inject(function(_$compile_, _$document_, _$rootScope_) {
7         $compile = _$compile_;
8         $rootScope = _$rootScope_;
9         $document = _$document_;
10    }));
11 });
```

Nothing fancy. Let's move to test it without using custom options:

```
1 describe('hardcoded options', function() {
2     beforeEach(function() {
3         element = angular.element(
4             '<div>' +
5             '  <button tooltip title="foo" data-placement="left">Btn</button>' +
6             '</div>'
7         );
8         $compile(element)($rootScope);
9         angular.element(document.body).append(element);
10    });
11
12    afterEach(function() {
13        element.remove();
14    });
15
16    it('should show the tooltip on the left', function() {
```



```
17     button = $document.find('button');
18     button.trigger('mouseenter');
19     var tooltip = $document.find('.tooltip-inner');
20
21     expect(tooltip.text()).toBe('foo');
22   });
23 });
```

This particular plugin has a behavior which will complicate our tests. When we test our directives, our element is an unattached element and well, that is usually fine, because our directive will normally work within the element itself. The issue here is that the `tooltip` plugin will not modify the element but create a **sibling** element. Because of that, we need to attach our element to the page and once it is attached, the plugin will be able to find it and then create the tooltip itself.

Ok, that part is fine, but that creates another issue. By attaching our element to the page, it will be there on the next test, and we don't want that.

To fix that, we just simply need to remove the element after each test. Only the element with the directive? No, we also need to delete the tooltip element. Something like:

```
1 $document.find('.tooltip-inner').remove();
```

Or you can wrap your element in a `<div>` so the tooltip will be created within that `<div>` and then you just need to delete it to get rid of everything.

Having this in place, we grab the button, we trigger the `mouseenter` event, we grab the tooltip and we assert that it is there.

Also, we need to test that it also works with custom options:

```
1 describe('dynamic options', function() {
2   beforeEach(function() {
3     element = angular.element(
4       '<div>' +
5         '<button tooltip options="options">Btn</button>' +
6       '</div>'
7     );
8
9     $rootScope.options = {
10       title: 'dynamic',
11       placement: 'right'
12     };
13
14     $compile(element)($rootScope);
```

```
15     angular.element(document.body).append(element);
16   });
17
18   afterEach(function() {
19     element.remove();
20   });
21
22   it('should show the tooltip on the right', function() {
23     var button = $document.find('button');
24     button.trigger('mouseenter');
25     var tooltip = $document.find('.tooltip-inner');
26
27
28     expect(tooltip.text()).toBe('dynamic');
29   });
30 });
```

Same rules as before. We just need to pass an options attribute to the directive and check that it is working as we expect.

Check the tests [here](#)<sup>4</sup>

## Summary

You can write a whole book about this topic, but the basics are here. To wrap a jQuery plugin you need to check how it is used and replicate that inside a directive. There are more complex plugins that needs more attributes and more options, but the same rules still applies.

I highly recommend you to rewrite some simple plugins into “native” angular code. That is an awesome way to learn more about Angular and directives.

---

<sup>4</sup><http://plnkr.co/edit/qwpdz097SnXvAXonfk45?p=preview>

# Appendix A - Using ngModelController

`ngModelController` provides useful methods and properties to work with input objects. With it, you can do custom validations (both local and remote), value formatting, parsing...

In this appendix we are going to learn a couple of examples of `ngModelController` usage.

# Local validation

When we update our model, `ngModelController` will run all the validators associated with that input. Where are those validators stored? In an object called `$validators`. How can we add new validators to it?

```
1 // ngModel is our reference to the controller
2 ngModel.$validators.name = function(modelValue, viewValue) {
3   // Do something and return either true or false
4   // depending if the validation passed or not
5   return result;
6 }
```

So we need to give it a name and assign a function to it. The function will receive the value of the model, the first parameter is the local model and the second is the value of the model in the view. Normally they have the same value.

Let's say I am working on a Google application and only gmail users are able to register. I need a way to validate an input to only accept gmail addresses. Let's do it:

```
1 ngModel.$validators.isGmail = function(modelValue, viewValue) {
2   var value = modelValue || viewValue;
3   return /\S+@gmail\.com/.test(value);
4 };
```

If our model matches the regular expression (an email address ending with `@gmail.com`), it will return true, meaning that our validator was successful. If not, it will return false indicating that our input is `$invalid`.

Let's put that into a directive:

```
1 angular.module('app')
2   .directive('gmailValidator', function() {
3     return {
4       require: 'ngModel',
5       link: function(scope, element, attrs, ngModel) {
6         ngModel.$validators.isGmail = function(modelValue, viewValue) {
7           var value = modelValue || viewValue;
8           return /\S+@gmail\.com/.test(value);
9         };
10      }
11    };
12  });
```

Now, we just need to use it like:

```
1 <form name="userForm" novalidate>
2   <input ng-model="user.email" gmail-validator name="email" />
3 </form>
```

Now, our input will be able to tackle this validation for us! The form will be never \$valid if the email is not a gmail one.

**Email**

Saved email 1: foo@gmail.com  
Saved email 2:

Check it [here](http://plnkr.co/edit/rCzLRStWAqp4tnELZH7z?p=preview)<sup>5</sup>

## The tests

Let's start with the classic skeleton:

---

<sup>5</sup><http://plnkr.co/edit/rCzLRStWAqp4tnELZH7z?p=preview>

```
1 describe('Directive: gmailValidator', function() {
2   var $scope;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function($compile, $rootScope) {
7     $scope = $rootScope;
8     var element = angular.element(
9       '<form name="form">' +
10       '<input ng-model="user.email" name="email" gmail-validator />' +
11       '</form>'
12     );
13     $scope.user = { email: ''};
14     $compile(element)($scope);
15   }));
16 });
```

Since a form with a name is accessible through the scope, we wrap our input inside one, that way we will be able to work with the input.

For the tests:

```
1 it('should pass with a gmail email', function() {
2   $scope.form.email.$setViewValue('johndoe@gmail.com');
3   $scope.$digest();
4   expect($scope.user.email).toEqual('johndoe@gmail.com');
5   expect($scope.form.email.$valid).toBe(true);
6 });
7 it('should not pass with a hotmail mail', function() {
8   $scope.form.email.$setViewValue('janedoe@hotmail.com');
9   $scope.$digest();
10  expect($scope.user.email).toBeUndefined();
11  expect($scope.form.email.$valid).toBe(false);
12 });
```

There is something I didn't explain yet. When does `ngModelController` run the validators? when it calls `$setViewValue`. `$setViewValue` is what we call when we want to change the view value. For example, when we change the value of the input (by writing on it), angular will call this `$setViewValue` for us which will between other things, run all the validators.

Here in the test, we can simulate that we wrote `johndoe@gmail.com` calling `$setViewValue` so we trigger our validator. Then we just need to see that the model has been updated and the email input is valid.

On the second test, we try an invalid email and look, if any validator returns false, our model will contain undefined instead of the invalid value.

Check it [here](#)<sup>6</sup>

## Summary

Writing custom validators is as easy as writing functions that will return either true or false depending if our model is valid given some value.

---

<sup>6</sup><http://plnkr.co/edit/bBoDK5LR83mxPg08LdDT?p=preview>

# Remote validation

`ngModelController` now supports remote validation. Well, to be honest, it is something that always existed, but now it is really really easy to use.

Like with `local validation`, `ngModelController` will execute all the async validators it has inside the `$asyncValidators` array to see whether the model associated is valid or not.

The difference here is that an async validator should return a promise. Angular will run the async validators and then using `$q.all` will wait until all the promises has been fulfilled.

That said, let's do an example. The most typical example is having a form where the user has to put his username. We want to make sure that the username is not already in use, and since we need to check that in the backend, we will need to make a request and that is... well, async.

For this directive, use [this plunker](#)<sup>7</sup>.

So let's think... We need to do a `$http.get` to some endpoint that will return **true** if the username exist and **false** if it doesn't exist.

We could do something like:

```
1 ngModel.$asyncValidators.uniqueUsername = function(modelValue, viewValue) {  
2   var value = modelValue || viewValue;  
3  
4   return $http.get('/api/checkuser/' + value);  
5 };
```

Yeah, but the thing is, our server will return a **2xx** response if it exist and a **4xx** if it doesn't, uhm so if it exist, it will succeed and the validation will pass, and we need the other way around. Let's fix it:

---

<sup>7</sup><http://plnkr.co/edit/S3KNwPKxo78FnHsNth0H>



```
1 ngModel.$asyncValidators.uniqueUsername = function(modelValue, viewValue) {
2   var value = modelValue || viewValue;
3
4   return $http.get('/api/checkuser/' + value)
5     .then(function resolved() {
6       return $q.reject('exists');
7     }, function rejected() {
8       return true;
9     });
10  };
```

Now if it succeed we return a rejection (so the validation will fail) and if it fails, well, just returning true is fine, it failed already.

If we put that inside a directive, we can achieve:

```
1 angular.module('app')
2   .directive('checkUser', function($http, $q) {
3     return {
4       require: 'ngModel',
5       link: function(scope, element, attrs, ngModel) {
6         ngModel.$asyncValidators.uniqueUsername =
7           function(modelValue, viewValue) {
8
9             var value = modelValue || viewValue;
10
11             return $http.get('/api/checkuser/' + value)
12               .then(function resolved() {
13                 return $q.reject('exists');
14               }, function rejected() {
15                 return true;
16               });
17           };
18       }
19     };
20  });
```

Now, we just need to use it like:

```

1 <form name="userForm" novalidate>
2   <input ng-model="user.username" check-user name="username" required/>
3 </form>

```

Notice how this time we are using `required`. It is not required (pun intended), but since the sync validators runs before the async ones, if the field is empty, it won't fire the async ones. Makes the input a bit more usable :)

**Username**


Check it [here](#)<sup>8</sup>

## The tests

Tests are much the same as the previous chapter, so I am going to paste they here and explain what changes.

Skeleton:

```

1 describe('Directive: checkUser', function() {
2   var $scope, $httpBackend;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function($compile, $rootScope, _$httpBackend_) {
7     $scope = $rootScope;
8     $httpBackend = _$httpBackend_;
9     var element = angular.element(
10      '<form name="form">' +
11      '  <input ng-model="user.username" name="username" check-user />' +
12      '</form>'
13    );
14    $scope.user = { username: '' };
15    $compile(element)($scope);
16  }));
17 });

```

Tests:

---

<sup>8</sup><http://plnkr.co/edit/wYCqwQ3AV9Jf3XtpwfsT?p=preview>

```
1  it('should pass with a free name', function() {
2    $httpBackend.expectGET('/api/checkuser/john').respond(401);
3
4    $scope.form.username.$setViewValue('john');
5    $httpBackend.flush();
6    $scope.$digest();
7    expect($scope.user.username).toEqual('john');
8    expect($scope.form.username.$valid).toBe(true);
9  });
10 it('should not pass if the name is taken', function() {
11   $httpBackend.expectGET('/api/checkuser/john').respond(201);
12
13   $scope.form.username.$setViewValue('john');
14   $httpBackend.flush();
15   $scope.$digest();
16   expect($scope.user.username).toBeUndefined();
17   expect($scope.form.username.$valid).toBe(false);
18 });
```

In this case, since we are reaching our backend, we tell our test that we are expecting a GET (they will fail if the GET is never made). Then we set the view value, we simulate that the request is being done and we make our assertions.

As easy as the last one :)

Check it [here](#)<sup>9</sup>

## Summary

Async validators are really cool. In the past they needed a bit of manual work, but thanks to `$asyncValidators`, we can talk to our server or do any other async operation. Just make sure you return a promise this time ;)

---

<sup>9</sup><http://plnkr.co/edit/PGu9FTDOqQ8XTW0qyOQF?p=preview>

# Formatters and Parsers

When you change your input or the model associated with it, `ngModelController` runs a series of functions. Those functions are called `$formatters` and `$parsers`.

When you change your input, all the functions inside the `$parsers` array will run. On the other hand, if the associated model changes, the `$formatters` functions will execute.

Those arrays work as a pipeline. They will be executed in order passing its return value through to the next.

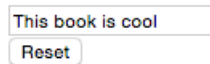
In the case `$parsers`, they will run in order and the last returned value will be the input for the first `$validator`. In the case of `$formatters`, they run in reverse array order and the last return value will be used as the actual DOM value.

Let's see an example, shall we? For today example, we are going to implement an input limited by a maximum number of characters.

```
1 angular.module('app')
2 .directive('twitterInput', function() {
3     return {
4         require: 'ngModel',
5         link: function(scope, element, attrs, ngModel) {
6
7         }
8     };
9 });
```

How can we make this work? We need a way to filter our input, and that is what a `$parser` function does:

```
1 function valueFilter(value){
2     if(value && value.length > 5){
3         value = value.substring(0, 5);
4     }
5
6     return value;
7 }
```



This

If our value's length gets bigger than 5 characters, we cut it to be 5 characters long. This way our associated model won't get longer than 5 characters. That is good, but if we try it, the model gets filtered correctly, but the input still accepts more characters. How can we fix that?

`ngModelController` has a `$render` method. That method does nothing by default, but the input directive implemented it so when we trigger it, it will update the value of the input but based on view value.

To use it, we need to change the view's value and then trigger a render:

```
1 function valueFilter(value){
2   if(value && value.length > 5){
3     value = value.substring(0, 5);
4     ngModel.$setViewValue(value);
5     ngModel.$render();
6   }
7
8   return value;
9 }
```



This

Now, if the value's length is bigger than 5, we cut the part we don't need and we set this new value to be the new view value and then, we render the input again. This way, when we have 5 characters on the input, it won't accept more. Said in other words... If we have `hello` written in our input and we try to put `hellow`, our filter will cut that `hellow` into `hello` and update the input with that value. We will never see that `w` being written.

We now need to add this function to the `$parsers` pipeline:

```
1 ngModel.$parsers.push(valueFilter);
```

Now, what happens if we modify the associated model programmatically ? The input will receive the new value, but no filtering is being done. Why? We learned before that the `$parsers` pipeline will only run when the model changes **from the view** and not programmatically. How can we fix that? We just need to do add this function in the `$formatters` pipeline:

```
1 ngModel.$formatters.push(valueFilter);
```

Now it works perfectly.

Let's add an extra feature. If there is an attribute called `twitter-length`, we use it as the max length. First we create a `maxLength` variable:

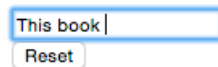
```
1 var maxLength = attrs.twitterLength || 5;
```

And we change the hardcoded 5 to use `maxLength`:

```
1 function valueFilter(value){
2   if(value && value.length > maxLength){
3     value = value.substring(0, maxLength);
4     ngModel.$setViewValue(value);
5     ngModel.$render();
6   }
7
8   return value;
9 }
```

Now we can use this directive like:

```
1 <form name="twitter">
2   <input name="twitt" ng-model="message" twitter-input twitter-length="10"/>
3 </form>
```



This book

Check it [here](http://plnkr.co/edit/tZ1mVcM50wiP0ORL5jkM?p=preview)<sup>10</sup>

## The tests

We will test this directive using the same patterns we used in this appendix:

---

<sup>10</sup><http://plnkr.co/edit/tZ1mVcM50wiP0ORL5jkM?p=preview>

```
1 describe('Directive: checkUser', function() {
2   var $scope, $compile;
3
4   beforeEach(module('app'));
5
6   beforeEach(inject(function(_$compile_, $rootScope) {
7     $scope = $rootScope;
8     $compile = _$compile_;
9   }));
10 });
```

This time we have a simpler skeleton. Since we have an optional parameter in here, we need two different templates, so I splitted the tests in two describe blocks:

```
1 describe('with default length', function() {
2   beforeEach(function() {
3     var element = angular.element(
4       '<form name="form">' +
5       '  <input ng-model="twitt" name="twitt" twitter-input />' +
6       '</form>'
7     );
8     $compile(element)($scope);
9   });
10
11   it('should limit the input correctly', function() {
12     $scope.form.twitt.$setViewValue('I have a lot of characters');
13     $scope.$digest();
14     expect($scope.form.twitt.$viewValue).toBe('I hav');
15     expect($scope.twitt).toEqual('I hav');
16   });
17 });
```

When using the default length, we need to make sure that if we set a big string as the view value, both the associated model and the input will have the filtered value.

```
1 describe('with custom length', function() {
2   beforeEach(function() {
3     var element = angular.element(
4       '<form name="form">' +
5       '  <input ng-model="twitt" name="twitt" +
6       '    twitter-input twitter-length="10" />' +
7       '</form>'
8     );
9     $compile(element)($scope);
10  });
11
12  it('should limit the input correctly', function() {
13    $scope.form.twitt.$setViewValue('I have a lot of characters');
14    $scope.$digest();
15    expect($scope.form.twitt.$viewValue).toBe('I have a l');
16    expect($scope.twitt).toEqual('I have a l');
17  });
18 });
```

For the cases where we use the optional attribute, we assert the same things as before, but making sure that the new maximum length is respected.

Check the tests [here](http://plnkr.co/edit/8eGAJuVe2bEt7Cwfp1x?p=preview)<sup>11</sup>

## Summary

This pipelines are really useful. In the past they were used to create custom validations, and even when that is not true anymore, they can be useful to transform our input. For example: upper case what we write, transform a string into tags...

---

<sup>11</sup><http://plnkr.co/edit/8eGAJuVe2bEt7Cwfp1x?p=preview>