SOURCE: http://www.sitepoint.com/practical-guide-angularjs-directives/

Directives are the most important components of any AngularJS application. Although AngularJS ships with wide range of directives, you will often need to create application specific directives. This tutorial will give an overview of custom directives and explain how to use them in real world Angular projects. At the end of the tutorial, I will guide you through the creation of a simple note taking app with the help of Angular directives.

**Overview**

A directive is something that introduces new syntax. Directives are markers on a DOM element which attach a special behavior to it. For example, static HTML does not know how to create and display a date picker widget. To teach HTML this new syntax we need a directive. The directive will somehow create an element that behaves like a date picker. We will see how this is achieved subsequently.

If you’ve written an Angular application before, then you have used directives, whether you realized it or not. You might have used simple directives like ng-model,ng-repeat,ng-show, etc. All these directives attach special behavior to DOM elements. For example, ng-repeat repeats a specific element and ng-show conditionally shows an element. If you want to make an element draggable/droppable you might create a directive for that too. The basic idea behind directive is very simple. It makes your HTML truly interactive by attaching event listeners to the elements and/or transforming the DOM.

**The jQuery Perspective**

Just imagine how you create a date picker with jQuery. We first add a normal input field to the HTML and then in jQuery we call $(element).datePicker() to actually convert it to a date picker. But, think about it. When a designer comes and examines the markup, can he/she immediately guess what the field actually is? Is it just a plain input field or a date picker? You have to look into the jQuery code to be sure. The Angular approach is to use a directive to extend HTML. So, a directive for a date picker can look like this:

<date-picker></date-picker>

Or it could look like this:

<input type="text" date-picker/>

This approach to creating UI components is intuitive and clear. You can simply look at the element to know what it is supposed to be!

**Building Custom Directives:**

An Angular directive comes in four flavors in terms of appearance.

1. A new HTML element (<date-picker></date>).
2. An attribute on an element (<input type="text" date-picker/>).
3. As a class (<input type="text" class="date-picker"/>).
4. As comment (<!--directive:date-picker-->).

Of course, we have control over how our directive will appear in the HTML. Now, let’s see how a typical directive is written in Angular. It is registered in the same way as a controller, but it returns a simple object (directive definition) that has several properties to configure the directive. The following code shows a simple Hello World directive.

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

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

return {

restrict: 'AE',

replace: 'true',

template: '<h3>Hello World!!</h3>'

};

});

In the previous code, the app.directive() function registers a new directive in our module. The first argument to this function is the directive name. The second argument is a function which returns a directive definition object. If your directive has dependencies on external objects/services such as $rootScope, $http, or $compile, they can be injected here. The directive can be used in HTML as an element like this:

<hello-world/>

Or:

<hello:world/>

Or, as an attribute like this:

<div hello-world></div>

Or:

<div hello:world/>

If you want to be HTML5 compliant, you can prefix the attribute name with x- or data-. So, the following markups will match helloWorld directive.

<div data-hello-world></div>

OR

<div x-hello-world></div>

**Note:**

While matching directives, Angular strips the prefix x- or data- from element/attribute names. Then it converts - or : delimited strings to camelCase and matches with the registered directives. That’s why we have used the helloWorld directive as hello-world in the HTML.

Though the above directive does nothing more than show static text, we have some interesting points to explore. We have used three properties in the directive definition object to configure the directive. Let’s explore the role that each one plays.

* restrict – This provides a way to specify how a directive should be used in HTML (remember a directive can appear in four ways). In this case we have set it to 'AE'. So, the directive can be used as a new HTML element or an attribute. To allow this directive to be used as a class we can set restrict to 'AEC'.
* template – This specifies the HTML markup that will be produced when the directive is compiled and linked by Angular. This does not have to be a simple string. The template can be complex, often involving other directives, expressions ({{ }}), etc. In most cases you want to use templateUrl instead of template. So, ideally you should place the template in a separate HTML file and make templateUrl point to it.
* replace – This specifies if the generated template will replace the HTML element on which the directive is attached. In our case we have used the directive as <hello-world></hello-world>, and replace is set to true. So, after the directive is compiled, the produced output template replaces <hello-world></hello-world>. The final output is <h3>Hello World!!</h3>. If you set replace to false, the default, the output template will be inserted into the element on which the directive is invoked.

Open up this [plunker](http://plnkr.co/edit/GKI339z2VDdZTOE2bGFP), right click on “Hello World!!”, and do an inspect element to visualize things.

**The link Function and Scope**

The template produced by a directive is meaningless unless it’s compiled against the right scope. By default a directive does not get a new child scope. Rather, it gets the parent’s scope. This means that if the directive is present inside a controller it will use that controller’s scope.

To utilize the scope, we can make use of a function called link. This is configured by the link property of the definition object. Let’s change our helloWorld directive so that when the user types a color name into an input field, the background color of Hello World text changes automatically. Also, when a user clicks on the text Hello World, the background color should reset to white. The HTML markup is shown below.

<body ng-controller="MainCtrl">

<input type="text" ng-model="color" placeholder="Enter a color" />

<hello-world/>

</body>

The modified helloWorld directive is shown below:

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

return {

restrict: 'AE',

replace: true,

template: '<p style="background-color:{{color}}">Hello World',

link: function(scope, elem, attrs) {

elem.bind('click', function() {

elem.css('background-color', 'white');

scope.$apply(function() {

scope.color = "white";

});

});

elem.bind('mouseover', function() {

elem.css('cursor', 'pointer');

});

}

};

});

Notice the link function used in the directive. It takes three arguments:

* scope – The scope passed to the directive. In this case it’s the same as the parent controller scope.
* elem – The jQLite (a subset of jQuery) wrapped element on which the directive is applied. If you have included jQuery in the HTML before AngularJS is included, this becomes jQuery wrapped instead of jQLite. As the element is already wrapped with jQuery/jQLite, there is no need to again wrap it inside $() for DOM manipulations.
* attrs – An object representing normalized attributes attached to the element on which the directive is applied. For example, you can attach attributes in HTML like: <hello-world some-attribute></hello-world> and access it in the link function as attrs.someAttribute.

The link function is mainly used for attaching event listeners to DOM elements, watching model properties for changes, and updating the DOM. In the previous directive snippet, we attached two listeners, click and mouseover. The click handler resets the background color of the <p>, while the mouseover handler changes the cursor to pointer. The template has an expression {{color}} which changes whenever the model color changes in the parent scope, thereby changing the background color of Hello World. Here is a [plunker](http://plnkr.co/edit/14q6WxHyhWuVxEIqwww1) demonstrating this concept.

**The compile Function**

The compile function is used to perform any DOM transformation before the link function runs. It accepts the following arguments.

* tElement – The element on which the directive is applied.
* attrs – The normalized list of attributes declared on the element.

Just note that the compile function does not have access to the scope, and must return a link function. But, if there is no compile function you can configure the link function as usual. The compile function can be written as:

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

return {

compile: function(tElem,attrs) {

//do optional DOM transformation here

return function(scope,elem,attrs) {

//linking function here

};

}

};

});

Most of the time, you will be working with the link function only. This is because most of the directives are concerned with registering event listeners, watchers, updating the DOM, etc., which are done inside the link function. Directives like ng-repeat, which need to clone and repeat the DOM element several times, use the compile function before the link function runs. This leads to a question of why two separate functions are needed. Why can’t we have just one? To answer this we need to understand how directives are compiled by Angular!

**How Directives are Compiled**

When the application bootstraps, Angular starts parsing the DOM using the $compile service. This service searches for directives in the markup and matches them against registered directives. Once all the directives have been identified, Angular executes their compile functions. As previously mentioned, the compile function returns a link function which is added to the list of link functions to be executed later. This is called the compile phase. If a directive needs to be cloned multiple times (e.g. ng-repeat), we get a performance benefit as the compile function runs once for the cloned template, but the link function runs for each cloned instance. That’s why the compile function does not receive a scope.

After the compile phase is over the linking phase, where the collected link functions are executed one by one, starts. This is where the templates produced by the directives are evaluated against correct scope and are turned into live DOM which react to events!

**Changing a Directive’s Scope**

By default a directive gets the parent’s scope. But we don’t want that in all cases. If we are exposing the parent controller’s scope to the directives, they are free to modify the scope properties. In some cases your directive may want to add several properties and functions to the scope that are for internal use only. If we are doing these things to parent’s scope, we are polluting it. So, we have two other options:

* A child scope – This scope prototypically inherits the parent’s scope.
* An isolated scope – A new scope that does not inherit from the parent and exists on its own.

The scope can be configured with the scope property of the directive definition object. An example of this is shown in the following snippet.

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

return {

scope: true, // use a child scope that inherits from parent

restrict: 'AE',

replace: 'true',

template: '<h3>Hello World!!</h3>'

};

});

The above code asks Angular to give the directive a new child scope that prototypically inherits from parent scope. The other option, an isolated scope, is shown below.

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

return {

scope: {}, // use a new isolated scope

restrict: 'AE',

replace: 'true',

template: '<h3>Hello World!!</h3>'

};

});

This directive uses a new isolated scope that does not inherit from the parent. Isolated scopes are good when we want to create reusable components. By isolating the scope we guarantee that the directive is self contained and can be easily plugged into an HTML app. This provides the parent scope from becoming polluted, as it is not accessible inside the directive. In our modified helloWorld directive if you set scope to {} the code won’t work anymore. It will create an isolated scope for the directive and the expression {{color}} will now refer to that isolated scope property (not parent scope) which is undefined.

Isolating the scope does not mean that you have no access to the parent scope’s properties. There are techniques that allow you to access the parent scope’s properties and also watch for changes on them. We will discuss these techniques, and some more advanced concepts like Controller functions in part two of this series. Part two will also walk you through the creation of a fully fledged note taking app using Angular directives, so stay tuned!