Course: AngularJS Directives Fundamentals

Placement: Module 3 (decorator directives)

===

Ctrl + Shift + C does inspect element

Author heavily uses Plunkr throughout the course

=====

Plunkr

=====

What’s cool about Plunkr is that when you’re running it, it auto-reloads the view based on your changes

Ctrl + arrow key lets you move between files

===

By default, a custom directive is restricted to being both an element and an attribute

===

ng-show=”!!user.address” ?

Three different types of directives: Components, decorators and structural/templating.

Component: The canonical custom directive, sometimes called a widget.

Decorator: The most commonly used type of directive. The decorator adds additional functionality to an existing tag or modifies that tag’s display. Examples- ng-click, ng-show, ng-hide. Almost always implemented as an attribute and almost never have a template.

Structural/Templating: Example - ng-repeat. Manipulates the DOM structurally to produce a display.

replace: true makes the custom directive tag not appear in the html

When you do this, you need to make sure your templateURL code is wrapped in a root element (just use a div)

The replace attribute is considered deprecated…

Use a prefix for your directive names in production code

“We’re beginning to see that directives are almost like a mini ecosystem inside of angular where everything you can do inside of regular angular can be done inside a directive.

scope: true // inherited scope

scope: false // shared scope, the default

scope: {} // isolated scope

---

\*\*\*\*\*\*\*\*\*\*\*

directive.js

\*\*\*\*\*\*\*\*\*\*\*

scope: {

   user: “=”

}

\*\*\*\*\*\*\*\*\*\*\*

index.html

\*\*\*\*\*\*\*\*\*\*\*

<user-info-card user=”user1”></user-info-card>

The attribute user=”user1” gives the directive scope access to the controller’s scope

===

In the case of two inherited directives on the same element, they use the same scope. Because they use the same scope, they could easily conflict with each other by trying to write to the same variables on the scope.

Best practice: Try not to put two inherited directives on the same element.

In general, component directives should use isolated scopes but decorated directives should use inherited or (preferably) shared scope.

===

The !! operator converts the value to a boolean and ensures a boolean type.

===

@ used for simple string values

===

initialCollapsed: ‘@collapsed’

The attribute typed by the user is collapsed (ex/ collapsed=“true”) but the controller sees initialCollapsed.

This naming convention can be used for the other operators (=, &)

===

& is for taking in functions

===

All the event handling directives in Angular are decorator directives

===

Attribute directives applied to an HTML5 video tag

angular.module("app").directive("spacebarSupport", function() {

 return {

   restrict: "A",

   link: function (scope, el, attrs) {

     $('body').on('keypress', function (evt) {

       var vidEl = el[0];

       if(evt.keyCode == 32) {

         if(vidEl.paused) {

           vidEl.play();

         } else {

           vidEl.pause();

         }

       }

     })

   }

 }

});

angular.module("app").directive("eventPause", function () {

 return {

   restrict: 'A',

   scope: {

     eventPause: "&"

   },

   link: function (scope, el, attrs) {

     el.on('pause', function (event) {

       scope.$apply(function() {

         scope.eventPause();

       })

     })

   }

 }

})

Confused why scope.$apply is needed.

Author’s explanation: The HTML5 video has fired the pause event, but that’s actually happening outside of Angular’s knowledge (why?). So if we were to try and do anything outside of Angular, in Angular nothing happens outside of a digest cycle. So we’re going to have to start a new digest cycle because an external event has happened.