Angular Directives

Exploring the awesome power of reusable code!

Presented by
Peter Pavlovich
Chief Technology Officer
Censinet, Inc.
pavlovich@gmail.com

Introductions …

- Peter Pavlovich
  - Front-End Evangelist and Technology Addict
  - Current: Chief Technology Officer, Censinet
    - Vue/React/Angular/Meteor/Polymer/Aurelia, Node, Scala/Java, OrientDB/Titan/MongoDB
  - Entrepreneur, Open Source Contributor and Community Member
  - Instructor, Mentor, Speaker and Leader.
  - http://linkedin.com/in/peterpavlovich
  - pavlovich@gmail.com

Current Work

- Greenfield UI, middle tier and backend
    - Vue/React/Rails/Postgres
    - Angular/Meteor/Node/MongoDB/Postgres
    - Java/Embedded C
- Open Source contributor
    - Meteor Metadata-driven Modeling Pkg.
    - Angular/Meteor Forms Package
    - Multiple 'side-projects'

Current Work

- Mentor for 55 startup companies co-located within "Greentown Labs", an incubator lab facility in Boston, MA
    - Focused on Green Energy & Clean Tech.
    - Office and Lab space
    - Access to expert advisors/mentors
    - Training, funding help, general support.

Plan for our session ...

- Angular 1
  - A few slides to orient us to the framework
  - Introduction to Angular 1 Directives
  - Interactive live coding of progressively complex example directives.
- Angular 2
  - Custom directives
- Q & A

Angular?

- Angular is:
  - JavaScript-based client-side MV* framework
  - Super powerful, easy to code in.
  - Flexible, efficient, productive for coders.
  - “Extends” HTML through ‘directives’
  - 2-way data binding: views <-> data.

Why use Angular?

- Very expressive, readable, maintainable code
  - Intentions are clear from reading HTML
  - Clean separation: Intention vs. implementation
  - HIGHLY testable (everything is testable)
  - Nice integrations with Meteor, Firebase and others allows angular to own the DOM but get its data reactively.
  - Plan for the future: Angular 2 is almost here!

Problem statement

(how will Angular help)

- Simplify view code
  - Reduce code complexity, volume.
  - HTML is more readable, intent clearer
- Simplify 'controller' code
  - Actions are cleaner
  - 2-way data-binding simplifies DB ops.

Angular Toolbox

- Modules: Organization/Architecture
- Modules provide HTML extensions via “tags” which bind instances of the three categories of objects defined in the MVC pattern into reusable “Components”:
  - “Services” = Models
  - “Templates” = Views
  - “Controllers” = Controllers

Angular: Gut check!

- To write directives, we should understand:
  - What Angular is, deep down.
  - What happens when we load angular.js

Angular: Gut Check!

- At its core, Angular is:
  - ‘Tag-to-HTML-and-JavaScript’ pre-processor
  - Tag registry / definition DSL
- Tags = keywords in the DOM
- Tags are organized into libraries called modules
- Angular ships with a core module (ng) + others
- You can create your own libraries

Loading angular.js:

- Phase 1:
  - Creates ‘tag’ registry.
  - Loads tag creation DSL.
  - Loads core ‘ng’ module and registers its tags.
  - Waits until all other JS files are loaded.

Loading angular.js

- Phase 2:
  - Need to discover WHERE to look for ‘tags’ to replace or augment.
  - Two methods:
    - Automatic discovery
    - Manual ‘Bootstrapping’

Automatic Discovery

- Scans DOM for ‘ng-app’ tag.
  - <element ng-app=“ModuleName”>
- Ensures module ‘ModuleName’ plus ALL of its direct and implied dependencies are loaded.
- Scans element and all of its children for ‘tags’ defined in the module or its dependencies.
- Applies ‘tags’ to specified elements.
- Rescans resulting structure in case new ‘tags’ were introduced in last round.
- Rinse, repeat … until no new tags introduced.

Autodiscovery Restrictions

- Can only ‘autodiscover’ ONE ng-app tag per HTML document.
- Nesting of ng-apps is not permitted
- Unlike other places in angular, snake-case module names in the HTML are NOT converted to CamelCase names in JavaScript

Manual Bootstrapping

- YOU manually ‘bootstrap’ angular:
    - use angular.bootstrap(element, modules)

  element: Pointer to a real DOM element: e.g. document, angular.element(x)

  modules: Array of module names to ‘use’.
- Loads named module(s) + dependencies.
- Applies ‘tags’ to specified elements.
- Scans element plus its children for ‘tags’ defined in the module or its dependencies.
- Applies ‘tags’ to specified elements.
- Rescans resulting structure for new ‘tags’
- Rinse, repeat … until no new tags introduced.

Manual Bootstrapping Notes

- Can bootstrap as many modules as you would like into a given element.
- You can bootstrap the same or different sets of modules into any number of elements in a given HTML document.
- Any given element in the DOM can ONLY be bootstrapped ONCE. Watch out for nesting.

So, what are ‘tags’?

Directives!

What can they do?

- Modify DOM elements
  - Add/remove styles/attributes
  - Add/remove content
  - Completely replace elements with content
  - Apply other directives (recursively ...)

Time to code!

- Preparations:
  - Download angular.js
  - Install some sort of web server
    - mac: python -mSimpleHTTPServer port
    - win: python -m http.server port

Starting point

```
<!DOCTYPE html>
<html>
  <body ng-app>
    <h1>Hello World!</h1>
    <script src="angular.js"></script>
  </body>
</html>
```

Demo

ad01

Modules = Organization

- Module: Named group of angular constructs
- Modules organize your Angular code
  - Namespacing
  - Isolation
  - Reuse
  - Dependency Management

Create your own!

```
<script type="application/javascript">
  angular.module(‘myModule’,[‘req1’,’req2']);
</script>
```

Use it!

```
<!DOCTYPE html>
<html>
  <body ng-app="myModule">
    <h1>Hello World!</h1>

    <script src=“angular.js"></script>
    <script type="application/javascript">
      angular.module('myModule',[]);
    </script>

  </body>
</html>
```

Scope and Controllers

- Controller = context factory
  - Populates context in constructor
    - Temporary data store (key/value pairs)
    - Used to populate view
    - context object can be:
      - the ‘scope’ object injected by angular
      - the controller itself (‘controller-as’ syntax)
  - Provides a ‘closure’ for the scope (if used)
- Assigned using the ‘ng-controller’ directive or as part of a directive definition
- Visibility limited to DOM element and children.

Assign controller

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
  <body ng-app="myModule">

    <div ng-controler="MyCtrl">
      <h1>Hello {{name}}!</h1>

    </div>
    <script src=“angular.js"></script>
    <script type="application/javascript">
      angular.module('myModule',[])
        .controller('MyCtrl', function($scope){
          $scope.name = 'Peter';
        });
    </script>
  </body>
</html>
```

Assign controller ‘as’

```
<!DOCTYPE html>
<html lang='en'>
  <body ng-app='myModule'>
    <div ng-controller="MyCtrl as vm">
      <h1>Hello {{vm.name + ' ' + name2}}!</h1>
    </div>
    <script src='angular.js'></script>
    <script type="application/javascript">
      angular.module('myModule',[])
        .controller('MyCtrl', function($scope){
          var vm = this;
          vm.name = 'Peter';
          $scope.name2 = 'John'
        });
    </script>
  </body>
</html>
```

Controllers … check!

- Controllers are useful, but they don’t get me all the way to where I want to be which is a way to package up my view and my behavior plus other dependencies into one useful reusable component.
- Enter …. directives!

Defining Directives

- Defined in the context of a module
- Have a name (unique within the module)
- Defined with a function returning a specification object with various attributes.
- module.directive(‘name’, function(){return {}})

Defining Directives: General form

```
angular.module('myModule', [])
  .directive('newtext', function () {
    return {
      optionName: optionValue,
                 …
                 …
    }
  });
```

Using Directives

- Reference the directive name inside HTML
- Assuming a directive named ‘demo’:
  - <any demo>…</any> (attribute-based)
  - <demo>…</demo> (element-based)
  - <any class="demo">…</any> (class-based)
  - <!-- directive:demo --> (comment-based)
- Directive must be “visible” where referenced

What are the options?

- controller: function literal or reference to one/null
- restrict: String containing one or more of 'E','A','C','M' ('EA')
- template: string literal/function literal returning a string/null
- templateUrl: string/function returning a string/null
- link: function/null
- transclude: true/'element'/falsey
- replace: true/falsey
- require: string/array or strings/null
- scope: true/falsey/object literal ({})
- controllerAs: string/null
- bindToController: true/object literal ({})/falsey
- compile: function returning linking object or function/null
- multiElement: true/falsey
- priority: number (0)
- terminal: true/falsey
- templateNamespace: 'html'/'svg'/'math'

https://docs.angularjs.org/api/ng/service/$compile
https://docs.angularjs.org/guide/directive

Scope and Controllers in Directives

- Used for the same purpose as in views
- Directives modify DOM, updating or replacing elements with new ones.
- Directive scope/controllers act on new/changed DOM without a ‘ng-controller’
- Influences ‘tagged’ element and its contents.
- Better control over scope isolation/contents

controller

```
controller: function($scope,$element,$attrs){…}

controller: myControllerFn

controller: ‘myControllerFn as identifier’
```

- Value is:
  - a literal, inline function or
  - reference to a function visible from here
  - default is null (no controller)
- Must use these exact parameter names.
- ‘as identifier’ => {{identifer.property}} in template
- If you define a $onInit function on your controller, it will be called after all bindings are

Demo

- Let's play with some code!
  - Basic directive
  - Using the directive
    - attribute, element

ad03

restrict

```
restrict: "A|E|C|M“
```

- Restrict specifies which type of ‘tag’ you can use to invoke your directive.
  - A = attribute
  - E = element
  - C = CSS Class
  - M = Comment
- Default = “AE”

Demo

- Play with restricting
- Demo minification issues

ad04 / 5 / 6 / 7

Explicit Function Annotation or ‘Bracket Notation’

```
controller: [‘$scope’,’$element’,’$attrs’,function(a,b,c){…}]

var myControllerFn = function(a,b,c){…}
…
controller: [‘$scope’,’$element’,’$attrs’,myControllerFn]
```

- If you plan to minify:
  - use whatever variable names you want in the function definitions.
  - the injector matches against the ‘string’ names that precede the function in the array
  - injection is done positionally, not by name.

Force check on stict-DI

- <any ng-app="someName" ng-strict-di="boolean"… >
- angular.bootstrap(any, [`someName'], {strictDi= true});
  - default value is false!
- Bracket notation also applies to your definitions of other angular constructs like services and directives themselves.
- Note, if you don't use any of the controller constructor function arguments or inject anything, you don't HAVE to use bracket notation, but it is recommended that you do to establish the habit.

Demo

- Bracket notation

ad08

template

```
template: "<h2>New text!</h2>“
template: function(tElement, tAttributes){return " ... "}
```

- The eventual value assigned to ‘template’:
  - is an HTML template string
  - can contain binding expressions
- Function can modify the element or attributes
- Any bindings still in ‘{{varName}}’ format

Demo

- Templates

ad10

templateUrl

```
templateUrl: “templates/myTemplate“
templateUrl: function(tElement,tAttributes){return "..."}
```

- The eventual value assigned to ‘templateUrl’:
  - is a string url pointing to a template file
  - can contain binding expressions
- Function can modify the element or attributes
- Any bindings still in ‘{{varName}}’ format

Demo

- Template URLs

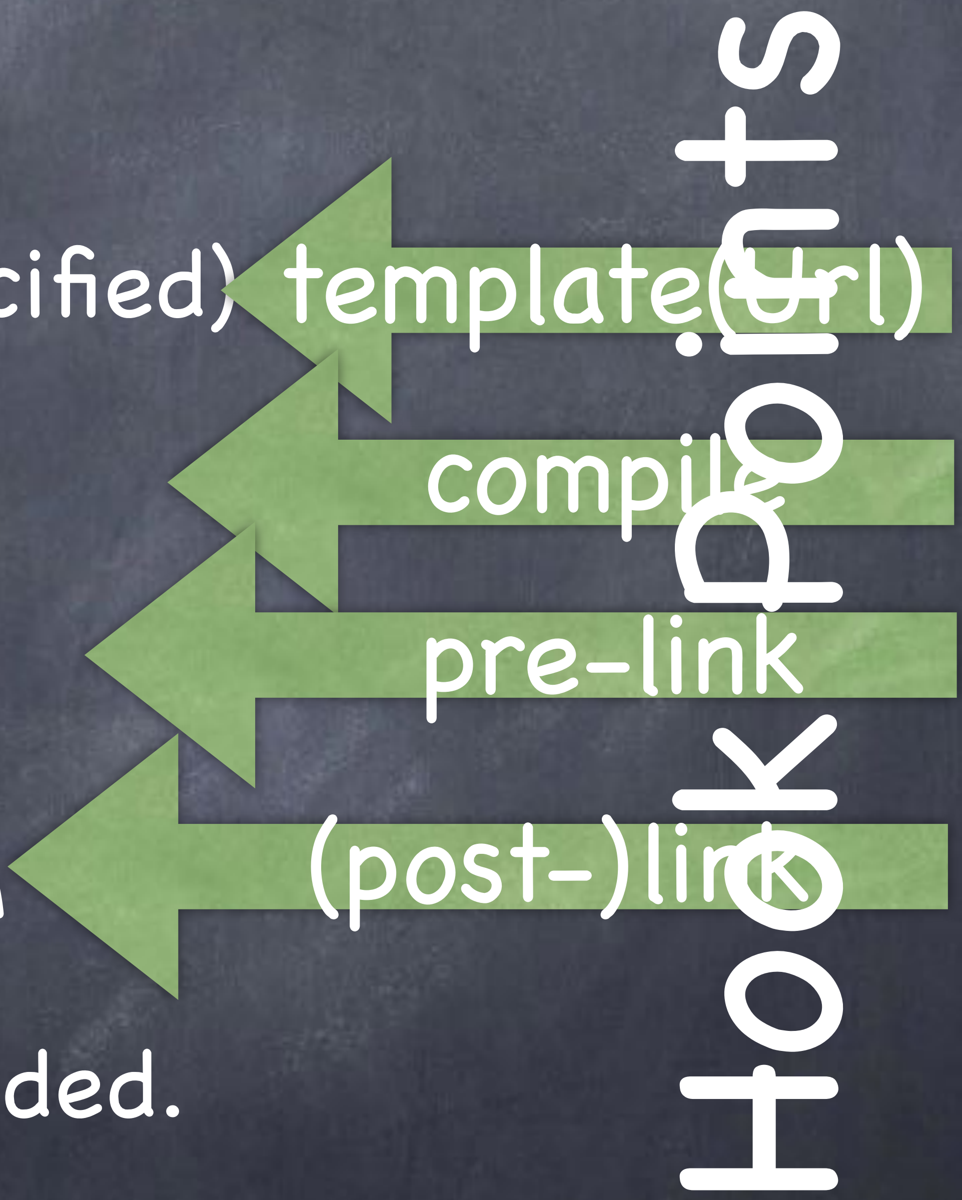
Ad12

Directive Lifecycle

- We’ve seen that directives can manipulate the DOM for us. They can replace contents with ‘templates’, then they can fill in those templates with data and ‘hook up’ event handlers for us.
- Angular does this in a number of ‘steps’

Steps to directive heaven

1. Get the template (if any specified)
2. Compile the template
3. Fill-in the blanks
4. Stick the results in the DOM
5. Re-scan and reprocess if needed.



Hook Point Uses …

- **template(Url)**: Get the template contents
- **compile:** Change template **DOM** / element **attribs**
  - Happens ONCE per use of the directive even if the directive is a repeater type.
  - Make changes you want to make to the TEMPLATE DOM before it is compiled to a function and locked in. You have access to the attributes of the original target element here.
  - Optimization construct: Put stuff here that only needs to happen once per use of your directive.
  - ‘replacement’ function is created and locked in. Wants to be ‘called’ with some view model object (i.e. the $scope)

Hook Point Uses ...

- **pre-link:** Make changes to view model object
    - The view model object is about to be passed to the function created from your template.
    - Make any additions or deletions to that object to prepare it for it ‘binding’ to your template text through that function.
    - No point in making any changes to the element’s contents here (even though you have access to it). Its contents will be replaced shortly by the output of the function being called with the scope.

Hook Point Uses …

- **post-(link)**: Hook up event handlers/listeners
  - The New DOM has been inserted as the target element's contents.
  - All data and events specified in your template have been 'hooked up'
  - Use the post-link to add watchers, register listeners for local/global events or changes to data held on the view model (scope).
  - Do not make changes to the DOM without doing a manual recompile/replace of the affected elements

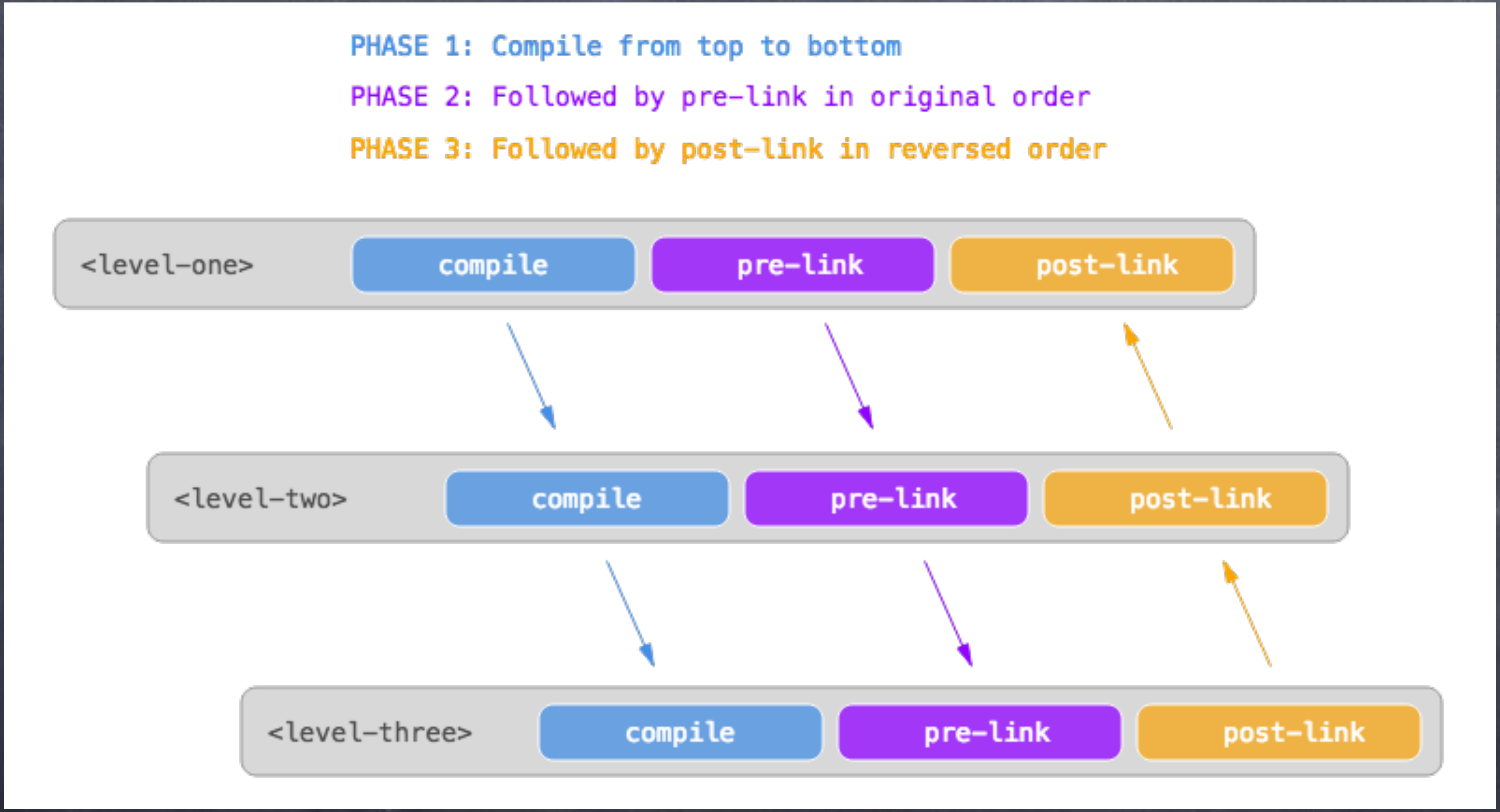
Nested Directives

- Look at this HTML with custom directives:

```
<level-one>
    <level-two>
        <level-three>
            Hello {{name}}
        </level-three>
    </level-two>
</level-one>
```

This example and graphics gratefully pilfered from : Jurgen Van de Moere's blog (http://goo.gl/BckkLT)

Order of hooks …


PHASE 1: Compile from top to bottom
PHASE 2: Followed by pre-link in original order
PHASE 3: Followed by post-link in reversed order
<level-one>
compile
pre-link
post-link
<level-two>
compile
pre-link
post-link
<level-three>
compile
pre-link
post-link

Why?

Compilation Parent->Child

- Parent gets compiled then children.
- Note that NO DOM substitution has taken place yet. Original DOM still intact.
- Parents and Children can 'see' the original DOM tree before any manipulations take place and act accordingly

Pre-link
Parent->Child

- Parent gets to fix up its attributes and view model before the child does.
- Allows parent to set/update values on view models before the children update their own.
- Since child view models can inherit from their parents', it is important that the parent state be known before the children make decisions prior to linking phase where the data will be substituted into templates.

Post-link
Child->Parent

- Child gets to hook up event handlers and/or recompile itself before the parents do. Thus, if a parent recompiles, the ‘final’ state of the child elements are already known. Otherwise, child DOM changes could get clobbered by a parent recompile during post-linking.
- Allows children to set/update values on view models either shared with or inheriting from its parents’ view models.
- Parent-initiated VM changes will trigger any child-initiated watches allowing the children to adjust to any changes the parents make in their own post-linking functions.

I don’t need all that

- We can supply one, two or all of the compile, pre- and post-link hook functions.
- But, often, all we want is to hook up event handlers and listeners or do some post-processing/recompilation based on ‘finished’ state after everyone else is done playing.
- For this, we can just include a ‘link’ property which functions as the ‘post-link’ property would.

link

```
link: function(scope, element, attributes, ctrls, transclude)
{…}
```

- The eventual value assigned to ‘link’:
  - is a literal function or reference to one
  - default is null
  - params (injected positionally, not by name):
    - scope: injected scope supplied by angular
    - element: element referencing the directive
    - attributes: attribs referencing the directive
    - ctrls: array of controllers required in
    - transclude: a function … see later!
- Purpose: To handle events & manipulate the DOM
- Ignored if you provide a compile function.

Demo

- Let's use the link function to set up a child/ parent directive interaction that we talked about!
- Back to the code!

transclude

```
transclude: true | false
```

- The eventual value assigned to ‘transclude’:
  - is a boolean value:
    - true: ‘wrap’ content of original
    - false: discard original content.
    - default is false.

Example gratefully pilfered from Tero Parviainen's excellent blog post: http://goo.gl/YUU4zj

Demo

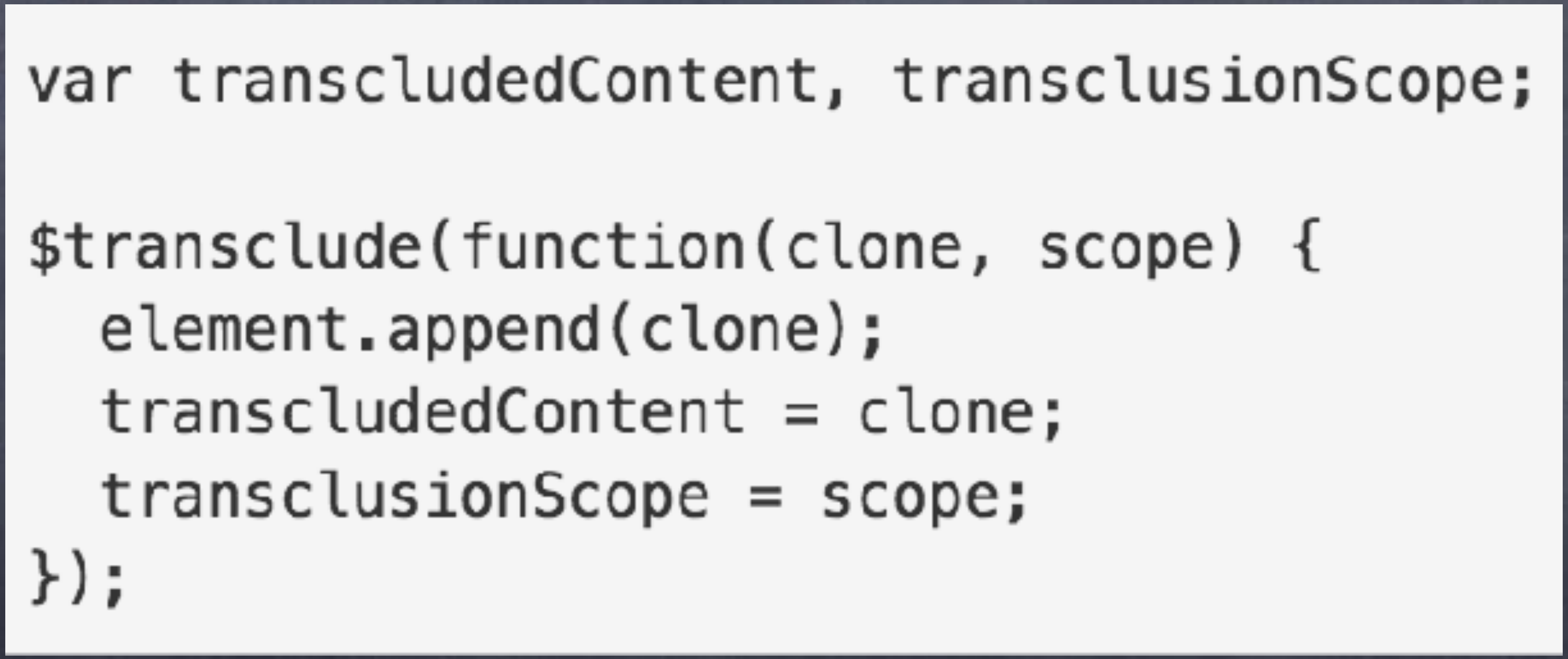
- Transclude

ad18

Transclude function

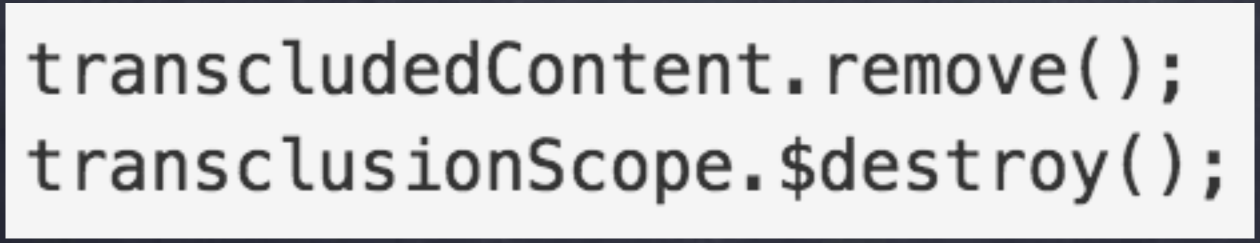
```
link: function(scope, element, attributes, ctrl, transclude) {…}
post: function(scope, element, attributes, ctrl, transclude) {…}
```

```
controller: function($scope,$element,$attrs,$transclude){…}
```



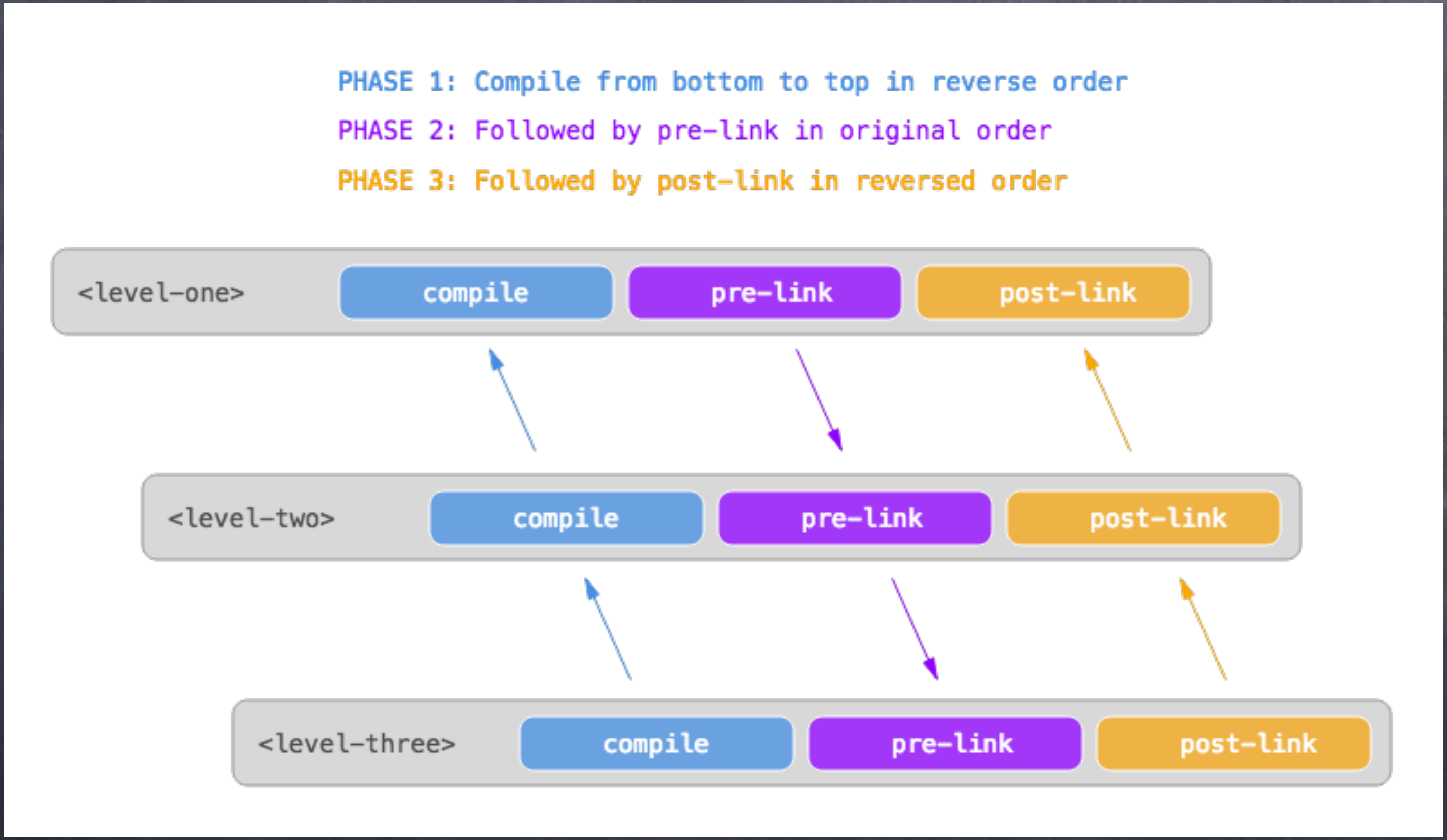
```
var transcludedContent, transclusionScope;

$transclude(function(clone, scope) {
  element.append(clone);
  transcludedContent = clone;
  transclusionScope = scope;
});
```



```
transcludedContent.remove();
transclusionScope.$destroy();
```

Transclude Hook Order …


PHASE 1: Compile from bottom to top in reverse order
PHASE 2: Followed by pre-link in original order
PHASE 3: Followed by post-link in reversed order
<level-one>
compile
pre-link
post-link
<level-two>
compile
pre-link
post-link
<level-three>
compile
pre-link
post-link

Why?

This example and graphics gratefully pilfered from : Jurgen Van de Moere's blog (http://goo.gl/BckkLT)

Compile order reversed!

- Need to ‘insert’ the content of myself into my template at the <ng-transclude> marker
- So … I need to know what that content is! I need to start all the way at the bottom to ensure I don’t miss any other transcludes!

Demo

- Transclude

ad18

replace

```
replace: true | false
```

- The eventual value assigned to ‘replace’:
  - is a boolean value:
    - true: replace the ‘tagged’ element
    - false: retain the ‘tagged’ element as a container.
    - default is false

Demo

- Replace

require

```
(1) require:'[prefix]directiveName'
(2) require:'[[prefix]directiveName', …]
(3) require:'{key1:[prefix]directiveName',key2:…}
```

- Retrieve the controller from the named directive. Where to find that directive is controlled using a prefix to the directive name you are looking for:
  - '' - Get from current element. Error if not found.
  - ^ - Get from current/parents. Error if not found.
  - ^^- Get from element's parents. Error if not found.
- '?' in front of Prefixes return null if missing, not error.
- Value passed as final (4th) param to linking function will be either (1) a controller, (2) array of controllers or (3) an object keyed as the 'input' object with values being the requested controllers.

Demo

- Replace

ad20

Scopes in directives

- Directives have a scope
  - injected into controller and link functions
  - can be your parent's scope or your own.
  - if your own, can have parent's as prototype
  - can be completely isolated from parent

scope

```
scope: true | falsey | {@|=|&}
```

- The eventual value assigned to 'scope':
  - is a boolean value or an object literal
    - true: own scope w/prototypically inherited parent
    - falsey: use parent scope
    - {aName: 'prefix[name]', ...}: use an isolated scope
      - Prefixes: = (2-way), @ (1-way), & (function)
    - default is falsey

Demo

- Replace

ad21

controllerAs

```
controllerAs: stringIdentifier
```

- Binds the view template using the stringIdentifier rather than the implied ‘this’
- If you are using controller: ‘someControllerName’, you can combine using controller: ‘someName as identifer’.

bindToController

```
bindToController: true|falsey|{@|=|&}
```

- If true and the scope is set to an object => bind the scope properties directly to the controller not the scope object.
- Uses an isolated scope for the scope
- If you set the require option to an object as well, this will bind those required controllers directly on this controller according to the keys in that object.
- All bindings are guaranteed to be finished before calling $onInit function on the controller (if defined).

Demo

- Bind to controller

ad23

compile

```
compile: function(element, attributes, transclude) {…}
compile: myCompileFn
```

- The eventual value assigned to ‘compile’:
  - is a literal function or reference to one
  - default is null
  - params (injected positionally, not by name):
    - element: element referencing the directive
    - attributes: attribs referencing the directive
    - transclude: function used in transclusion
  - Returns ‘Link’ function or object with keys ‘pre’ (pre-link fn) and ‘post’ (post-link fn).

compile examples

```
.directive('myName', function(){
  return {
    compile: function(){
      . . .
      return {
        pre: function(scope, element, attributes, transclude){},
        post: function(scope, element, attributes, transclude){}
      };
    }
  }
})

.directive('myName', function(){
  return {
    compile: function(){
       . . .
      return function(scope, element, attributes, transclude){};
    }
  }
})
```

priority

```
priority: number
```

- The eventual value assigned to ‘priority’:
  - is a number
  - default is 0
- If more than one directive applies to an element, the priority assigned to the directive determines the order in which they run.
- Higher numbers get executed first.
- You can use negative number to ensure your directive gets run last.

terminal

```
terminal: boolean
```

- The eventual value assigned to ‘terminal’:
  - is a boolean
  - default is false
  - If true, this will be the final directive run on this element.

templateNamespace

```
templateNamespace: ‘html’|’svg’|’math’
```

- The eventual value assigned to ‘terminal’:
  - is a string, one of the choices shown
  - default is ‘html’
  - Used to identify directives which create elements which must be inserted into the DOM in a ‘special way’.
    - Only two cases so far are SVG and MATH elements. See the docs online for more info.

multiElement

```
multiElement: true | false
```

- The eventual value assigned to ‘multiElement’:
  - is a boolean
  - When true, the HTML compiler will collect DOM nodes between nodes with the attributes directive-name-start and directive-name-end, and group them together as the directive elements.
  - Expect this to be used on directives which are not strictly behavioral (such as ngClick), and which do not manipulate or replace child nodes (such as ngInclude).

New for 1.5+ …

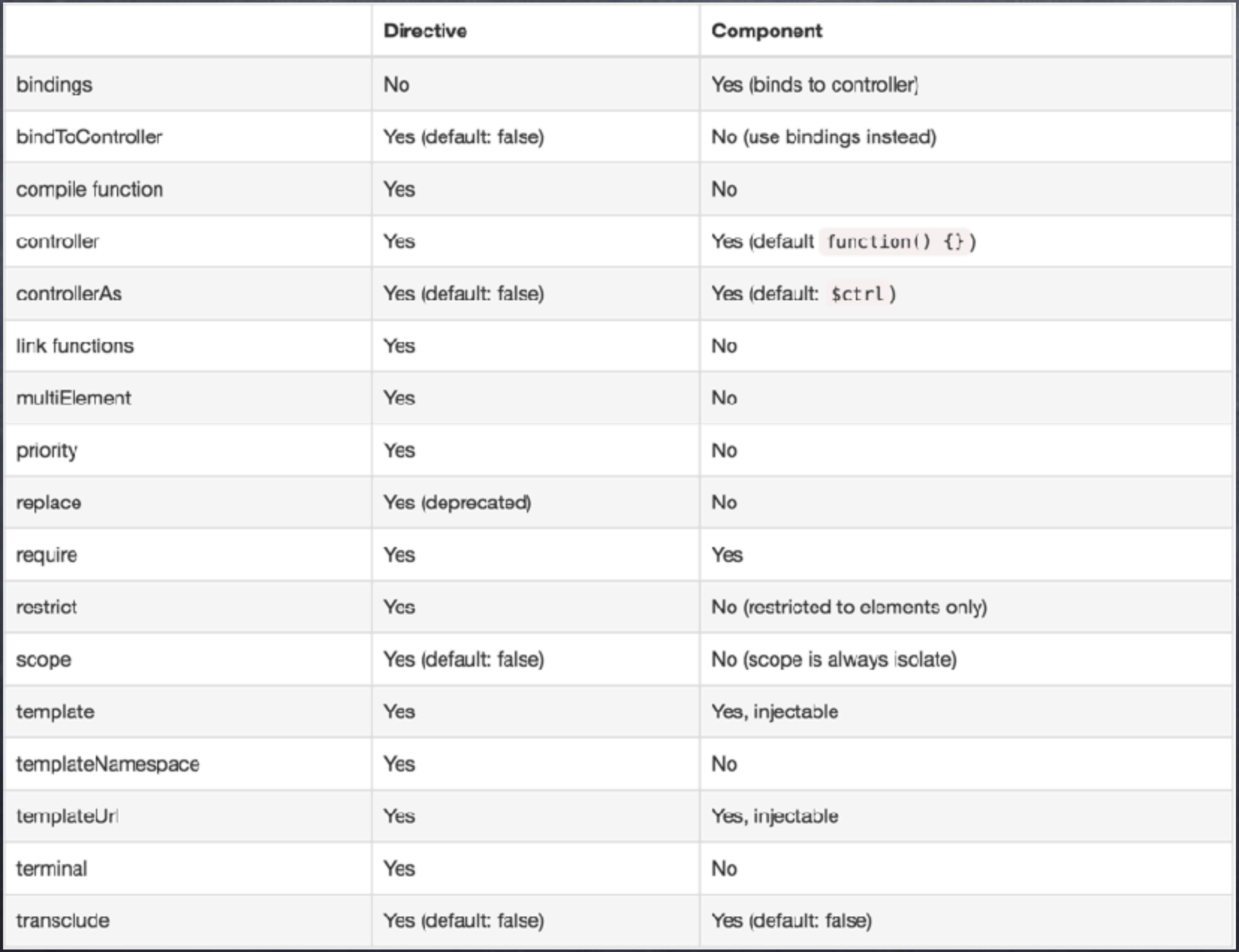
‘Component’ directives

- New ‘convenience’ type of directive definition
  - Provides a limited set of DDO options
    - adds ‘bindings’ to the DDO
      - is an alias for ‘bindToController’
    - restrict: ‘E’ (don’t change)
    - scope: {} (isolated, don’t change)
    - controllerAs: ‘$ctrl’ (can override)

‘Component’ directives

- Common DDO options you can set:
  - controller
  - controllerAs (default: $ctrl)
  - template, templateUrl
  - transclude
  - require
  - bindings

Component vs Directive



	Directive	Component
bindings	No	Yes (binds to controller)
bindToController	Yes (default: false)	No (use bindings instead)
compile function	Yes	No
controller	Yes	Yes (default `function() {}`)
controllerAs	Yes (default: false)	Yes (default: `$ctrl`)
link functions	Yes	No
multiElement	Yes	No
priority	Yes	No
replace	Yes (deprecated)	No
require	Yes	Yes
restrict	Yes	No (restricted to elements only)
scope	Yes (default: false)	No (scope is always isolate)
template	Yes	Yes, injectable
templateNamespace	Yes	No
templateUrl	Yes	Yes, injectable
terminal	Yes	No
transclude	Yes (default: false)	Yes (default: false)

Why use components?

- Advantages of Components:
    - simpler configuration than plain directives
    - promote sane defaults and best practices
    - optimized for component-based architecture
    - writing component directives will make it easier to upgrade to Angular 2

When not to use components?

- For directives that rely on DOM manipulation, adding event listeners etc, because the compile and link functions are unavailable
- When you need advanced directive definition options like priority, terminal, multi-element
- When you want a directive that is triggered by an attribute or CSS class, rather than an element (restricted to elements only)

Examples

```
var myMod = angular.module(...);
myMod.component('myComp', {
  template: '<div>My name is {{$ctrl.name}}</div>',
  controller: function() {
    this.name = 'shahar';
  }
});

myMod.component('myComp', {
  template: '<div>My name is {{$ctrl.name}}</div>',
  bindings: {name: '@'}
});

myMod.component('myComp', {
  templateUrl: 'views/my-comp.html',
  controller: 'MyCtrl',
  controllerAs: 'ctrl',
  bindings: {name: '@'}
});
```

Angular 2/6?

Types of Directives in Angular 2

- Component: Directive with a view template.
- Structural: Alters the structure of the DOM by removing or adding elements.
  - *ngFor, *ngIf
- Attribute: Alters the appearance or behavior of DOM elements through the manipulation of its attributes/properties.

Components

```
@Component({
  changeDetection?: ChangeDetectionStrategy: change detection strategy for component
  viewProviders?: Provider[]: injectables available to the component and view children
  moduleId?: string
  templateUrl?: string: URL for the template for this component
  template?: string: Literal template code for this component
  styleUrls?: string[]: Array of links to style sheets for this component
  styles?: string[]: Array of style entries to apply to this component
  animations?: any[]: list of animations of this component
  encapsulation?: ViewEncapsulation: encapsulation strategy used by this componen
  interpolation?: [string, string]: custom interpolation markers used in this template
  entryComponents?: Array<Type<any>|any[]>: List of components auto inserted in this view
  preserveWhitespaces?: boolean

// inherited from core/Directive
  selector?: string: Identifier for component
  inputs?: string[]: Properties of the component others can provide values for
  outputs?: string[]: Emmitters that consumers of the component can listen to
  host?: {[key: string]: string}: map of class prop/host bindings for events/props/attribs
  providers?: Provider[]: injectables to be provided to the component and children
  exportAs?: name under which the component instance is exported in a template
  queries?: {[key: string]: any}: Queries to be injected into the component
})

See https://angular.io/api/core/Component for more info
```

Attribute Directives

- Import Directive, ElementRef + Input
- Use @Directive instead of @Component
- Define no template
- Apply changes to ElementRef using NG's API

```
import { Directive, ElementRef, Input } from '@angular/core';

@Directive({ selector: '[myHighlight]' })
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

Attribute Directives

```
import { Directive, ElementRef, HostListener, Input }
  from '@angular/core';

@Directive({ selector: ‘[myHighlight]' })
export class HighlightDirective {

  constructor(private el: ElementRef) { }

  @Input() defaultColor: string;

  @Input('myHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || this.defaultColor || 'red');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

Structural Directives

- Name-prefixed with a '*'
  - Used to make structural DOM changes

Structural Directives

```
<div *ngIf="hero" >{{hero.name}}</div>
```

↓

```
<div template="ngIf hero">{{hero.name}}</div>
```

↓

```
<ng-template [ngIf]="hero">
  <div>{{hero.name}}</div>
</ng-template>
```

Structural Directives

```
import {Directive, Input, TemplateRef, ViewContainerRef} from '@angular/core';

/**
 * ### Syntax
 *
 * - `<div *myUnless="condition">...</div>`
 * - `<div template="myUnless condition">...</div>`
 * - `<template [myUnless]="condition"><div>...</div></template>`
 *
 */
@Directive({ selector: '[myUnless]'})
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @Input() set myUnless(condition: boolean) {
    if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }
}
```

References

- https://blog.angularindepth.com/exploring-angular-dom-abstractions-80b3ebcfc02
- http://angular.io
- http://angularjs.org
-

Questions?

Thank you!