FAQ

(where "frequently" === at least once)

Section 1

Q: Can you explain the concept of "a directive" to me?

A: A directive is your chance to extend HTML with a new element specific to your needs. You use div, input, table, ul, etc. all the time when creating your pages. What if you could create one for costs-graph, search-form, or social-media-links if those are things you find yourself repeating over and over?

Why not extend HTML with some language specific to what you specifically are building? That's what a directive lets you do. Add a new element to the page that can have visible and invisible parts as well as interactive elements.

Q: You said in the course that directives have a scope? Why do they have a scope?

A: The flippant answer is that Google says that they should, but I think we can infer that they wanted them to be able to have their own scope so they could be more like HTML elements.

HTML elements normally take all of their content, styling cues, etc. from attributes applied directly to the elements themselves and not just from outer context. That allows us to have two divs one after another which appear different from each other because each has a different CSS class.

When building our own directives we need easy ways to say that one instance needs to use different data or has a different appearance from another by setting attributes of it and without worrying that that will carry over into our other directive instances.

Q: Can you explain isolate scopes in more depth?

A: Be sure to read the question just before this one because the two really go together.

While AngularJS allows us to have directives which inherit their scope from a parent they really come into their own when you isolate them from that scope.

Imagine you wanted to get a third party directive for displaying graphs within your pages. However, if it did not have an isolated scope you would suddenly have to worry about how that directive named its own variables and what variables it was creating in the shared scope because they would all be in the scope of the controller

with which you wrap it.

That makes development more difficult and more dangerous (or at least more error prone). Some developers would have naming conventions like namespace_someName to prevent collisions but others would inevitably pick names like "color" or "data" and run headlong into other data in outside scopes.

Isolate scope protects you from that.

Q: What happens if I have a variable in the scope of a directive with the same name as a variable in the scope of a controller which wraps it?

A: Thanks to isolate scope. Nothing. The directive is working in its own world and does not need to worry about polluting the scope of another directive or controller which may wrap it. Both can choose to use the same variable names with impunity.

Section 2

Q: Do you have some real world examples of directives?

A: AngularJS has pages of them, ng-repeat, ng-click, or anything on this page: https://docs.angularjs.org/api/ng/directive

But beyond those examples there are projects like AngularStrap (http://mgcrea.github.io/angular-strap/) which wraps Bootstrap controls in AngularJS wrappers and sites like http://ngmodules.org/ which has hundreds of examples.

Q: When do I need to use $observe in my directive?

A: $observe is almost never needed because most of the time we are using isolate scope in our directives and $watch and $observe are exactly the same for bound variables in an isolate scope. I reference this which is considered one of the authoritative explations online: http://stackoverflow.com/questions/14876112/difference-between-the-observe-and-watch-methods

I have never needed $observe in a directive.

Q: Isn't "controller as" syntax obsolete?

A: No! I do not believe that at all. If anything I believe that Google is steering developers away from $scope and toward something more like "controller as" in the future.

Q: Where can I learn about "controller as" outside of a directive?

A: I like Todd Motto's explanation pretty well: http://toddmotto.com/digging-into-angulars-controller-as-syntax/

I also like that he talks about how it interacts with $watch, directives (also covered in the course), and ngRoute.

Q: What exactly is “ng-transclude” attribute doing within the template?

A: It's specifying the exact spot the contents of the directive (seen out in the HTML) will be placed when the directive is replaced with the template for that directive.

Q: Are we limited to a single top-level tag in the template if I’m using transclusion?

Q: What happens if I place the “ng-transclude” attribute in multiple tags in the template?

Q: What happens if I don’t use the “ng-transclude” attribute at all, but I set “transclude: true”?

A: Man, you have a lot of transclusion questions.

There is no restriction on the template that it should only have one top level tag as far as I know. I crafted an example with two different divs, each at the topmost level of the template and was able to embed the transcluded material into each one individually or both without problems. See also the next answer.

Multiple uses of ng-transclude just embeds the same content multiple spots in the compiled output. So use it as many times as you need the inner material to repeat in your template.

If you don’t use the ng-transclude at all but set transclude: true, nothing happens. It has no place to insert the contents so it does nothing.

Section 3

Q: What about communication between directives using $emit/$broadcast?

A: I really recommend against that. When you're using events to communicate it can be very difficult to reason about why something happened. "This code over here triggered because of an event. Was it this code that generates that event or over there or over there?"

When functions are called directly there's often a call stack in the browser which I can trace back through to see the origin of a particular call and know why it happened.

I feel like this is reinforced by AngularJS itself. If you look through the framework you will not see many examples of where it is either emitting or sinking events itself. There is a place for communication via a loose mechanism like this but do not use it a lot. I think you will regret it.

Q: Why are there different functions? Why controller and pre-link and post-link?

A: That's an excellent question. The best answer I have is that, as shown in the course videos, they get called

at different times within the life-cycle of a directive. Different points in time get different functions.

But you can imagine a world where there was only one function and the phase the directive was in during its life cycle was a parameter to the function. So this solution was probably not the only one possible, but it's the one which was picked.

Q: Is the $scope that a controller can inject the same as the scope passed to the link function?

A: Yes. And the very next question is a good follow up to that.

Q: Why don't link function parameters have a $ at the beginning?

A: A controller gets all its parameters via dependency injection. If you don't need a particular parameter, don't list it and you won't get it. So the naming conventions it follows are the standard ones for dependency injection in AngularJS. Getting a scope is done via injecting $scope whether we're talking about the controller in a directive or the controller attached to a view.

By contrast, the link gets a standard set of parameters, it does not get them via dependency injection and they arrive in a specific order. You can name that first parameter anything you like ($scope, scope, or penguins), but you're always going to get the scope there. So I assume that AngularJS names it slightly differently in the description of the link function to highlight that you cannot change the order of those parameters like you could in the controller.