Angular JS

Main sources for this training

Code examples:

https://github.com/bhovhannes/trainings/tree/master/angular

ng-book:

https://www.ng-book.com

or

https://drive.google.com/a/attask.com/folderview?id=0B6kIiS_4mYqERXFXRUpGYTNFbEE&usp=sharing

Links to other used sources will be provided during the training

0.

The Basics of Angular JS

Let’s start with
the basics

What is Angular JS ?

It is a **framework** that is primarily used to build single-page web applications

AngularJS makes it easy to build interactive, modern web applications by increasing the level of abstraction between the developer and common web app development tasks.

The AngularJS team describes it as a "**structural framework for dynamic web apps.**"

What does Angular JS give to you ?

- Separation of application logic, data models, and views
- Ajax services
- Dependency injection
- Routing
- Testing
- more …

1.

Data-binding

Introducing data-binding in Angular JS

01-data-binding

https://github.com/bhovhannes/trainings/blob/master/angular/examples/01-data-binding/index.html

```
<!DOCTYPE html>
<html ng-app>
    <head>
        <title>01-data-binding</title>
        <script src="https://ajax.googleapis.
com/ajax/libs/angularjs/1.2.27/angular.js"></script>
    </head>
    <body>
        <input ng-model="name" type="text" placeholder="Your name">
        <h1>Hello {{ name }}</h1>
    </body>
</html>
```

How data-binding works ?

Angular JS creates **live** templates and uses them as a view. Individual components of the view are **dynamically interpolated** live.

Interpolation is when the view is evaluated with one or more variable substitutions.

Interpolation on a view “Hello {{ **name** }}” with variable named “**name**” which value is equal to “Foo” will return “Hello Foo”.

Any time Angular thinks the model value could change, it calls **$digest()** to check if value is ‘dirty’ (**dirty-checking**). And if it is, Angular updates the view.

02-clock

https://github.com/bhovhannes/trainings/blob/master/angular/examples/02-clock/index.html

```
<!DOCTYPE html>
<html ng-app>
    <head>
        <title>02-clock</title>
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.27/angular.
js"></script>
        <script>
            function MyController($interval, $scope) {
                var updateClock = function() {
                    $scope.clock = new Date();
                };
                $interval(updateClock, 1000);
                updateClock();
            };
        </script>
    </head>
    <body ng-controller="MyController">
        <h1>Now is {{ clock }}</h1>
    </body>
</html>
```

$scope

The **$scope** object is simply a JavaScript object whose properties are all available to the view and with which the controller can interact.

$scope acts as a data model for view. The model *observes* the change through dirty checking, and if the model changes the value, the view updates with the change.

All properties found on the $scope object are *automatically* accessible to the view.

2. Modules

A main way to define an AngularJS app

03-clock-module

https://github.com/bhovhannes/trainings/blob/master/angular/examples/03-clock-module/index.html

In Angular, a module is the *main* way to define an AngularJS app.

Using modules allows to:

1. Keep global namespace clean
2. Make it easy to share code between applications
3. Load different parts of the code in any order

Module setter/getter syntax

```
var myModule = angular.module('myApp', ['dep1', 'dep2']);
```

When called with 2 arguments, defines a new module, named "myApp", which depends on modules "dep1" and "dep2".

```
var myModule = angular.module('myApp');
```

When called with 1 argument, returns an existing module, named "myApp".

04-module-properties

https://github.com/bhovhannes/trainings/blob/master/angular/examples/04-module-properties/index.html

`name (String)`

Name of the module as a string

`requires (String[])`

List of module names, which are loaded before the module itself is loaded (i.e. list of module dependencies)

3.

Expressions

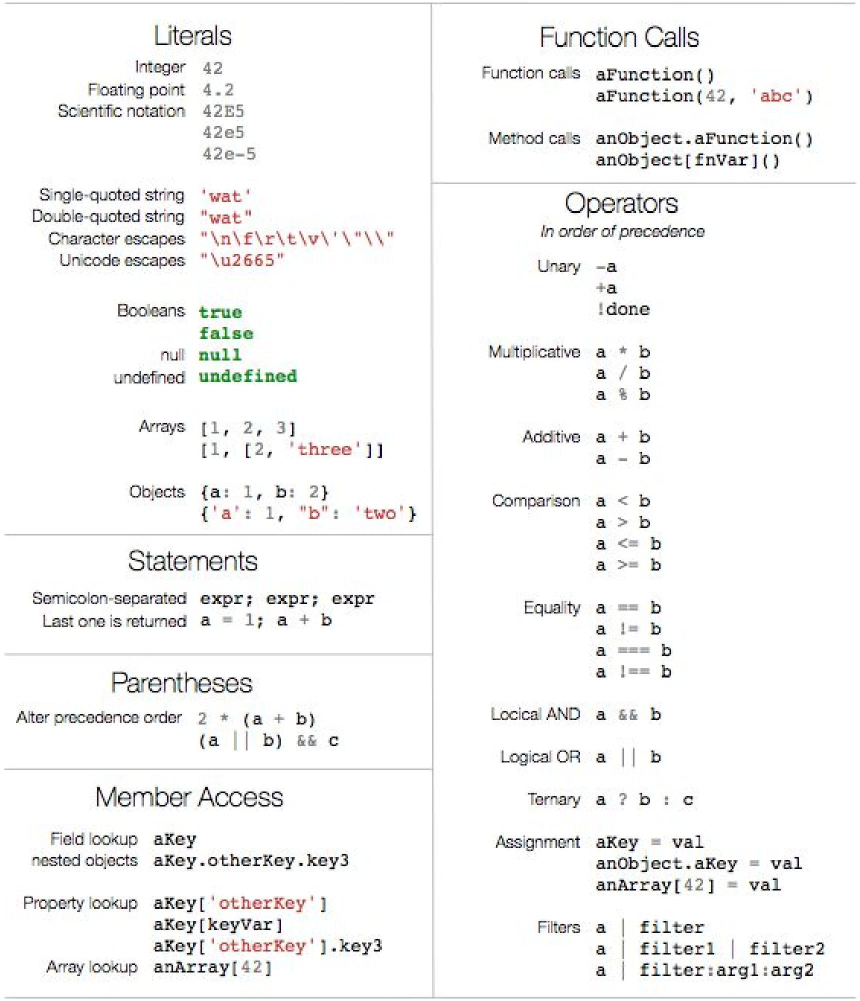
```
{{ 'Hello' + name }}
```

Angular Expressions

- Angular expressions are JavaScript-like code snippets that are usually placed in bindings such as `{{ expression }}`
- Angular expressions differ from JavaScript expressions

Angular Expressions vs. JavaScript Expressions

- **Context**: JavaScript expressions are evaluated against the global window. In Angular, expressions are evaluated against a **scope** object.
- **Forgiving**: In JavaScript, trying to evaluate undefined properties generates `ReferenceError` or `TypeError`. In Angular, expression evaluation is forgiving to `undefined` and `null`.
- **No Control Flow Statements**: You cannot use the following in an Angular expression: conditionals, loops, or exceptions.
- **No Function Declarations**: You cannot declare functions in an Angular expression.
- **No RegExp Creation With Literal Notation**: You cannot create regular expressions in an Angular expression.
- **No New Operator**: You cannot create new objects using `new` `operator` in an Angular expression.
- **No Comma And Void Operators**: You cannot use `,` or `void` `operators` in an Angular expression.
- **Filters**: You can use **filters** within expressions to format data before displaying it.



Literals

Integer	`42`
Floating point	`4.2`
Scientific notation	`42E5`
	`42e5`
	`42e-5`
Single-quoted string	`'wat'`
Double-quoted string	`"wat"`
Character escapes	`"\n\f\r\t\v\'\"\\"`
Unicode escapes	`"\u2665"`
Booleans	`true`
	`false`
null	`null`
undefined	`undefined`
Arrays	`[1, 2, 3]`
	`[1, [2, 'three']]`
Objects	`{a: 1, b: 2}`
	`{'a': 1, "b": 'two'}`

Statements

Semicolon-separated	`expr; expr; expr`
Last one is returned	`a = 1; a + b`

Parentheses

Alter precedence order	`2 * (a + b)`
	`(a || b) && c`

Member Access

Field lookup	`aKey`
nested objects	`aKey.otherKey.key3`
Property lookup	`aKey['otherKey']`
	`aKey[keyVar]`
	`aKey['otherKey'].key3`
Array lookup	`anArray[42]`

Function Calls

Function calls	`aFunction()`
	`aFunction(42, 'abc')`
Method calls	`anObject.aFunction()`
	`anObject[fnVar]()`

Operators

In order of precedence

Unary	`-a`
	`+a`
	`!done`
Multiplicative	`a * b`
	`a / b`
	`a % b`
Additive	`a + b`
	`a - b`
Comparison	`a < b`
	`a > b`
	`a <= b`
	`a >= b`
Equality	`a == b`
	`a != b`
	`a === b`
	`a !== b`
Locical AND	`a && b`
Logical OR	`a || b`
Ternary	`a ? b : c`
Assignment	`aKey = val`
	`anObject.aKey = val`
	`anArray[42] = val`
Filters	`a | filter`
	`a | filter1 | filter2`
	`a | filter:arg1:arg2`

What is allowed ?

Original source

05-expressions

https://github.com/bhovhannes/trainings/blob/master/angular/examples/05-expressions/index.html

Let’s play around with example.

4.

Filters

{{ 'awesome' | uppercase }}

Using filters in your view

```
{{ 123.456789 | number }}        <!-- outputs 123.456789 -->
```

with params:

```
{{ 123.456789 | number:2 }}      <!-- outputs 123.46 -->
```

with multiple params:

```
{{ [‘Kevin’, ’Bob’, ’Dave’] | number:sortFn:true }}
```

multiple filters can be applied using |:

```
{{ 123.456789 | number:2 | currency }}      <!-- outputs $123.46 -->
```

Built-in filters

https://docs.angularjs.org/api/ng/filter

filter
Selects a subset of items from array and returns it as a new array.

currency
Formats a number as a currency (ie $1,234.56). When no currency symbol is provided, default symbol for current locale is used.

number
Formats a number as text.

date
Formats date to a string based on the requested format.

json
Allows you to convert a JavaScript object into JSON string.

lowercase
Converts string to lowercase.

uppercase
Converts string to uppercase.

limitTo
Creates a new array or string containing only a specified number of elements. The elements are taken from either the beginning or the end of the source array, string or number, as specified by the value and sign (positive or negative) of limit. If a number is used as input, it is converted to a string.

orderBy
Orders a specified array by the expression predicate. It is ordered alphabetically for strings and numerically for numbers.

Accessing filters from JS

https://github.com/bhovhannes/trainings/blob/master/angular/examples/05-filters-in-js/index.html

The **$filter** service can be used to retrieve filter by name.

```
<!DOCTYPE html>
<html ng-app>
    <head>
        <title>05-filters-in-js</title>
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.27
/angular.js"></script>
        <script>
            function MyController($scope, $filter) {
                $scope.getTodayAsString = function() {
                    return $filter('date')(new Date(), 'd-M-y');
                };
            }
        </script>
    </head>
    <body ng-controller="MyController">
        Today is {{ getTodayAsString() }}
    </body>
</html>
```

Custom filters

Creating your own filter is easy!

Just register a new filter factory function with your module:

```
angular.module(‘myModule’).filter(‘myFilter’, function() {
    return function (value, filterParam1, …, filterParamN ) {
        var filterResult;
        filterResult = …;
        return filterResult;
    };
});
```

Custom filters

Angular executes the filter only when the inputs to the function change.

Best practice: the filter function should be a pure function.

06-filter-hexToRgba

https://github.com/bhovhannes/trainings/blob/master/angular/examples/06-filter-hexToRgba/index.html

5.

Providers

https://docs.angularjs.org/guide/providers

5 recipes

Injector

In Angular, objects are instantiated and wired together automatically by the injector service - *$injector*

The injector creates two types of objects, services and specialized objects.

objects whose API is defined by the developer writing the service

objects that conform to a specific Angular framework API:

- *controllers,*
- *directives,*
- *filters,*
- *animation*

Provider recipes

https://docs.angularjs.org/guide/providers

https://code.angularjs.org/1.4.8/docs/api/auto/service/$provide

You should tell injector how to create objects.

To do that, you should register a “*recipe*” for creating your object with the provider.

There are 5 recipe types:

- *provider*
- *factory*
- *service*
- *value*
- *constant*

Value recipe

https://docs.angularjs.org/guide/providers

Use value recipe if you want to share the same single value (of any type) across your module.

Registration

```
var myApp = angular.module('myApp', []);
myApp.value('clientId', 'a12345654321x');
```

Usage

```
myApp.controller('DemoController', ['clientId', function(clientId) {
    console.log( clientId );
}]);
```

Factory recipe

https://docs.angularjs.org/guide/providers

Factory recipe is more powerful version of Value recipe.

Use it if you want to do something before creating your value, or if your value depends on other values.

Registration

```
var myApp = angular.module('myApp', []);
myApp.factory('clientId', function () {
     var clientIdPrefix = 'a';
     return clientIdPrefix + '12345654321x';
});
```

Usage is the same as in case of value recipe

Factory recipe

https://docs.angularjs.org/guide/providers

You can inject other services when using factory recipe

Registration

```
var myApp = angular.module('myApp', []);
myApp.factory('apiToken', ['clientId', function (clientId) {
    var tokenPrefix = 'secret';
    return tokenPrefix + clientId;
}]);
```

Service recipe

https://docs.angularjs.org/guide/providers

Consider this:

```
function RocketLauncher(apiToken) {
    this.launchedCount = 0;
    this.launch = function() {
        // Make a request to the remote API and include the apiToken ...
        this.launchedCount++;
    }
}
```

Using Factory recipe:

```
myApp.factory('rocketLauncher', ["apiToken", function(apiToken) {
    return new RocketLauncher(apiToken);
}]);
```

The same using Service recipe:

```
myApp.service('rocketLauncher', ["apiToken", RocketLauncher]);
```

Service recipe

https://docs.angularjs.org/guide/providers

- The Service recipe produces a service just like the Value or Factory recipes, but it does so by invoking a constructor with the *new* operator.

  The constructor can take zero or more arguments, which represent dependencies needed by the instance of this type.

- Use Service recipe when you want to inject instance of a certain type, and don’t need to control how this instance is being created.

Module Initialization Phases

https://github.com/bhovhannes/trainings/blob/master/angular/examples/07-module-phases/index.html

Modules have 2 initialization phases: **config** and **run**

`myApp.config( ... )` - allows to configure services before they are used

`myApp.run( ... )` - services can be used here and cannot be reconfigured anymore

Provider recipe

https://docs.angularjs.org/guide/providers

Provider recipe is the core recipe type and Value, Factory and Service recipes are just syntactic sugar on top of it.

It is the most verbose recipe with the most abilities, but for most services it's overkill.

Use the Provider recipe only when you want to expose an API for application-wide configuration that must be made before the application starts.

Provider recipe

https://docs.angularjs.org/guide/providers

The Provider recipe is syntactically defined as a custom type that implements a $get method.

This method is a factory function just like the one as in the Factory recipe.

In fact, if you define a Factory recipe, an empty Provider type with the $get method set to your factory function is automatically created under the hood.

08-providers

https://github.com/bhovhannes/trainings/blob/master/angular/examples/08-providers/index.html

Constant recipe

https://docs.angularjs.org/guide/providers

Constant recipe = Value recipe + availability in config phase

Registration

```
var myApp = angular.module('myApp', []);
myApp.constant(API_PASSWORD, 'pass');
```

Usage

```
myApp.controller('DemoController', ['API_PASSWORD', function(API_PASSWORD) {
    console.log( API_PASSWORD );
}]);
```

In config phase

```
myApp.config(['API_PASSWORD', 'MyProvider', function(API_PASSWORD, MyProvider) {
    //...
}]);
```

6.

Bootstrap process

https://docs.angularjs.org/guide/bootstrap

ng-app
and
angular.bootstrap

Automatic Initialization

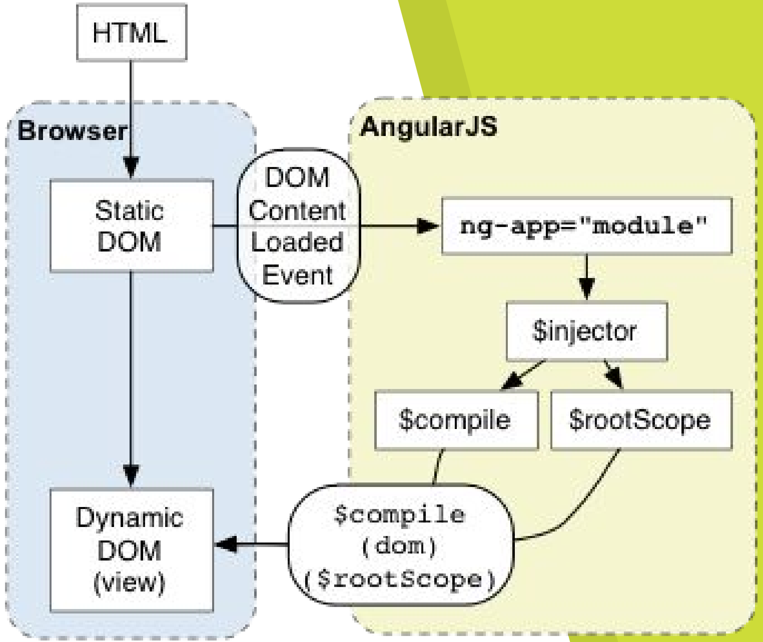
https://docs.angularjs.org/guide/bootstrap

Angular initializes automatically upon ***DOMContentLoaded*** event or when the ***angular.js*** script is evaluated if at that time ***document.readyState*** is set to '***complete***'.

At this point Angular looks for the **ng-app** directive which designates your application root.

If the **ng-app** directive is found then Angular will:

1) load the **module** associated with the directive.
2) create the application **injector**
3) compile the DOM treating the **ng-app** directive as the root of the compilation. This allows you to tell it to treat only a portion of the DOM as an Angular application.



Manual Initialization

https://github.com/bhovhannes/trainings/blob/master/angular/examples/09-bootstrap/index.html

This is the sequence that your code should follow:

1) After the page and all of the code is loaded, find the root element of your AngularJS application, which is typically the root of the document.
2) Call angular.bootstrap to compile the element into an executable, bi-directionally bound application.

Example:

```
<body>
    <div id='my-app'></div>
    <script>
        var el = document.getElementById('my-app');
        angular.bootstrap(el, ['module1', 'module2']);
    </script>
</body>
```

7.

Intro to ngMock

angular.mock.module
and
angular.mock.inject

Overview

The *ngMock* module provides support to inject and mock Angular services into unit tests.

In addition, *ngMock* also extends various core *ng* services such that they can be inspected and controlled in a synchronous manner within test code.

ngMock is distributed separately as **angular-mocks.js**

angular.mock.module

http://www.bradoncode.com/blog/2015/05/24/ngmock-fundamentals-angularjs-unit-testing/

In tests, we can’t use ng-app because we don’t load html page into browser.

So how can we tell Angular that we need a certain module?

The *angular.mock.module()* provides a mechanism to initialise Angular modules.

angular.mock.module

http://www.bradoncode.com/blog/2015/05/24/ngmock-fundamentals-angularjs-unit-testing/

Register an existing module named ‘demo’:

```
angular.mock.module(‘demo’)
```

Register multiple existing modules:

```
angular.mock.module(‘demo’, ‘clock’)
```

Register completely new module:

```
angular.mock.module( function ($provide) {
    $provide.constant(‘pi’, 3.14);
} );
```

angular.mock.inject

http://www.bradoncode.com/blog/2015/05/27/ngmock-fundamentals-angularjs-testing-inject/

The *angular.mock.inject()* works in a pair with *angular.mock.module()*.

The *angular.mock.inject()* allows to inject instances into our functions. It

1) scans for service definitions in all modules specified by *angular.mock.module()*,
2) creates an instance of found service (if not created previously),
3) pass the instance to our function as an argument.

8.

Dependency Injection

More about *$injector*

DI for five-year-olds

When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.

What you should be doing is stating a need, “I need something to drink with lunch”, and then we will make sure you have something when you sit down to eat.

Annotation types

Annotation by Inference

```
    injector.invoke( function($http, greeter){} );
```

Explicit annotation

```
    var fn = function($http, greeter){};
    fn.$inject = [‘$http’, ‘greeter’];
    injector.invoke( fn );
```

Inline annotation

```
    injector.invoke( [
        ‘$http’, ‘greeter’, function($http, greeter){}
    ] );
```

$injector API

https://github.com/bhovhannes/trainings/blob/master/angular/examples/10-injector/

`annotate( function | array )` - returns an array of names of services that will be injected into the function at the time of invocation

`get( name:`*`string`*` )` - returns an instance of the service

`has( name:`*`string`*` )` - checks if the given service exists

$injector API

https://github.com/bhovhannes/trainings/blob/master/angular/examples/10-injector/

`instantiate( Type:`*`function`*`, [locals:`*`Object`*`] )` - creates a new instance of the JS *Type*.

It takes a constructor and invokes the *new* operator with all the arguments specified.

If *locals* are given, they will be added to constructor arguments.

`invoke( fn:`*`function`*`, [self:`*`Object`*`] [locals:`*`Object`*`] )` - calls *fn* with supplied arguments.

If *locals* are given, they will be added to *fn* arguments.

9.

Scopes

More about *scopes*

What are scopes?

https://github.com/bhovhannes/trainings/blob/master/angular/examples/11-scopes/

Scope is an object that refers to the application model.

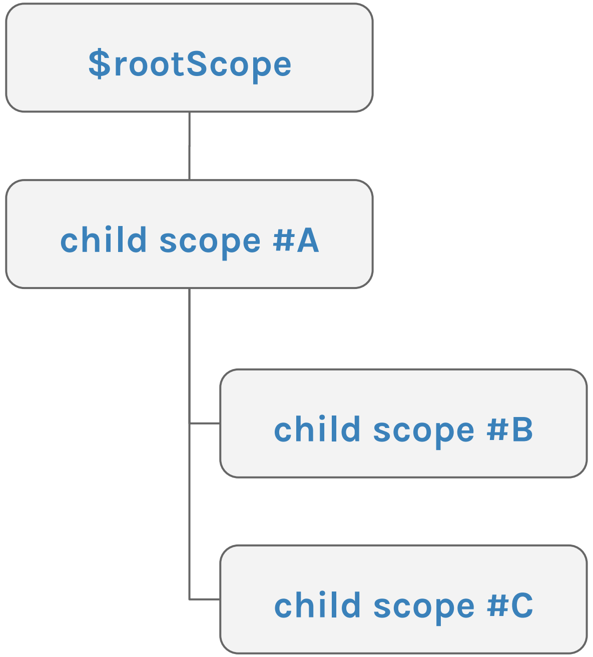
Scope is an execution context for expressions.

Scopes are arranged in hierarchical structure which mimic the DOM structure of the application.

Scopes can watch expressions and propagate events.

Scope structure

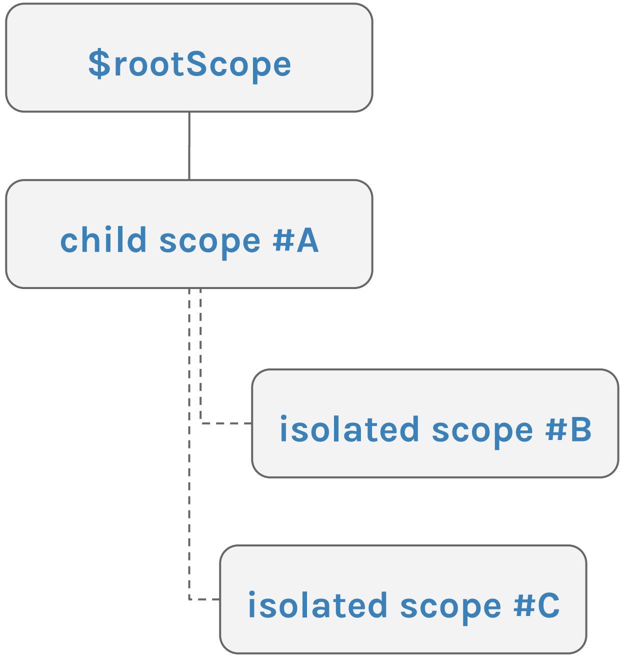
https://github.com/bhovhannes/trainings/blob/master/angular/examples/11-scopes/child-scopes.html



A "child scope" (prototypically) inherits properties from its parent scope.

Scope structure

https://github.com/bhovhannes/trainings/blob/master/angular/examples/11-scopes/isolated-scopes.html



An "isolated scope" does not inherit any properties from its parent scope and is completely isolated.

Walking through scopes

https://github.com/bhovhannes/trainings/blob/master/angular/examples/11-scopes/child-scopes.html

Don’t walk through scopes

`scope.$parent` - reference to parent scope

`scope.$root` - reference to root scope

`scope.$id` - unique identifier of the scope

Scope has a number of $$-prefixed properties, which Angular uses to be able to fully traverse through scope hierarchy.

Although you can use them, this is strongly discouraged, as they may change in next version of Angular.

Retrieving scopes from the DOM

Scopes are attached to the DOM, and can be retrieved for debugging purposes.

`angular.element($0).scope()` - returns scope associated with the element

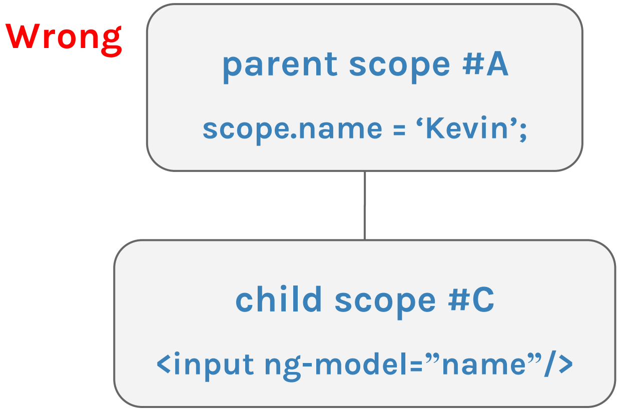
`angular.element($0).isolateScope()` - returns isolated scope associated with the element

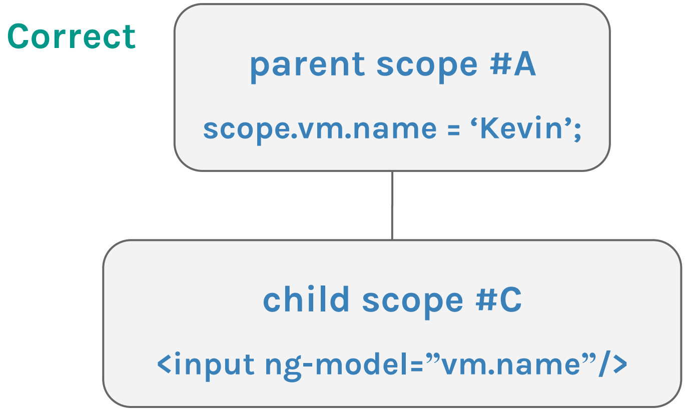
$scope.vm best practice

https://github.com/bhovhannes/trainings/blob/master/angular/examples/11-scopes/always-use-vm.html

https://github.com/bhovhannes/trainings/blob/master/angular/examples/11-scopes/issue-without-vm.html

Don’t put variables directly in scope.
Declare scope.vm object and put variables there.

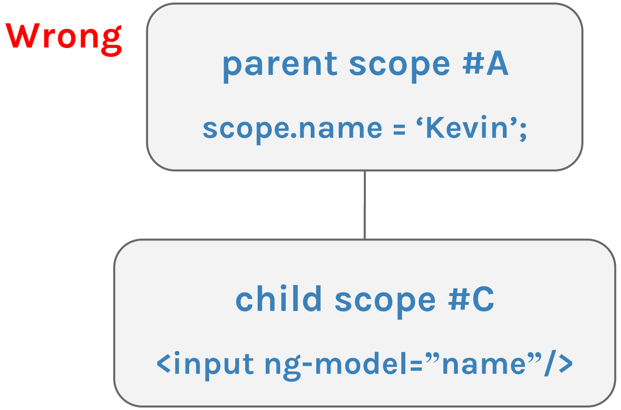


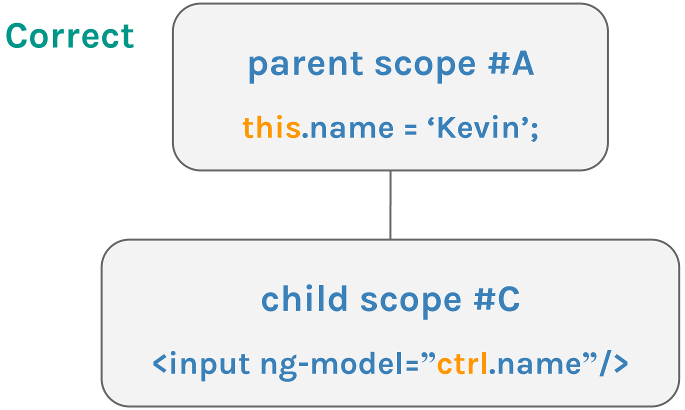


controllerAs best practice

https://github.com/bhovhannes/trainings/blob/master/angular/examples/11-scopes/controller-as-syntax.html

Don't reinvent wheel and use **$scope.vm**!
Use ***controllerAs*** syntax where it is possible.





Events

https://github.com/bhovhannes/trainings/blob/master/angular/examples/12-events/events.html

It is possible to send events (holding arbitrary data) from one scope to another.

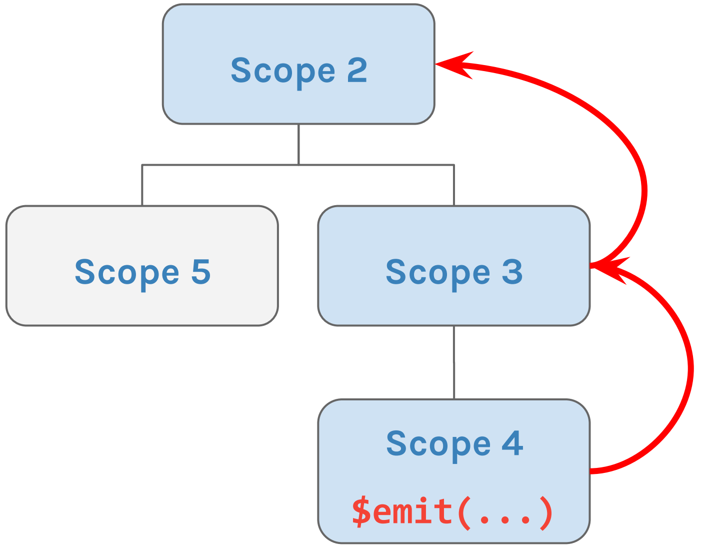
That can be accomplished using either **$emit** and **$broadcast** methods available on scope objects.

Scope can subscribe to particular event using **$on** method.

$on method returns deregistration function, which will remove the event listener when called.

Events - $emit

https://github.com/bhovhannes/trainings/blob/master/angular/examples/12-events/events.html

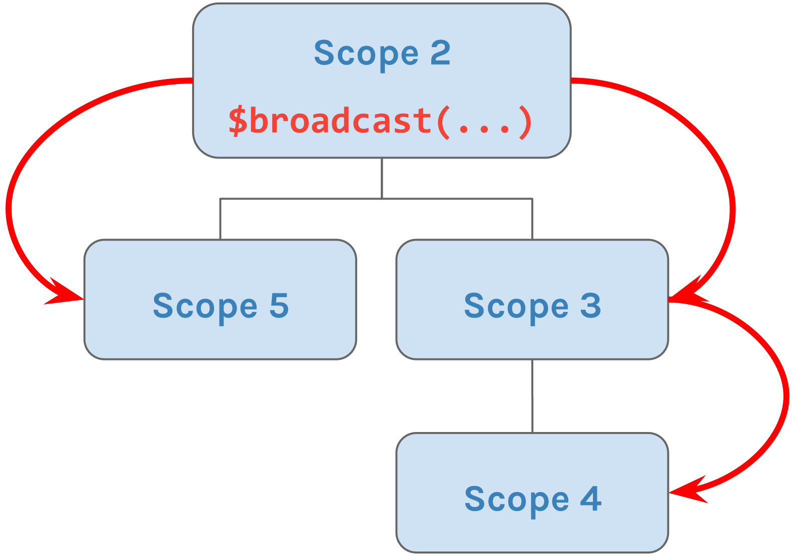


$emit sends event upwards through scope hierarchy.

Each of receiver scopes can stop propagation of $emit-ed event further through scope hierarchy.

Events - $broadcast

https://github.com/bhovhannes/trainings/blob/master/angular/examples/12-events/events.html



$broadcast sends event downwards through scope hierarchy to all scopes.

It is impossible to stop propagation of **$broadcast**-ed event.

$rootScope as an event bus

https://github.com/bhovhannes/trainings/blob/master/angular/examples/12-events/eventbus.html

Since **$rootScope** is always available and can be injected everywhere, it can act as an **event bus**.

Different application components can use it to communicate to each other via **$emit**-ing events on **$rootScope**.

While it is not the best way for communication between application components, it still has its value.

Watching for expression changes

https://github.com/bhovhannes/trainings/blob/master/angular/examples/13-digest/watch.html

You can monitor changes of expression value using **watchers**.

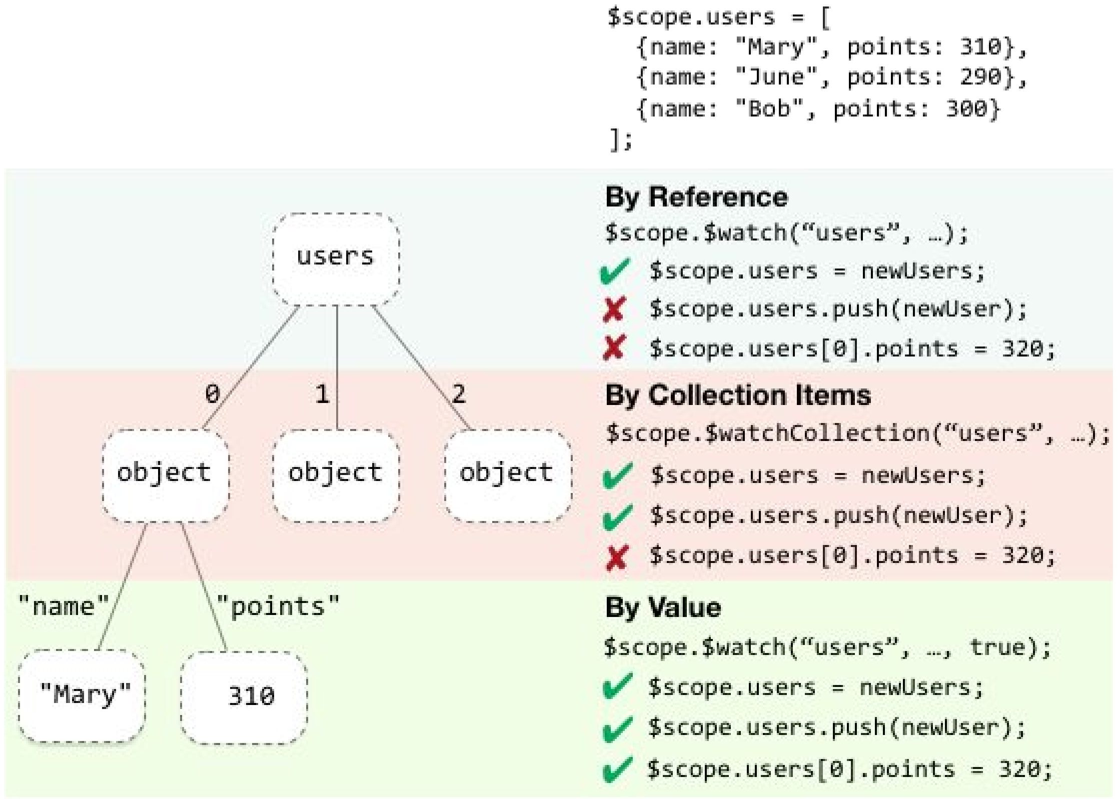
There are 3 ways to register a watcher on some scope:

```
$watch(expression, listener, [objectEquality])

$watchGroup(expressions, listener)

$watchCollection(obj, listener)
```

All methods return deregistration function, which will remove the watcher when called.

Scope $watch strategies

https://docs.angularjs.org/guide/scope



```
$scope.users = [
  {name: "Mary", points: 310},
  {name: "June", points: 290},
  {name: "Bob", points: 300}
];
```

By Reference

```
$scope.$watch("users", …);
✓ $scope.users = newUsers;
✗ $scope.users.push(newUser);
✗ $scope.users[0].points = 320;
```

By Collection Items

```
$scope.$watchCollection("users", …);
✓ $scope.users = newUsers;
✓ $scope.users.push(newUser);
✗ $scope.users[0].points = 320;
```

By Value

```
$scope.$watch("users", …, true);
✓ $scope.users = newUsers;
✓ $scope.users.push(newUser);
✓ $scope.users[0].points = 320;
```

Scope $watch performance

https://docs.angularjs.org/guide/scope

- `$watch(expr, listener)` is the fastest.
  If you need to have a watcher, try to use this one.

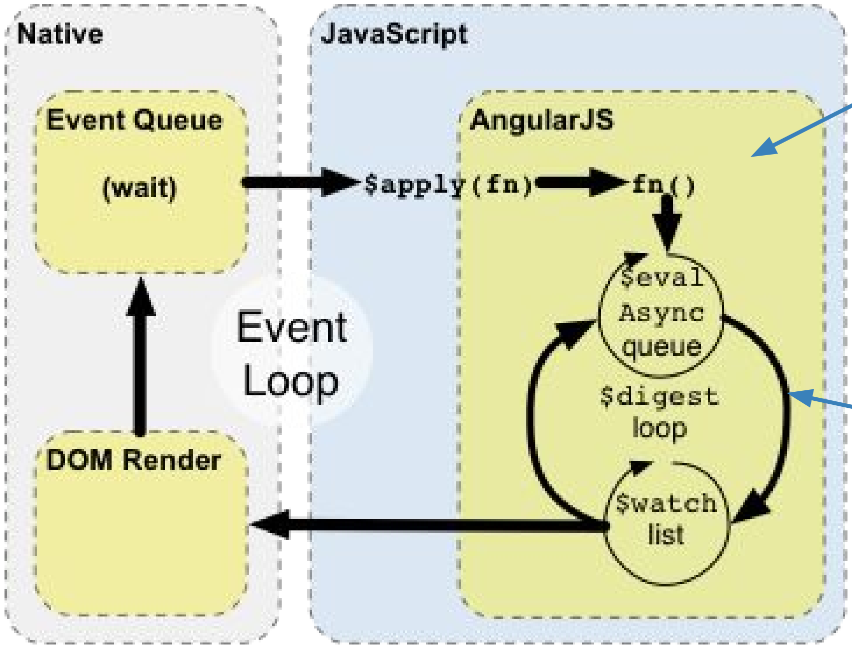
- `$watch(expr, listener, true)` is the slowest.
  Avoid using it.

- If `$watch(expr, listener)` is not enough, you can try to use `$watchCollection(obj, listener)`.

  Speed will be somewhere in the middle.

Integration with the browser event loop

https://docs.angularjs.org/guide/scope



Angular execution context

max 10 iterations

Entering Angular execution context

https://github.com/bhovhannes/trainings/blob/master/angular/examples/13-digest/apply.html

- You can use **$apply()** to enter the Angular execution context from JavaScript.
- In most places (controllers, services) **$apply** has already been called for you by the directive which is handling the event.
- An explicit call to **$apply** is needed only when implementing custom event callbacks, or when working with third-party library callbacks.

Exceptions during $apply

https://github.com/bhovhannes/trainings/blob/master/angular/examples/13-digest/apply-exceptions.html

Wrong

```
$scope.someAction();
$scope.$apply();
```

Correct

```
$scope.$apply(function() {
     $scope.someAction();
});
```

all exceptions in `someAction` will be handled by `$exceptionHandler` service

$apply vs $digest

https://github.com/bhovhannes/trainings/blob/master/angular/examples/13-digest/apply-vs-digest.html

```
function $apply(expr) {
    try {
        return $eval(expr);
    } catch (e) {
        $exceptionHandler(e);
    } finally {
        $rootScope.$digest();
    }
}
```

$apply() triggers **$digest** on **$rootScope**, so it is slower than **$scope.$digest()** call.

Evaluating expressions

https://docs.angularjs.org/guide/scope

`$eval(expression, [locals])`

exceptions will not be caught

`$evalAsync(expression, [locals])`

a) if called outside of *$digest* cycle, triggers a new one

b) exceptions are being caught by *$exceptionHandler* service

Creating and destroying scopes

https://docs.angularjs.org/guide/scope

`$new(isolate, [parent])`

creates a new scope

`$destroy()`

a) broadcasts `$destroy` event on the scope
b) removes scope with its children from its parent scope
c) prepares scope for garbage collection

10.

Controllers

Add behavior to scopes

Controllers

https://github.com/bhovhannes/trainings/blob/master/angular/examples/14-controllers/

A controller is defined by a JavaScript constructor function.

```
function DemoController($scope) {
     $scope.msg = ‘Hello world’;
}

var module = angular.module(‘demo’);
module.controller(‘DemoCtrl’, DemoController);
```

controller name

Creating controllers

ng-controller directive instantiates a new Controller object, using the specified Controller's constructor function.

Angular also creates new controllers when *controller* property in directive definition is set to a function.

A new child scope is created and made available as an injectable parameter to the Controller's constructor function as *$scope*.

If the controller has been attached using the *controller as* syntax then the controller instance will be assigned to a property on the new scope.

Creating controllers manually

https://github.com/bhovhannes/trainings/blob/master/angular/examples/14-controllers/

Usually useful for controller tests, that involves injecting the *$rootScope* and *$controller* services.

```
var scope;
beforeEach( angular.mock.inject(function($rootScope, $controller) {
    scope = $rootScope.$new();
    $controller('DemoCtrl', {$scope: scope});
}));
```

Controller best practices

- Try to keep your controllers as small as possible.
- Do not use controllers to manipulate DOM, as it greatly affects controller testability.
- Do not use controllers to manage the life-cycle of other components (for example, to create service instances).
- Do not use controllers to share code or state across controllers. Use angular services instead.

11.

Directives

Writing custom directives

What are Directives?

https://docs.angularjs.org/guide/directive

Directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's *HTML compiler* ($compile service) to attach a specified behavior to that DOM element (e.g. via event listeners), or even to transform the DOM element and its children.

When Angular bootstraps your application, the HTML compiler traverses the DOM matching directives against the DOM elements.

Matching directives

https://docs.angularjs.org/guide/directive

We say an element **matches** a directive when the directive is part of its declaration.

These inputs both match **ngModel** directive:

```
<input ng-model="foo"/>
<input data-ng-model="foo"/>
```

And this element matches **person** directive:

```
<person>{{ name }}</person>
```

Normalization

https://docs.angularjs.org/guide/directive

Angular **normalizes** an element's tag and attribute name to determine which elements match which directives.

Normalizing means converting to camelCase.

The normalization process is as follows:

a) Strip `x-` and `data-` from the front of the element/attributes.

b) Convert the `:`, `-`, or `_`-delimited name to **camelCase**.

Normalization example

https://docs.angularjs.org/guide/directive

The following forms are all equivalent and match the **ngBind** directive:

```
<div ng-controller="Controller">
  <span ng-bind="name"></span>
  <span ng:bind="name"></span>
  <span ng_bind="name"></span>
  <span data-ng-bind="name"></span>
  <span x-ng-bind="name"></span>
</div>
```

Directive types

https://docs.angularjs.org/guide/directive

Directives can be matched based on element names (E), attributes (A), class names (C), as well as comments (M):

```
<my-dir></my-dir>

<span my-dir="exp"></span>

<!-- directive: my-dir exp -->

<span class="my-dir: exp;"></span>
```

Creating directives

https://docs.angularjs.org/guide/directive

To register a directive, you use the `module.directive()`.

`module.directive()` takes the **normalized** directive name followed by a **factory function**.

This factory function should return an object with the different options (**DDO - Directive Definition Object**) to tell `$compile` how the directive should behave when matched.

Always prefix directive you create to avoid collisions with other directives.

Directive Definition Object (DDO)

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

The directive definition object provides instructions to the compiler. The attributes are:

`multiElement`	`restrict`
`priority`	`templateNamespace`
`terminal`	`template`
`scope`	`templateUrl`
`bindToController`	`replace`
`controller`	`transclude`
`require`	`compile`
`controllerAs`	`link`

priority

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/priority.html

Priority is used to specify order in which directive is being compiled.

When there are multiple directives applied to the same element, directives with greater numerical priority are compiled first.

The compilation order of directives with the same priority is undefined.

By default `priority: 0`.

terminal

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/terminal.html

Terminal option is used when you need to prevent compilation of other directives applied to the same element.

When there are multiple directives applied to the same element, directives with numerical priority less than that of terminal directives are skipped.

By default terminal: false.

scope

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/scope.html

Scope option is used to specify how scope should be created for directive.

scope: true - prototypically inherited child scope will be created

scope: false - no scope will be created, directive will use first parent scope as its scope

scope: { } - a new isolated scope will be created

passing parameters to directive

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/passing-params.html

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/passing-params-expr.html

When *scope:* { } is used, there are 3 ways to pass a value from outside to directive:

localName: '@attr' - one-way binding between *localName* in directive scope and *attr* in outer scope.

localModel: '=model - two-way binding between *localModel* in directive scope and *model* in outer scope.

localFn: '&fn' - calling *localFn* in directive scope will evaluate expression *fn*. You may also specify locals for evaluation.

scope limitations

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/scope-limitation.html

Only one scope can be attached to a single DOM node. That causes the following limitations:

no scope + no scope => will use their parent's scope

child scope + no scope => will share one single child scope

child scope + child scope => will share one single child scope

isolated scope + no scope => The isolated directive will use it's own created isolated scope. The other directive will use its parent's scope

isolated scope + child scope => Won't work!

isolated scope + isolated scope => Won't work!

replace

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/replace.html

replace: true - the template will replace the directive's element.

replace: false - the template will replace the contents of the directive's element.

By default **replace: false**.

restrict

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/restrict.html

E - Element name (default): `<my-directive></my-directive>`

A - Attribute (default): `<div my-directive="exp"></div>`

C - Class: `<div class="my-directive: exp;"></div>`

M - Comment: `<!-- directive: my-directive exp -->`

By default **restrict**: ‘EA’.

template

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/template.html

Template option is used to specify a directive template either as a string (of HTML code for directive) or as a function which returns string (of HTML code for directive)

```
template: ‘<div>aaa</div>’

template: function(tElement, tAttrs) {
     return ‘<div>aaa</div>’;
}
```

templateUrl

TemplateUrl option is used to specify a url for directive template either as a string (url to a file with directive template code) or as a function which returns string (url to a file with directive template code)

```
templateUrl: ‘../templates/demo.html’

templateUrl: function(tElement, tAttrs) {
     return ‘../templates/demo.html’;
}
```

templateNamespace

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/templateNamespace.html

templateNamespace option is used to specify a document type used by the markup in the directive template.

Possible values are:

> *html* - All root nodes in the template are HTML. Root nodes may also be top-level elements such as <svg> or <math>.
>
> *svg* - The root nodes in the template are SVG elements.
>
> *math* - The root nodes in the template are MathML elements.

By default templateNamespace: 'html'.

controller

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/controller.html

```
controller: ['$scope', function($scope) {}]
```

The controller is instantiated before the pre-linking phase and can be accessed by other directives.

This allows the directives to communicate with each other and augment each other's behavior.

The controller is injectable (and supports bracket notation) with the following locals:

- **$scope** - Current scope associated with the element
- **$element** - Current element
- **$attrs** - Current attributes object for the element
- **$transclude** - A transclude linking function

bindToController

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/bindToController.html

When ***scope:* { }** and ***controllerAs*** are used, **bindToController: true** will allow a component to have its properties bound to the controller, rather than to scope.

When the controller is instantiated, the initial values of the isolate scope bindings are already available.

compile

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/compile.html

The *compile* function deals with **transforming the template DOM**. Since most directives do not do template transformation, it is not used often.

```
compile: function(tElement, tAttrs) {
}
```

tElement - The element where the directive has been declared. It is safe to do template transformation on the element and child elements only.

tAttrs - Normalized list of attributes declared on this element shared between all directive compile functions.

Attributes

https://github.com/bhovhannes/trainings/blob/master/angular/examples/15-directives/attributes.html

https://code.angularjs.org/1.4.8/docs/api/ng/service/$compile#attributes

Attributes object is passed as a parameter in the **link()** or **compile()** functions. It allows:

1) access attribute values using their normalized names

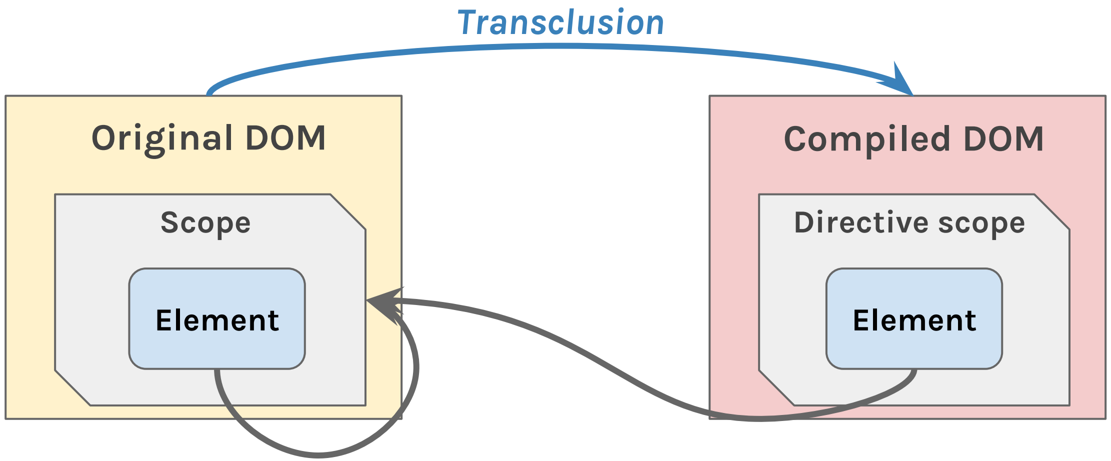
2) observe value changes of attributes containing interpolation via **$observe**

$observe() is the only way to easily get the actual value of attribute because during the linking phase the interpolation hasn't been evaluated yet and so the value is at this time set to **undefined**.

Transclusion

https://code.angularjs.org/1.4.8/docs/api/ng/service/$compile#transclusion

Transclusion is the process of extracting a collection of DOM elements from one part of the DOM and copying them to another part of the DOM, while maintaining their connection to the original AngularJS scope from where they were taken.



ngTransclude

https://github.com/bhovhannes/trainings/blob/master/angular/examples/16-transclusion/ngTransclude.html

transclude: true - tells that directive requests transclusion.

Transclusion is often used with **ngTransclude** directive.

ngTransclude marks the insertion point for the transcluded DOM of the nearest parent directive that uses transclusion.

ngTransclude automatically cares about moving DOM elements and about maintaining connection of transcluded content to its original scope.

Transclusion function

https://github.com/bhovhannes/trainings/blob/master/angular/examples/16-transclusion/transclusion-function.html

When a directive requests transclusion, the compiler extracts its contents and provides a **transclusion function** (**$transclude**) to the directive's **link** function and **controller**.

$transclude is a special linking function that will return the compiled contents linked to a new **transclusion scope**.

If you want to manually control the insertion and removal of the transcluded content in your directive then you must use **$transclude**.

Transclusion function

https://github.com/bhovhannes/trainings/blob/master/angular/examples/16-transclusion/transclusion-function.html

When you call $transclude you can pass in a clone attach function:

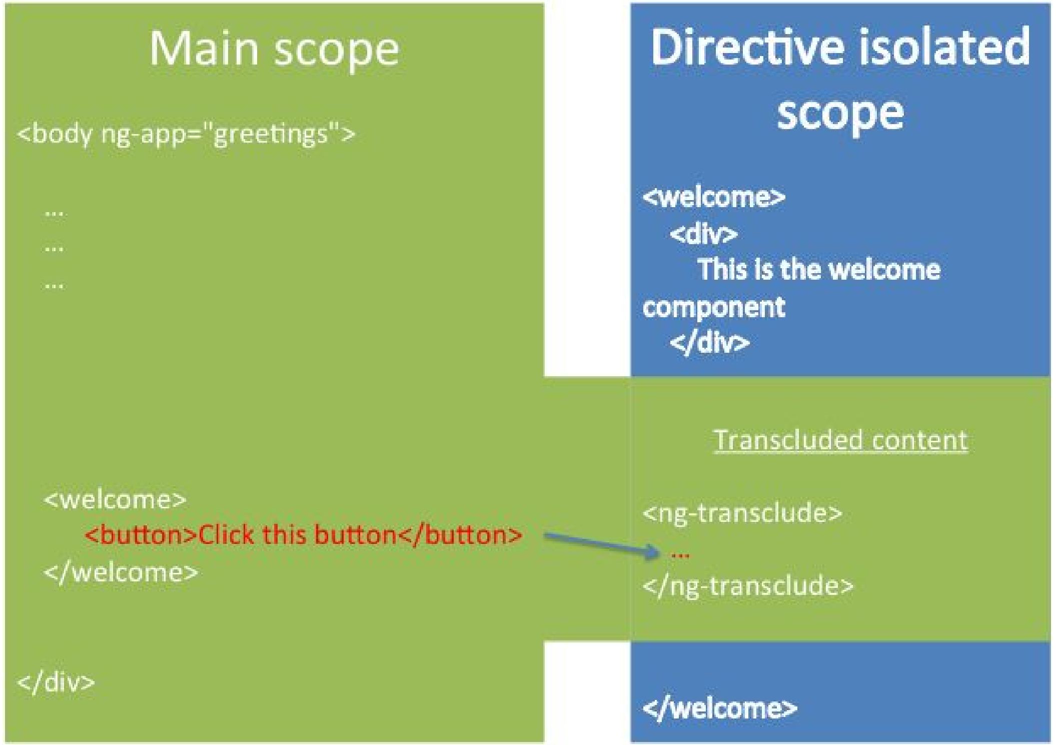
```
function(clone, transclusionScope) { ... }
```

clone - a fresh compiled copy of your transcluded content. Is array of DOM nodes.

transclusionScope - the newly created transclusion scope, to which the clone is bound.

Transclusion scopes

https://code.angularjs.org/1.4.8/docs/api/ng/service/$compile#transclusion



Transclusion scopes

https://github.com/bhovhannes/trainings/blob/master/angular/examples/16-transclusion/transclusion-scope.html

When you call a transclude function it returns a DOM fragment that is pre-bound to a **transclusion scope**.

Transclusion scope **is a child of the directive's scope**. It gets destroyed when the directive scope gets destroyed.

Transclusion scope **prototypically inherits the properties of the scope from which it was taken.**

CREDITS

Special thanks to all the people who made and released these awesome resources for free:

- Simple line icons by Mirko Monti
- E-commerce icons by Virgil Pana
- Streamline iconset by Webalys
- Presentation template by SlidesCarnival