AngularJS Training

# List three ways to communicate between modules of your app.

1. Services
2. Events/Watches: scope.$on and scope.$broadcast
3. Assigning models on $rootScope
4. Between controllers using $parent, $$childHead, $$nextSibling
5. Between controllers using ControllerAs

$$childHead is the isolated scope of a directive

# Which means of communication between modules of your application are easily testable?

1. Controllers – in unit testing, controllers are usually instantiated.
2. Services – services are injected, so in a test a real or mock service can be used.
3. Events – for testing events on $rootScope, rootScope must be injected into the test.

# The most popular e2e testing tool for AngularJS is Protractor. There are also others which rely on similar mechanisms. Describe how e2e testing of AngularJS applications work.

The e2e tests are executed against a running app, that is a fully initialized system. They most often ***spawn a browser instance and involve the actual input of commands through the user interface***. The written code is evaluated by an automation program, such as a ***Selenium*** server (webdriver). That program sends commands to a browser instance, then evaluates the visible results and reports back to the user.

The ***assertions*** are handled by another library, for Protractor the default is ***Jasmine***. Before Protractor, there was a module called Angular Scenarios, which usually was executed through Karma, and is now deprecated. Should you want to e2e test hybrid apps, you could use another Selenium server, called Appium.

Testing can be handled manually, or it can be delegated to continuous integration servers, either custom or ones provided by Travis, SauceLabs, and Codeship.

# List a few ways to improve performance in an AngularJS app.

* ***Disable debug data***
  + Scopes cannot be appended to elements, so scope not accessible in console
  + In app.config 🡪 $compileProvider.debugInfoEnabled(false);
  + Testing tools like Protractor and Batarang need debug data to run, but you can disable debug data for production
* **Strict di mode**
  + Use ng-strict-di **(ex/** **<html ng-app=’myApp’ *ng-strict-di*> )**
  + With Strict di, you must ***explicitly annotate your injected functions and modules***
  + Ensures that the code will work even when minified/uglified
  + Performance is improved because Angular does not have to dynamically discover a function’s dependencies.

# What is $rootScope and how does it relate to $scope?

$rootScope is the parent object of all $scope Angular objects created in a web page.

# Glossary of terms

## $apply

* Propagates model changes from outside the Angular realm.
* A digest is also called after the apply.
* Useful when integrating 3rd pty libs

## $broadcast

* Dispatches events down the scope’s tree descendants.
* $scope.$on listeners (which are bound to this event) will be invoked.

## Child Scope vs. Isolated Scope

* A child scope prototypically inherits from the parent scope
* An isolated scope does not inherit from parent
* Scope is the glue between controller and view

## $compile

This service converts a html string in a fully functional DOM element. The resulting DOM would have all linking, events working just like a DOM element. This uses $parse internally for evaluating expressions. e.g usage of $compile would be

|  |  |
| --- | --- |
| 1  2 | var html = '<div ng-click='clickme();'>{{text}}</div>';  $compile(html)($scope); |

$compile is mostly used inside custom directives and doesn’t have much use outside.

## $digest

$scope.$digest iterates thru all watches

## $emit

* Dispatches events up the scope’s tree ancestors.
* $scope.$on listeners (which are bound to this event) will be invoked.
* An emit (vs. broadcast) may have a slight performance increase if you need to communicate with a modal window service, or a modal window directive)

## Directives

### Phases

#### Initialization, compilation and linking phases

* **Initialization**: this happens when a directive is found for the first time in the DOM tree traversal
* **Compilation**: Angular manipulates the DOM node(s) for each node the directive appears on. It also has a chance to modify the DOM node before a scope is attached to it;
* **Linking**: Angular attaches event listeners to the Html template to make it interactive, and attaches a scope to the directive. Linking takes place AFTER compilation of the Html template.
  + ***Controller & Pre-link***: gets executed before post-link functions. The controller facilitates communication btwn directives. If there are nested directives, all pre-link functions get executed first, then post-link from bottom up.
  + ***Post-link***: gets executed from bottom up, after pre-link functions have executed.

\*\* in most cases, developers use just the controller and post-link functions.

\*\* using the “require:[‘^^myCtrl1’, ‘myCtrl2’]” option allows the link function to interact with a controller

\*\* ***BEST PRACTICE***: Use controller when you need to expose an API to other directives.

## Order of execution

(see <https://www.toptal.com/angular-js/angular-js-demystifying-directives> )

Ex/

<div parentDir>

<div childDir>

<div grandChildDir>

</div>

</div>

</div>

The execution order of the functions is as follows:

* **Compile Phase**
  + **Compile Function**: parentDir
  + **Compile Function**: childDir
  + **Compile Function**: grandChildDir
* **Controller & Pre-Link Phase**
  + **Controller Function**: parentDir
  + **Pre-Link Function**: parentDir
  + **Controller Function**: childDir
  + **Pre-Link Function**: childDir
  + **Controller Function**: grandChildDir
  + **Pre-Link Function**: grandChildDir
* **Post-Link Phase**
  + **Post-Link Function**: grandChildDir
  + **Post-Link Function**: childDir
  + **Post-Link Function**: parentDir

## $injector

Used to retrieve object instances as defined by provider, instantiate types, invoke methods, and load modules.

* Annotations are needed for dependency injection (JavaScript does not have annotations).
  + 3 ways of annotating functions
    - Inferred (does not work if code is minified/obfuscated) - $injector.invoke(function(serviceA){});
    - annotated – myFunction.$inject = [‘ServiceA’]
    - inline annotation - $injector.invoke( **[**‘ServiceA’, function(ServiceA){} **]** );

## Interpolation and data-binding

Interpolation markup with embedded [expressions](https://docs.angularjs.org/guide/expression) is used by Angular to provide data-binding to text nodes and attribute values.

An example of interpolation is shown below:

<a ng-href="img/{{username}}.jpg">Hello {{username}}!</a>

### How text and attribute bindings work

During the compilation process the [compiler](https://docs.angularjs.org/api/ng/service/$compile) uses the [$interpolate](https://docs.angularjs.org/api/ng/service/$interpolate) service to see if text nodes and element attributes contain interpolation markup with embedded expressions.

If that is the case, the **compiler adds an interpolateDirective to the node and registers** [**watches**](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) on the computed interpolation function, which will update the corresponding text nodes or attribute values as part of the normal [digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) cycle.

\*\*\* The interpolateDirective sets up the watches in the preLink function

## $interpolate

This service is used to evaluate angular expressions. It knows how to process a string with embedded interpolation expressions. i.e. ***it’s like a simple string-based template language***.

You can run an entire string against a scope, and interpolate will give the result. e.g would be

|  |  |
| --- | --- |
|  | var string = 'My Name is {{name}}';  $scope.name = 'Manish';  $interpolate(string)($scope); //this will result in My Name is Manish |

## $parse

This service is used as a getter/setter for single variables only; it is concerned with individual expressions only (name, extension). It is a read-write service.

|  |  |
| --- | --- |
|  | $scope.text = 'abc';  $parse('text')($scope);  //this will result in abc  $parse('text').assign($scope,'xyz'); |

## ngBind vs. {{ }}

* ngBind replaces the ***text content*** of the Html element.
* ngBind is preferred over {{expression}} as the raw template expression might temporarily be displayed before Angular fully compiles the Html
* ngBind is an element attribute, so it makes the bindings invisible until the page loads
* ngCloak is an alternative to ngBind

## ngModel

* Binds the value of an html input control to the model
* It binds input, select, textarea html controls

## Scope Inheritance

***Ex/*** Scope Life Cycle when Ang compiles the ng-controller directive: (<https://www.youtube.com/watch?v=V4nocbBMaFU> )

* + <div ng-controller=”***emp***”> {{name}}

<div ng-controller=***’empDetails’***> {{name}} earns {{salary}} in {{dept}} </div>

</div>

1. This new ‘***emp’*** scope will be attached to the DOM, on the respective div element
2. Ang then creates a new ‘***emp’*** controller instance, which references the same scope object ‘emp’ on the DOM
3. The ‘emp’ controller will now get executed, and new ‘***name***’ prop is added to scope.
4. The compiler will evaluate the data binding expressions ({{name}}

* ‘***empDetails’*** is the inner block, and a new child scope is created under ‘emp’ par scope
  + The ***inner scope*** inherits all parent scope properties
  + Newly-created properties in ‘***empDetails’*** are isolated, but {{name}} is on the parent scope. If {{name}} is overwritten in child controller, it will only update in the inner scope (the parent {{name}} will not change).
* ***Console notes*** (open f12 console, elem tab, click on ng-controller=”emp” outer scope):
  + $0 displays the curr scope object (i.e. whatever elem is selected in console)
  + Angular.element($0).scope shows this scope $id and scope properties
  + This child scope is inherited from ***\_\_proto*** object, which you can expand in consol
* $rootScope
  + If ***rootScope properties*** are modified in child scopes, that rootScope property will indeed get updated everywhere!!!
  + ***\*\* BEST PRACTICE*** – Not recommended to share properties on root scope across other scopes. It’s avail for convenience , but not good practice.

## Scope Hierarchies

* An Angular application contains only one root scope, but many child scopes
* Child scopes prototypically inherit from parent
* When evaluating an expression ({{username}}, Angular searches the current scope; if the prop is not found, it searches up the scope

## Scope Life Cycle

1. **Creation** - $rootScope is created when Ang bootstraps the app ($injector); directives may create child scopes during linking phase
2. **Watcher registration** – directives register watches on the scope during the template linking phase
3. **Model mutation** – When models on the scope change, angular will execute the $apply; if the model changes occur outside of Ang controllers/services, you will manually execute $apply
4. **Model Observation** – At the end of $apply, angular performs a $digest cycle which propagates thru all child scopes; the $digest iterates thru all watches to check for model mutation (i.e. if the model changes, then the listener is called for that watch)
5. **Scope destruction** – when child scopes are no longer needed, a $destroy must be executed (i.e. $digest will no longer visit that child scope; its memory is reallocated)

## Service vs. Factory vs. Provider

* **Service** 
  + ***Instantiated with the ‘new’ keyword***, so you add properties to ‘this’ and the service will return ‘this’.

app.service(‘***myService’***), function(){

\_artist = ‘’;

this.setArtist = function(artist){

\_artist = artist;

};

});

* + When declaring a service as an injectable argument, you are provided with an instance of the function.

* **Factory**
  + It creates an object, so you add properties and return that object. When injected into a controller, those properties are available.
  + When declaring a service, you are provided with the value that is returned by invoking the function reference.

app.factory(‘myFact’), function(){

var service = { };

return service;

});

* The factory method is the most common way of getting objects into AngularJS dependency injection system. It is very flexible and can contain sophisticated creation logic. Since factories are regular functions, we can also take advantage of a new lexical scope to simulate "private" variables. This is very useful as we can hide implementation details of a given service.
* Provider – are services you can only pass into app.config() function; this provides module-wide configuration for your service

## Service and Factory Differences

* **Services are instantiated** via ***new fn()*** so they return an instance, whereas **Factories are invoked** so they can return anything.

## Strict DI mode

* Use ng-strict-di **(ex/** **<html ng-app=’myApp’ *ng-strict-di*> )**
* With Strict di, you must ***explicitly annotate your injected functions and modules***
* Ensures that the code will work even when minified/uglified
* Performance is improved because Angular does not have to dynamically discover a function’s dependencies.

## Transclusion

* The inclusion of the directive’s DOM element content into the directive’s template (i.e. including some content from one template to another)
* The compile phase is executed in reverse order, then pre-link and post-link is executed normally; this is because Angular checks if child directives also have a transclude, then bubbles up all the DOM changes before adding it to the original directive’s template.

## $watch

* Created when *data binding* is setup on a view to a var on $scope
* Created using $scope.$watch 🡪
  + Registered with two parameters: ***Value (WatchExpr) function and Listener function***
* Should be ***idempotent*** – returns the same value if called many times
* Angular often calls $scope.$digest
  + A digest iterates thru all watches to check for var changes
  + If a var changes is detected, then that watch’s listener function is executed to update html data binding

Ex/ ***$scope.$watch***(function(scope) { return scope.data.myVar },

function(newValue, oldValue) {

document.getElementById("").innerHTML =

"**" + newValue + "**";

}

);