**Which tool have you used to analyse performance of your angularjs application.**

**List at least three ways to communicate between modules of your application using core AngularJS functionality or How do you share data between controllers?.**

Common ways to communicate between modules of your application using core AngularJS functionality include:

* Using services
* Using events
* By assigning models on $rootScope
* Directly between controllers, using $parent, $$childHead, $$nextSibling, etc.
* Directly between controllers, using ControllerAs, or other forms of inheritance

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

Using a service is definitely easy to test. Services are injected, and in a test either a real service can be used or it can be mocked.

Events can be tested. In unit testing controllers, they usually are instantiated. For testing events on $rootScope, it must be injected into the test.

Testing $rootScope against the existence of some arbitrary models is testable, but sharing data through $rootScope is not considered a good practice.

For testing direct communication between controllers, the expected results should probably be mocked. Otherwise, controllers would need to be manually instantiated to have the right context.

**When a scope is terminated, two similar “destroy” events are fired. What are they used for, and why are there two?**

The first one is an AngularJS event, “$destroy”, and the second one is a jqLite / jQuery event “$destroy”. The first one can be used by AngularJS scopes where they are accessible, such as in controllers or link functions.

Consider the two below happening in a directive’s postLink function. The AngularJS event:

scope.$on(‘$destroy’, function () {

// handle the destroy, i.e. clean up.

});

And

element.on(‘$destroy’, function () {

// respectful jQuery plugins already have this handler.

// angular.element(document.body).off(‘someCustomEvent’);

});

The jqLite / jQuery event is called whenever a node is removed, which may just happen without scope teardown.

**How do you reset a “$timeout”, and disable a “$watch()”?**

The key to both is assigning the result of the function to a variable.

To cleanup the timeout, just “.cancel()” it:

var customTimeout = $timeout(function () {

// arbitrary code

}, 55);

$timeout.cancel(customTimeout);

The same applies to “$interval()”.

To disable a watch, just call it.

// .$watch() returns a deregistration function that we store to a variable

var deregisterWatchFn = $rootScope.$watch(‘someGloballyAvailableProperty’, function (newVal) {

if (newVal) {

// we invoke that deregistration function, to disable the watch

deregisterWatchFn();

...

}

});

**How does interpolation, e.g. “{{ someModel }}”, actually work?**

It relies on $interpolation, a service which is called by the compiler. It evaluates text and markup which may contain AngularJS expressions. **For every interpolated expression, a “watch()” is set**. $interpolation returns a function, which has a single argument, “context”. By calling that function and providing a scope as context, the expressions are “$parse()”d against that scope.

**How does the digest phase work or how $scope.$apply() works?**

In a nutshell, on every digest cycle all scope models are compared against their previous values. That is *dirty checking*. If change is detected, the watches set on that model are fired. Then another digest cycle executes, and so on until all models are stable.

It is probably important to mention that there is no “.$digest()” polling. That means that every time it is being called deliberately. As long as core directives are used, we don’t need to worry, but when external code changes models the digest cycle needs to be called manually. Usually to do that, “.$apply()” or similar is used, and not “.$digest()” directly.

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

The two officially recommended methods for production are **disabling debug data** and enabling strict DI mode.

The first one can be enabled through the $compileProvider:

myApp.config(function ($compileProvider) {

$compileProvider.debugInfoEnabled(false);

});

That tweak disables appending scope to elements, making scopes inaccessible from the console. The second one can be set as a directive:

<html ng-app=“myApp” ng-strict-di>

The performance gain lies in the fact that the injected modules are annotated explicitly, hence they don’t need to be discovered dynamically.

You don’t need to annotate yourself, just use some automated build tool and library for that.

Two other popular ways are:

* **Using one-time binding where possible**. Those bindings are set, e.g. in “{{ ::someModel }}” interpolations by prefixing the model with two colons. In such a case, no watch is set and the model is ignored during digest.
* Making $**httpProvider use applyAsync**:

myApp.config(function ($httpProvider) {

$httpProvider.useApplyAsync(true);

});

**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.

**What are the DOM and the BOM?**

The DOM is the Document Object Model. It’s the view part of the UI. Whatever we are changing in page elements is reflected in the DOM.

BOM is the Browser Object Model, which specificies the global browser objects like window, localstorage, and console.

**What is the difference between ng-show/ng-hide and ng-if directives?**

ng-show/ng-hide will always insert the DOM element, but will display/hide it based on the condition. ng-if will not insert the DOM element until the condition is not fulfilled.

ng-if is better when we needed the DOM to be loaded conditionally, as it will help load page bit faster compared to ng-show/ng-hide.

**If you were to migrate from Angular 1.4 to Angular 1.5, what is the main thing that would need refactoring?**

Changing .directive to .component to adapt to the new Angular 1.5 components.

**How would you specify that a scope variable should have one-time binding only?**

By using “::” in front of it.

**What makes the angular.copy() method so powerful?**

It creates a deep copy of the variable.

A deep copy of a variable means it doesn’t point to the same memory reference as that variable. Usually assigning one variable to another creates a “shallow copy”, which makes the two variables point to the same memory reference. Therefore if one is changed, the other changes as well.

**What is the role of services in AngularJS and name any services made available by default?**

Few of the inbuilt services in AngularJS are:

* the $http service: The $http service is a core Angular service that facilitates communication with the remote HTTP servers via the browser’s XMLHttpRequest object or via JSONP
* the $log service: Simple service for logging. Default implementation safely writes the message into the browser’s console
* the $anchorScroll: it scrolls to the element related to the specified hash or (if omitted) to the current value of $location.hash() Why should one know about AngularJS Services, you may ask. Well, understanding the purpose of AngularJS Services helps bring modularity to AngularJS code

**== vs ===**

Double equals is officially known as the abstract equality comparison operator while triple equals is termed the strict equality comparison operator. The difference between them can be summed up as follows: **Abstract equality will attempt to resolve the data types via type coercion before making a comparison. Strict equality will return false if the types are different.** Consider the following example:

|  |  |
| --- | --- |
| console.log(3 == "3"); // true    console.log(3 === "3"); // false. |  |

Using two equal signs returns true because the string “3” is [converted](https://developer.mozilla.org/en-US/docs/Mozilla/js-ctypes/Using_js-ctypes/Type_conversion) to the number 3 before the comparison is made. Three equal signs sees that the types are different and returns false. Here’s another: