ANGULAR

**What should be the maximum number of concurrent “watches”? Bonus: How would you keep an eye on that number?**

To reduce memory consumption and improve performance it is a good idea to limit the number of watches on a page to 2,000. A utility called ng-stats can help track your watch count and digest cycles.

**How do you share data between controllers?**

Create an AngularJS service that will hold the data and inject it inside of the controllers.

Using a service is the cleanest, fastest and easiest way to test. However, there are couple of other ways to implement data sharing between controllers, like:

– Using events

– Using $parent, nextSibling, controllerAs, etc. to directly access the controllers

– Using the $rootScope to add the data on (not a good practice)

Of course, instead of using a factory or a service, you can also directly hold the shared data in the root scope and reference it from any controller. Although this actually works fine, it has a few disadvantages:

* Whatever is present in the root scope is inherited by all scopes
* You need to use some naming conventions to prevent multiple modules or libraries from overwriting each other’s data

***ng-show/ng-hide:***

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

**Dirty Checking**

Dirty checking is a simple process that boils down to a very basic concept: It checks whether a value has changed that hasn’t yet been synchronized across the app.

Our Angular app keeps track of the values of the current watches (in the watch object, for those who are curious). Angular walks down the $watch list, and, if the updated value has *not* changed from the old value, it continues down the list. If the value *has* changed, the app records the new value and continues down the $watch list.

Once Angular has run through the entire $watch list, if any value changed, the app will fall back into the $watch loop until it detects that nothing has changed.

Why run the loop all over again? If we update a value in the $watch list that updates another value, Angular won’t detect the update unless we rerun the loop.

If the loop runs ten times or more, our Angular app throws an exception, and the app dies. If Angular doesn’t throw this exception, our app could launch into an infinite loop, with bad results.

**What is a digest cycle in AngularJS?**In each digest cycle Angular compares the old and the new version of the scope model values. The digest cycle is triggered automatically. We can also use $apply() if we want to trigger the digest cycle manually.

Digest loop is designed to run dirty check over POJO models associated with Angular scope instances, so it only needs to run when a model might be changed. In a single page Web application running inside Browser, the following actions/events could lead to a model change

* DOM events
* XHR responses firing callbacks
* Browser's location changes
* Timers (setTimout, setInterval) firing the callbacks

Correspondingly, Angular trigger digest loop at, for instance

* input directives+ngModel, ngClick, ngMouseOver etc.
* $http and $resource
* $location
* $timeout

So some key practices for avoid angular performance pitfalls are

* Watch callback should be coded as simpler/efficient as possible, for example detach complicated algorithm code to, for example, worker threads
* Proactively remove a watch if it is not used anymore
* Prefer to call $scope.$digest() instead of $scope.$apply() if applicable, $digest() only run part of scope tree and ensure the models associated under the subtree is reflecting to view. But $apply() will run against entire scope tree, it will iterate through more watches.

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

By using “ :: ” in front of it. This allows you to check if the candidate is aware of the available variable bindings in AngularJS.

**How to cancel $timeout?**

To reset a timeout and/or $interval, assign the result of the function to a variable and then call the .cancel() function:

**var** customTimeout = $timeout(**function**() {

*// arbitrary code*

}, 55);

$timeout.cancel(customTimeout);

**Explain what is a $**scope**in AngularJS.**

The $scope is glue between a controller and view (HTML). It transfers data from the controller to view and vice-versa.

angularJS creates and injects a different $scope object to each controller in an application. So, the data and methods attached to $scope inside one controller cannot be accessed in another controller. With the nested controller, child controller will inherit the parent controller's scope object. Therefore, child controller can access properties added in parent controller but parent controller cannot access properties added in child controller.

Scope functions:

$emit() Dispatches the specified event upwards till $rootScope.

$broadcast() Dispatches the specified event downwards to all child scopes.

**What are Directives?**

You use it to create reusable web elements

Directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS’s HTML compiler ($compile) to attach a specified behavior to that DOM element (e.g. via event listeners), or even to transform the DOM element and its children. Angular comes with a set of these directives built-in, like ngBind, ngModel, and ngClass. Much like you create controllers and services, you can create your own directives for Angular to use. When Angular bootstraps your application, the HTML compiler traverses the DOM matching directives against the DOM elements.

***Structure of Directive :***

**var** directiveDefinitionObject = {

priority: 0,

template: '<div></div>',

*// templateUrl: 'directive.html', // or // function(tElement, tAttrs) { ... },*

transclude: false,

restrict: 'A',

templateNamespace: 'html',

scope: false,

controller: **function**(

$scope,

$element,

$attrs,

$transclude,

otherInjectables

) { ... },

controllerAs: 'stringIdentifier',

bindToController: false,

require: 'siblingDirectiveName',

compile: **function** **compile**(tElement, tAttrs, transclude) {

**return** {

pre: **function** **preLink**(scope, iElement, iAttrs, controller) { ... },

post: **function** **postLink**(scope, iElement, iAttrs, controller) { ... }

};

}

};

The **compile** function allows the directive to manipulate the DOM before it is compiled and linked thereby allowing it to add/remove/change directives, as well as, add/remove/change other DOM elements.

The **controller** function facilitates directive communication. Sibling and child directives can request the controller of their siblings and parents to communicate information.

The **pre**-link function allows for private $scope manipulation before the post-link process begins.

The **post**-link method is the primary workhorse method of the directive.

Commonly, not all of the functions are needed. In most circumstances, developers will simply create a controller and post-link function following the pattern below.

-- --- -- -- here link refers to post-link

.directive("directiveName",function () {

return {

controller: function() {

// controller code here...

},

link: function() {

// post-link code here...

}

}

})



***Scope of directives***

**SCOPE: FALSE (shared)**

This is the default value of scope in the directive definition. In this case, the directive has the same scope as its parent controller.

**SCOPE: TRUE (inherited)**

Angular JS will create a new scope by inheriting parent scope (usually controller scope, else application’s root Scope).

*Note*: Any changes made to this new scope will not reflect back to the parent scope. However, since the new scope is inherited from the parent scope, any changes made in the parent scope i.e. controller will be reflected in the directive scope.

**SCOPE: {} (isolated)**

New scope created for the directive, but it will not be inherited from the parent scope. This new scope also known as Isolated scope because it is completely detached from its parent scope.

***TRANSCLUDE in Directives :***

Consider a directive called *myDirective* in an element, and that element is enclosing some other content, let's say:

<div my-directive>

<button>some button</button>

<a href="#">and a link</a>

</div>

If *myDirective* is using a template, you'll see that the content of <div my-directive> will be replaced by your directive template. So having:

app.directive('myDirective', function(){

return{ template: '<div class="something"> This is my directive content</div>'}

});

will result in this render:

<div class="something"> This is my directive content</div>

Notice that the content of your original element <div my-directive> **will be lost** (or better said, replaced). So, say good-bye to these buddies:

<button>some button</button>

<a href="#">and a link</a>

So, what if you want to keep your <button>... and <a href>... in the DOM? You'll need something called transclusion. The concept is pretty simple: **Include the content from one place into another**. So now your directive will look something like this:

app.directive('myDirective', function(){

return{

transclude: true,

template: '<div class="something"> This is my directive content</div> <ng-transclude></ng-transclude>'}});

This would render:

<div class="something"> This is my directive content

<button>some button</button>

<a href="#">and a link</a>

</div>.

In conclusion, you basically use transclude when you want to preserve the contents of an element when you're using a directive.

**Type of directives**

* **A** : Specifies that Directive will be used as an attribute, like <div item-widget></div>, as was done in last example. This is also the default behavior if restrict is not declared. Preferred if just modifying the behavior of existing elements.
* **E** : Specifies that Directive will be used as an Element. Like <item-widget></item-widget>. Preferred if creating new content.
* **C** : Specifies that Directive can be used as class name in existing HTML elements. Like <div class=”item-widget”></div>
* **M** : Specifies that Directive can be used as HTML comments. Like <!– Using directive: item-widget–>

**filters**

they’re very powerful for transforming our data e.g lowercase ,number ,orderBy ,uppercase

app.filter('makeUppercase', function () {

return function (item) {

return item.toUpperCase();

};

});

<div ng-controller="PersonCtrl as person">

<p>

{{ person.username | makeUppercase }}

</p>

</div>

---------------------------------------------------

app.filter('startsWithLetter', function () {

return function (items, letter) {

var filtered = [];

var letterMatch = new RegExp(letter, 'i');

for (var i = 0; i < items.length; i++) {

var item = items[i];

if (letterMatch.test(item.name.substring(0, 1))) {

filtered.push(item);

}

}

return filtered;

};

});

<input type="text" ng-model="letter">

<ul>

<li ng-repeat="friend in person.friends | startsWithLetter:letter:number:somethingElse:anotherThing">

{{ friend }}

</li>

</ul>

**What is the difference between link and compile in Angular.js?**

Compile function is used for template DOM Manipulation and to collect all the directives.

Link function is used for registering DOM listeners as well as instance DOM manipulation and is executed once the template has been cloned.





So why is transclusion reversing the order in which the compile functions are called? because AngularJS needs to process the element's DOM content before it can make it available inside the template.

However, the element's child elements can also contain directives that apply transclusion themselves.

So AngularJS has to recursively traverse the DOM first to check if transclusion is enabled in child elements and then compile the DOM backwards to make sure all DOM changes correctly "bubble up" again to the top before the processed DOM is ready to be added to the original directive's template.

**9. What are the advantages of using Angular.js framework?**

Angular.js framework has the following advantages:

* Supports two way data-binding
* Supports MVC pattern
* Support static template and angular template
* Can add custom directive
* Supports REST full services
* Supports form validations
* Support both client and server communication
* Support dependency injection
* Applying Animations
* Event Handlers

**NG directives built in**

* ng-app : Initializes application.
* ng-model : Binds HTML controls to application data.
* ng-Controller : Attaches a controller class to view.
* ng-repeat : Bind repeated data HTML elements. Its like a for loop.
* ng-if : Bind HTML elements with condition.
* ng-show : Used to show the HTML elements.
* ng-hide : Used to hide the HTML elements.
* ng-class : Used to assign CSS class.
* ng-src : Used to pass the URL image etc.

**Where should we implement the DOM manipulation in AngularJS?**

In the directives. DOM Manipulations should not exist in controllers, services or anywhere else but in directives.

**$digest vs $apply**

scope.$digest() will fire watchers on the current scope, and on all of its children, too. scope.$apply will evaluate passed function and run $rootScope.$digest().

The first one is faster, as it needs to evaluate watchers for current scope and its children. The second one is slower, as it needs to evaluate watchers for$rootScope and all it's child scopes.

**services in angular**

Services are singleton components and all components of the application (into which the service is injected) will work with single instance of the service. An AngularJS service allows developing of business logic without depending on the View logic which will work with it.

**What is an interceptor? What are common uses of it?**

An interceptor is a middleware code where all the $http requests go through.

**Service vs factory**

a service is a constructor function whereas a factory is not. Somewhere deep inside of this Angular world, there’s this code that calls Object.create() with the service constructor function, when it gets instantiated. However, a factory function is really just a function that gets called, which is why we have to return an object explicitly.





dependency injection

Singleton Pattern