ANGULAR NOTES WITH SCRIPT JS CODE BY JOBHUNTER TEAMS

What is AngularJS

AngularJS is not a library, it's a framework that embraces extending HTML into a more expressive and readable format

AngularJS is a structural framework for dynamic web apps

A framework that lets you use HTML as your template language and extend HTML's syntax to express your application's components clearly and succinctly

Angular is an opinionated framework on how a CRUD application should be built

This looks like normal HTML, with some new markup. In AngularJS, a file like this is called a template. When AngularJS starts your application, it parses and processes this new markup from the template using the compiler. The loaded, transformed and rendered DOM is then called the view.

The first kind of new markup are the directives. They apply special behavior to attributes or elements in the HTML. In the example above we use the ng-app attribute, which is linked to a directive that automatically initializes our application. AngularJS also defines a directive for the input element that adds extra behavior to the element. The ng-model directive stores/updates the value of the input field into/from a variable.

Custom directives to access the DOM: In AngularJS, the only place where an application should access the DOM is within directives. This is important because artifacts that access the DOM are hard to test. If you need to access the DOM directly you should write a custom directive for this. The directives guide explains how to do this.

The second kind of new markup are the double curly braces {{ expression | filter }}: When the compiler encounters this markup, it will replace it with the evaluated value of the markup. An expression in a template is a JavaScript-like code snippet that allows AngularJS to read and write variables. Note that those variables are not global variables. Just like variables in a JavaScript function live in a scope, AngularJS provides a scope for the variables accessible to expressions. The values that are stored in variables on the scope are referred to as the model in the rest of the documentation. Applied to the example above, the markup directs AngularJS to "take the data we got from the input widgets and multiply them together".

The example above also contains a filter. A filter formats the value of an expression for display to the user. In the example above, the filter currency formats a number into an output that looks like money.

The important thing in the example is that AngularJS provides live bindings: Whenever the input values change, the value of the expressions are automatically recalculated and the DOM is updated with their values. The concept behind this is two-way data binding.

Adding UI logic: Controllers

Let's add some more logic to the example that allows us to enter and calculate the costs in different currencies and also pay the invoice.

angular.module('invoice1', [])

.controller('InvoiceController', function InvoiceController() {

this.qty = 1;

this.cost = 2;

this.inCurr = 'EUR';

this.currencies = ['USD', 'EUR', 'CNY'];

this.usdToForeignRates = {

USD: 1,

EUR: 0.74,

CNY: 6.09

};

this.total = function total(outCurr) {

return this.convertCurrency(this.qty \* this.cost, this.inCurr, outCurr);

};

this.convertCurrency = function convertCurrency(amount, inCurr, outCurr) {

return amount \* this.usdToForeignRates[outCurr] / this.usdToForeignRates[inCurr];

};

this.pay = function pay() {

window.alert('Thanks!');

};

});

What changed?

First, there is a new JavaScript file that contains a controller. More accurately, the file specifies a constructor function that will be used to create the actual controller instance. The purpose of controllers is to expose variables and functionality to expressions and directives.

Besides the new file that contains the controller code, we also added an ng-controller directive to the HTML. This directive tells AngularJS that the new InvoiceController is responsible for the element with the directive and all of the element's children. The syntax InvoiceController as invoice tells AngularJS to instantiate the controller and save it in the variable invoice in the current scope.

We also changed all expressions in the page to read and write variables within that controller instance by prefixing them with invoice. . The possible currencies are defined in the controller and added to the template using ng-repeat. As the controller contains a total function we are also able to bind the result of that function to the DOM using {{ invoice.total(...) }}.

Again, this binding is live, i.e. the DOM will be automatically updated whenever the result of the function changes. The button to pay the invoice uses the directive ngClick. This will evaluate the corresponding expression whenever the button is clicked.

In the new JavaScript file we are also creating a module at which we register the controller. We will talk about modules in the next section.

The following graphic shows how everything works together after we introduced the controller:

View-independent business logic: Services

Right now, the InvoiceController contains all logic of our example. When the application grows it is a good practice to move view-independent logic from the controller into a service, so it can be reused by other parts of the application as well. Later on, we could also change that service to load the exchange rates from the web, e.g. by calling the Fixer.io exchange rate API, without changing the controller.

Let's refactor our example and move the currency conversion into a service in another file:

angular.module('finance2', [])

.factory('currencyConverter', function() {

var currencies = ['USD', 'EUR', 'CNY'];

var usdToForeignRates = {

USD: 1,

EUR: 0.74,

CNY: 6.09

};

var convert = function(amount, inCurr, outCurr) {

return amount \* usdToForeignRates[outCurr] / usdToForeignRates[inCurr];

};

return {

currencies: currencies,

convert: convert

};

});

What changed?

We moved the convertCurrency function and the definition of the existing currencies into the new file finance2.js. But how does the controller get a hold of the now separated function?

This is where Dependency Injection comes into play. Dependency Injection (DI) is a software design pattern that deals with how objects and functions get created and how they get a hold of their dependencies. Everything within AngularJS (directives, filters, controllers, services, ...) is created and wired using dependency injection. Within AngularJS, the DI container is called the injector.

To use DI, there needs to be a place where all the things that should work together are registered. In AngularJS, this is the purpose of the modules. When AngularJS starts, it will use the configuration of the module with the name defined by the ng-app directive, including the configuration of all modules that this module depends on.

In the example above: The template contains the directive ng-app="invoice2". This tells AngularJS to use the invoice2 module as the main module for the application. The code snippet angular.module('invoice2', ['finance2']) specifies that the invoice2 module depends on the finance2 module. By this, AngularJS uses the InvoiceController as well as the currencyConverter service.

Now that AngularJS knows of all the parts of the application, it needs to create them. In the previous section we saw that controllers are created using a constructor function. For services, there are multiple ways to specify how they are created (see the service guide). In the example above, we are using an anonymous function as the factory function for the currencyConverter service. This function should return the currencyConverter service instance.

Back to the initial question: How does the InvoiceController get a reference to the currencyConverter function? In AngularJS, this is done by simply defining arguments on the constructor function. With this, the injector is able to create the objects in the right order and pass the previously created objects into the factories of the objects that depend on them. In our example, the InvoiceController has an argument named currencyConverter. By this, AngularJS knows about the dependency between the controller and the service and calls the controller with the service instance as argument.

The last thing that changed in the example between the previous section and this section is that we now pass an array to the module.controller function, instead of a plain function. The array first contains the names of the service dependencies that the controller needs. The last entry in the array is the controller constructor function. AngularJS uses this array syntax to define the dependencies so that the DI also works after minifying the code, which will most probably rename the argument name of the controller constructor function to something shorter like a.

Accessing the backend

Let's finish our example by fetching the exchange rates from the Fixer.io exchange rate API. The following example shows how this is done with AngularJS:

angular.module('invoice3', ['finance3'])

.controller('InvoiceController', ['currencyConverter', function InvoiceController(currencyConverter) {

this.qty = 1;

this.cost = 2;

this.inCurr = 'EUR';

this.currencies = currencyConverter.currencies;

this.total = function total(outCurr) {

return currencyConverter.convert(this.qty \* this.cost, this.inCurr, outCurr);

};

this.pay = function pay() {

window.alert('Thanks!');

};

}]);

Data Binding

CONTENTS

Data Binding in Classical Template Systems

Data Binding in AngularJS Templates

Related Topics

Data-binding in AngularJS apps is the automatic synchronization of data between the model and view components. The way that AngularJS implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa.

Data Binding in Classical Template Systems

Most templating systems bind data in only one direction: they merge template and model components together into a view. After the merge occurs, changes to the model or related sections of the view are NOT automatically reflected in the view. Worse, any changes that the user makes to the view are not reflected in the model. This means that the developer has to write code that constantly syncs the view with the model and the model with the view.

Data Binding in AngularJS Templates

AngularJS templates work differently. First the template (which is the uncompiled HTML along with any additional markup or directives) is compiled on the browser. The compilation step produces a live view. Any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. The model is the single-source-of-truth for the application state, greatly simplifying the programming model for the developer. You can think of the view as simply an instant projection of your model.

Because the view is just a projection of the model, the controller is completely separated from the view and unaware of it. This makes testing a snap because it is easy to test your controller in isolation without the view and the related DOM/browser dependency.

n AngularJS, a Controller is defined by a JavaScript constructor function that is used to augment the AngularJS Scope.

Controllers can be attached to the DOM in different ways. For each of them, AngularJS will instantiate a new Controller object, using the specified Controller's constructor function:

the ngController directive. A new child scope will be created and made available as an injectable parameter to the Controller's constructor function as $scope.

a route controller in a $route definition.

the controller of a regular directive, or a component directive.

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

Use controllers to:

Set up the initial state of the $scope object.

Add behavior to the $scope object.

Do not use controllers to:

Manipulate DOM — Controllers should contain only business logic. Putting any presentation logic into Controllers significantly affects its testability. AngularJS has databinding for most cases and directives to encapsulate manual DOM manipulation.

Format input — Use AngularJS form controls instead.

Filter output — Use AngularJS filters instead.

Share code or state across controllers — Use AngularJS services instead.

Manage the life-cycle of other components (for example, to create service instances).

In general, a Controller shouldn't try to do too much. It should contain only the business logic needed for a single view.

The most common way to keep Controllers slim is by encapsulating work that doesn't belong to controllers into services and then using these services in Controllers via dependency injection. This is discussed in the Dependency Injection and Services sections of this guide.

Setting up the initial state of a $scope object

Typically, when you create an application you need to set up the initial state for the AngularJS $scope. You set up the initial state of a scope by attaching properties to the $scope object. The properties contain the view model (the model that will be presented by the view). All the $scope properties will be available to the template at the point in the DOM where the Controller is registered.

The following example demonstrates creating a GreetingController, which attaches a greeting property containing the string 'Hola!' to the $scope:

var myApp = angular.module('myApp',[]);

myApp.controller('GreetingController', ['$scope', function($scope) {

$scope.greeting = 'Hola!';

}]);

We create an AngularJS Module, myApp, for our application. Then we add the controller's constructor function to the module using the .controller() method. This keeps the controller's constructor function out of the global scope.

We have used an inline injection annotation to explicitly specify the dependency of the Controller on the $scope service provided by AngularJS. See the guide on Dependency Injection for more information.

We attach our controller to the DOM using the ng-controller directive. The greeting property can now be data-bound to the template:

<div ng-controller="GreetingController">

{{ greeting }}

</div>

Adding Behavior to a Scope Object

In order to react to events or execute computation in the view we must provide behavior to the scope. We add behavior to the scope by attaching methods to the $scope object. These methods are then available to be called from the template/view.

The following example uses a Controller to add a method, which doubles a number, to the scope:

var myApp = angular.module('myApp',[]);

myApp.controller('DoubleController', ['$scope', function($scope) {

$scope.double = function(value) { return value \* 2; };

}]);

Once the Controller has been attached to the DOM, the double method can be invoked in an AngularJS expression in the template:

<div ng-controller="DoubleController">

Two times <input ng-model="num"> equals {{ double(num) }}

</div>

As discussed in the Concepts section of this guide, any objects (or primitives) assigned to the scope become model properties. Any methods assigned to the scope are available in the template/view, and can be invoked via AngularJS expressions and ng event handler directives (e.g. ngClick).

Simple Spicy Controller Example

To illustrate further how Controller components work in AngularJS, let's create a little app with the following components:

A template with two buttons and a simple message

A model consisting of a string named spice

A Controller with two functions that set the value of spice

The message in our template contains a binding to the spice model which, by default, is set to the string "very". Depending on which button is clicked, the spice model is set to chili or jalapeño, and the message is automatically updated by data-binding.

<div ng-controller="SpicyController">

<button ng-click="chiliSpicy()">Chili</button>

<button ng-click="jalapenoSpicy()">Jalapeño</button>

<p>The food is {{spice}} spicy!</p>

</div>

AngularJS services are substitutable objects that are wired together using dependency injection (DI). You can use services to organize and share code across your app.

AngularJS services are:

Lazily instantiated – AngularJS only instantiates a service when an application component depends on it.

Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.

AngularJS offers several useful services (like $http), but for most applications you'll also want to create your own.

Note: Like other core AngularJS identifiers, built-in services always start with $ (e.g. $http).

Using a Service

To use an AngularJS service, you add it as a dependency for the component (controller, service, filter or directive) that depends on the service. AngularJS's dependency injection subsystem takes care of the rest.

<div id="simple" ng-controller="MyController">

<p>Let's try this simple notify service, injected into the controller...</p>

<input ng-init="message='test'" ng-model="message" >

<button ng-click="callNotify(message);">NOTIFY</button>

<p>(you have to click 3 times to see an alert)</p>

</div>

Creating Services

Application developers are free to define their own services by registering the service's name and service factory function, with an AngularJS module.

The service factory function generates the single object or function that represents the service to the rest of the application. The object or function returned by the service is injected into any component (controller, service, filter or directive) that specifies a dependency on the service.

Registering Services

Services are registered to modules via the Module API. Typically you use the Module factory API to register a service:

var myModule = angular.module('myModule', []);

myModule.factory('serviceId', function() {

var shinyNewServiceInstance;

// factory function body that constructs shinyNewServiceInstance

return shinyNewServiceInstance;

});

Note that you are not registering a service instance, but rather a factory function that will create this instance when called.

Dependencies

Services can have their own dependencies. Just like declaring dependencies in a controller, you declare dependencies by specifying them in the service's factory function signature.

For more on dependencies, see the dependency injection docs.

The example module below has two services, each with various dependencies:

var batchModule = angular.module('batchModule', []);

/\*\*

\* The `batchLog` service allows for messages to be queued in memory and flushed

\* to the console.log every 50 seconds.

\*

\* @param {\*} message Message to be logged.

\*/

batchModule.factory('batchLog', ['$interval', '$log', function($interval, $log) {

var messageQueue = [];

function log() {

if (messageQueue.length) {

$log.log('batchLog messages: ', messageQueue);

messageQueue = [];

}

}

// start periodic checking

$interval(log, 50000);

return function(message) {

messageQueue.push(message);

}

}]);

/\*\*

\* `routeTemplateMonitor` monitors each `$route` change and logs the current

\* template via the `batchLog` service.

\*/

batchModule.factory('routeTemplateMonitor', ['$route', 'batchLog', '$rootScope',

function($route, batchLog, $rootScope) {

return {

startMonitoring: function() {

$rootScope.$on('$routeChangeSuccess', function() {

batchLog($route.current ? $route.current.template : null);

});

}

};

}]);

In the example, note that:

The batchLog service depends on the built-in $interval and $log services.

The routeTemplateMonitor service depends on the built-in $route service and our custom batchLog service.

Both services use the array notation to declare their dependencies.

The order of identifiers in the array is the same as the order of argument names in the factory function.

Registering a Service with $provide

You can also register services via the $provide service inside of a module's config function:

angular.module('myModule', []).config(['$provide', function($provide) {

$provide.factory('serviceId', function() {

var shinyNewServiceInstance;

// factory function body that constructs shinyNewServiceInstance

return shinyNewServiceInstance;

});

}]);

This technique is often used in unit tests to mock out a service's dependencies.

Unit Testing

The following is a unit test for the notify service from the Creating AngularJS Services example above. The unit test example uses a Jasmine spy (mock) instead of a real browser alert.

var mock, notify;

beforeEach(module('myServiceModule'));

beforeEach(function() {

mock = {alert: jasmine.createSpy()};

module(function($provide) {

$provide.value('$window', mock);

});

inject(function($injector) {

notify = $injector.get('notify');

});

});

it('should not alert first two notifications', function() {

notify('one');

notify('two');

expect(mock.alert).not.toHaveBeenCalled();

});

it('should alert all after third notification', function() {

notify('one');

notify('two');

notify('three');

expect(mock.alert).toHaveBeenCalledWith("one\ntwo\nthree");

});

it('should clear messages after alert', function() {

notify('one');

notify('two');

notify('third');

notify('more');

notify('two');

notify('third');

expect(mock.alert.calls.count()).toEqual(2);

expect(mock.alert.calls.mostRecent().args).toEqual(["more\ntwo\nthird"]);

});

Scope is an object that refers to the application model. It 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 characteristics

Scopes provide APIs ($watch) to observe model mutations.

Scopes provide APIs ($apply) to propagate any model changes through the system into the view from outside of the "AngularJS realm" (controllers, services, AngularJS event handlers).

Scopes can be nested to limit access to the properties of application components while providing access to shared model properties. Nested scopes are either "child scopes" or "isolate scopes". A "child scope" (prototypically) inherits properties from its parent scope. An "isolate scope" does not. See isolated scopes for more information.

Scopes provide context against which expressions are evaluated. For example {{username}} expression is meaningless, unless it is evaluated against a specific scope which defines the username property.

Scope as Data-Model

Scope is the glue between application controller and the view. During the template linking phase the directives set up $watch expressions on the scope. The $watch allows the directives to be notified of property changes, which allows the directive to render the updated value to the DOM.

Both controllers and directives have reference to the scope, but not to each other. This arrangement isolates the controller from the directive as well as from the DOM. This is an important point since it makes the controllers view agnostic, which greatly improves the testing story of the applications.

angular.module('scopeExample', [])

.controller('MyController', ['$scope', function($scope) {

$scope.username = 'World';

$scope.sayHello = function() {

$scope.greeting = 'Hello ' + $scope.username + '!';

};

}]);

In the above example notice that the MyController assigns World to the username property of the scope. The scope then notifies the input of the assignment, which then renders the input with username pre-filled. This demonstrates how a controller can write data into the scope.

Similarly the controller can assign behavior to scope as seen by the sayHello method, which is invoked when the user clicks on the 'greet' button. The sayHello method can read the username property and create a greeting property. This demonstrates that the properties on scope update automatically when they are bound to HTML input widgets.

Logically the rendering of {{greeting}} involves:

retrieval of the scope associated with DOM node where {{greeting}} is defined in template. In this example this is the same scope as the scope which was passed into MyController. (We will discuss scope hierarchies later.)

Evaluate the greeting expression against the scope retrieved above, and assign the result to the text of the enclosing DOM element.

You can think of the scope and its properties as the data which is used to render the view. The scope is the single source-of-truth for all things view related.

From a testability point of view, the separation of the controller and the view is desirable, because it allows us to test the behavior without being distracted by the rendering details.

it('should say hello', function() {

var scopeMock = {};

var cntl = new MyController(scopeMock);

// Assert that username is pre-filled

expect(scopeMock.username).toEqual('World');

// Assert that we read new username and greet

scopeMock.username = 'angular';

scopeMock.sayHello();

expect(scopeMock.greeting).toEqual('Hello angular!');

});

Scope Hierarchies

Each AngularJS application has exactly one root scope, but may have several child scopes.

The application can have multiple scopes, because some directives create new child scopes (refer to directive documentation to see which directives create new scopes). When new scopes are created, they are added as children of their parent scope. This creates a tree structure which parallels the DOM where they're attached.

When AngularJS evaluates {{name}}, it first looks at the scope associated with the given element for the name property. If no such property is found, it searches the parent scope and so on until the root scope is reached. In JavaScript this behavior is known as prototypical inheritance, and child scopes prototypically inherit from their parents.

This example illustrates scopes in application, and prototypical inheritance of properties. The example is followed by a diagram depicting the scope boundaries.

<div class="show-scope-demo">

<div ng-controller="GreetController">

Hello {{name}}!

</div>

<div ng-controller="ListController">

<ol>

<li ng-repeat="name in names">{{name}} from {{department}}</li>

</ol>

</div>

</div>

Notice that AngularJS automatically places ng-scope class on elements where scopes are attached. The <style> definition in this example highlights in red the new scope locations. The child scopes are necessary because the repeater evaluates {{name}} expression, but depending on which scope the expression is evaluated it produces different result. Similarly the evaluation of {{department}} prototypically inherits from root scope, as it is the only place where the department property is defined.

Retrieving Scopes from the DOM.

Scopes are attached to the DOM as $scope data property, and can be retrieved for debugging purposes. (It is unlikely that one would need to retrieve scopes in this way inside the application.) The location where the root scope is attached to the DOM is defined by the location of ng-app directive. Typically ng-app is placed on the <html> element, but it can be placed on other elements as well, if, for example, only a portion of the view needs to be controlled by AngularJS.

To examine the scope in the debugger:

Right click on the element of interest in your browser and select 'inspect element'. You should see the browser debugger with the element you clicked on highlighted.

The debugger allows you to access the currently selected element in the console as $0 variable.

To retrieve the associated scope in console execute: angular.element($0).scope() or just type $scope

Scope Events Propagation

Scopes can propagate events in similar fashion to DOM events. The event can be broadcasted to the scope children or emitted to scope parents.

angular.module('eventExample', [])

.controller('EventController', ['$scope', function($scope) {

$scope.count = 0;

$scope.$on('MyEvent', function() {

$scope.count++;

});

}]);

Scope Life Cycle

The normal flow of a browser receiving an event is that it executes a corresponding JavaScript callback. Once the callback completes the browser re-renders the DOM and returns to waiting for more events.

When the browser calls into JavaScript the code executes outside the AngularJS execution context, which means that AngularJS is unaware of model modifications. To properly process model modifications the execution has to enter the AngularJS execution context using the $apply method. Only model modifications which execute inside the $apply method will be properly accounted for by AngularJS. For example if a directive listens on DOM events, such as ng-click it must evaluate the expression inside the $apply method.

After evaluating the expression, the $apply method performs a $digest. In the $digest phase the scope examines all of the $watch expressions and compares them with the previous value. This dirty checking is done asynchronously. This means that assignment such as $scope.username="angular" will not immediately cause a $watch to be notified, instead the $watch notification is delayed until the $digest phase. This delay is desirable, since it coalesces multiple model updates into one $watch notification as well as guarantees that during the $watch notification no other $watches are running. If a $watch changes the value of the model, it will force additional $digest cycle.

Creation

The root scope is created during the application bootstrap by the $injector. During template linking, some directives create new child scopes.

Watcher registration

During template linking, directives register watches on the scope. These watches will be used to propagate model values to the DOM.

Model mutation

For mutations to be properly observed, you should make them only within the scope.$apply(). AngularJS APIs do this implicitly, so no extra $apply call is needed when doing synchronous work in controllers, or asynchronous work with $http, $timeout or $interval services.

Mutation observation

At the end of $apply, AngularJS performs a $digest cycle on the root scope, which then propagates throughout all child scopes. During the $digest cycle, all $watched expressions or functions are checked for model mutation and if a mutation is detected, the $watch listener is called.

Scope destruction

When child scopes are no longer needed, it is the responsibility of the child scope creator to destroy them via scope.$destroy() API. This will stop propagation of $digest calls into the child scope and allow for memory used by the child scope models to be reclaimed by the garbage collector.

Scopes and Directives

During the compilation phase, the compiler matches directives against the DOM template. The directives usually fall into one of two categories:

Observing directives, such as double-curly expressions {{expression}}, register listeners using the $watch() method. This type of directive needs to be notified whenever the expression changes so that it can update the view.

Listener directives, such as ng-click, register a listener with the DOM. When the DOM listener fires, the directive executes the associated expression and updates the view using the $apply() method.

When an external event (such as a user action, timer or XHR) is received, the associated expression must be applied to the scope through the $apply() method so that all listeners are updated correctly.

Directives that Create Scopes

In most cases, directives and scopes interact but do not create new instances of scope. However, some directives, such as ng-controller and ng-repeat, create new child scopes and attach the child scope to the corresponding DOM element. You can retrieve a scope for any DOM element by using an angular.element(aDomElement).scope() method call. See the directives guide for more information about isolate scopes.

Controllers and Scopes

Scopes and controllers interact with each other in the following situations:

Controllers use scopes to expose controller methods to templates (see ng-controller).

Controllers define methods (behavior) that can mutate the model (properties on the scope).

Controllers may register watches on the model. These watches execute immediately after the controller behavior executes.

See the ng-controller for more information.

Scope $watch Performance Considerations

Dirty checking the scope for property changes is a common operation in AngularJS and for this reason the dirty checking function must be efficient. Care should be taken that the dirty checking function does not do any DOM access, as DOM access is orders of magnitude slower than property access on JavaScript object.

Scope $watch Depths

Dirty checking can be done with three strategies: By reference, by collection contents, and by value. The strategies differ in the kinds of changes they detect, and in their performance characteristics.

Watching by reference (scope.$watch (watchExpression, listener)) detects a change when the whole value returned by the watch expression switches to a new value. If the value is an array or an object, changes inside it are not detected. This is the most efficient strategy.

Watching collection contents (scope.$watchCollection (watchExpression, listener)) detects changes that occur inside an array or an object: When items are added, removed, or reordered. The detection is shallow - it does not reach into nested collections. Watching collection contents is more expensive than watching by reference, because copies of the collection contents need to be maintained. However, the strategy attempts to minimize the amount of copying required.

Watching by value (scope.$watch (watchExpression, listener, true)) detects any change in an arbitrarily nested data structure. It is the most powerful change detection strategy, but also the most expensive. A full traversal of the nested data structure is needed on each digest, and a full copy of it needs to be held in memory.

Integration with the browser event loop

The diagram and the example below describe how AngularJS interacts with the browser's event loop.

The browser's event-loop waits for an event to arrive. An event is a user interaction, timer event, or network event (response from a server).

The event's callback gets executed. This enters the JavaScript context. The callback can modify the DOM structure.

Once the callback executes, the browser leaves the JavaScript context and re-renders the view based on DOM changes.

AngularJS modifies the normal JavaScript flow by providing its own event processing loop. This splits the JavaScript into classical and AngularJS execution context. Only operations which are applied in the AngularJS execution context will benefit from AngularJS data-binding, exception handling, property watching, etc... You can also use $apply() to enter the AngularJS execution context from JavaScript. Keep in mind that 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.

Enter the AngularJS execution context by calling scope.$apply(stimulusFn), where stimulusFn is the work you wish to do in the AngularJS execution context.

AngularJS executes the stimulusFn(), which typically modifies application state.

AngularJS enters the $digest loop. The loop is made up of two smaller loops which process $evalAsync queue and the $watch list. The $digest loop keeps iterating until the model stabilizes, which means that the $evalAsync queue is empty and the $watch list does not detect any changes.

The $evalAsync queue is used to schedule work which needs to occur outside of current stack frame, but before the browser's view render. This is usually done with setTimeout(0), but the setTimeout(0) approach suffers from slowness and may cause view flickering since the browser renders the view after each event.

The $watch list is a set of expressions which may have changed since last iteration. If a change is detected then the $watch function is called which typically updates the DOM with the new value.

Once the AngularJS $digest loop finishes, the execution leaves the AngularJS and JavaScript context. This is followed by the browser re-rendering the DOM to reflect any changes.

Here is the explanation of how the Hello world example achieves the data-binding effect when the user enters text into the text field.

During the compilation phase:

the ng-model and input directive set up a keydown listener on the <input> control.

the interpolation sets up a $watch to be notified of name changes.

During the runtime phase:

Pressing an 'X' key causes the browser to emit a keydown event on the input control.

The input directive captures the change to the input's value and calls $apply("name = 'X';") to update the application model inside the AngularJS execution context.

AngularJS applies the name = 'X'; to the model.

The $digest loop begins

The $watch list detects a change on the name property and notifies the interpolation, which in turn updates the DOM.

AngularJS exits the execution context, which in turn exits the keydown event and with it the JavaScript execution context.

The browser re-renders the view with the updated text.

Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies.

The AngularJS injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

Using Dependency Injection

Dependency Injection is pervasive throughout AngularJS. You can use it when defining components or when providing run and config blocks for a module.

Services, directives, filters, and animations are defined by an injectable factory method or constructor function, and can be injected with "services", "values", and "constants" as dependencies.

Controllers are defined by a constructor function, which can be injected with any of the "service" and "value" as dependencies, but they can also be provided with "special dependencies". See Controllers below for a list of these special dependencies.

The run method accepts a function, which can be injected with "services", "values" and, "constants" as dependencies. Note that you cannot inject "providers" into run blocks.

The config method accepts a function, which can be injected with "providers" and "constants" as dependencies. Note that you cannot inject "services" or "values" into configuration.

The provider method can only be injected with other "providers". However, only those that have been registered beforehand can be injected. This is different from services, where the order of registration does not matter.

See Modules for more details about run and config blocks and Providers for more information about the different provider types.

Factory Methods

The way you define a directive, service, or filter is with a factory function. The factory methods are registered with modules. The recommended way of declaring factories is:

angular.module('myModule', [])

.factory('serviceId', ['depService', function(depService) {

// ...

}])

.directive('directiveName', ['depService', function(depService) {

// ...

}])

.filter('filterName', ['depService', function(depService) {

// ...

}]);

Module Methods

We can specify functions to run at configuration and run time for a module by calling the config and run methods. These functions are injectable with dependencies just like the factory functions above.

angular.module('myModule', [])

.config(['depProvider', function(depProvider) {

// ...

}])

.run(['depService', function(depService) {

// ...

}]);

Controllers

Controllers are "classes" or "constructor functions" that are responsible for providing the application behavior that supports the declarative markup in the template. The recommended way of declaring Controllers is using the array notation:

someModule.controller('MyController', ['$scope', 'dep1', 'dep2', function($scope, dep1, dep2) {

...

$scope.aMethod = function() {

...

}

...

}]);

Unlike services, there can be many instances of the same type of controller in an application.

Moreover, additional dependencies are made available to Controllers:

$scope: Controllers are associated with an element in the DOM and so are provided with access to the scope. Other components (like services) only have access to the $rootScope service.

resolves: If a controller is instantiated as part of a route, then any values that are resolved as part of the route are made available for injection into the controller.

Dependency Annotation

AngularJS invokes certain functions (like service factories and controllers) via the injector. You need to annotate these functions so that the injector knows what services to inject into the function. There are three ways of annotating your code with service name information:

Using the inline array annotation (preferred)

Using the $inject property annotation

Implicitly from the function parameter names (has caveats)

Inline Array Annotation

This is the preferred way to annotate application components. This is how the examples in the documentation are written.

For example:

someModule.controller('MyController', ['$scope', 'greeter', function($scope, greeter) {

// ...

}]);

Here we pass an array whose elements consist of a list of strings (the names of the dependencies) followed by the function itself.

When using this type of annotation, take care to keep the annotation array in sync with the parameters in the function declaration.

$inject Property Annotation

To allow the minifiers to rename the function parameters and still be able to inject the right services, the function needs to be annotated with the $inject property. The $inject property is an array of service names to inject.

var MyController = function($scope, greeter) {

// ...

}

MyController.$inject = ['$scope', 'greeter'];

someModule.controller('MyController', MyController);

In this scenario the ordering of the values in the $inject array must match the ordering of the parameters in MyController.

Just like with the array annotation, you'll need to take care to keep the $inject in sync with the parameters in the function declaration.

Implicit Annotation

Careful: If you plan to minify your code, your service names will get renamed and break your app.

The simplest way to get hold of the dependencies is to assume that the function parameter names are the names of the dependencies.

someModule.controller('MyController', function($scope, greeter) {

// ...

});

Given a function, the injector can infer the names of the services to inject by examining the function declaration and extracting the parameter names. In the above example, $scope and greeter are two services which need to be injected into the function.

One advantage of this approach is that there's no array of names to keep in sync with the function parameters. You can also freely reorder dependencies.

However this method will not work with JavaScript minifiers/obfuscators because of how they rename parameters.

Tools like ng-annotate let you use implicit dependency annotations in your app and automatically add inline array annotations prior to minifying. If you decide to take this approach, you probably want to use ng-strict-di.

Because of these caveats, we recommend avoiding this style of annotation.

Using Strict Dependency Injection

You can add an ng-strict-di directive on the same element as ng-app to opt into strict DI mode:

<!doctype html>

<html ng-app="myApp" ng-strict-di>

<body>

I can add: {{ 1 + 2 }}.

<script src="angular.js"></script>

</body>

</html>

Strict mode throws an error whenever a service tries to use implicit annotations.

Consider this module, which includes a willBreak service that uses implicit DI:

angular.module('myApp', [])

.factory('willBreak', function($rootScope) {

// $rootScope is implicitly injected

})

.run(['willBreak', function(willBreak) {

// AngularJS will throw when this runs

}]);

When the willBreak service is instantiated, AngularJS will throw an error because of strict mode. This is useful when using a tool like ng-annotate to ensure that all of your application components have annotations.

If you're using manual bootstrapping, you can also use strict DI by providing strictDi: true in the optional config argument:

angular.bootstrap(document, ['myApp'], {

strictDi: true

});

Why Dependency Injection?

This section motivates and explains AngularJS's use of DI. For how to use DI, see above.

For in-depth discussion about DI, see Dependency Injection at Wikipedia, Inversion of Control by Martin Fowler, or read about DI in your favorite software design pattern book.

There are only three ways a component (object or function) can get a hold of its dependencies:

The component can create the dependency, typically using the new operator.

The component can look up the dependency, by referring to a global variable.

The component can have the dependency passed to it where it is needed.

The first two options of creating or looking up dependencies are not optimal because they hard code the dependency to the component. This makes it difficult, if not impossible, to modify the dependencies. This is especially problematic in tests, where it is often desirable to provide mock dependencies for test isolation.

The third option is the most viable, since it removes the responsibility of locating the dependency from the component. The dependency is simply handed to the component.

function SomeClass(greeter) {

this.greeter = greeter;

}

SomeClass.prototype.doSomething = function(name) {

this.greeter.greet(name);

}

In the above example SomeClass is not concerned with creating or locating the greeter dependency, it is simply handed the greeter when it is instantiated.

This is desirable, but it puts the responsibility of getting hold of the dependency on the code that constructs SomeClass.

To manage the responsibility of dependency creation, each AngularJS application has an injector. The injector is a service locator that is responsible for construction and lookup of dependencies.

Here is an example of using the injector service:

// Provide the wiring information in a module

var myModule = angular.module('myModule', []);

Teach the injector how to build a greeter service. Notice that greeter is dependent on the $window service. The greeter service is an object that contains a greet method.

myModule.factory('greeter', function($window) {

return {

greet: function(text) {

$window.alert(text);

}

};

});

Create a new injector that can provide components defined in our myModule module and request our greeter service from the injector. (This is usually done automatically by AngularJS bootstrap).

var injector = angular.injector(['ng', 'myModule']);

var greeter = injector.get('greeter');

Asking for dependencies solves the issue of hard coding, but it also means that the injector needs to be passed throughout the application. Passing the injector breaks the Law of Demeter. To remedy this, we use a declarative notation in our HTML templates, to hand the responsibility of creating components over to the injector, as in this example:

<div ng-controller="MyController">

<button ng-click="sayHello()">Hello</button>

</div>

function MyController($scope, greeter) {

$scope.sayHello = function() {

greeter.greet('Hello World');

};

}

When AngularJS compiles the HTML, it processes the ng-controller directive, which in turn asks the injector to create an instance of the controller and its dependencies.

injector.instantiate(MyController);

This is all done behind the scenes. Notice that by having the ng-controller ask the injector to instantiate the class, it can satisfy all of the dependencies of MyController without the controller ever knowing about the injector.

This is the best outcome. The application code simply declares the dependencies it needs, without having to deal with the injector. This setup does not break the Law of Demeter.

In AngularJS, templates are written with HTML that contains AngularJS-specific elements and attributes. AngularJS combines the template with information from the model and controller to render the dynamic view that a user sees in the browser.

These are the types of AngularJS elements and attributes you can use:

Directive — An attribute or element that augments an existing DOM element or represents a reusable DOM component.

Markup — The double curly brace notation {{ }} to bind expressions to elements is built-in AngularJS markup.

Filter — Formats data for display.

Form controls — Validates user input.

The following code snippet shows a template with directives and curly-brace expression bindings:

<html ng-app>

<!-- Body tag augmented with ngController directive -->

<body ng-controller="MyController">

<input ng-model="foo" value="bar">

<!-- Button tag with ngClick directive, and

string expression 'buttonText'

wrapped in "{{ }}" markup -->

<button ng-click="changeFoo()">{{buttonText}}</button>

<script src="angular.js"></script>

</body>

</html>

AngularJS expressions are JavaScript-like code snippets that are mainly placed in interpolation bindings such as <span title="{{ attrBinding }}">{{ textBinding }}</span>, but also used directly in directive attributes such as ng-click="functionExpression()".

For example, these are valid expressions in AngularJS:

1+2

a+b

user.name

items[index]

AngularJS Expressions vs. JavaScript Expressions

AngularJS expressions are like JavaScript expressions with the following differences:

Context: JavaScript expressions are evaluated against the global window. In AngularJS, expressions are evaluated against a scope object.

Forgiving: In JavaScript, trying to evaluate undefined properties generates ReferenceError or TypeError. In AngularJS, expression evaluation is forgiving to undefined and null.

Filters: You can use filters within expressions to format data before displaying it.

No Control Flow Statements: You cannot use the following in an AngularJS expression: conditionals, loops, or exceptions.

No Function Declarations: You cannot declare functions in an AngularJS expression, even inside ng-init directive.

No RegExp Creation With Literal Notation: You cannot create regular expressions in an AngularJS expression. An exception to this rule is ng-pattern which accepts valid RegExp.

No Object Creation With New Operator: You cannot use new operator in an AngularJS expression.

No Bitwise, Comma, And Void Operators: You cannot use Bitwise, , or void operators in an AngularJS expression.

If you want to run more complex JavaScript code, you should make it a controller method and call the method from your view. If you want to eval() an AngularJS expression yourself, use the $eval() method.

Example

<span>

1+2={{1+2}}

</span>

Interpolation markup with embedded expressions is used by AngularJS 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 uses the $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 on the computed interpolation function, which will update the corresponding text nodes or attribute values as part of the normal digest cycle.

Note that the interpolateDirective has a priority of 100 and sets up the watch in the preLink function.

How the string representation is computed

If the interpolated value is not a String, it is computed as follows:

undefined and null are converted to ''

if the value is an object that is not a Number, Date or Array, $interpolate looks for a custom toString() function on the object, and uses that. Custom means that myObject.toString !== Object.prototype.toString.

if the above doesn't apply, JSON.stringify is used.

Binding to boolean attributes

Attributes such as disabled are called boolean attributes, because their presence means true and their absence means false. We cannot use normal attribute bindings with them, because the HTML specification does not require browsers to preserve the values of boolean attributes. This means that if we put an AngularJS interpolation expression into such an attribute then the binding information would be lost, because the browser ignores the attribute value.

In the following example, the interpolation information would be ignored and the browser would simply interpret the attribute as present, meaning that the button would always be disabled.

Disabled: <input type="checkbox" ng-model="isDisabled" />

<button disabled="{{isDisabled}}">Disabled</button>

For this reason, AngularJS provides special ng-prefixed directives for the following boolean attributes: disabled, required, selected, checked, readOnly , and open.

These directives take an expression inside the attribute, and set the corresponding boolean attribute to true when the expression evaluates to truthy.

Disabled: <input type="checkbox" ng-model="isDisabled" />

<button ng-disabled="isDisabled">Disabled</button>

ngAttr for binding to arbitrary attributes

Web browsers are sometimes picky about what values they consider valid for attributes.

For example, considering this template:

<svg>

<circle cx="{{cx}}"></circle>

</svg>

We would expect AngularJS to be able to bind to this, but when we check the console we see something like Error: Invalid value for attribute cx="{{cx}}". Because of the SVG DOM API's restrictions, you cannot simply write cx="{{cx}}".

With ng-attr-cx you can work around this problem.

If an attribute with a binding is prefixed with the ngAttr prefix (denormalized as ng-attr-) then during the binding it will be applied to the corresponding unprefixed attribute. This allows you to bind to attributes that would otherwise be eagerly processed by browsers (e.g. an SVG element's circle[cx] attributes). When using ngAttr, the allOrNothing flag of $interpolate is used, so if any expression in the interpolated string results in undefined, the attribute is removed and not added to the element.

For example, we could fix the example above by instead writing:

<svg>

<circle ng-attr-cx="{{cx}}"></circle>

</svg>

If one wants to modify a camelcased attribute (SVG elements have valid camelcased attributes), such as viewBox on the svg element, one can use underscores to denote that the attribute to bind to is naturally camelcased.

For example, to bind to viewBox, we can write:

<svg ng-attr-view\_box="{{viewBox}}">

</svg>

Other attributes may also not work as expected when they contain interpolation markup, and can be used with ngAttr instead. The following is a list of known problematic attributes:

size in <select> elements (see issue 1619)

placeholder in <textarea> in Internet Explorer 10/11 (see issue 5025)

type in <button> in Internet Explorer 11 (see issue 14117)

value in <progress> in Internet Explorer = 11 (see issue 7218)

Known Issues

Dynamically changing an interpolated value

You should avoid dynamically changing the content of an interpolated string (e.g. attribute value or text node). Your changes are likely to be overwritten, when the original string gets evaluated. This restriction applies to both directly changing the content via JavaScript or indirectly using a directive.

For example, you should not use interpolation in the value of the style attribute (e.g. style="color: {{ 'orange' }}; font-weight: {{ 'bold' }};") and at the same time use a directive that changes the content of that attribute, such as ngStyle.

Embedding interpolation markup inside expressions

Note: AngularJS directive attributes take either expressions or interpolation markup with embedded expressions. It is considered bad practice to embed interpolation markup inside an expression:

<div ng-show="form{{$index}}.$invalid"></div>

You should instead delegate the computation of complex expressions to the scope, like this:

<div ng-show="getForm($index).$invalid"></div>

function getForm(index) {

return $scope['form' + index];

}

You can also access the scope with this in your templates:

<div ng-show="this['form' + $index].$invalid"></div>

Using filters in view templates

Filters can be applied to expressions in view templates using the following syntax:

{{ expression | filter }}

E.g. the markup {{ 12 | currency }} formats the number 12 as a currency using the currency filter. The resulting value is $12.00.

Filters can be applied to the result of another filter. This is called "chaining" and uses the following syntax:

{{ expression | filter1 | filter2 | ... }}

Filters may have arguments. The syntax for this is

{{ expression | filter:argument1:argument2:... }}

E.g. the markup {{ 1234 | number:2 }} formats the number 1234 with 2 decimal points using the number filter. The resulting value is 1,234.00.

When filters are executed

In templates, filters are only executed when their inputs have changed. This is more performant than executing a filter on each $digest as is the case with expressions.

There are two exceptions to this rule:

In general, this applies only to filters that take primitive values as inputs. Filters that receive Objects as input are executed on each $digest, as it would be too costly to track if the inputs have changed.

Filters that are marked as $stateful are also executed on each $digest. See Stateful filters for more information. Note that no AngularJS core filters are $stateful.

Using filters in controllers, services, and directives

You can also use filters in controllers, services, and directives.

For this, inject a dependency with the name <filterName>Filter into your controller/service/directive. E.g. a filter called number is injected by using the dependency numberFilter. The injected argument is a function that takes the value to format as first argument, and filter parameters starting with the second argument.

The example below uses the filter called filter. This filter reduces arrays into sub arrays based on conditions. The filter can be applied in the view template with markup like {{ctrl.array | filter:'a'}}, which would do a fulltext search for "a". However, using a filter in a view template will reevaluate the filter on every digest, which can be costly if the array is big.

The example below therefore calls the filter directly in the controller. By this, the controller is able to call the filter only when needed (e.g. when the data is loaded from the backend or the filter expression is changed).

<div ng-controller="FilterController as ctrl">

<div>

All entries:

<span ng-repeat="entry in ctrl.array">{{entry.name}} </span>

</div>

<div>

Entries that contain an "a":

<span ng-repeat="entry in ctrl.filteredArray">{{entry.name}} </span>

</div>

</div>

Creating custom filters

Writing your own filter is very easy: just register a new filter factory function with your module. Internally, this uses the filterProvider. This factory function should return a new filter function which takes the input value as the first argument. Any filter arguments are passed in as additional arguments to the filter function.

The filter function should be a pure function, which means that it should always return the same result given the same input arguments and should not affect external state, for example, other AngularJS services. AngularJS relies on this contract and will by default execute a filter only when the inputs to the function change. Stateful filters are possible, but less performant.

Note: Filter names must be valid AngularJS Expressions identifiers, such as uppercase or orderBy. Names with special characters, such as hyphens and dots, are not allowed. If you wish to namespace your filters, then you can use capitalization (myappSubsectionFilterx) or underscores (myapp\_subsection\_filterx).

The following sample filter reverses a text string. In addition, it conditionally makes the text upper-case.

<div ng-controller="MyController">

<input ng-model="greeting" type="text"><br>

No filter: {{greeting}}<br>

Reverse: {{greeting|reverse}}<br>

Reverse + uppercase: {{greeting|reverse:true}}<br>

Reverse, filtered in controller: {{filteredGreeting}}<br>

</div>

Stateful filters

It is strongly discouraged to write filters that are stateful, because the execution of those can't be optimized by AngularJS, which often leads to performance issues. Many stateful filters can be converted into stateless filters just by exposing the hidden state as a model and turning it into an argument for the filter.

If you however do need to write a stateful filter, you have to mark the filter as $stateful, which means that it will be executed one or more times during the each $digest cycle.

<div ng-controller="MyController">

Input: <input ng-model="greeting" type="text"><br>

Decoration: <input ng-model="decoration.symbol" type="text"><br>

No filter: {{greeting}}<br>

Decorated: {{greeting | decorate}}<br>

</div>

What is a Module?

You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.

Why?

Most applications have a main method that instantiates and wires together the different parts of the application.

AngularJS apps don't have a main method. Instead modules declaratively specify how an application should be bootstrapped. There are several advantages to this approach:

The declarative process is easier to understand.

You can package code as reusable modules.

The modules can be loaded in any order (or even in parallel) because modules delay execution.

Unit tests only have to load relevant modules, which keeps them fast.

End-to-end tests can use modules to override configuration.

The Basics

I'm in a hurry. How do I get a Hello World module working?

<div ng-app="myApp">

<div>

{{ 'World' | greet }}

</div>

</div>

Important things to notice:

The Module API

The reference to myApp module in <div ng-app="myApp">. This is what bootstraps the app using your module.

The empty array in angular.module('myApp', []). This array is the list of modules myApp depends on.

Recommended Setup

While the example above is simple, it will not scale to large applications. Instead we recommend that you break your application to multiple modules like this:

A module for each feature

A module for each reusable component (especially directives and filters)

And an application level module which depends on the above modules and contains any initialization code.

You can find a community style guide to help yourself when application grows.

The above is a suggestion. Tailor it to your needs.

<div ng-controller="XmplController">

{{ greeting }}

</div>

Overview

AngularJS's HTML compiler allows the developer to teach the browser new HTML syntax. The compiler allows you to attach behavior to any HTML element or attribute and even create new HTML elements or attributes with custom behavior. AngularJS calls these behavior extensions directives.

HTML has a lot of constructs for formatting the HTML for static documents in a declarative fashion. For example if something needs to be centered, there is no need to provide instructions to the browser how the window size needs to be divided in half so that the center is found, and that this center needs to be aligned with the text's center. Simply add an align="center" attribute to any element to achieve the desired behavior. Such is the power of declarative language.

However, the declarative language is also limited, as it does not allow you to teach the browser new syntax. For example, there is no easy way to get the browser to align the text at 1/3 the position instead of 1/2. What is needed is a way to teach the browser new HTML syntax.

AngularJS comes pre-bundled with common directives which are useful for building any app. We also expect that you will create directives that are specific to your app. These extensions become a Domain Specific Language for building your application.

All of this compilation takes place in the web browser; no server side or pre-compilation step is involved.

Compiler

Compiler is an AngularJS service which traverses the DOM looking for attributes. The compilation process happens in two phases.

Compile: traverse the DOM and collect all of the directives. The result is a linking function.

Link: combine the directives with a scope and produce a live view. Any changes in the scope model are reflected in the view, and any user interactions with the view are reflected in the scope model. This makes the scope model the single source of truth.

Some directives such as ng-repeat clone DOM elements once for each item in a collection. Having a compile and link phase improves performance since the cloned template only needs to be compiled once, and then linked once for each clone instance.

Directive

A directive is a behavior which should be triggered when specific HTML constructs are encountered during the compilation process. The directives can be placed in element names, attributes, class names, as well as comments. Here are some equivalent examples of invoking the ng-bind directive.

<span ng-bind="exp"></span>

<span class="ng-bind: exp;"></span>

<ng-bind></ng-bind>

<!-- directive: ng-bind exp -->

A directive is just a function which executes when the compiler encounters it in the DOM. See directive API for in-depth documentation on how to write directives.

Here is a directive which makes any element draggable. Notice the draggable attribute on the <span> element.

angular.module('drag', []).

directive('draggable', function($document) {

return function(scope, element, attr) {

var startX = 0, startY = 0, x = 0, y = 0;

element.css({

position: 'relative',

border: '1px solid red',

backgroundColor: 'lightgrey',

cursor: 'pointer',

display: 'block',

width: '65px'

});

element.on('mousedown', function(event) {

// Prevent default dragging of selected content

event.preventDefault();

startX = event.screenX - x;

startY = event.screenY - y;

$document.on('mousemove', mousemove);

$document.on('mouseup', mouseup);

});

function mousemove(event) {

y = event.screenY - startY;

x = event.screenX - startX;

element.css({

top: y + 'px',

left: x + 'px'

});

}

function mouseup() {

$document.off('mousemove', mousemove);

$document.off('mouseup', mouseup);

}

};

});

AngularJS <script> Tag

This example shows the recommended path for integrating AngularJS with what we call automatic initialization.

<!doctype html>

<html xmlns:ng="http://angularjs.org" ng-app>

<body>

...

<script src="angular.js"></script>

</body>

</html>

Place the script tag at the bottom of the page. Placing script tags at the end of the page improves app load time because the HTML loading is not blocked by loading of the angular.js script. You can get the latest bits from http://code.angularjs.org. Please don't link your production code to this URL, as it will expose a security hole on your site. For experimental development linking to our site is fine.

Choose: angular-[version].js for a human-readable file, suitable for development and debugging.

Choose: angular-[version].min.js for a compressed and obfuscated file, suitable for use in production.

Place ng-app to the root of your application, typically on the <html> tag if you want AngularJS to auto-bootstrap your application.

If you choose to use the old style directive syntax ng: then include xml-namespace in html when running the page in the XHTML mode. (This is here for historical reasons, and we no longer recommend use of ng:.)

Automatic Initialization

AngularJS 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 AngularJS looks for the ngApp directive which designates your application root. If the ngApp directive is found then AngularJS will:

load the module associated with the directive.

create the application injector

compile the DOM treating the ngApp directive as the root of the compilation. This allows you to tell it to treat only a portion of the DOM as an AngularJS application.

<!doctype html>

<html ng-app="optionalModuleName">

<body>

I can add: {{ 1+2 }}.

<script src="angular.js"></script>

</body>

</html>

As a best practice, consider adding an ng-strict-di directive on the same element as ng-app:

<!doctype html>

<html ng-app="optionalModuleName" ng-strict-di>

<body>

I can add: {{ 1+2 }}.

<script src="angular.js"></script>

</body>

</html>

This will ensure that all services in your application are properly annotated. See the dependency injection strict mode docs for more.

Manual Initialization

If you need to have more control over the initialization process, you can use a manual bootstrapping method instead. Examples of when you'd need to do this include using script loaders or the need to perform an operation before AngularJS compiles a page.

Here is an example of manually initializing AngularJS:

<!doctype html>

<html>

<body>

<div ng-controller="MyController">

Hello {{greetMe}}!

</div>

<script src="http://code.angularjs.org/snapshot/angular.js"></script>

<script>

angular.module('myApp', [])

.controller('MyController', ['$scope', function ($scope) {

$scope.greetMe = 'World';

}]);

angular.element(function() {

angular.bootstrap(document, ['myApp']);

});

</script>

</body>

</html>

Note that we provided the name of our application module to be loaded into the injector as the second parameter of the angular.bootstrap function. Notice that angular.bootstrap will not create modules on the fly. You must create any custom modules before you pass them as a parameter.

You should call angular.bootstrap() after you've loaded or defined your modules. You cannot add controllers, services, directives, etc after an application bootstraps.

Note: You should not use the ng-app directive when manually bootstrapping your app.

This is the sequence that your code should follow:

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.

Call angular.bootstrap to compile the element into an executable, bi-directionally bound application.

JavaScript is a dynamically typed language which comes with great power of expression, but it also comes with almost no help from the compiler. For this reason we feel very strongly that any code written in JavaScript needs to come with a strong set of tests. We have built many features into AngularJS which make testing your AngularJS applications easy. With AngularJS, there is no excuse for not testing.

Separation of Concerns

Unit testing, as the name implies, is about testing individual units of code. Unit tests try to answer questions such as "Did I think about the logic correctly?" or "Does the sort function order the list in the right order?"

In order to answer such a question it is very important that we can isolate the unit of code under test. That is because when we are testing the sort function we don't want to be forced into creating related pieces such as the DOM elements, or making any XHR calls to fetch the data to sort.

While this may seem obvious it can be very difficult to call an individual function on a typical project. The reason is that the developers often mix concerns resulting in a piece of code which does everything. It makes an XHR request, it sorts the response data, and then it manipulates the DOM.

With AngularJS, we try to make it easy for you to do the right thing. For your XHR requests, we provide dependency injection, so your requests can be simulated. For the DOM, we abstract it, so you can test your model without having to manipulate the DOM directly. Your tests can then assert that the data has been sorted without having to create or look at the state of the DOM or to wait for any XHR requests to return data. The individual sort function can be tested in isolation.

With great power comes great responsibility

AngularJS is written with testability in mind, but it still requires that you do the right thing. We tried to make the right thing easy, but if you ignore these guidelines you may end up with an untestable application.

Dependency Injection

AngularJS comes with dependency injection built-in, which makes testing components much easier, because you can pass in a component's dependencies and stub or mock them as you wish.

Components that have their dependencies injected allow them to be easily mocked on a test by test basis, without having to mess with any global variables that could inadvertently affect another test.

Additional tools for testing AngularJS applications

For testing AngularJS applications there are certain tools that you should use that will make testing much easier to set up and run.

Karma

Karma is a JavaScript command line tool that can be used to spawn a web server which loads your application's source code and executes your tests. You can configure Karma to run against a number of browsers, which is useful for being confident that your application works on all browsers you need to support. Karma is executed on the command line and will display the results of your tests on the command line once they have run in the browser.

Karma is a NodeJS application, and should be installed through npm/yarn. Full installation instructions are available on the Karma website.

Jasmine

Jasmine is a behavior driven development framework for JavaScript that has become the most popular choice for testing AngularJS applications. Jasmine provides functions to help with structuring your tests and also making assertions. As your tests grow, keeping them well structured and documented is vital, and Jasmine helps achieve this.

In Jasmine we use the describe function to group our tests together:

describe("sorting the list of users", function() {

// individual tests go here

});

And then each individual test is defined within a call to the it function:

describe('sorting the list of users', function() {

it('sorts in descending order by default', function() {

// your test assertion goes here

});

});

Grouping related tests within describe blocks and describing each individual test within an it call keeps your tests self documenting.

Finally, Jasmine provides matchers which let you make assertions:

describe('sorting the list of users', function() {

it('sorts in descending order by default', function() {

var users = ['jack', 'igor', 'jeff'];

var sorted = sortUsers(users);

expect(sorted).toEqual(['jeff', 'jack', 'igor']);

});

});

Jasmine comes with a number of matchers that help you make a variety of assertions. You should read the Jasmine documentation to see what they are. To use Jasmine with Karma, we use the karma-jasmine test runner.

angular-mocks

AngularJS also provides the ngMock module, which provides mocking for your tests. This is used to inject and mock AngularJS services within unit tests. In addition, it is able to extend other modules so they are synchronous. Having tests synchronous keeps them much cleaner and easier to work with. One of the most useful parts of ngMock is $httpBackend, which lets us mock XHR requests in tests, and return sample data instead.

Testing a Controller

Because AngularJS separates logic from the view layer, it keeps controllers easy to test. Let's take a look at how we might test the controller below, which provides $scope.grade, which sets a property on the scope based on the length of the password.

angular.module('app', [])

.controller('PasswordController', function PasswordController($scope) {

$scope.password = '';

$scope.grade = function() {

var size = $scope.password.length;

if (size > 8) {

$scope.strength = 'strong';

} else if (size > 3) {

$scope.strength = 'medium';

} else {

$scope.strength = 'weak';

}

};

});

Because controllers are not available on the global scope, we need to use angular.mock.inject to inject our controller first. The first step is to use the module function, which is provided by angular-mocks. This loads in the module it's given, so it is available in your tests. We pass this into beforeEach, which is a function Jasmine provides that lets us run code before each test. Then we can use inject to access $controller, the service that is responsible for instantiating controllers.

describe('PasswordController', function() {

beforeEach(module('app'));

var $controller;

beforeEach(inject(function(\_$controller\_){

// The injector unwraps the underscores (\_) from around the parameter names when matching

$controller = \_$controller\_;

}));

describe('$scope.grade', function() {

it('sets the strength to "strong" if the password length is >8 chars', function() {

var $scope = {};

var controller = $controller('PasswordController', { $scope: $scope });

$scope.password = 'longerthaneightchars';

$scope.grade();

expect($scope.strength).toEqual('strong');

});

});

});

Notice how by nesting the describe calls and being descriptive when calling them with strings, the test is very clear. It documents exactly what it is testing, and at a glance you can quickly see what is happening. Now let's add the test for when the password is less than three characters, which should see $scope.strength set to "weak":

describe('PasswordController', function() {

beforeEach(module('app'));

var $controller;

beforeEach(inject(function(\_$controller\_){

// The injector unwraps the underscores (\_) from around the parameter names when matching

$controller = \_$controller\_;

}));

describe('$scope.grade', function() {

it('sets the strength to "strong" if the password length is >8 chars', function() {

var $scope = {};

var controller = $controller('PasswordController', { $scope: $scope });

$scope.password = 'longerthaneightchars';

$scope.grade();

expect($scope.strength).toEqual('strong');

});

it('sets the strength to "weak" if the password length <3 chars', function() {

var $scope = {};

var controller = $controller('PasswordController', { $scope: $scope });

$scope.password = 'a';

$scope.grade();

expect($scope.strength).toEqual('weak');

});

});

});

Now we have two tests, but notice the duplication between the tests. Both have to create the $scope variable and create the controller. As we add new tests, this duplication is only going to get worse. Thankfully, Jasmine provides beforeEach, which lets us run a function before each individual test. Let's see how that would tidy up our tests:

describe('PasswordController', function() {

beforeEach(module('app'));

var $controller;

beforeEach(inject(function(\_$controller\_){

// The injector unwraps the underscores (\_) from around the parameter names when matching

$controller = \_$controller\_;

}));

describe('$scope.grade', function() {

var $scope, controller;

beforeEach(function() {

$scope = {};

controller = $controller('PasswordController', { $scope: $scope });

});

it('sets the strength to "strong" if the password length is >8 chars', function() {

$scope.password = 'longerthaneightchars';

$scope.grade();

expect($scope.strength).toEqual('strong');

});

it('sets the strength to "weak" if the password length <3 chars', function() {

$scope.password = 'a';

$scope.grade();

expect($scope.strength).toEqual('weak');

});

});

});

We've moved the duplication out and into the beforeEach block. Each individual test now only contains the code specific to that test, and not code that is general across all tests. As you expand your tests, keep an eye out for locations where you can use beforeEach to tidy up tests. beforeEach isn't the only function of this sort that Jasmine provides, and the documentation lists the others.

Testing Filters

Filters are functions which transform the data into a user readable format. They are important because they remove the formatting responsibility from the application logic, further simplifying the application logic.

myModule.filter('length', function() {

return function(text) {

return ('' + (text || '')).length;

}

});

describe('length filter', function() {

var $filter;

beforeEach(inject(function(\_$filter\_){

$filter = \_$filter\_;

}));

it('returns 0 when given null', function() {

var length = $filter('length');

expect(length(null)).toEqual(0);

});

it('returns the correct value when given a string of chars', function() {

var length = $filter('length');

expect(length('abc')).toEqual(3);

});

});

Testing Directi

As applications grow in size and complexity, it becomes unrealistic to rely on manual testing to verify the correctness of new features, catch bugs and notice regressions. Unit tests are the first line of defense for catching bugs, but sometimes issues come up with integration between components which can't be captured in a unit test. End-to-end tests are made to find these problems.

We have built Protractor, an end to end test runner which simulates user interactions that will help you verify the health of your AngularJS application.

Using Protractor

Protractor is a Node.js program, and runs end-to-end tests that are also written in JavaScript and run with node. Protractor uses WebDriver to control browsers and simulate user actions.

For more information on Protractor, view getting started or the api docs.

Protractor uses Jasmine for its test syntax. As in unit testing, a test file is comprised of one or more it blocks that describe the requirements of your application. it blocks are made of commands and expectations. Commands tell Protractor to do something with the application such as navigate to a page or click on a button. Expectations tell Protractor to assert something about the application's state, such as the value of a field or the current URL.

If any expectation within an it block fails, the runner marks the it as "failed" and continues on to the next block.

Test files may also have beforeEach and afterEach blocks, which will be run before or after each it block regardless of whether the block passes or fails.

In addition to the above elements, tests may also contain helper functions to avoid duplicating code in the it blocks.

Here is an example of a simple test:

describe('TODO list', function() {

it('should filter results', function() {

// Find the element with ng-model="user" and type "jacksparrow" into it

element(by.model('user')).sendKeys('jacksparrow');

// Find the first (and only) button on the page and click it

element(by.css(':button')).click();

// Verify that there are 10 tasks

expect(element.all(by.repeater('task in tasks')).count()).toEqual(10);

// Enter 'groceries' into the element with ng-model="filterText"

element(by.model('filterText')).sendKeys('groceries');

// Verify that now there is only one item in the task list

expect(element.all(by.repeater('task in tasks')).count()).toEqual(1);

});

});

This test describes the requirements of a ToDo list, specifically, that it should be able to filter the list of items.

Example

See the angular-seed project for more examples, or look at the embedded examples in the AngularJS documentation (For example, $http has an end-to-end test in the example under the protractor.js tag).

Caveats

Protractor does not work out-of-the-box with apps that bootstrap manually using angular.bootstrap. You must use the ng-app directive.

CSS classes used by AngularJS

ng-scope

Usage: AngularJS applies this class to any element for which a new scope is defined. (see scope guide for more information about scopes)

ng-isolate-scope

Usage: AngularJS applies this class to any element for which a new isolate scope is defined.

ng-binding

Usage: AngularJS applies this class to any element that is attached to a data binding, via ng-bind or {{}} curly braces, for example. (see databinding guide)

ng-invalid, ng-valid

Usage: AngularJS applies this class to a form control widget element if that element's input does not pass validation. (see input directive)

ng-pristine, ng-dirty

Usage: AngularJS ngModel directive applies ng-pristine class to a new form control widget which did not have user interaction. Once the user interacts with the form control, the class is changed to ng-dirty.

ng-touched, ng-untouched

Usage: AngularJS ngModel directive applies ng-untouched class to a new form control widget which has not been blurred. Once the user blurs the form control, the class is changed to ng-touched.

ngModel

Much of ngAria's heavy lifting happens in the ngModel directive. For elements using ngModel, special attention is paid by ngAria if that element also has a role or type of checkbox, radio, range or textbox.

For those elements using ngModel, ngAria will dynamically bind and update the following ARIA attributes (if they have not been explicitly specified by the developer):

aria-checked

aria-valuemin

aria-valuemax

aria-valuenow

aria-invalid

aria-required

aria-readonly

aria-disabled

Example

<form>

<custom-checkbox role="checkbox" ng-model="checked" required

aria-label="Custom checkbox" show-attrs>

Custom checkbox

</custom-checkbox>

</form>

<hr />

<b>Is checked:</b> {{ !!checked }}

ngAria will also add tabIndex, ensuring custom elements with these roles will be reachable from the keyboard. It is still up to you as a developer to ensure custom controls will be accessible. As a rule, any time you create a widget involving user interaction, be sure to test it with your keyboard and at least one mobile and desktop screen reader.

ngValue and ngChecked

To ease the transition between native inputs and custom controls, ngAria now supports ngValue and ngChecked. The original directives were created for native inputs only, so ngAria extends support to custom elements by managing aria-checked for accessibility.

Example

<custom-checkbox ng-checked="val"></custom-checkbox>

<custom-radio-button ng-value="val"></custom-radio-button>

Becomes:

<custom-checkbox ng-checked="val" aria-checked="true"></custom-checkbox>

<custom-radio-button ng-value="val" aria-checked="true"></custom-radio-button>

ngDisabled

The disabled attribute is only valid for certain elements such as button, input and textarea. To properly disable custom element directives such as <md-checkbox> or <taco-tab>, using ngAria with ngDisabled will also add aria-disabled. This tells assistive technologies when a non-native input is disabled, helping custom controls to be more accessible.

Example

<md-checkbox ng-disabled="disabled"></md-checkbox>

Becomes:

<md-checkbox disabled aria-disabled="true"></md-checkbox>

You can check whether a control is legitimately disabled for a screen reader by visiting chrome://accessibility and inspecting the accessibility tree.

ngRequired

The boolean required attribute is only valid for native form controls such as input and textarea. To properly indicate custom element directives such as <md-checkbox> or <custom-input> as required, using ngAria with ngRequired will also add aria-required. This tells accessibility APIs when a custom control is required.

Example

<md-checkbox ng-required="val"></md-checkbox>

Becomes:

<md-checkbox ng-required="val" aria-required="true"></md-checkbox>

ngReadonly

The boolean readonly attribute is only valid for native form controls such as input and textarea. To properly indicate custom element directives such as <md-checkbox> or <custom-input> as required, using ngAria with ngReadonly will also add aria-readonly. This tells accessibility APIs when a custom control is read-only.

Example

<md-checkbox ng-readonly="val"></md-checkbox>

Becomes:

<md-checkbox ng-readonly="val" aria-readonly="true"></md-checkbox>

ngShow

The ngShow directive shows or hides the given HTML element based on the expression provided to the ngShow attribute. The element is shown or hidden by removing or adding the .ng-hide CSS class onto the element.

In its default setup, ngAria for ngShow is actually redundant. It toggles aria-hidden on the directive when it is hidden or shown. However, the default CSS of display: none !important, already hides child elements from a screen reader. It becomes more useful when the default CSS is overridden with properties that don’t affect assistive technologies, such as opacity or transform. By toggling aria-hidden dynamically with ngAria, we can ensure content visually hidden with this technique will not be read aloud in a screen reader.

One caveat with this combination of CSS and aria-hidden: you must also remove links and other interactive child elements from the tab order using tabIndex=“-1” on each control. This ensures screen reader users won't accidentally focus on "mystery elements". Managing tab index on every child control can be complex and affect performance, so it’s best to just stick with the default display: none CSS. See the fourth rule of ARIA use.

Example

.ng-hide {

display: block;

opacity: 0;

}

<div ng-show="false" class="ng-hide" aria-hidden="true"></div>

Becomes:

<div ng-show="true" aria-hidden="false"></div>

Note: Child links, buttons or other interactive controls must also be removed from the tab order.

ngHide

The ngHide directive shows or hides the given HTML element based on the expression provided to the ngHide attribute. The element is shown or hidden by removing or adding the .ng-hide CSS class onto the element.

The default CSS for ngHide, the inverse method to ngShow, makes ngAria redundant. It toggles aria-hidden on the directive when it is hidden or shown, but the content is already hidden with display: none. See explanation for ngShow when overriding the default CSS.

ngClick and ngDblclick

If ng-click or ng-dblclick is encountered, ngAria will add tabindex="0" to any element not in a node blacklist: Button Anchor Input Textarea Select Details/Summary To fix widespread accessibility problems with ng-click on div elements, ngAria will dynamically bind a keypress event by default as long as the element isn't in the node blacklist. You can turn this functionality on or off with the bindKeypress configuration option. ngAria will also add the button role to communicate to users of assistive technologies. This can be disabled with the bindRoleForClick configuration option. For ng-dblclick, you must still manually add ng-keypress and a role to non-interactive elements such as div or taco-button to enable keyboard access.

Example

html <div ng-click="toggleMenu()"></div> Becomes: html <div ng-click="toggleMenu()" tabindex="0"></div>

ngMessages

The ngMessages module makes it easy to display form validation or other messages with priority sequencing and animation. To expose these visual messages to screen readers, ngAria injects aria-live="assertive", causing them to be read aloud any time a message is shown, regardless of the user's focus location.

Example

<div ng-messages="myForm.myName.$error">

<div ng-message="required">You did not enter a field</div>

<div ng-message="maxlength">Your field is too long</div>

</div>

Becomes:

<div ng-messages="myForm.myName.$error" aria-live="assertive">

<div ng-message="required">You did not enter a field</div>

<div ng-message="maxlength">Your field is too long</div>

</div>

Disabling attributes

The attribute magic of ngAria may not work for every scenario. To disable individual attributes, you can use the config method. Just keep in mind this will tell ngAria to ignore the attribute globally.

<div ng-click="someFunction" show-attrs>

&lt;div&gt; with ng-click and bindRoleForClick, tabindex set to false

</div>

<script>

angular.module('ngAria\_ngClickExample', ['ngAria'], function config($ariaProvider) {

$ariaProvider.config({

bindRoleForClick: false,

tabindex: false

});

})

.directive('showAttrs', function() {

return function(scope, el, attrs) {

var pre = document.createElement('pre');

el.after(pre);

scope.$watch(function() {

var attrs = {};

Array.prototype.slice.call(el[0].attributes, 0).forEach(function(item) {

if (item.name !== 'show-attrs') {

attrs[item.name] = item.value;

}

});

return attrs;

}, function(newAttrs, oldAttrs) {

pre.textContent = JSON.stringify(newAttrs, null, 2);

}, true);

}

});

</script>

Common Accessibility Patterns

Accessibility best practices that apply to web apps in general also apply to AngularJS.

Text alternatives: Add alternate text content to make visual information accessible using these W3C guidelines. The appropriate technique depends on the specific markup but can be accomplished using offscreen spans, aria-label or label elements, image alt attributes, figure/figcaption elements and more.

HTML Semantics: If you're creating custom element directives, Web Components or HTML in general, use native elements wherever possible to utilize built-in events and properties. Alternatively, use ARIA to communicate semantic meaning. See notes on ARIA use.

Focus management: Guide the user around the app as views are appended/removed. Focus should never be lost, as this causes unexpected behavior and much confusion (referred to as "freak-out mode").

Announcing changes: When filtering or other UI messaging happens away from the user's focus, notify with ARIA Live Regions.

Color contrast and scale: Make sure content is legible and interactive controls are usable at all screen sizes. Consider configurable UI themes for people with color blindness, low vision or other visual impairments.

Progressive enhancement: Some users do not browse with JavaScript enabled or do not have the latest browser. An accessible message about site requirements can inform users and improve the experience.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ANGULAR INTERVIEW QUESTION

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

AngularJS Interview Question #1

What are the basic steps to unit test an AngularJS filter?

(Question provided by Daniel Lamb)

Inject the module that contains the filter.

Provide any mocks that the filter relies on.

Get an instance of the filter using $filter('yourFilterName').

Assert your expectations.

Dependency injection is a powerful software design pattern that Angular employs to compose responsibilities through an intrinsic interface. However, for those new to the process, it can be puzzling where you need to configure and mock these dependencies when creating your isolated unit tests. The open-source project “Angular Test Patterns” is a free resource that is focused on dispelling such confusion through high-quality examples.

This question is useful since it can give you a feel for how familiar the candidate is with automated testing (TDD, BDD, E2E), as well as open up a conversation about approaches to code quality.

Sources:

https://github.com/daniellmb/angular-test-patterns/blob/master/patterns/filter.md

https://docs.angularjs.org/guide/unit-testing

AngularJS Interview Question #2

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

(Question provided by Daniel Lamb)

TL;DR Summary: 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.

Jank happens when your application cannot keep up with the screen refresh rate. To achieve 60 frames-per-second, you only have about 16 milliseconds for your code to execute. It is crucial that the scope digest cycles are as short as possible for your application to be responsive and smooth. Memory use and digest cycle performance are directly affected by the number of active watches. Therefore, it is best to keep the number of watches below 2,000. The open-source utility ng-stats gives developers insight into the number of watches Angular is managing, as well as the frequency and duration of digest cycles over time.

Caution: Be wary of relying on a “single magic metric” as the golden rule to follow. You must take the context of your application into account. The number of watches is simply a basic health signal. If you have many thousands of watches, or worse, if you see that number continue to grow as you interact with your page. Those are strong indications that you should look under the hood and review your code.

This question is valuable as it gives insight into how the candidate debugs runtime issues while creating a discussion about performance and optimization.

Sources:

https://github.com/kentcdodds/ng-stats

http://jankfree.org

Author Bio

Daniel has over 15 years experience specializing in large-scale front-end architecture and web development. Hire Daniel Now.

AngularJS Interview Question #3

How do you share data between controllers?

(Question provided by Tome Pejoski)

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)

The methods above are all correct, but are not the most efficient and easy to test.

Here is a good video explanation on egghead.io.

AngularJS Interview Question #4

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

(Question provided by Tome Pejoski)

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.

We only need to keep in mind what the difference between these directives is, so deciding which one to use totally depends on the task requirements.

AngularJS Interview Question #5

What is a digest cycle in AngularJS?

(Question provided by Tome Pejoski)

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.

For more information, take a look in the ng-book explanation: The Digest Loop and $apply

AngularJS Interview Question #6

Where should we implement the DOM manipulation in AngularJS?

(Question provided by Tome Pejoski)

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

Here is a detailed explanation.

AngularJS Interview Question #7

Is it a good or bad practice to use AngularJS together with jQuery?

(Question provided by Tome Pejoski)

It is definitely a bad practice. We need to stay away from jQuery and try to realize the solution with an AngularJS approach. jQuery takes a traditional imperative approach to manipulating the DOM, and in an imperative approach, it is up to the programmer to express the individual steps leading up to the desired outcome.

AngularJS, however, takes a declarative approach to DOM manipulation. Here, instead of worrying about all of the step by step details regarding how to do the desired outcome, we are just declaring what we want and AngularJS worries about the rest, taking care of everything for us.

Here is a detailed explanation.

Author Bio

Tome is a senior software engineer at Netcetera and has worked on various AngularJS, 2 of which are enterprise-level. Hire Tome Now.

hire-angularjs-developers.png

Hire top AngularJS developers now

CodementorX will find you the best engineers for your project.

AngularJS Interview Question #8

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

(Question provided by Jad Salhani)

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

AngularJS Interview Question #9

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

(Question provided by Jad Salhani)

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

AngularJS Interview Question #10

What is the difference between one-way binding and two-way binding?

(Question provided by Jad Salhani)

– One way binding implies that the scope variable in the html will be set to the first value its model is bound to (i.e. assigned to)

– Two way binding implies that the scope variable will change it’s value everytime its model is assigned to a different value

AngularJS Interview Question #11

Explain how $scope.$apply() works

(Question provided by Jad Salhani)

$scope.$apply re-evaluates all the declared ng-models and applies the change to any that have been altered (i.e. assigned to a new value) Explanation: scope.scope.apply() is one of the core angular functions that should never be used explicitly, it forces the angular engine to run on all the watched variables and all external variables and apply the changes on their values

Source: https://docs.angularjs.org/api/ng/type/$rootScope.Scope

AngularJS Interview Question #12

What directive would you use to hide elements from the HTML DOM by removing them from that DOM not changing their styling?

(Question provided by Jad Salhani)

The ngIf Directive, when applied to an element, will remove that element from the DOM if it’s condition is false.

AngularJS Interview Question #13

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

(Question provided by Jad Salhani)

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.

Sources:

https://docs.angularjs.org/api/ng/function/angular.copy

https://en.wikipedia.org/wiki/Object\_copying

AngularJS Interview Question #14

How would you make an Angular service return a promise? Write a code snippet as an example

(Question provided by Jad Salhani)

To add promise functionality to a service, we inject the “$q” dependency in the service, and then use it like so:

angular.factory('testService', function($q) {

return {

getName: function() {

var deferred = $q.defer();

//API call here that returns data

testAPI.getName().then(function(name) {

deferred.resolve(name);

});

return deferred.promise;

}

};

});

The $q library is a helper provider that implements promises and deferred objects to enable asynchronous functionality.

Source: https://docs.angularjs.org/api/ng/service/$q

15128858\_738189639662306\_946498680252809464\_o.jpg

Author Bio

Jad is a 5-star rated Codementor who has helped mentor many AngularJS users. Hire Jad now.

AngularJS Interview Quesion #15

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

(Question provided by Harikishore Tadigotla)

AngularJS Services are objects that provide separation of concerns to an AngularJS app. These can be created using a factory method or a service method. 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.

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

Services are the best may to evolve reusable API within and AngularJS app.

Overview:

AngularJS Services help create reusable components.

A Service can be created either using the service() method or the factory() method.

A typical service can be injected into another service or into an AngularJS Controller.

Sources:

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

http://www.tutorialspoint.com/angularjs/angularjs\_services.htm

3d9f646.jpg

Author Bio

Harikishore is a former lead developer for Nokia and has been a senior systems analyst for JP Morgan before. Hire Harikishore Now.

AngularJS Interview Question #16

When creating a directive, it can be used in several different ways in the view. Which ways for using a directive do you know? How do you define the way your directive will be used?

(Question provided by Nuno Brites)

When you create a directive, it can be used as an attribute, element or class name. To define which way to use, you need to set the restrict option in your directive declaration.

The restrict option is typically set to:

‘A’ – only matches attribute name ‘E’ – only matches element name

‘C’ – only matches class name

These restrictions can all be combined as needed:

‘AEC’ – matches either attribute or element or class name

For more information, feel free to check out the AngularJS documentation.

AngularJS Interview Question #17

When should you use an attribute versus an element?

(Question provided by Nuno Brites)

Use an element when you are creating a component that is in control of the template. Use an attribute when you are decorating an existing element with new functionality.

This topic is important so developers can understand the several ways a directive can be used inside a view and when to use each way.

Source: https://docs.angularjs.org/api/ng/service/$compile#directive-definition-object

Author Bio

Nuno is a full stack software engineer with 6 years of experience developing enterprise apps. Hire Nuno Now.

AngularJS Interview Question #18

How do you reset a $timeout, $interval(), and disable a $watch()?

(Question provided by Fouad Kada)

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);

To disable $watch(), we call its deregistration function. $watch() then returns a deregistration function that we store to a variable and that will be called for cleanup:

var deregisterWatchFn = $scope.$on('$destroy', function() {

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

deregisterWatchFn();

});

Author Bio

Fouad is a software engineer with 3 years of experience with AngularJS. Hire Fouad Now.

hire-angularjs-developers.png

Hire top AngularJS developers now

CodementorX will help you find you the best engineers for your project.

AngularJS Interview Question #19

Explain what is a $scope in AngularJS

(Question provided by Ashish Kumar)

Scope is an object that refers to the application model. It 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. Scopes are objects that refer to the model. They act as glue between controller and view.

This question is important as it will judge a developer's knowledge about a $scope object, and it is one of the most important concepts in AngularJS. Scope acts like a bridge between view and model.

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

AngularJS Interview Question #20

What are Directives?

(Question provided by Ashish Kumar)

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.

This question is important because directives define the UI while defining a single page app. Candidates need to be very clear about how to create a new custom directive or use the existing ones already pre-build in AngularJS.

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

AngularJS Interview Question #21

What is DDO Directive Definition Object?

(Question provided by Ashish Kumar)

“DDO is an object used while creating a custome directive. A standard DDO object has following parameters.

var directiveDefinitionObject = {

priority: 0,

template: '<div></div>', // or // function(tElement, tAttrs) { ... },

// or

// 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', // or // ['^parentDirectiveName', '?optionalDirectiveName', '?^optionalParent'],

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

return {

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

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

};

// or

// return function postLink( ... ) { ... }

}

// or

// link: {

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

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

// }

// or

// link: function postLink( ... ) { ... }

};

This question mainly judges whether candidate knows about creating custom directives.

Read more at https://docs.angularjs.org/guide/directive

photo.jpg

Author Bio

Ashish has 7 years of experience in software development and currently holds a 5-star rating on Codementor.io. Hire Ashish Now

AngularJS Interview Question #22

What is a singleton pattern and where we can find it in Angularjs?

(Question provided by Jon Oyanguren López)

Is a great pattern that restricts the use of a class more than once. We can find singleton pattern in angular in dependency injection and in the services.

In a sense, if the candidate does 2 times ‘new Object()‘ without this pattern, the candidate will be alocating 2 pieces of memory for the same object. With singleton pattern, if the object exists, it'll be reused.

Source: http://joelhooks.com/blog/2013/05/01/when-is-a-singleton-not-a-singleton/

AngularJS Interview Question #23

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

(Question provided by Jon Oyanguren López)

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

The interceptor is a factory that are registered in $httpProvider. There are 2 types of requests that go through the interceptor, request and response (with requestError and responseError respectively). This piece of code is very useful for error handling, authentication or middleware in all the requests/responses.

Source: https://docs.angularjs.org/api/ng/service/$http

Jon

Author Bio

Jon works as a full stack developer in AngularJS, JavaScript, and .NET. He is the CTO of iDriveYourCar and innovUp, and a teacher at CMU University. Hire Jon Now.

Angular Interview Question #24

How would you programatically change or adapt the template of a directive before it is executed and transformed?

(Question provided by Johann de Swardt)

The candidate should use the compile function. The compile function gives access to the directive’s template before transclusion occurs and templates are transformed, so changes can safely be made to DOM elements. This is very useful for cases where the DOM needs to be constructed based on runtime directive parameters.

Read more about it here.

Angular Interview Question #25

How would you validate a text input field for a twitter username, including the @ symbol?

(Question provided by Johann de Swardt)

The developer should use the ngPatterndirective to perform a regex match that matches Twitter usernames. The same principal can be applied to validating phone numbers, serial numbers, barcodes, zip codes and any other text input.

The official documentation can be found here.

Angular Interview Question #26

How would you implement application-wide exception handling in your Angular app?

(Question provided by Johann de Swardt)

Angular has a built-in error handler service called $exceptionHandler which can easily be overriden as seen below:

myApp.factory('$exceptionHandler', function($log, ErrorService) {

return function(exception, cause) {

if (console) {

$log.error(exception);

$log.error(cause);

}

ErrorService.send(exception, cause);

};

});

This is very useful for sending errors to third party error logging services or helpdesk applications. Errors trapped inside of event callbacks are not propagated to this handler, but can manually be relayed to this handler by calling $exceptionHandler(e) from within a try catch block.

Author Bio

Johann

Johann has 13 years of professional programming experience, and he’s a full-stack Javascript developer specialising in NodeJS, AngularJS and Ionic apps. Hire Johann Now.

AngularJS Interview Question #27

How do you hide an HTML element via a button click in AngularJS?

(Question provided by Nishant Kumar)

This can be done by using the ng-hide directive in conjunction with a controller to hide an HTML element on button click.

<div ng-controller="MyCtrl">

<button ng-click="hide()">Hide element</button>

<p ng-hide="isHide">Hello World!</p>

</div>

function MyCtrl($scope) {

$scope.isHide = false;

$scope.hide = function() {

$scope.isHide = true;

};

}

AngularJS Interview Question #28

How would you react on model changes to trigger some further action? For instance, say you have an input text field called email and you want to trigger or execute some code as soon as a user starts to type in their email.

(Question provided by Nishant Kumar)

This can be achieved by using $watch function in the controller.

function MyCtrl($scope) {

$scope.email = '';

$scope.$watch('email', function(newValue, oldValue) {

if ($scope.email.length > 0) {

console.log('User has started writing into email');

}

});

}

AngularJS Interview Question #29

How do you disable a button depending on a checkbox’s state?

(Question provided by Nishant Kumar)

The developer can use the ng-disabled directive and bind its condition to the checkbox’s state.

<body ng-app>

<label><input type="checkbox" ng-model="checked"/>Disable Button</label>

<button ng-disabled="checked">Select me</button>

</body>