SCOPE

http://oricalvo.com
ori.calvo@gmail.com

Scope Tree

- At runtime, multiple scope instances are created
- Organized into a tree structure
  - $rootScope is the root
- Each directive is bound to exactly one scope
  - However, the directive may decide to traverse the scope tree and monitors other scopes
- A directive is linked to a new scope or to an existing scope (surrounding)
  - A matter of definition

Scope Type

- Scope instance is represented by a JavaScript class named Scope

```
function Scope() {
     this.$id = nextUid();

     this.$$phase = this.$parent = this.$$watchers =
                    this.$$nextSibling = this.$$prevSibling =
                    this.$$childHead = this.$$childTail = null;

     this.$root = this;
     this.$$destroyed = false;
 }

Scope.prototype = {
     constructor: Scope,
     ...
}
```

Scope or Not ? This is the …

- How many scope instances are created for the following markup

```
<div ng-controller="HomeCtrl">
     <input ng-model="name" />

     <div>Hello, {{name}}</div>

     <ul>
         <li ng-repeat="point in points">
             <span>{{point.x}}, {{point.y}}</span>
         </li>
     </ul>
</div>
```

```
angular.module("MyApp", [])
     .controller("HomeCtrl", function ($scope) {
         $scope.points = [
             { x: 1, y: 2 },
             { x: 2, y: 3 },
             { x: 3, y: 4 },
         ];
     });
```

- *4 exactly ! Can you explain ?*

Why do we need it ?

- Scope is the link between directives and data model
  - The directive monitors the scope and automatically updates the DOM
- However, we usually use an ng-controller directive which causes Angular to instantiate a new object
- The data model can be stored inside the controller object making the scope instance redundant

Controller with no $scope

- Start with using the as syntax

```
<div ng-controller="HomeCtrl as ctrl">
     <div>Hello, {{ctrl.name}}</div>
     <button ng-click="ctrl.run()">Click me</button>
</div>
```

- Continue with writing plain JavaScript class

```
function HomeCtrl() {
     this.name = "Ng";
}

HomeCtrl.prototype.run = function () {
     console.log("run");
}
```

- Register the controller

```
angular.module("MyApp", [])
     .controller("HomeCtrl", HomeCtrl);
```

AltJS

- Most AltJS languages like Dart and Typescript offers class syntax (compiles to prototype)
- Using the as syntax we can easily integrate Angular with AltJS classes

```
module MyApp {
     class HomeCtrl {
         name: string;
         constructor() {
             this.name = "Ng";
         }

         run() {
             console.log("run");
         }
     }

     angular.module("MyApp", [])
         .controller("HomeCtrl", HomeCtrl);
};
```

as Syntax – How does it work ?

- Even when using the as syntax Angular still creates a scope instance

```
<div ng-controller="HomeCtrl as ctrl">
     <div>Hello, {{ctrl.name}}</div>
     <button ng-click="ctrl.run()">Click me</button>
</div>
```

- The scope instance is extended with an attribute named ctrl which references the controller instance
- An expression like ctrl.name is evaluated against the scope instance as any other expression

Still, why do we need a Scope ?

- According to previous slides it looks as if we can manage without $scope object
- So why does Angular create one ?
- Possible reasons
  - Scope inheritance
  - Scope API
  - ng-repeat use case

Scope Inheritance

- Angular allows the developer to inject data into one scope and be able to read it from a child scope

```
<div ng-controller="HomeCtrl">
     <div>Hello, {{name}}</div>
     <div ng-controller="ChildCtrl">
         <div>Hello child, {{name}}</div>
     </div>
</div>
```

- ChildCtrl does not need to set the name
  - It is taken from the parent

```
angular.module("MyApp", [])
     .controller("HomeCtrl", function ($scope) {
         $scope.name = "Ng";
     })
     .controller("ChildCtrl", function ($scope) {
     });
```

Scope Inheritance

- Is implemented by using JavaScript prototypical inheritance

```
function $new(isolate, parent) {
     var child;

     parent = parent || this;
     if (isolate) {
         child = new Scope();
     } else {
         if (!this.$$ChildScope) {
             this.$$ChildScope = function ChildScope() {
             };
             this.$$ChildScope.prototype = this;
         }
         child = new this.$$ChildScope();
     }

     child.$parent = parent;

     return child;
}
```

Scope Inheritance – Are you sure ?

- Automatically being able to read data from parent scope is error prone
- Be aware that setting the data on the child scope does not update the parent scope
- Below code is cleaner, don't you think ?

```
<div ng-controller="HomeCtrl as home">
     <div>Hello, {{home.name}}</div>

     <div ng-controller="ChildCtrl as child">
         <div>Hello child, {{home.name}}</div>
     </div>
</div>
```

http://oricalvo.com
ori.calvo@gmail.com

Scope API

- Beside being the bridge between data model and directive the Scope type offers some important API
  - Register watchers
  - Initiate dirty checking
  - Cleanup
  - Evaluate expressions
  - Publish/subscribe events

http://oricalvo.com
ori.calvo@gmail.com

Watcher

- Each scope contains a list of watchers
  - Named $$watchers
- A watcher consists of
  - Expression to be monitored
  - Listener to be notified when expression changes
  - The result of evaluating the expression
  - Other management flags

Registering a Watcher

- There are three different registration watchers
- Lets start with $watch

```
function HomeCtrl($scope) {
     $scope.name = "Ng";

     $scope.$watch("name", function (newValue, oldValue) {
         console.log("Name changed: " + oldValue + " --> " + newValue);
     });
}
```

- The expression is evaluated against $scope
  - Can use a function instead of an expression
- The function is a listener to be notified when expression changes

Registering a Watcher


```
function $watch(watchExp, listener, objectEquality) {
     var get = $parse(watchExp);

     var scope = this,
         array = scope.$$watchers,
         watcher = {
             fn: listener,
             last: initWatchVal,
             get: get,
             exp: watchExp,
             eq: !!objectEquality
         };

    array.unshift(watcher);

     return function deregisterWatch() {
         arrayRemove(array, watcher);
         lastDirtyWatch = null;
     };
}
```

```
function initWatchVal() {}
```

- initWatchVal is a special value to mark the watcher as uninitialized

Watcher Lifecycle

- Upon registration expression is not evaluated
- The watcher is considered as uninitialized
- The expression is evaluated on the next digest cycle
  - The last field is updated
  - The listener is always notified
    - Even if no change is detected
    - newValue and oldValue are the same
- During future digest cycles the listener is notified only if a change is detected

Comparing old and new Values

- Angular uses plain equal operator (==) to compare last to current value
- Great for comparing primitive values like String and Boolean
  - Not ideal for comparing objects/arrays

```
function $digest() {
     ...

     if ((value = watch.get(current)) !== (last = watch.last)) {
         watch.last = watch.eq ? copy(value, null) : value;
         watch.fn(value, ((last === initWatchVal) ? value : last), current);
     }

     ...
}
```

$watchCollection

- Watching an object

```
$scope.contact = {id: 1, name: “Ori”};

$scope.$watchCollection(
     function () {
         return $scope.contact;
     },
     function (newValue, oldValue) {
         console.log("Contact changed");
     });
```

- Watching an array

```
$scope.nums = [1,2,3];

$scope.$watchCollection(
     function () {
         return $scope.nums;
     },
     function (newValue, oldValue) {
         console.log("Nums changed");
     });
```

$watchCollection

- Detects changes inside an object
  - Examines only direct attributes
  - Comparison is done using plain equal operator

changeDetected is increased each time a change is detected

```
function $watchCollection(obj, listener) {
     var newValue;
     var oldValue;
     var veryOldValue;
     var changeDetected = 0;
     var changeDetector = $parse(obj, $watchCollectionInterceptor);
     ...

     function $watchCollectionInterceptor(_value) {
         ...
         return changeDetected;
     }

     function $watchCollectionAction() {
         ...
         listener(newValue, veryOldValue, self);
     }

     return this.$watch(changeDetector, $watchCollectionAction);
}
```

$watchGroup

- Receives an array of expressions (or functions)
- Registers each expression using $watch
- If one (or more) of the expressions changes fires the listener (only once)

```
$scope.contact = {
     id: 1,
     name: "Ori",
     email: "ori@gmail.com",
};

$scope.$watchGroup(
     ["contact.name", "contact.email"],
     function (newValue, oldValue) {
         console.log("Contact changed");
     });
```

$watchGroup - Challenge

- $watchGroup's listener accepts newValue and oldValue parameters
  - Each represents an array of new values and old values for all expressions
- When a single watcher is notified of a change $watchGroup cannot send immediately notification to the client since it still does not hold all values
- The notification must be queued and fired later

$watchGroup


```
function $watchGroup(watchExpressions, listener) {
     var oldValues = new Array(watchExpressions.length);
     var newValues = new Array(watchExpressions.length);
     var changeReactionScheduled = false;

     forEach(watchExpressions, function (expr, i) {
         var unwatchFn = self.$watch(expr, function watchGroupSubAction(value, oldValue) {
             newValues[i] = value;
             oldValues[i] = oldValue;

             if (!changeReactionScheduled) {
                 changeReactionScheduled = true;

                 self.$evalAsync(function () {
                     listener(newValues, oldValues, self);
                 });
             }
         });
     });
}
```

http://oricalvo.com
ori.calvo@gmail.com

$evalAsync

- Queues a request to execute an expression/function at a later time
- The request is executed at the next digest cycle
- If requested outside of a $apply phase a new cycle is enforced using $browser.defer
- If requested during a digest phase a new digest iteration is enforced immediately after the current one
- No additional cycle when executed during $apply phase (outside of $digest phase)

$evalAsync – When to use ?

- Whenever you face the same problem as $watchGroup
  - You registered multiple watchers
  - You want to execute some code after all watchers were notified
  - However, there is no guarantee that any watcher will be notified
- $evalAsync is executed after all watchers were executed

$watchGroup - Performance

- $watchGroup calls $evalAsync when a single expression changes
- However, the notification for a change is sent during a digest cycle
- This means that another digest cycle is enforced ☹
- However, another cycle must be performed because of the expression change (regardless of $evalAsync) ☺

Deep Watch

- $watch can be used to monitor a graph of objects
- Angular monitors the whole graph

```
$scope.contact = {
     id: 1,
     name: "Ori",
     address: {
         city: "Rehovot",
         street: "Yehiel Paldi“
     }
};
```

```
$scope.$watch(
     function () {
         return $scope.contact;
     },
     function (newValue, oldValue) {
         console.log("Contact changed");
     },
     true);
```

```
$scope.onClick = function () {
     $scope.contact.address.city += "X";
}
```

angular.equals

- Deep watching is implemented using angular.equals
- This is a general purpose API for comparing graph of objects
- Ignores fields that start with $

```
function equals(o1, o2) {
     var keySet = {};
     for (key in o1) {
         if (key.charAt(0) === '$' || isFunction(o1[key])) continue;
         if (!equals(o1[key], o2[key])) return false;
         keySet[key] = true;
     }

     for (key in o2) {
         if (!keySet.hasOwnProperty(key) && key.charAt(0) !== '$' &&
             o2[key] !== undefined && !isFunction(o2[key])) return false;
     }
     return true;
}
```

Initiate Dirty Checking

- As explained in previous chapter Angular by itself don't know when to perform dirty checking
- Most directive/service initiates dirty checking after invoking external code which may change the data model
- For example, ng-click

```
element.on(eventName, function (event) {
     var callback = function () {
         fn(scope, { $event: event });
     };

     scope.$apply(callback);
});
```

$apply

- Executes user function and then performs a digest cycle starting from the root scope
- Guards against sub-invocation of $apply

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

$apply - Performance

- Angular is pessimistic
- All built-in directives use $apply
- Angular cannot determines the scope of a change and therefore traverses all scopes
  - However, this behavior has performance impact
- We, as application writers do know the scope of a single change and may choose to “refresh” only that part of the DOM

$digest

- Walks the list of scope's watchers
- Asks every watcher for the current value
- Compares it to previous value
- If value changed, notifies the watcher
- Works in a recursive manner
  - Through out the scope tree
- If a change detected, runs another cycle
  - Stops if no change occurred
  - Or, if maximum allowed cycles was reached (10)

Digest Cycle


```
function $digest() {
     do {
         dirty = false; current = target;

         do {
             if ((watchers = current.$$watchers)) {
                 length = watchers.length;
                 while (length--) {
                     try {
                         watch = watchers[length];
                         if (watch changed) {
                             dirty = true;
                             watch.fn(newValue, lastValue);
                         }
                      }
                      catch (e) {
                         $exceptionHandler(e);
                     }
                 }
             }

             if (!(next = (current.$$childHead || (current !== target && current.$$nextSibling)))) {
                 while (current !== target && !(next = current.$$nextSibling)) {
                     current = current.$parent;
                 }
             }
         } while ((current = next));

         if (dirty && !(ttl--)) {
            throw $rootScopeMinErr('infdig', '{0} $digest() iterations reached. Aborting!\n', TTL);
         }
     } while (dirty); }
```

What the hell is this ?

Digest Cycle Performance

- See http://jsperf.com/angularjs-digest
- 100 scope instances
- Each scope has 100 watchers
- 10,000 watchers total
- On my machine (Core i7 3.5GHz)
  - Angular 1.0.2 → 237 cycles per seconds
  - Angular 1.3.3 → 2083 cycles per seconds
- Angular 2.0 should be even better

http://oricalvo.com
ori.calvo@gmail.com

Digest Cycle Performance

- The previously performance test uses watchers with expression (not a function)
- When using a function the performance might degrade significantly
- It is your responsibility to write efficient watchers
  - No DOM
  - No blocking method
  - No complex algorithm

$$postDigest

- Executes a function at the end of a digest cycle
- Only after model stabilization
- Does not force a digest cycle

```
function HomeCtrl($scope) {
     $scope.onClick = function () {
         $scope.$$postDigest(function () {
             console.log("POST digest");
         });
     }
}
```

```
function $digest() {
     do { // "while dirty" loop
     ...
     }
     while (dirty || asyncQueue.length);

     while (postDigestQueue.length) {
         try {
             postDigestQueue.shift()();
         }
         catch (e) {
             $exceptionHandler(e);
         }
     }
}
```

$$postDigest – When to use ?

- $evalAsync executes at the next digest cycle
  - Is the first step before traversing the scope tree
  - Before model is stable
  - Might be too soon
- Think of animation
  - You normally want to animate the whole change
  - Therefore need to wait for model stabilization

$applyAsync

- Executes a function at the next digest cycle
- Schedule a $apply request using browser.defer

```
function $applyAsync(expr) {
     var scope = this;

     expr && applyAsyncQueue.push($applyAsyncExpression);
     scheduleApplyAsync();

     function $applyAsyncExpression() {
         scope.$eval(expr);
     }
}
```

```
function $digest() {
     if (this === $rootScope && applyAsyncId !== null) {
         $browser.defer.cancel(applyAsyncId);
         flushApplyAsync();
     }
     ...
}
```

```
function scheduleApplyAsync() {
     if (applyAsyncId === null) {
         applyAsyncId = $browser.defer(function() {
             $rootScope.$apply(flushApplyAsync);
         });
     }
}
```

- Executing $digest before the timer causes $applyAsync requests to be executed but the timer is cancelled

$applyAsync vs. $evalAsync

- $applyAsync
  - Queues an $apply request
  - If invoked during apply phase does not cause additional apply
- $evalAsync
  - Queues a $digest request
  - If invoked during digest cycle does not cause additional digest
- Think of $evalAsync as $digestAsync

$on

- Scope instance allows you to subscribe to events
- Later on you can raise an event using
  - $emit
  - $broadcast

```
function AuthService($rootScope) {
     this.$rootScope = $rootScope;
}

AuthService.prototype.logout = function () {
     this.$rootScope.$broadcast("logout");
}
```

```
function HomeCtrl($scope, AuthService) {
     $scope.$on("logout", function () {
         console.log("User logged out");
     });

     $scope.logout = function () {
         AuthService.logout();
     }
}
```

$emit vs. $broadcast

- $emit behaves like most DOM events
  - Triggers at the specified scope and through its ancestors until $rootScope is reached
- $broadcast triggers at the specified scope and through its children
  - In a recursive manner
- You may consider using custom event mechanism

$broadcast Performance

- Scope tree may be deep
- $broadcast need to walk through many Scope instances → Not efficient
- Angular maintains a counter per event name at each scope instance
- In case counter is zero no need to traverse children

```
$broadcast: function(name, args) {
     var target = this,current = target, next = target;

     while ((current = next)) {
         ...
         if (!(next = ((current.$$listenerCount[name] && current.$$childHead) ||
             (current !== target && current.$$nextSibling)))) {
             while (current !== target && !(next = current.$$nextSibling)) {
                 current = current.$parent;
             }
         }
     }
}
```

$on – Cleanup

- Registering to an event using $on means that as long as the scope instance is a live the registered handler (+ dependencies) is alive too
- You are responsible for deregistering as soon as possible

```
var off = $scope.$on("logout", function () {
     console.log("User logged out");

     off();
});
```

Scope Disposal

- Some Scope instances are short lived
- Think about a controller inside ng-if
  - The controller's scope should be destroyed each time ng-if is false

```
<div ng-controller="HomeCtrl">
     <button ng-click="toggleAdmin()">{{(showAdmin ? "Hide Admin" : "Show Admin")}}</button>

     <div ng-if="showAdmin">
         <div ng-controller="AdminCtrl">
             <h1>Admin Zone</h1>
         </div>
     </div>
</div>
```

Scope Disposal

- Broadcasts destroy event
  - Controller may use this event to implement custom disposal behavior
- Detaches current scope (and its children) from the scope tree
- Disables public API
- Clears internal references like $$listeners and $$watchers

Scope Disposal


```
function $destroy() {
     var parent = this.$parent;

     this.$broadcast('$destroy');
     this.$$destroyed = true;

     if (parent.$$childHead == this) parent.$$childHead = this.$$nextSibling;
     if (parent.$$childTail == this) parent.$$childTail = this.$$prevSibling;

     if (this.$$prevSibling) this.$$prevSibling.$$nextSibling = this.$$nextSibling;
     if (this.$$nextSibling) this.$$nextSibling.$$prevSibling = this.$$prevSibling;

     this.$destroy = this.$digest = this.$apply = this.$evalAsync = this.$applyAsync = noop;
     this.$on = this.$watch = this.$watchGroup = function () { return noop; };

     this.$$listeners = {};

     this.$parent = this.$$nextSibling = this.$$prevSibling = this.$$childHead =
         this.$$childTail = this.$root = this.$$watchers = null;
}
```

Controller Disposal

- Controller instance is a user defined object which Angular has no idea how it looks like
  - Therefore there is no Controller disposal logic
- Use $destroy event to simulate that

```
function AdminCtrl($scope) {
     console.log("AdminCtrl created");

     $scope.$on("$destroy", function () {
         console.log("AdminCtrl destroyed");
     });
}
```

$eval

- A small wrapper around $parse
- Evaluates the expression against the current scope
- Can overrides the values inside scope with locals

```
function HomeCtrl($scope) {
     $scope.x = 10;
     $scope.y = 20;

     $scope.message = "NONE";

     $scope.run = function () {
         $scope.message = $scope.$eval("x + ' ' + y", { x: 11 });
     }
}
```

Summary

- There is no magic with Angular dirty checking mechanism
- You should profile your application to see that the dirty checking is kept simple
  - 1 cycle per user action
  - 2 iterations per cycle
- Use $watch to update DOM not state