UNDERSTANDING ANGULARJS LIFECYCLE

Ori calvo @2015

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

Objectives

- Get deeper into AngularJS building blocks
- Dependency Injection
- Digest cycle
- Watchers
- Modules

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 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>
```

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

- 4 exactly ! Can you explain ?

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,
     ...
}
```

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

Scope Hierarchy

- A controller can be nested under a parent controller
  - Very common scenario
- A controller usually represents a UI component which is modular and encapsulated
- So, should we care ? Yes …
  - State
  - Communication

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

Scope Hierarchy


```
<body ng-controller="RootCtrl">
     <div class="home-view" ng-controller="HomeCtrl">
          Name: <input type="text" ng-model="name" />
         <span>Hello, {{name}}</span>
     </div>
</body>
```

```
function RootCtrl($scope) {
     $scope.name = "Roni";
}
```

```
function HomeCtrl($scope) {
     //$scope.name = "Ori";
}
```

Scope Hierarchy

- Controller's scope inherits from its parent scope
- This is a prototypically inheritance
  - When reading an attribute from child scope the value may come from parent scope
  - When writing a value into child scope it will never be written into the parent !!!
- Angular supports isolated scope which does not inherit from its parent
  - However, you can always access the parent object using $scope.$parent

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>
```

$rootScope

- The default root scope which is created automatically
  - Even when no ng-controller is present
- Is attached to the ng-app DOM element
- You can put some data into $rootScope and it will be available in all other scopes
- A way to share data between different controllers
- Like any other global state it should be avoided

$rootScope


```
<body>
     <div class="home-view" ng-controller="HomeCtrl"></div>

     <div class="second-view" ng-controller="SecondCtrl">
         <span>{{globalData}}</span>
     </div>
</body>
```

```
function HomeCtrl($scope, $rootScope) {
     $rootScope.globalData = "This is a global data";
}
```

```
function SecondCtrl($scope, $rootScope) {
}
```

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

Why do we need $scope ?

- 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
- Hence, the data model can be stored inside the controller object making the scope instance redundant

Avoiding $scope

- Assuming scope inheritance is not used, it would be nice to attach bound properties to the controller itself
- Allows you to use prototype based JavaScript classes

```
<div class="home-view" ng-controller="HomeCtrl as ctrl">
     <div>{{ctrl.counter}}</div>
     <button ng-click="ctrl.inc()">Inc</button>
</div>
```

```
function HomeCtrl($scope) {
     this.counter = 0;
}

HomeCtrl.prototype.inc = function () {
     this.counter++;
}
```

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 curly braces expression

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

So, can we forget $scope ?

- According to previous slides it looks as if we can manage without $scope object
- Not really
- $scope is very useful
  - Scope API
  - Custom directive which resembles ng-repeat

Controller Ctor – Second Look

- You can change the order of the parameters and everything still works

```
function HomeCtrl($rootScope, $scope) {
     $rootScope.globalData = "This is a global data";
}
```

- You cannot change their names

```
function HomeCtrl($rootScope2, $scope) {
     $rootScope.globalData = "This is a global data";
}
```

- Exception is thrown

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

Dependency Injection

- When Angular invokes a method
  - For example, a controller's constructor
- It analyzes the method and tries to understand the list of dependencies to be injected
- A parameter that cannot be resolved creates an error
- Injection logic is encapsulated under a built-in service named $injector

$injector Service

- Usually you are not using it directly
- $injector knows how to resolve a service name to a service reference

```
$injector.invoke(function ($injector, $rootScope) {
     console.log(!!$injector);
     console.log(!!$rootScope);
});
```

- $injector does not offer registration methods – See module later
- But how does $injector detect the dependency list?

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

Dependency Injection – How?

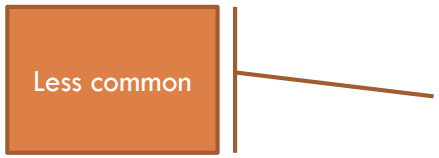
- $injector looks for special metadata attached to the function being invoked
- If not found it uses toString on the specified function and try to parse the dependency list from the function's source code itself !!!

```
$injector.invoke(function ($rootScope) {
     console.log(!!$rootScope);
});
```

- This means that we just need to name the parameters correctly
- However, what if we do not control parameter's name?

Injection Metadata

- A minification tool usually changes function parameters names
- This means that Angular sees minified unrecognized parameters names and fails
- We can specify dependency metadata manually




```
var func = function (rs) {
     console.log(!!rs);
}
func.$inject = ["$rootScope"];
$injector.invoke(func);
```

```
$injector.invoke(["$rootScope", function ($rootScope) {
     console.log(!!$rootScope);
}]);
```

Controller with Metadata

- Previous technique for manually specifying dependency list can be applied to controllers too
  - Previous technique for manually specifying dependency list can be applied to controllers too

```
var HomeCtrl = ["$scope", function HomeCtrl(aaa) {
     console.log(!!aaa);
}];
```

```
HomeCtrl.$inject = ["$scope"]; function HomeCtrl(aaa) {
     console.log(!!aaa);
};
```

Avoiding Globals

- All examples up until now declared a controller in the global scope
- This is considered a bad practice
  - Increases the chances for name collision
- We would like to hide the controller from global scope but still to be instantiable by Angular
- Enter the module world …
- Starting Angular 1.3 you cannot longer define a global controller

Module

- A container for other Angular entities like controllers, directives and providers
- You can think of it as a namespace
  - But not only
  - May contain some initialization and configuration logic
- Has dependencies on other modules
- Usually you define at least one module and put some initialization code inside it

Module

- Define a new module named myApp

```
angular.module(“myApp", []);
```

- The second parameter (empty array) represents the dependency list
  - You should keep it
  - Without it the module function behaves differently
- Getting a reference to an existing module

```
var myApp = angular.module("myApp");
console.log(!!myApp);
```

Load on Demand

- A module is loaded by Angular only if requested explicitly
- Not loading a module means that all controller/services/directives that are defined inside the module are not available to the application
- In many cases a module is loaded because it is specified as a dependency of other module being loaded

```
angular.module('MyApp', ["ngRoute", "ngSanitize"])
```

- How does the root module is loaded ?

ng-app Directive

- ng-app directive may specify the name of a module
- Angular waits for DOM ready event and then initializes the module and its dependencies

```
<html ng-app="myApp">
    ...
</html>
```

- Once loaded, all controllers/services/directives are available to the application

Controller Registration

- Once you define a module you can register your controller into it
  - Thus avoiding global controller function
- Angular automatically looks for the controller inside the module

```
angular.module("myApp").controller("HomeCtrl", function ($scope) {
     console.log("HomeCtrl ctor");
});
```

Controller Registration + Metadata

- Below code is considered best practice for controller registration

```
angular.module("myApp").controller("HomeCtrl", ["$scope", function ($scope) {
     console.log("HomeCtrl ctor");
}]);
```

- You may consider ng-annotate for automatic generation of metadata

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

Scripts Loading

- In most cases, modules/controllers are defined inside separate files
- This means that your project may contain tens of files
- Angular does not help with script loading
- You should add reference to every script inside your HTML
- Or, use other libraries to load scripts asynchronously
  - RequireJS is a well known solution
  - Community has no strong opinion regarding RequireJS integration

Automatic DOM Update


```
angular.module("myApp").controller("HomeCtrl", function ($scope) {
    $scope.contacts = [];
    $scope.add = function () {
        $scope.contacts.push({
                name: $scope.name
            });
    }
});
```

```
<div class="home-view" ng-controller="HomeCtrl">
    <ul>
        <li ng-repeat="contact in contacts">
            <span>{{contact.name}}</span>
        </li>
    </ul>
    <div>
        <input type="text" ng-model="name" />
        <button ng-click="add()">Add</button>
    </div>
</div>
```

- Is the newly added contact displayed ?

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

Automatic DOM Update

- Traditionally, automatic DOM update is implemented using observation mechanism
  - For example, KnockoutJS
- Using observation means
  - Model object is written in a special way
  - The developer must change model values using dedicated API which informs the framework about the change
  - Knockout uses ()
  - Backbone uses get and set methods

Detecting Changes

- In previous slides we explained that Angular supports plain JavaScript object as the Model
- However, plain JavaScript object does not support observation capabilities
- This means that Angular has no way to detect that a change was done to the model
- Almost
  - Object.observe (ECMA Script v6)
  - Dirty checking

Dirty Checking – How ?

- By default angular does not monitor any object
- A controller/directive/service may request Angular to monitor a specific expression by installing a watcher
  - For example, ng-model directive
- The watcher is held inside the scope instance
- Angular stores the original value of the expression
- When requested, Angular fetches the current value and compares it to the previous one
- If value changes it informs the directive
- The directive updates the DOM

Digest Cycle

- Walks the scope tree
- For each scope iterates the list of watchers
- Asks every watcher for the current value
- Compares it to previous value
- If values differ, notifies the watcher
- Runs another cycle
  - Stops if no change occurred
  - Or, if maximum allowed cycles was reached (10)

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

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

Dirty Checking – When ?

- Angular by itself don't know when to perform dirty checking
- Most directives/services initiate 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 traverse 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 part of the DOM
  - Use $digest instead

Dirty Checking – Side Effects

- There are cases where you listen to DOM events which are outside of Angular spectrum
- In those cases Angular cannot invoke the digest cycle and DOM is not updated
- Use $scope.$apply

```
angular.module("myApp").controller("HomeCtrl", function ($scope) {
    $scope.status = "Running ...";

    setTimeout(function () {
        $scope.$apply(function(){
            $scope.status = "Done";
        });
    }, 1500);
});
```

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 methods
- 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 2nd parameter is a function to be notified when expression changes

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");
     });
```

$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");
     });
```

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";
}
```

$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

$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 to
- 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>
```

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

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");
     });
}
```

Summary

- Don't declare global controllers
- Prefer using the as syntax
- Module is a must
- Use $watch for monitoring changes in the DOM
- Automatic DOM update is implemented using naïve dirty checking mechanism
  - A source for performance issues