What are Scopes?

Scope (api/ng/type/$rootScope.Scope) is an object that refers to the application model. It is an execution context for expressions (guide/expression). Scopes are arranged in hierarchical structure which mimic the DOM structure of the application. Scopes can watch expressions (guide/expression) and propagate events.

Scope characteristics

Scopes provide APIs ($watch (api/ng/type/$rootScope.Scope#$watch)) to observe model mutations.

Scopes provide APIs ($apply (api/ng/type/$rootScope.Scope#$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 (guide/directive#isolating-the-scope-of-a-directive) for more information.

Scopes provide context against which expressions (guide/expression) 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 (guide/compiler) phase the directives (api/ng/provider/$compileProvider#directive) set up `$watch` (api/ng/type/$rootScope.Scope#$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.

script.js () index.html () Edit in Plunker

```
angular.module('scopeExample', [])
.controller('MyController', ['$scope', function($scope) {
  $scope.username = 'World';

  $scope.sayHello = function() {
    $scope.greeting = 'Hello ' + $scope.username + '!';
  };
}]);
```

Your name: greet

{{greeting}}

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 (guide/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 (api/ng/service/$rootScope), but may have any number of child scopes.

The application can have multiple scopes, because directives (guide/directive) can create new child 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.

The section Directives that Create Scopes (guide/scope#directives-that-create-scopes) has more info about which directives create scopes.

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.

index.html () | script.js () | style.css () | Edit in Plunker

```
<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` (api/ng/directive/ngApp) 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:

1. 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.
2. The debugger allows you to access the currently selected element in the console as `$0` variable.
3. To retrieve the associated scope in console execute: `angular.element($0).scope()`

The `scope()` function is only available when `$compileProvider.debugInfoEnabled()` (api/ng/provider/$compileProvider#debugInfoEnabled) is true (which is the default).

Scope Events Propagation

Scopes can propagate events in similar fashion to DOM events. The event can be broadcasted (api/ng/type/$rootScope.Scope#$broadcast) to the scope children or emitted (api/ng/type/$rootScope.Scope#$emit) to scope parents.

script.js () index.html () Edit in Plunker

```
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 `$apply` (api/ng/type/$rootScope.Scope#$apply) method. Only model modifications which execute inside the `$apply` method will be properly accounted for by AngularJS. For example if a directive listens on DOM events, such as `ng-click` (api/ng/directive/ngClick) it must evaluate the expression inside the `$apply` method.

After evaluating the expression, the `$apply` method performs a `$digest` (api/ng/type/$rootScope.Scope#$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 `$watch` es are running. If a `$watch` changes the value of the model, it will force additional `$digest` cycle.

1. **Creation**

   The root scope (api/ng/service/$rootScope) is created during the application bootstrap by the $injector (api/auto/service/$injector). During template linking, some directives create new child scopes.

2. **Watcher registration**

   During template linking, directives register watches (api/ng/type/$rootScope.Scope#$watch) on the scope. These watches will be used to propagate model values to the DOM.

3. **Model mutation**

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

4. **Mutation observation**

   At the end of `$apply` , AngularJS performs a $digest (api/ng/type/$rootScope.Scope#$digest) cycle on the root scope, which then propagates throughout all child scopes. During the `$digest` cycle, all `$watch` ed expressions or functions are checked for model mutation and if a mutation is detected, the `$watch` listener is called.

5. **Scope destruction**

   When child scopes are no longer needed, it is the responsibility of the child scope creator to destroy them via scope.$destroy() (api/ng/type/$rootScope.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 (guide/compiler) matches directives (api/ng/provider/$compileProvider#directive) against the DOM template. The directives usually fall into one of two categories:

- Observing directives (api/ng/provider/$compileProvider#directive), such as double-curly expressions `{{expression}}` , register listeners using the $watch() (api/ng/type/$rootScope.Scope#$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 (api/ng/directive/ngClick), register a listener with the DOM. When the DOM listener fires, the directive executes the associated expression and updates the view using the $apply() (api/ng/type/$rootScope.Scope#$apply) method.

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

Directives that Create Scopes

In most cases, directives (api/ng/provider/$compileProvider#directive) and scopes interact but do not create new instances of scope. However, some directives, such as ng-controller (api/ng/directive/ngController) and ng-repeat (api/ng/directive/ngRepeat), create new child scopes and attach the child scope to the corresponding DOM element.

A special type of scope is the `isolate` scope, which does not inherit prototypically from the parent scope. This type of scope is useful for component directives that should be isolated from their parent scope. See the directives guide (guide/directive#isolating-the-scope-of-a-directive) for more information about isolate scopes in custom directives.

Note also that component directives, which are created with the .component() (api/ng/type/angular.Module#component) helper always create an isolate scope.

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 (api/ng/directive/ngController)).
- Controllers define methods (behavior) that can mutate the model (properties on the scope).
- Controllers may register watches (api/ng/type/$rootScope.Scope#$watch) on the model. These watches execute immediately after the controller behavior executes.

See the ng-controller (api/ng/directive/ngController) 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.

```
$scope.users = [
  {name: "Mary", points: 310},
  {name: "June", points: 290},
  {name: "Bob", points: 300}
];
```

By Reference

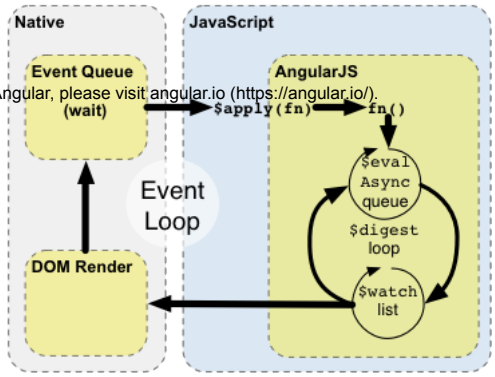
- Watching *by reference* (scope.$watch (api/ng/type/$rootScope.Scope#$watch) `(watchExpression, listener)`) detects a change when the whole value returned by the watch expression switches to a new value. If the value is an array or an object, changes inside it are not detected. This is the most efficient strategy.
- Watching *collection contents* (scope.$watchCollection (api/ng/type/$rootScope.Scope#$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.$watch (api/ng/type/$rootScope.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.

1. 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).
2. The event's callback gets executed. This enters the JavaScript context. The callback can modify the DOM structure.
3. 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.



1. Enter the AngularJS execution context by calling scope (guide/scope) . $apply (api/ng/type/$rootScope.Scope#$apply) `(stimulusFn)` , where `stimulusFn` is the work you wish to do in the AngularJS execution context.
2. AngularJS executes the `stimulusFn()` , which typically modifies application state.
3. AngularJS enters the $digest (api/ng/type/$rootScope.Scope#$digest) loop. The loop is made up of two smaller loops which process $evalAsync (api/ng/type/$rootScope.Scope#$evalAsync) queue and the $watch (api/ng/type/$rootScope.Scope#$watch) list. The $digest (api/ng/type/$rootScope.Scope#$digest) loop keeps iterating until the model stabilizes, which means that the $evalAsync (api/ng/type/$rootScope.Scope#$evalAsync) queue is empty and the $watch (api/ng/type/$rootScope.Scope#$watch) list does not detect any changes.
4. The $evalAsync (api/ng/type/$rootScope.Scope#$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.
5. The $watch (api/ng/type/$rootScope.Scope#$watch) list is a set of expressions which may have changed since last iteration. If a change is detected then the `$watch` function is called which typically updates the DOM with the new value.
6. Once the AngularJS $digest (api/ng/type/$rootScope.Scope#$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.

1. During the compilation phase:
    1. the ng-model (api/ng/directive/ngModel) and input (api/ng/directive/input) directive (guide/directive) set up a `keydown` listener on the **`<input>`** control.
    2. the interpolation (api/ng/service/$interpolate) sets up a $watch (api/ng/type/$rootScope.Scope#$watch) to be notified of `name` changes.
2. During the runtime phase:
    1. Pressing an '`X`' key causes the browser to emit a `keydown` event on the input control.
    2. The input (api/ng/directive/input) directive captures the change to the input's value and calls $apply (api/ng/type/$rootScope.Scope#$apply) `("name = 'X';")` to update the application model inside the AngularJS execution context.
    3. AngularJS applies the `name = 'X';` to the model.
    4. The $digest (api/ng/type/$rootScope.Scope#$digest) loop begins
    5. The $watch (api/ng/type/$rootScope.Scope#$watch) list detects a change on the `name` property and notifies the interpolation (api/ng/service/$interpolate), which in turn updates the DOM.
    6. AngularJS exits the execution context, which in turn exits the `keydown` event and with it the JavaScript execution context.
    7. The browser re-renders the view with the updated text.