AngularJs Performance Improvement Guide

Purpose/Justification

The Project Scope Statement (PSS) identifies the preliminary scope of a system. The PSS establishes a common understanding of the project scope among project stakeholders. Most important, it establishes not only what is in scope but also what is out of scope for the project.

Minimize/Avoid Watchers

**Watches are set on:**

* $scope.$watch
* {{ }} type bindings
* Most directives (i.e. ng-show)
* Scope variables scope: { bar: '='}
* Filters {{ value | myFilter }}
* ng-repeat

**Watchers (digest cycle) run on**

User action (ng-click etc). Most built in directives will call $scope.apply upon completion which triggers the digest cycle.

* ng-change
* ng-model
* $http events (so all ajax calls)
* $q promises resolved
* $timeout
* $interval
* Manual call to $scope.apply and $scope.digest

Avoid ng-repeat without “track By”.

If you have to use ng-repeat use infinite scrolling or pagination

In addition to infinite scroll, make sure to use **track by** when possible.

For example, a unique step id, is a good value to track by when doing an ng-repeat.

**<li ng-repeat="Task in Tasks track by Task.Id></li>**

Use Bind once when possible

Angular 1.3 added **‘::’** notation to allow one time binding. In summary, Angular will wait for a value to stabilize after its first series of digest cycles, and will use that value to render the DOM element. After that, Angular will remove the watcher forgetting about that binding.

If you are on pre-1.3 version of Angular you can use this library to achieve similar results:  
<https://github.com/Pasvaz/bindonce>

Use $watchCollection instead of $watch

**$watch** with only 2 parameters, is fast. However, Angular supports a 3rd parameter to this function, that can look like this: **$watch ('value', function(){}, true)**. The third parameter, tells Angular to perform deep checking, meaning to check every property of the object, which could be very expensive.

To address this performance issue, angular added **$watchCollection('value', function(){}).** **$watchColleciton** acts almost like $watch with a 3rd parameter, except it only checks the first layer of object’s properties, thus greatly improving the performance.

**Official doc:**  
<https://code.angularjs.org/1.3.15/docs/api/ng/type/$rootScope.Scope#$watchCollection>

Avoid repeated filters and cache data whenever possible

One time binding does not seem to play well with filters. There seems to be work arounds to make it works, but I think it’s cleaner and more intuitive to simply assign the needed value to a variable (or set it as a property on an object, if you are dealing with a lot of variables).

For example,

instead of:

**{{'DESCRIPTION' | translate }}**

You can do:  
– In JavaScript

**$scope.description:$translate.instant('DESCRIPTION')**  
– In HTML **{{::description}}**

Or instead of:

**{{step.time\_modified | timeFormatFilter}}**

You can Do

* In JavaScript

**var timeFormatFilter = $filter('timeFormatFilter');**

**step.time\_modified = timeFormatFilter(step.time\_modified);**

* In HTML

**{{::Path.time\_modified}}**

Debounce ng-model

If you know there is going to be a lot of changes coming from an ng-model, you can de-bounce the input.

For example if you have a search input like Google, you can de-bounce it by setting the following ng-model option: **ng-model-options="{ debounce: 250 }.**

This will ensure that the digest cycle due to the changes in this input model will get triggered no more than once per 250ms.

<https://docs.angularjs.org/api/ng/directive/ngModelOptions>

Use ng-if instead of ng-show

**ng-show** will render an element, and use **display:none** to hide it, **ng-if** will actually removes the element from DOM, and will re-create it, if it’s needed.

You may need ng-show for an element that toggles on an off often, but for 95% of the time, ng-if is a better way to go.

Use console.time to benchmark your functions

**console.time** is a great API, that particularly helpful when debugging issues with Angular performance. It’s better to place a number of those calls throughout the code, to confirm that re-factoring was improving the performance.

<https://developer.mozilla.org/en-US/docs/Web/API/Console/time>

*Note: If console.time is not precise enough for your needs, you can get a more accurate reading using performance.now(). You will have to do your own math, if you choose to take this path.*

Use native JavaScript or Lodash for slow functions

Lodash boosts your application performance by simply rewriting some of the basic logic instead of relying on inbuilt AngularJS methods. If Lodash is not included in your application, then you would probably need to re-write everything in native JavaScript.

Use scope.$evalAsync

If you try to manually activate the digest cycle while it’s already running, you could get an error. To prevent this from happening, use **scope.$evalAsync** instead of $apply when you need to initiate the digest cycle. It queues up operations to be executed at the end of the current cycle without setting off a new one.

Use $cacheFactory

If you need to store data that you might need to recalculate later, use the **$cacheFactorydirective** . It works like any other memorization method.

Tight Scoping

Keep your variables scoped tightly so that the JavaScript garbage collector can free up some memory every now and then.