Top 5 AngularJS Development Anti-Pattterns

for large scale enterprise applications

Donald Lamar Davis II

XTIVIA Enterprise Architect

A Publication of
XTIVIA

TABLE OF CONTENTS

1. Introduction

When developing an AngularJS application there are quite a few blogs, documents and tutorials about how to develop an AngularJS application. What many of those items, especially tutorials, do not cover, are the practices to avoid when creating a scalable and performant application that will work well with large sets of data. As the application is executing within the browser and is therefore a single user application the recommendations below will be centered upon the following themes.

- Simplify application logic and structure.
- Conflicting JavaScript framework usage.
- Minimize browser to server round trips and JavaScript processing.
- Retrieving data sets in an efficient manner.
- Proper memory management.

"Performance is a topic near and dear to my heart, which is why I wrote this whitepaper about creating a scalable, performant AngularJS application. I sincerely hope it adds value for our customers."

-- Don Davis,
XTIVIA Enterprise Architect

As you can tell by the themes we have chosen performance is a topic near and dear to my heart. There are quite a few other anti-patterns described on the web and they are very helpful tips. We will list those quickly here and go into detail about some practices in the following chapters. The following chapters will describe the Anti-Patterns tied to the themes we listed earlier.

2. Anti-Patterns & Resolution

2.1 Simplify Application Logic and Structure

2.1.1 Anti-Pattern #1 – Controller "Bloat"

One of the first places to look for anti-patterns in your application is "bloated" controllers. This is typically a controller that incorporates several practices that are anti-patterns. The following are examples of this -

1. Large number of uses of $scope. For example $scope.variableA = "some text".
2. Storing model objects as part of the current scope. For example $scope.objectA = {name:'Model Name'}.
3. Direct manipulation of the DOM with jQuery calls. For example $('.class1').value('some text').
4. Implementing services as methods of the controller. For example $scope.callService = function(paramOne) { $http.get(paramOne); };

Now that we have explained some of the common sub-optimal practices for controller implementation we will describe the best practices for implementing these patterns. The first two bullet items can be overcome with the use of "controller as" when using a controller. This is an example of using "controller as" to implement a controller that adheres to the SRP (Single Responsibility Principle) model.

The view will have html that defines the "controller as" attribute.

```
<div ng-controller="Company as company">
  <h1></h1>
  <article ng-controller="Customers as customers">
    <h2></h2>
    <ul ng-repeat="c in customers.customers">
      <li></li>
    </ul>
  </article>
</div>
```

2.1.1 Anti-Pattern #1 – Controller “Bloat” (continued)

The corresponding controller implementation removes the $scope references for readability. Another benefit is that it removes some of the scoping issues with nesting or extending controllers using the Directive Controller pattern (more about this pattern later).

```
angular.module('app')
    .controller('Customers', [function() {
      var vm = this;
      vm.title = 'Customers';
      vm.customers = [
        {name: 'HP'}, {name: 'DELL'}, {name: 'Amazon'}, {name:
'Google'}
        ];
    }]);
```

As can be seen from the example the implementation of the ViewModel(vm) removes a lot of messy $scope assignments and makes the HTML easier to read and understand the current scope by using the “Controller as vm” naming convention.

As mentioned earlier Directive Controllers also can benefit from this same implementation pattern. The Directive controller also corrects the third implementation mistake listed above. It does this by abstracting DOM (Document Object Model) manipulation into a controller that can be dependency injected and customized via inheritance for reuse in multiple locations in an application.

2.1.1 Anti-Pattern #1 – Controller “Bloat” (continued)

```
angular.module('directivesModule')
.directive('directiveWithControllerAs', function () {

      var controller = function () {
              var vm = this;

              function init() {
                  vm.items = angular.copy(vm.datasource);
              }

              init();

              vm.addItem = function () {
                  vm.add();

                  //Add new customer to directive scope
                  vm.items.push({
                      name: 'New Directive Controller Item'
                  });
              };
      };

      var template = '<button ng-click="vm.addItem()">Add
Item</button>' +
                     '<ul><li ng-repeat="item in vm.items">{{
::item.name }}</li></ul>';

      return {
          restrict: 'EA', //Default for 1.3+
          scope: {
              datasource: '=',
              add: '&',
          },
          controller: controller,
          controllerAs: 'vm',
          bindToController: true, //required in 1.3+ with
controllerAs
          template: template
      };
  });
```

2.1.1 Anti-Pattern #1 – Controller "Bloat" (continued)

To solve the fourth and final implementation mistake would require the implementation of a factory to implement the service functionality. A factory is used in this case as a singleton is sufficient for my example needs. If multiple instances of a service are required, then a service is used. The following is an example of the factory implementation.

```
angular.module('app')
    .factory('SimpleService', ['$http',function($http) {
        return {
            serviceCall: function(paramOne) {
                return $http.get(paramOne);
            }
        }
    }]);
```

After this service is defined it can be injected as a dependency for a controller.

2.2 Avoid Conflicting JavaScript Framework Usage

2.2.1 Anti-Pattern #2 – JQuery Usage

We recommend limiting jQuery usage in an AngularJS application and in particular for DOM manipulation. This is mainly because of the following three reasons –

1. When you execute jQuery code, you need to call $digest() yourself to update AngularJS watchers. For many cases, there is an AngularJS solution which is tailored for AngularJS and can be a better fit inside the AngularJS application compared to jQuery (e.g. ng-click or the event system).
2. Single page applications as the name implies spend a lot of time on the same page. As the application is operating within the current page any applications that are not designed to reclaim memory efficiently (which AngularJS does) will cause the page to exhibit slower and slower performance as functionality is exercised on the current page.
3. Cleaning up is not actually the easiest thing to do and analyze in a browser based application. There is no way to call a garbage collector from the script (in the browser). You may also end up with detached DOM trees. These DOM trees create a duplicate of the DOM memory for each detached DOM Tree. So if there are two detached DOM trees then 3x memory is consumed for the current page's DOM.

As AngularJS provides an alternative implementation to jQuery's selector mechanism it is recommended to use this for DOM selection and manipulation as often as possible. Also as mentioned in point 1 above, directives and events are preferred for acting on the browser DOM and working with browser events.

2.3 Minimize Browser to Server Round Trips and JavaScript Processing

2.3.1 Anti-Pattern #3 – RequireJS

This anti-pattern may be slightly controversial for some developers. We have found that there are performance issues with extensive use of this framework in AngularJS applications. We will describe the performance implications of using this framework in large AngularJS applications.

In the case of RequireJS there are two pain points we have experienced. The first pain point is that large numbers of JavaScript files can cause performance limitations in transiting from one route to another. The reason for this is that as a view is loaded and the required modules need to be loaded for dependency injection in the controllers, etc. It forces JavaScript loading, parsing and compiling just as you need the best performance while loading the new view. It also causes conflicts with template caching for the same reasons. The other issue we have noticed is a little bit harder to see but causes more performance issues. RequiresJS uses AJAX to load and inject JavaScript files. Whereas JavaScript is normally loaded via the script html tag. If the script tag is used the look ahead pre-parser (or browser pre-loader) is enabled and for most modern browsers increases JavaScript heavy applications by ~20 percent. There is a tool for minimizing the number of files loaded by RequireJS, but this is only one of the issues with using RequireJS in a large application the use of r.js does not eliminate the pre-parser short circuit described above.

So what do we prefer to use in these situations. We are big fans of gulp.js and livereload to allow speed of development for local development. For production deployments, again gulp.js and creating an optimized application directory structure that can be deployed to a web server allows for a great deal of flexibility, scalability and performance at the browser level.

2.4 Retrieving Data Sets in an Efficient Manner

2.4.1 Anti-Pattern #4 – Did Not Implement Sorting and Filtering in Application Services

This anti-pattern is usually found when an application has implemented a naive service to retrieve all customers for a company. Then the application will implement a view to render this service data as a series of ng-repeat directives. This is very fast for 1-1000 elements, but it will quickly slow or even stop working as it approaches ~2000 elements. This is due to the need to call $digest() for each element added. To overcome this limitation there are several ways to implement the service so that it does not return the entire data set. The service would require an interface for applying sort options, starting element index and length of the results set requested. It is beyond the scope of this white paper to describe all of the implementation options. At XTIVIA we have implemented performant services that serve data from relational databases, search engines (SOLR and ElasticSearch), NoSQL databases (MongoDB, CouchDB, etc.) and other enterprise data stores/systems.

2.5 Render Data Sets in an Efficient Manner

2.4.1 Anti-Pattern #5 – Did Not Listen for $destroy

When creating a directive that removes elements from a browser DOM or listens to DOM events it is necessary to listen to the $destroy event. If the directive does not listen to $destroy it will cause memory leaks every time the scope is destroyed. The result is similar to the shadow tree memory leak described above. The browser DOM memory usage will increase as the application is used. A couple of example approaches for handling the $destroy event are described below.

In a directive a timer may be used to handle timed actions.

```
var promise = $interval(function () {}, 1000);

scope.$on('$destroy', function () {
  $interval.cancel(promise);
});
```

In a directive a DOM event can be listened to. The following shows destruction of this event handler.

```
var windowClick = function () {
   ...
};

angular.element(window).on('click', windowClick);

scope.$on('$destroy', function () {
  angular.element(window).off('click', windowClick);
});
```

3. Conclusion

Thank you for the opportunity to share some performance oriented anti-patterns that are commonly found in new AngularJS implementations.

ANGULARJS
by Google

Got questions?

We have the answers!

XTIVIA has been helping customers build fast, custom applications for decades.

Our deep knowledge and know-how have been honed with each project we build for our customers. We know how to deliver successful projects and look forward to helping you make your business better.

Contact us today:
info@xtivia.com
Call 1-888-685-3101 ext. 2