AngularJS Design and Best Practices

by Vlad Costel Ungureanu
for “Learn Stuff” and “CGM Romania”

Thinking about AngularJS

- ✓ Understand what you need to do before you start
- ✓ Use wireframes or any form of representation before you start
- ✓ Think about how the data flow of the view
- ✓ Remember it's still java Script and it's not magical
- ✓ Refresh your knowledge before you implement; it only takes a couple of minutes
- ✓ **If you aim for reusability you will have a very bad time**
- ✓ Aim for separation of concerns, reusability will come over time when you are tempted to copy + paste

Naming Conventions

Element	Naming style	Example	usage
Modules	lowerCamelCase	angularApp	
Controllers	functionality + 'Ctrl'	adminCtrl	
Directives	lowerCamelCase	userInfo	
Filters	lowerCamelCase	userFilter	
Services	UpperCamelCase	User	constructor
Factories	lowerCamelCase	dataFactory	others

Structure

✓ Following the standards MVC structure:

```
templates/
   login.html
   feed.html
app/
   app.js
   controllers/
      LoginController.js
      FeedController.js
   directives/
      FeedEntryDirective.js
   services/
      LoginService.js
      FeedService.js
   filters/
      CapatalizeFilter.js
```

Structure

✓ Following the feature base file structure:

```
app/
   app.js
   Feed/
      feed.html
      FeedController.js
      FeedEntryDirective.js
      FeedService.js
   Login/
      login.html
      LoginController.js
      LoginService.js
   Shared/
      CapatalizeFilter.js
```

LEARN stuff IO

Structure

✓ Assets: refer to all files you need that are not AngualrJS code and are part of any project

```
assets/
  img/     // Images and icons for your app
  css/     // All styles and style related files (SCSS or LESS files)
  js/      // JavaScript files written for your app that are not for angular
  libs/
```

Communication between Elements

- ✓ By using $rootScope
- ✓ By using local storage
- ✓ By using cookies
- ✓ By serializing objects in paths or headers
- ✓ By using broadcast, emit and on
- ✓ By sharing an injectable, singleton service
- ✓ By using component composition and required scope

Fat Controller Approach

- ✓ Controller contains all the model required by the view
- ✓ It uses dependency injection to gain access to services and factories
- ✓ It uses the services and factories to initialize values, control data flows, communicate with backend
- ✓ It uses two way value binding for variables and methods in order to provide data and behaviour to directives, components or hierarchies of view elements
- ✓ Logic might get entangled in the controller
- ✓ Similar to several practices in other programming or scripting languages

Recommended Approach

- ✓ Controller contains the model required by the view or partial view, but some values are found in directive/component scopes as they need to be isolated or contained
- ✓ It uses dependency injection to gain access to services and factories which contain the actual logic of the application
- ✓ It only wrappers some of this functionality in functions exposed through the scope
- ✓ It relies on isolated scopes of directives/components and their child scopes for specific data and view related functionality
- ✓ It relies on directive or components for view manipulation
- ✓ Directives or components form hierarchies with their own scopes and child scopes

Composition Approach

- ✓ Controllers are slim and only contain mandatory scope data and functions
- ✓ Components are fine grained, and are organized in logical hierarchies that compose features
- ✓ Scopes are inherited when possible for communication between components
- ✓ If components are reusable then communication is event based
- ✓ It is based on the composite pattern
- ✓ It is harder to implement and requires vision and patience
- ✓ It ensures loose coupling and changeable components

General practices

- ✓ When a controller uses constructor injection it will start execution only when all elements are injected. Using $inject will start controller and then perform the injection
- ✓ Don’t pipe filters as they will be re-run at each $digest cycle
- ✓ Don’t pollute the $rootScope
- ✓ Avoid unnecessary data bindings and implicit watchers
- ✓ Aim for reusability only if it makes sense, if not use YAGNI
- ✓ Test your code as you go along, since TDD is difficult to implement (the view elements might change several times before they are usable and the tests you write keep changing)

Preferred usage

- ✓ $timeout instead of setTimeout
- ✓ $interval instead of setInterval
- ✓ $window instead of window
- ✓ $document instead of document
- ✓ $http instead of $.ajax
- ✓ $location instead of window.location or $window.location
- ✓ $cookies instead of document.cookie

Controller practices

- ✓ Aim for thin controllers
- ✓ Avoid manipulating DOM in controllers
- ✓ Avoid including business logic in controllers
- ✓ Use ‘controller as’ syntax and capture ‘this’ in your controllers

```
var vm = this;
//a clearer visual connection on how is defined on the view
vm.title = 'Some title';
vm.description = 'Some description';
```

- ✓ Since ‘this’ is depends on the execution context we should avoid using ‘this’ in controllers
- ✓ ngBind is preferred over double brackets

LEARN stuff IO

Directive practices

- ✓ DOM manipulation should only be done in directives
- ✓ Reusable components should always have an isolated scope
- ✓ Directives need to be very well refined
- ✓ Respect single responsibility principle
- ✓ If you need to append thins (popups, tooltips etc.) append them to the body

Template Caching

```
angular
   .module('someModule', [])
   .run(['$templateCache', function injectTemplates($templateCache) {

      $templateCache.put('someDirectiveTemplate', [
         '<div class="my-awesome-header">',
         ' </h2> My template <h2>',
         '</div>'
       ]);


// in directive:
 if (angular.isDefined($templateCache.get('someDirectiveTemplate'))) {
         directive.template = $templateCache.get('someDirectiveTemplate').toString();
       } else {
         directive.templateUrl = 'someDirectiveTemplate.html';
       }
```

Request Caching

```
// create a new cache with a capacity of 10
var myCache = $cacheFactory(myCache', { capacity: 10 });

// use the new cache for this request
$http.get('/someService + itemId, { cache: myCache })
  .success(function (data) {
    // process response
  })
  .error(function (data, status, headers, config) {
    // handle error
  });
```

Some design patterns in AngularJS

- ✓ Inversion of control – Dependency Injection
- ✓ MVC/MVVM/MVW – Controller, Views and Data Binding
- ✓ Singleton: Services and Factories as DI usages
- ✓ Factory: Factories
- ✓ Observer: watchers
- ✓ Façade: Services and Factories as wrappers over complex functionality

THANK YOU!

Vlad Costel Ungureanu
ungureanu_vlad_costel@yahoo.com