An AngularJS Style Guide for Closure Users at Google

This is the external version of a document that was primarily written for Google engineers. It describes a recommended style for AngularJS apps that use Closure, as used internally at Google. Members of the broader AngularJS community should feel free to apply (or not apply) these recommendations, as relevant to their own use cases.

This document describes style for AngularJS apps in google3. This guide supplements and extends the Google JavaScript Style Guide.

Style Note: Examples on the AngularJS external webpage, and many external apps, are written in a style that freely uses closures, favors functional inheritance, and does not often use JavaScript types. Google follows a more rigorous Javascript style to support JSCompiler optimizations and large code bases - see the javascript-style mailing list. This is not an Angular-specific issue, and is not discussed further in this style guide. (But if you want further reading: Martin Fowler on closures, much longer description, appendix A of the closure book has a good description of inheritance patterns and why it prefers pseudoclassical, Javascript, the Good Parts as a counter.)

1 Angular Language Rules

- Manage dependencies with Closure's goog.require and goog.provide
- Modules
- Modules should reference other modules using the "name" property
- Use the provided Angular externs file
- JSCompiler Flags
- Controllers and Scopes
- Directives
- Services

2 Angular Style Rules

- Reserve $ for Angular properties and services
- Custom elements.

3 Angular Tips, Tricks, and Best Practices

- Testing
- Consider using the Best Practices for App Structure
- Be aware of how scope inheritance works
- Use @ngInject for easy dependency injection compilation

4 Best practices links and docs

1 Angular Language Rules

Manage dependencies with Closure's goog.require and goog.provide

Choose a namespace for your project, and use goog.provide and goog.require.

```
goog.provide('hello.about.AboutCtrl');
goog.provide('hello.versions.Versions');
```

Why? Google BUILD rules integrate nicely with closure provide/require.

Modules

Your main application module should be in your root client directory. A module should never be altered other than the one where it is defined.

Modules may either be defined in the same file as their components (this works well for a module that contains exactly one service) or in a separate file for wiring pieces together.

Why? A module should be consistent for anyone that wants to include it as a reusable component. If a module can mean different things depending on which files are included, it is not consistent.

Modules should reference other modules using the Angular Module's "name" property

For example:

```
// file submodulea.js:
  goog.provide('my.submoduleA');
  my.submoduleA = angular.module('my.submoduleA', []);
  // ...
// file app.js
  goog.require('my.submoduleA');
  Yes: my.application.module = angular.module('hello', [my.submoduleA.name]);

      No: my.application.module = angular.module('hello', ['my.submoduleA']);
```

Why? Using a property of my.submoduleA prevents Closure presubmit failures complaining that the file is required but never used. Using the .name property avoids duplicating strings.

Use a common externs file

This maximally allows the JS compiler to enforce type safety in the presence of externally provided types from Angular, and means you don't have to worry about Angular vars being obfuscated in a confusing way.

Note to readers outside Google: the current externs file is located in an internal-to-Google directory, but an example can be found on github here.

JSCompiler Flags

Reminder: According to the JS style guide, customer facing code must be compiled.

Recommended: Use the JSCompiler (the closure compiler that works with js_binary by default) and ANGULAR_COMPILER_FLAGS_FULL from //javascript/angular/build_defs/build_defs for your base flags.

Note - if you are using @export for methods, you will need to add the compiler flag

```
"--generate_exports",
```

If you are using @export for properties, you will need to add the flags:

```
"--generate_exports",
"--remove_unused_prototype_props_in_externs=false",
"--export_local_property_definitions",
```

Controllers and Scopes

Controllers are classes. Methods should be defined on MyCtrl.prototype.

Google Angular applications should use the **'controller as'** style to export the controller onto the scope. This is fully implemented in Angular 1.2 and can be mimicked in pre-Angular 1.2 builds.

Pre Angular 1.2, this looks like:

```
/**
 * Home controller.
 *
 * @param {!angular.Scope} $scope
 * @constructor
 * @ngInject
 * @export
 */
hello.mainpage.HomeCtrl = function($scope) {
  /** @export */
  $scope.homeCtrl = this; // This is a bridge until Angular 1.2 controller-as
  /**
   * @type {string}
   * @export
   */
  this.myColor = 'blue';
};
/**
 * @param {number} a
 * @param {number} b
 * @export
 */
hello.mainpage.HomeCtrl.prototype.add = function(a, b) {
  return a + b;
};
```

And the template:

```
<div ng-controller="hello.mainpage.HomeCtrl"/>
  <span ng-class="homeCtrl.myColor">I'm in a color!</span>
  <span>{{homeCtrl.add(5, 6)}}</span>
</div>
```

After Angular 1.2, this looks like:

```
/**
 * Home controller.
 *
 * @constructor
 * @ngInject
 * @export
 */
hello.mainpage.HomeCtrl = function() {
  /**
```

```
    * @type {string}
    * @export
    */
   this.myColor = 'blue';
};
/**
 * @param {number} a
 * @param {number} b
 * @export
 */
hello.mainpage.HomeCtrl.prototype.add = function(a, b) {
  return a + b;
};
```

If you are compiling with property renaming, expose properties and methods using the @export annotation. Remember to @export the constructor as well.

And in the template:

```
<div ng-controller="hello.mainpage.HomeCtrl as homeCtrl"/>
  <span ng-class="homeCtrl.myColor">I'm in a color!</span>
  <span>{{homeCtrl.add(5, 6)}}</span>
</div>
```

Why? Putting methods and properties directly onto the controller, instead of building up a scope object, fits better with the Google Closure class style. Additionally, using 'controller as' makes it obvious which controller you are accessing when multiple controllers apply to an element. Since there is always a '.' in the bindings, you don't have to worry about prototypal inheritance masking primitives.

Directives

All DOM manipulation should be done inside directives. Directives should be kept small and use composition. Files defining directives should goog.provide a static function which returns the directive definition object.

```
goog.provide('hello.pane.paneDirective');
/**
 * Description and usage
 * @return {angular.Directive} Directive definition object.
 */
hello.pane.paneDirective = function() {
  // ...
};
```

Exception: DOM manipulation may occur in services for DOM elements disconnected from the rest of the view, e.g. dialogs or keyboard shortcuts.

Services

Services registered on the module with module.service are classes. Use module.service instead of module.provider or module.factory unless you need to do initialization beyond just creating a new instance of the class.

```
/**
 * @param {!angular.$http} $http The Angular http service.
 * @constructor
 */
hello.request.Request = function($http) {
  /** @type {!angular.$http} */
```

```
  this.http_ = $http;
};
hello.request.Request.prototype.get = function() {/*...*/};
```

In the module:

```
module.service('request', hello.request.Request);
```

2 Angular Style Rules

Reserve $ for Angular properties and services

Do not use $ to prepend your own object properties and service identifiers. Consider this style of naming reserved by AngularJS and jQuery.

Yes:

```
$scope.myModel = { value: 'foo' }
myModule.service('myService', function() { /*...*/ });
var MyCtrl = function($http) {this.http_ = $http;};
```

No:

```
$scope.$myModel = { value: 'foo' } // BAD
$scope.myModel = { $value: 'foo' } // BAD
myModule.service('$myService', function() { ... }); // BAD
var MyCtrl = function($http) {this.$http_ = $http;}; // BAD
```

Why? It's useful to distinguish between Angular / jQuery builtins and things you add yourself. In addition, $ is not an acceptable character for variables names in the JS style guide.

Custom elements

For custom elements (e.g. <ng-include src="template"></ng-include>), IE8 requires special support (html5shiv-like hacks) to enable css styling. Be aware of this restriction in apps targeting old versions of IE.

3 Angular Tips, Tricks, and Best Practices

These are not strict style guide rules, but are placed here as reference for folks getting started with Angular at Google.

Testing

Angular is designed for test-driven development.

The recommended unit testing setup is Jasmine + Karma (though you could use closure tests or js_test)

Angular provides easy adapters to load modules and use the injector in Jasmine tests.

- module
- inject

Consider using the Best Practices for App Structure

This directory structure doc describes how to structure your application with controllers in nested subdirectories and all components (e.g. services and directives) in a 'components' dir.

Be aware of how scope inheritance works

See The Nuances of Scope Prototypal Inheritance

Use @ngInject for easy dependency injection compilation

This removes the need to add myCtrl['$inject'] = ... to prevent minification from messing up Angular's dependency injection.

Usage:

```
/**
 * My controller.
 * @param {!angular.$http} $http
 * @param {!my.app.myService} myService
 * @constructor
 * @export
 * @ngInject
 */
my.app.MyCtrl = function($http, myService) {
  //...
};
```

4 Best practices links and docs

- Best Practices from Angular on GitHub
- Meetup video (not Google specific)