AngularJS Best Practices & Style Guide

Table of Contents

General

- ^When possible, use angular.element(), etc. instead of jQuery lookups and DOM manipulation.
- ^Don't wrap element inside of $(). All AngularJS elements are already jqobjects.
- ^Do not pollute your $scope. Only add functions and variables that are being used in the templates. In other words, don't put all of your functions & variables on the scope. If you have methods that are private to your controller, never to be used in the view/template, there is no reason to put them on the scope.
- ^Do not use $ prefix for the names of variables, properties and methods. This prefix is reserved for AngularJS usage.
- ^When you need to set the src of an image dynamically use ng-src instead of src with {{}}.

- ^When you need to set the href of an anchor tag dynamically use ng-href instead of href with {{}}
- ^Avoid using $rootScope. It's ok to use $rootScope for emitting an event, but storing data on the $rootScope should be avoided. Use a service for your data instead.
- ^When possible, avoid using "magic strings" by using Angular's constant service. More details on this blog post.

Files

Note: Some of these opinions about file structure, naming, etc are due to the fact that I use and recommend Browserify as part the build process.

- **^One module per file**: Each file should have only one module definition. Exceptions are your app definition file (usually app.js), and any modules that need a config module.

- **^Each file should get its own namespace**: The namespace should follow its directory structure. The module namespace is //path/to/file. Note that you leave the src root out of the filepath (the src root is typically /app). Ex: If your project is called LocAdmin, your file is a controller for the LocationsListing directive and is named LocationsListingCtrl it will likely have the following path on your filesystem: /LocAdmin/app/components/locationsListing.

```
.module('locAdmin.components.locationsListing.locationsListingCtrl', [
]);
```

- **^Each filename should match the controller/service/etc name**: A file with a .controller('mainCtrl') definition should be named mainCtrl.js

- **^Function name and file name should match**: Given the the function definition below, you would name your file locationsListingCtrl.js. Note that filenames should start with a lowercase letter.

```
function LocationsListingCtrl($scope) {
}
```

- **^Each file should be CommonJS compatible**: This means using module.exports and require(). Further explanations here(change the dropdown from Coffeescript to JavaScript to change the source code on that page). Note that at some point in the future this recommendation may

switch to ES6 Modules, using the ES6 Module transpiler from Square (or traceur).

```
// recommended
function LocationsListingCtrl() {
}

module.exports = angular
  .module('locAdmin.components.locationsListing.locationsListingCtrl', [
    require('./locationsListingService').name
  ])
  .controller('LocationsListingCtrl', LocationsListingCtrl);
```

- **^Directory structure**

```
/MyProject
--/src
----index.html (the index.html for the SPA)
----app.js (the app definition for the Angular app)
----/assets (images)
------logo.png
----/less (LESS files for things other than pages/directives)
------main.less (the main LESS file for the app. Should import LESS files from directives & pages
------variables.less
----/app (angular app files)
------/components (directives go here)
--------/myDirective (directory for myDirective directive
----------myDirective.js (directive file)
----------myDirective.tpl.html (directive template/partial)
----------myDirectiveCtrl.js (the directive's controller)
----------myDirectiveService.js (if directive requires a service, used ONLY by this directive)
----------myDirective.less (LESS file for this directive, if needed)
------/pages (top level pages/views go here. Subdirectories follow the same logic as the directives directory)
--------/main (the main page/view)
----------mainCtrl.js (the controller for the main view)
----------main.tpl.html (the template/partial for the main view)
----------main.less (LESS file for the main view)
------/services (shared services, used by multiple controllers, go here)
--------mySharedService.js
------/utils (other helper files used throughout the app
--------stringUtils.js
--------viewUtils.js
```

Modules

- **Definitions**: Declare modules without a variable using the setter and getter syntax

```
// avoid
var app = angular.module('app', []);
app.controller();
app.factory();

// recommended
angular
  .module('app', [])
  .controller()
  .factory();
```

- Note: Using `angular.module('app', []);` sets a module, whereas `angular.module('app');` gets the module. Only set once and get for all other instances.

- **Methods**: Pass functions into module methods rather than assign as a callback

```
// avoid
angular
  .module('app', [])
  .controller('MainCtrl', function MainCtrl () {

  })
  .service('SomeService', function SomeService () {

  });

// recommended
function MainCtrl () {

}
function SomeService () {

}
angular
  .module('app', [])
  .controller('MainCtrl', MainCtrl)
  .service('SomeService', SomeService);
```

- This aids with readability and reduces the volume of code "wrapped" inside the Angular framework

Back to top

Controllers

- **controllerAs syntax**: Controllers are classes, so use the `controllerAs` syntax at all times

```
<!-- avoid -->
<div ng-controller="MainCtrl">
  {{ someObject }}
</div>

<!-- recommended -->
<div ng-controller="MainCtrl as main">
  {{ main.someObject }}
</div>
```

- In the DOM we get a variable per controller, which aids nested controller methods, avoiding any `$parent` calls
- The `controllerAs` syntax uses `this` inside controllers, which gets bound to `$scope`

```
// avoid
function MainCtrl ($scope) {
  $scope.someObject = {};
  $scope.doSomething = function () {

  };
}

// recommended use this or self
function MainCtrl () {
  this.someObject = {};
  this.doSomething = function () {

  };
}

function MainCtrlTwo() {
  var self = this;
  self.someObject = {};
  self.doSomething = function() {
};
```

- Only use `$scope` in `controllerAs` when necessary; for example, publishing and subscribing events using `$emit`, `$broadcast`, `$on` or `$watch`. Try to limit the use of these, however, and treat `$scope` as a special use case
- **^controllerAs 'self'**: Capture the `this` context of the Controller using `self' (this is further explanation of MainCtrlTwo() in the example above)

```
// avoid
function MainCtrl () {
  this.doSomething = function () {
```

```
    };
}

// recommended
function MainCtrl (SomeService) {
  var self = this;
  self.doSomething = SomeService.doSomething;
}
```

Why? : Function context changes the `this` value, use it to avoid `.bind()` calls and scoping issues

- **Presentational logic only (MVVM)**: Presentational logic only inside a controller, avoid Business logic (delegate to Services)

```
// avoid
function MainCtrl () {

  var self = this;

  $http
    .get('/users')
    .success(function (response) {
      self.users = response;
    });

  vm.removeUser = function (user, index) {
    $http
      .delete('/user/' + user.id)
      .then(function (response) {
        self.users.splice(index, 1);
      });
  };

}

// recommended
function MainCtrl (UserService) {

  var self = this;

  UserService
    .getUsers()
    .then(function (response) {
      self.users = response;
    });

  self.removeUser = function (user, index) {
    UserService
      .removeUser(user)
```

```
        .then(function (response) {
          self.users.splice(index, 1);
        });
    };

}
```

Why? : Controllers should fetch Model data from Services, avoiding any Business logic. Controllers should act as a ViewModel and control the data flowing between the Model and the View presentational layer. Business logic in Controllers makes testing Services impossible.

Back to top

Services and Factory

- All Angular Services are singletons, using `.service()` or `.factory()` differs the way Objects are created.
  Services: act as a `constructor` function and are instantiated with the `new` keyword. Use `this` for public methods and variables (or var self=this, and use self as noted in the Controller As example above)

```
function SomeService () {
  this.someMethod = function () {

  };
}
angular
  .module('app')
  .service('SomeService', SomeService);
```

  Factory: Business logic or provider modules, return an Object or closure

- Always return a host Object instead of the revealing Module pattern due to the way Object references are bound and updated

```
function AnotherService () {
  var AnotherService = {};
  AnotherService.someValue = '';
  AnotherService.someMethod = function () {

  };
  return AnotherService;
}
angular
  .module('app')
  .factory('AnotherService', AnotherService);
```

Why? : Primitive values cannot update alone using the revealing module pattern

Back to top

Directives

- **Declaration restrictions**: Only use `custom element` and `custom attribute` methods for declaring your Directives (`{ restrict: 'EA' }`) depending on the Directive's role

```
<!-- avoid -->

<!-- directive: my-directive -->
<div class="my-directive"></div>

<!-- recommended -->

<my-directive></my-directive>
<div my-directive></div>
```

- Comment and class name declarations are confusing and should be avoided. Comments do not play nicely with older versions of IE. Using an attribute is the safest method for browser coverage.

- **^Templating**: Use external templates instead of inline string templates for larger html blocks. These templates should be stored in Angular's template cache. Inline String templates should only be used in rare circumstances (when the template is very short)

- **DOM manipulation**: Takes place only inside Directives (link function), never a controller/service

```
// avoid
function UploadCtrl () {
  $('.dragzone').on('dragend', function () {
    // handle drop functionality
  });
}
angular
  .module('app')
  .controller('UploadCtrl', UploadCtrl);

// recommended
function dragUpload () {
  return {
    restrict: 'EA',
```

```
      link: function ($scope, $element, $attrs) {
        $element.on('dragend', function () {
          // handle drop functionality
        });
      }
    };
  }
  angular
    .module('app')
    .directive('dragUpload', dragUpload);
```

- **Naming conventions**: Never `ng-*` prefix custom directives, they might conflict future native directives

```
  // avoid
  // <div ng-upload></div>
  function ngUpload () {
    return {};
  }
  angular
    .module('app')
    .directive('ngUpload', ngUpload);

  // recommended
  // <div drag-upload></div>
  function dragUpload () {
    return {};
  }
  angular
    .module('app')
    .directive('dragUpload', dragUpload);
```

- Directives and Filters are the *only* providers that have the first letter as lowercase; this is due to strict naming conventions in Directives. Angular hyphenates `camelCase`, so `dragUpload` will become `<div drag-upload></div>` when used on an element.
- **controllerAs**: Use the `controllerAs` syntax inside Directives as well

```
  // avoid
  function dragUpload () {
    return {
      controller: function ($scope) {

      }
    };
  }
  angular
    .module('app')
    .directive('dragUpload', dragUpload);

  // recommended
  function dragUpload () {
```

```
    return {
      controllerAs: 'dragUpload',
      controller: function () {

      }
    };
  }
  angular
    .module('app')
    .directive('dragUpload', dragUpload);
```

- **^Each directive should live in its own directory**: This directory will include the directive js file, the template, and any services that are specific to the directive. The parent director for all directives is typically /components.

- Example:

```
/components/listing/listing.js (the directive)
/components/listing/listing.tpl.html (the template/partial for the directive)
/components/listing/listingService.js (if the directive needs a service, used ONLY be this directive)
/components/listing/listingCtrl.js (the controller for this directive)
```

- **^Use isolate scope in directives whenever possible**: This isn't an absolute, hence the *whenever possible* phrase. Allowing directives to rely on inherited/shared scope can make the code brittle. Be explicit about what data your directive needs by passing it into the scope. Note that workarounds can be found if you have an element with multiple directives (since Angular only allows 1 directive per element to have an isolate scope).

Back to top

Filters

- **Global filters**: Create global filters using `angular.filter()` only. Never use local filters inside Controllers/Services

```
// avoid
function SomeCtrl () {
  this.startsWithLetterA = function (items) {
    return items.filter(function (item) {
      return /^a/i.test(item.name);
    });
  };
}
angular
  .module('app')
```

```
    .controller('SomeCtrl', SomeCtrl);

// recommended
function startsWithLetterA () {
  return function (items) {
    return items.filter(function (item) {
      return /^a/i.test(item.name);
    });
  };
}
angular
  .module('app')
  .filter('startsWithLetterA', startsWithLetterA);
```

- This enhances testing and reusability

Back to top

Routing resolves

- **^Promises**: When possible, resolve Controller dependencies in the `$stateProvider`(prefer `ui-router` instead of `ng-route`), not the Controller itself

```
// avoid
function MainCtrl (SomeService) {
  var _this = this;
  // unresolved
  _this.something;
  // resolved asynchronously
  SomeService.doSomething().then(function (response) {
    _this.something = response;
  });
}
angular
  .module('app')
  .controller('MainCtrl', MainCtrl);

// recommended
function config ($stateProvider) {
  $stateProvider
  .state('main', {
    url: '/main',
    controller: 'MainCtrl as mainCtrl',
    templateUrl: 'pages/main/main.tpl.html'
    resolve: {
      // resolve here
      locations: function () {
        return SomeService.getLocations();
```

```
    }
  });
}
angular
  .module('app')
  .config(config);

function MainCtrl (SomeService, locations) {
  var self = this;
  self.locations = locations;
}
```

Back to top

Publish and subscribe events

- **$rootScope**: Use only `$emit` as an application-wide event bus and remember to unbind listeners

```
// all $rootScope.$on listeners
$rootScope.$emit('customEvent', data);
```

- Hint: `$rootScope.$on` listeners are different from `$scope.$on` listeners and will always persist, so they need destroying when the relevant `$scope` fires the `$destroy` event

```
// call the closure
var unbind = $rootScope.$on('customEvent'[, callback]);
$scope.$on('$destroy', unbind);
```

- For multiple `$rootScope` listeners, use an Object literal and loop each one on the `$destroy` event to unbind all automatically

```
var rootListeners = {
  'customEvent1': $rootScope.$on('customEvent1'[, callback]),
  'customEvent2': $rootScope.$on('customEvent2'[, callback]),
  'customEvent3': $rootScope.$on('customEvent3'[, callback])
};
for (var unbind in rootListeners) {
  $scope.$on('$destroy', rootListeners[unbind]);
}
```

Back to top

Performance

- **One-time binding syntax**: In newer versions of Angular (v1.3.0-beta.10+), use the one-time binding syntax `{{ ::value }}`where it makes sense

```
// avoid
<h1>{{ vm.title }}</h1>
```

```
// recommended
<h1>{{ ::vm.title }}</h1>
```

Why? : Binding once removes the `$$watchers` count after the `undefined` variable becomes resolved, thus reducing performance in each dirty-check

- **Consider $scope.$digest**: Use `$scope.$digest` over `$scope.$apply` where it makes sense. Only child scopes will update

```
$scope.$digest();
```

Why? : `$scope.$apply` will call `$rootScope.$digest`, which causes the entire application `$$watchers` to dirty-check again. Using `$scope.$digest` will dirty check current and child scopes from the initiated `$scope`

Back to top

Angular wrapper references

- **$document and $window**: Use `$document` and `$window` at all times to aid testing and Angular references

```
// avoid
function dragUpload () {
  return {
    link: function ($scope, $element, $attrs) {
      document.addEventListener('click', function () {

      });
    }
  };
}

// recommended
function dragUpload () {
  return {
    link: function ($scope, $element, $attrs, $document) {
      $document.addEventListener('click', function () {

      });
    }
  };
}
```

- **$timeout and $interval**: Use `$timeout` and `$interval` over their native counterparts to keep Angular's two-way data binding up to date

```
// avoid
function dragUpload () {
  return {
```

```
      link: function ($scope, $element, $attrs) {
        setTimeout(function () {
          //
        }, 1000);
      }
    };
  }

  // recommended
  function dragUpload ($timeout) {
    return {
      link: function ($scope, $element, $attrs) {
        $timeout(function () {
          //
        }, 1000);
      }
    };
  }
```

Back to top

Comment standards

- **jsDoc**: Use jsDoc syntax to document function names, description, params and returns

```
  /**
   * @name SomeService
   * @desc Main application Controller
   */
  function SomeService (SomeService) {

    /**
     * @name doSomething
     * @desc Does something awesome
     * @param {Number} x First number to do something with
     * @param {Number} y Second number to do something with
     * @returns {Number}
     */
    this.doSomething = function (x, y) {
      return x * y;
    };

  }
  angular
    .module('app')
    .service('SomeService', SomeService);
```

Back to top

Minification and annotation

- **ng-annotate**: Use ng-annotate for Gulp as `ng-min` is deprecated, and comment functions that need automated dependency injection using `/** @ngInject */`

```
/**
 * @ngInject
 */
function MainCtrl (SomeService) {
  this.doSomething = SomeService.doSomething;
}
angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

- Which produces the following output with the `$inject` annotation

```
/**
 * @ngInject
 */
function MainCtrl (SomeService) {
  this.doSomething = SomeService.doSomething;
}
MainCtrl.$inject = ['SomeService'];
angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

Back to top

Angular docs

For anything else, including API reference, check the Angular documentation.