Welcome to Cybage

Document Name	AngularJS Best Practices
Version No.	V.1
Release Date	4th Nov 2014

Document Template History

Version No.	Authored / Modified by	Reviewed by, Date	Approved by, Date	Date Remark / Change History
V1.0	Yogesh Gaikwad			Compiled as on 4th Nov 2014
V1.0	Yogesh Gaikwad			Code snippet update as on 12th Dec 2014

This document contains guidelines for Angular framework based web applications. This document's primary motive is to implement code consistency and best practices to all AngularJS projects. By maintaining consistency in coding styles and conventions, we can ease the burden of code maintenance, and mitigate risk of breakage in the future.

By adhering to best practices, we ensure optimized page loading, performance and maintainable code.

Contents

1. OBJECTIVE

This is a set of coding conventions and rules for use in AngularJS programming.

These Standards are extracted from the multiple developers' community and past experience of Cybage on Angular projects.

Best Practices can help in reducing the instability of AngularJS applications. It is all about, "How to make it more efficient" rather than "How to write Angular code"?

2. FOLDER STRUCTURE

In case folder structure is not defined and base structure is required as a start-up, here is the skeleton for typical Angular web applications. It also promotes best practices at module level.

- YEOMEN (https://github.com/yeoman/generator-angular)
- Angular-seed (https://github.com/angular/angular-seed)

It is suggested to have modular approach for the application. Depend on project requirement structure can be altered.

3. SCRIPT LOADING

99% of code should be stored in and delivered as external .js files.

All <script> tags at the bottom

The primary goal is to make the page load as quickly as possible for the user. If the script file is added in the beginning of the page, the browser can't continue on until the entire file has been loaded. Thus, the user will have to wait longer before noticing any progress.

If JS files are added for some functionality like, on button click perform some action — go ahead and place those files at the bottom, just before the closing body tag. This will allow browser to render HTML objects while .js files are loading.

```
<p>Lorem Ip sum </p>
<script type="text/javascript" src="js/main.js"></script>
<script type="text/javascript" src="js/carousal.js"></script>
</body>
</html>
```

ng-app bootstrap

Place ng-app to the root of your application, typically on the <html> tag if you want angular to auto-bootstrap your application.

To run multiple applications in an HTML document you must manually bootstrap them using angular.bootstrap instead.

RequireJS to load files (optional)
Create and re-use modules without polluting the global namespace. The more polluted your global namespace is, the bigger the chance of a function/variable collision. That means you define a function called "foo" and another developer defines the function "foo" = one of the functions gets overwritten.

Structure your code into separate folders and files and requirejs will load them asynchronously when needed, so everything just works.

Build for production. RequireJS comes with its own build tool called R.JS that will concat and uglify your javascript modules into a single (or multiple) packages. This will improve your page speed as the user will have to make less script calls and load less content (as your JS is uglified).

4. FLASH OF UNSTYLED CONTENTS

To avoid showing {{}} when data is not loaded in the initial page, use below directives. This will hide dynamic elements till the time data is not ready for load.

- Hide it with ng-cloak directive
  - style [ng-cloak]{display: none}
  - <body ng-cloak>
- Ng-bind
  - <span ng-bind="projects.length || '?'">?</span>

5. MINIFICATION AND COMPILATION

No need to minify angular.js as it is already minified. In case code needs to be in single file only. Add user file after angular and compile using tools.

Compiler Settings: Disable property renaming or directive / variables / model data will not work in Angular

Minification: No need to minify angular.js as it is already minified. In case code needs to be in single file only. Add user file after angular and compile using tools.

6. DO I NEED DIRECTIVE?

If there is a need to write $(element) in controller then yes directive is required.

One way of writing directive is using prefix as below but prefix doesn't work in IE <**my**-component><**my**:component> so it better to avoid it and adopt alternatives.

- Add namespace in html <html xmlns:ng "http://www.angular.org">
- Instead of tag based elements use it as attribute <div **my**-component>

Ways to write directives

- <div x-my-component>
- <div data-my-component>
- <!-- directive:my-component -->
- <div class="my-component">
- <div my-component>

7. BUSINESS AND PRESENTATION LOGIC

Business logic belongs to model

1. Data processing should be kept in models. Server interaction and error handling from the model.
2. This way it can be shared among different controller and services
3. Easy to write test for them

Controllers

1. Move reusable logic to factories and keep the controller simple and focused on its view. Reusing controllers with several views is brittle and good end to end (e2e) test coverage is required to ensure stability across large applications.
2. Should not reference DOM
3. Should have View behaviour
   a. What happens if user does X?
   b. Where do I get X from?
4. Place bindable members at the top of the controller, alphabetized, and not spread through the controller code.
   Placing bindable members at the top makes it easy to read and helps you instantly identify which members of the controller can be bound and used in the View.

   ```
   /* recommended */
   function Sessions() {
     var vm = this;

     vm.gotoSession = gotoSession;
     vm.refresh = refresh;
   ```

5. Defer logic in a controller by delegating to services and factories.
   Logic may be reused by multiple controllers when placed within a service and exposed via a function.
   Logic in a service can more easily be isolated in a unit test, while the calling logic in the controller can be easily mocked.
   Removes dependencies and hides implementation details from the controller.

```
/* avoid */
function Order($http, $q, config, userInfo) {
  var vm = this;
  vm.checkCredit = checkCredit;
  vm.isCreditOk;
  vm.total = 0;

  function checkCredit() {
    var settings = {};
    // Get the credit service base URL from config
    // Set credit service required headers
    // Prepare URL query string or data object with request data
    // Add user-identifying info so service gets the right credit limit for this user.
    // Use JSONP for this browser if it doesn't support CORS
    return $http.get(settings)
      .then(function(data) {
        // Unpack JSON data in the response object
        // to find maxRemainingAmount
        vm.isCreditOk = vm.total <= maxRemainingAmount
      })
      .catch(function(error) {
        // Interpret error
        // Cope w/ timeout? retry? try alternate service?
        // Re-reject with appropriate error for a user to see
      });
  };
}


/* recommended */
function Order(creditService) {
  var vm = this;
  vm.checkCredit = checkCredit;
  vm.isCreditOk;
  vm.total = 0;

  function checkCredit() {
    return creditService.isOrderTotalOk(vm.total)
      .then(function(isOk) { vm.isCreditOk = isOk; })
      .catch(showServiceError);
  };
}
```

Services

1. Should avoid references to the DOM

2. Are singletons object/class. Created once in an application's lifetime and used through caching. E.g. navigation.

```
// service
angular
   .module('app')
   .service('logger', logger);

function logger() {
  this.logError = function(msg) {
    /* */
  };
}

// factory
angular
   .module('app')
   .factory('logger', logger);

function logger() {
   return {
      logError: function(msg) {
        /* */
      }
   };
}
```

3. Communication between 2 controllers should be in service and not in controller. Controller should be there to hook things between views and not for logical part.
4. Should have global presence
5. Minimal DOM references under services. Headache at the testing point.
6. Services are instantiated with the `new` keyword, use `this` for public methods and variables. Since these are so similar to factories, use a factory instead for consistency.
7. When calling a data service that returns a promise such as $http, return a promise in the calling function too.
   Promises can be chain together and take further action after the data call completes and resolves or rejects the promise.

```
/* recommended */
activate();

function activate() {
   /**
    * Step 1
    * Ask the getAvengers function for the
    * avenger data and wait for the promise
```

```
      */
    return getAvengers().then(function() {
       /**
        * Step 4
        * Perform an action on resolve of final promise
        */
       logger.info('Activated Avengers View');
    });
}

function getAvengers() {
     /**
      * Step 2
      * Ask the data service for the data and wait
      * for the promise
      */
     return dataservice.getAvengers()
        .then(function(data) {
           /**
            * Step 3
            * set the data and resolve the promise
            */
           vm.avengers = data;
           return vm.avengers;
     });
}
```

Directives

- Put all DOM manipulation here
- Avoid jQuery DOM manipulation
  Exception: DOM manipulation may occur in services for DOM elements disconnected from the rest of the view, e.g. dialogs or keyboard shortcuts.
- One directive per file. Name the file for the directive.
  It is easy to mash all the directives in one file, but difficult to then break those out so some are shared across apps, some across modules, some just for one module.

/* avoid */

```
/* directives.js */
angular
   .module('app.widgets')

   /* order directive that is specific to the order module */
   .directive('orderCalendarRange', orderCalendarRange)

   /* sales directive that can be used anywhere across the sales app */
```

```
    .directive('salesCustomerInfo', salesCustomerInfo)


function orderCalendarRange() {
    /* implementation details */
}

function salesCustomerInfo() {
    /* implementation details */
}
```

/* recommended */

```
/* calendarRange.directive.js */
/**
* @desc order directive that is specific to the order module
* @example <div acme-order-calendar-range></div>
*/
angular
    .module('sales.order')
    .directive('acmeOrderCalendarRange', orderCalendarRange);

function orderCalendarRange() {
    /* implementation details */
}
```

/* recommended */

```
/* customerInfo.directive.js */
/**
* @desc spinner directive that can be used anywhere across the sales app
* @example <div acme-sales-customer-info></div>
*/
angular
    .module('sales.widgets')
    .directive('acmeSalesCustomerInfo', salesCustomerInfo);

function salesCustomerInfo() {
    /* implementation details */
}
```

Templates

Angular gives us the option to define named templates in a special script tag. There are three types of templates, Static, Dynamic and Recursive.

Static templates:

- Must be defined inside the scope of the ng-app directive otherwise it and its content is ignored by Angular.
- The type attribute of the script tag must be “text/ng-template”
- The script tag must contain an id attribute with a unique value.

8. SCOPES ARE IMPORTANT

- Do not use [expensive] functions in expressions (in templates) – they are re-evaluated often
- Use $scope.$apply() when applying external data changes (e.g. callback from outside AngularJS)
- Never store references to DOM elements in the scope
- Do not access the DOM outside of a directive
- Scope has reference to model and not to be model. No inheritance requires.
- When doing bidirectional binding (ng-model) make sure binding is not directly applied to the scope properties. Unexpected behaviour in child scopes and it overrides parent scope.

9. EQUIVALANTS OF FAMILIAR FUNCTIONS

Use instead,

- window.setTimeout with $timeout
- $.ajax with $http
- $q promises

All these things play nicely with Angular evaluation loop, and you do not have to worry about calling $digest or $apply manually.

Check out the global APIs in Angular documentation – it has a rich set of utility functions that you need on a daily basis.

10. STRUCTURING 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.

- Usually one module for the entire application
- One module per third party reusable library
- Anything shared with other project? Make it module

- A module should only be created once, then retrieved from that point and after
  - Use angular.module('app', []); to set a module.
  - Use angular.module('app'); to get a module.
- For testing
  - Create test modules which override services (mock ngMock)
  - Test portion of the app (*)
- In case multiple modules are created per app
  - Group by functionality / feature and not by type
  - Group by view since views will be lazy loaded in near future

Module Definition

Declare modules without a variable using the setter syntax.

```
/* avoid */
var app = angular.module('app', [
   'ngAnimate',
   'ngRoute',
   'app.shared',
   'app.dashboard'
]);
```

Instead use the simple setter syntax.

```
/* recommended */
angular
    .module('app', [
      'ngAnimate',
      'ngRoute',
      'app.shared',
      'app.dashboard'
   ]);
```

Getters

When using a module, avoid using a variable and instead use chaining with the getter syntax. This produces more readable code and avoids variable collisions or leaks.

```
/* avoid */
var app = angular.module('app');
app.controller('SomeController', SomeController);

function SomeController() { }


/* recommended */
```

```
angular
    .module('app')
    .controller('SomeController', SomeController);

function SomeController() { }
```

Named vs Anonymous Functions

Use named functions instead of passing an anonymous function in as a callback. This produces more readable code, is much easier to debug, and reduces the amount of nested callback code.

```
/* avoid */
angular
    .module('app')
    .controller('Dashboard', function() { })
    .factory('logger', function() { });


/* recommended */
// dashboard.js
angular
    .module('app')
    .controller('Dashboard', Dashboard);

function Dashboard() { }


// logger.js
angular
    .module('app')
    .factory('logger', logger);

function logger() { }
```

11. TESTING

Angular is designed for test-driven development. The recommended unit testing setup is Jasmine/Mocha + Karma

Test Library

- Both Jasmine and Mocha are widely used in the AngularJS community. Both are stable, well maintained, and provide robust testing features.
- When using Mocha, also consider choosing an assert library such as Chai.

Test Runner

- Karma is easy to configure to run once or automatically when you change your code.
- Karma hooks into your Continuous Integration process easily on its own or through Grunt or Gulp.
- Some IDE's are beginning to integrate with Karma, such as WebStorm and Visual Studio.

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

12. REFERENCES

1. Cybage Angular Projects
2. Angular Style-guides
3. Angular Developer Blogs
4. Angular Books – O'reilly