Angular Standardization Guide

- Baskaran, Varadarajan

VERSION CONTROL

Prepared by:	Baskaran, Varadarajan
Date:	6/10/2015
Reviewed and Accepted by:	B.C.Subramanian
Date:	7/10/2015
Approved by:	Baskaran Varadarajan, Soundararajan Das
Date:	28/12/2015

VERSION HISTORY

Date	Version	Author	Description
6/10/2015	0.1	Baskaran	Initial Draft Version
9/10/2015	1.0	Bala, Das	Final Reviewed
28/12/2015	1.1	Baskaran	Final Approved

Table of Contents

Contents

1. JavaScript Scopes

- Wrap Angular components in an Immediately Invoked Function Expression (IIFE).

  Why?: An IIFE removes variables from the global scope. This helps prevent variables and function declarations from living longer than expected in the global scope, which also helps avoid variable collisions.

  Why?: When your code is minified and bundled into a single file for deployment to a production server, you could have collisions of variables and many global variables. An IIFE protects you against both of these by providing variable scope for each file.

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

// logger function is added as a global variable
function logger() { }

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

// storage function is added as a global variable
function storage() { }

/**
* recommended
*
* no globals are left behind
*/

// logger.js
(function() {
    'use strict';

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

```
    function logger() { }
})();

// storage.js
(function() {
    'use strict';

    angular
        .module('app')
        .factory('storage', storage);

    function storage() { }
})();
```

- Note: For brevity only, the rest of the examples in this guide may omit the IIFE syntax.
- Note: IIFE's prevent test code from reaching private members like regular expressions or helper functions which are often good to unit test directly on their own. However you can test these through accessible members or by exposing them through their own component. For example placing helper functions, regular expressions or constants in their own factory or constant.

2. MODULES

2.1 AVOID NAMING COLLISIONS

- Use unique naming conventions with separators for sub-modules.

  Why?: Unique names help avoid module name collisions. Separators help define modules and their submodule hierarchy. For example `app` may be your root module while `app.dashboard` and `app.users` may be modules that are used as dependencies of `app`.

2.2 DEFINITIONS (AKA SETTERS)

- Declare modules without a variable using the setter syntax.

  Why?: With 1 component per file, there is rarely a need to introduce a variable for the module.

```
/* 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'
    ]);
```

3. GETTERS

- When using a module, avoid using a variable and instead use chaining with the getter syntax.

  Why?: 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() { }
```

4. SETTING VS GETTING

- Only set once and get for all other instances.

  Why?: A module should only be created once, then retrieved from that point and after.

```
/* recommended */

// to set a module
angular.module('app', []);

// to get a module
angular.module('app');
```

5. Named vs Anonymous Functions

Use named functions instead of passing an anonymous function in as a callback.

Why?: This produces more readable code, is much easier to debug, and reduces the amount of nested callback code.

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

/* recommended */

// dashboard.js
angular
    .module('app')
    .controller('DashboardController', DashboardController);

function DashboardController() { }

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

function logger() { }
```

6. CONTROLLERS

6.1 CONTROLLERAS VIEW SYNTAX

- Use the `controllerAs` syntax over the `classic controller with $scope` syntax.
  Why?: Controllers are constructed, "newed" up, and provide a single new instance, and the `controllerAs` syntax is closer to that of a JavaScript constructor than the `classic $scope syntax`.
  Why?: It promotes the use of binding to a "dotted" object in the View (e.g. `customer.name` instead of `name`), which is more contextual, easier to read, and avoids any reference issues that may occur without "dotting".
  Why?: Helps avoid using `$parent` calls in Views with nested controllers.

```
<!-- avoid -->
<div ng-controller="CustomerController">
    {{ name }}
</div>

<!-- recommended -->
<div ng-controller="CustomerController as customer">
    {{ customer.name }}
</div>
```

6.2 CONTROLLERAS CONTROLLER SYNTAX

- Use the `controllerAs` syntax over the `classic controller with $scope` syntax.
- The `controllerAs` syntax uses `this` inside controllers which gets bound to `$scope`
  Why?: `controllerAs` is syntactic sugar over `$scope`. You can still bind to the View and still access `$scope` methods.
  Why?: Helps avoid the temptation of using `$scope` methods inside a controller when it may otherwise be better to avoid them or move the method to a factory, and reference them from the controller. Consider using `$scope` in a controller only when needed. For example when publishing and subscribing events using `$emit`, `$broadcast`, or `$on`.

```
/* avoid */
function CustomerController($scope) {
    $scope.name = {};
    $scope.sendMessage = function() { };
}

/* recommended - but see next section */
```

```
function CustomerController() {
    this.name = {};
    this.sendMessage = function() { };
}
```

6.3 CONTROLLERAS WITH VM

- Use a capture variable for `this` when using the `controllerAs` syntax. Choose a consistent variable name such as `vm`, which stands for ViewModel.
  Why?: The `this` keyword is contextual and when used within a function inside a controller may change its context. Capturing the context of `this` avoids encountering this problem.

```
/* avoid */
function CustomerController() {
    this.name = {};
    this.sendMessage = function() { };
}

/* recommended */
function CustomerController() {
    var vm = this;
    vm.name = {};
    vm.sendMessage = function() { };
}
```

Note: You can avoid any jshint warnings by placing the comment above the line of code. However it is not needed when the function is named using UpperCasing, as this convention means it is a constructor function, which is what a controller is in Angular.

```
/* jshint validthis: true */
var vm = this;
```

Note: When creating watches in a controller using `controller as`, you can watch the `vm.*` member using the following syntax. (Create watches with caution as they add more load to the digest cycle.)

```
<input ng-model="vm.title"/>

function SomeController($scope, $log) {
    var vm = this;
    vm.title = 'Some Title';

    $scope.$watch('vm.title', function(current, original) {
        $log.info('vm.title was %s', original);
```

```
        $log.info('vm.title is now %s', current);
    });
}
```

Note: When working with larger codebases, using a more descriptive name can help ease cognitive overhead & searchability. Avoid overly verbose names that are cumbersome to type.

```
<!-- avoid -->
<input ng-model="customerProductItemVm.title">

<!-- recommended -->
<input ng-model="productVm.title">
```

7. BINDABLE MEMBERS UP TOP

- Place bindable members at the top of the controller, alphabetized, and not spread through the controller code.

  Why?: 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.

  Why?: Setting anonymous functions in-line can be easy, but when those functions are more than 1 line of code they can reduce the readability. Defining the functions below the bindable members (the functions will be hoisted) moves the implementation details down, keeps the bindable members up top, and makes it easier to read.

```
/* avoid */
function SessionsController() {
    var vm = this;

    vm.gotoSession = function() {
      /* ... */
    };
    vm.refresh = function() {
      /* ... */
    };
    vm.search = function() {
      /* ... */
    };
    vm.sessions = [];
    vm.title = 'Sessions';
}

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

    vm.gotoSession = gotoSession;
    vm.refresh = refresh;
    vm.search = search;
    vm.sessions = [];
    vm.title = 'Sessions';

    ////////////
```

```
    function gotoSession() {
      /* */
    }

    function refresh() {
      /* */
    }

    function search() {
      /* */
    }
}
```

```
(function () {
    'use strict';

    angular
        .module('app')
        .controller('Speakers', Speakers);

    Speakers.$inject = ['$location', 'common', 'config', 'data

    function Speakers($location, common, config, datacontext)
        /*jshint validthis: true */
        var vm = this;

        var keyCodes = config.keyCodes;

        vm.filteredSpeakers = [];
        vm.gotoSpeaker = gotoSpeaker;
        vm.refresh = refresh;
        vm.search = search;
        vm.speakerSearch = '';
        vm.speakers = [];
        vm.title = 'Speakers';

        activate();

        function activate() {
            return getSpeakers();
        }

        function applyFilter() {
            vm.filteredSpeakers = vm.speakers.filter(speakerF
        }
```

Note: If the function is a 1 liner consider keeping it right up top, as long as readability is not affected.

```
/* avoid */
function SessionsController(data) {
    var vm = this;

    vm.gotoSession = gotoSession;
```

```
    vm.refresh = function() {
        /**
         * lines
         * of
         * code
         * affects
         * readability
         */
    };
    vm.search = search;
    vm.sessions = [];
    vm.title = 'Sessions';
}

/* recommended */
function SessionsController(sessionDataService) {
    var vm = this;

    vm.gotoSession = gotoSession;
    vm.refresh = sessionDataService.refresh; // 1 liner is OK
    vm.search = search;
    vm.sessions = [];
    vm.title = 'Sessions';
}
```

8. Function Declarations to Hide Implementation Details

- Use function declarations to hide implementation details. Keep your bindable members up top. When you need to bind a function in a controller, point it to a function declaration that appears later in the file. This is tied directly to the section Bindable Members Up Top. For more details see this post.

  Why?: 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. (Same as above.)

  Why?: Placing the implementation details of a function later in the file moves that complexity out of view so you can see the important stuff up top.

  Why?: Function declarations are hoisted so there are no concerns over using a function before it is defined (as there would be with function expressions).

  Why?: You never have to worry with function declarations that moving `var a` before `var b` will break your code because `a` depends on `b`.

  Why?: Order is critical with function expressions

```
/**
* avoid
* Using function expressions.
*/
function AvengersController(avengersService, logger) {
    var vm = this;
    vm.avengers = [];
    vm.title = 'Avengers';

    var activate = function() {
        return getAvengers().then(function() {
            logger.info('Activated Avengers View');
        });
    }

    var getAvengers = function() {
        return avengersService.getAvengers().then(function(data) {
            vm.avengers = data;
            return vm.avengers;
```

```
        });
    }

    vm.getAvengers = getAvengers;

    activate();
}
```

Notice that the important stuff is scattered in the preceding example. In the example below, notice that the important stuff is up top. For example, the members bound to the controller such as `vm.avengers` and `vm.title`. The implementation details are down below. This is just easier to read.

```
/*
* recommend
* Using function declarations
* and bindable members up top.
*/
function AvengersController(avengersService, logger) {
    var vm = this;
    vm.avengers = [];
    vm.getAvengers = getAvengers;
    vm.title = 'Avengers';

    activate();

    function activate() {
        return getAvengers().then(function() {
            logger.info('Activated Avengers View');
        });
    }

    function getAvengers() {
        return avengersService.getAvengers().then(function(data) {
            vm.avengers = data;
            return vm.avengers;
        });
    }
}
```

9. Defer Controller Logic to Services

- Defer logic in a controller by delegating to services and factories.

  Why?: Logic may be reused by multiple controllers when placed within a service and exposed via a function.

  Why?: Logic in a service can more easily be isolated in a unit test, while the calling logic in the controller can be easily mocked.

  Why?: Removes dependencies and hides implementation details from the controller.

  Why?: Keeps the controller slim, trim, and focused.

```
/* avoid */
function OrderController($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 OrderController(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(showError);
    };
}
```

10. KEEP CONTROLLERS FOCUSED

- Define a controller for a view, and try not to reuse the controller for other views. Instead, move reusable logic to factories and keep the controller simple and focused on its view.

  Why?: Reusing controllers with several views is brittle and good end-to-end (e2e) test coverage is required to ensure stability across large applications.

11. ASSIGNING CONTROLLERS

- When a controller must be paired with a view and either component may be re-used by other controllers or views, define controllers along with their routes.

  Note: If a View is loaded via another means besides a route, then use the `ng-controller="Avengers as vm"` syntax.

  Why?: Pairing the controller in the route allows different routes to invoke different pairs of controllers and views. When controllers are assigned in the view using `ng-controller`, that view is always associated with the same controller.

```
/* avoid - when using with a route and dynamic pairing is desired */

// route-config.js
angular
    .module('app')
    .config(config);

function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
          templateUrl: 'avengers.html'
        });
}

<!-- avengers.html -->
<div ng-controller="AvengersController as vm">
</div>

/* recommended */

// route-config.js
angular
    .module('app')
    .config(config);

function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'Avengers',
            controllerAs: 'vm'
        });
}

<!-- avengers.html -->
<div>
</div>
```

12. SERVICES

12.1 SINGLETONS

- 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.

  Note: All Angular services are singletons. This means that there is only one instance of a given service per injector.

```
// 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) {
          /* */
        }
   };
}
```

13. FACTORIES

13.1 SINGLE RESPONSIBILITY

- Factories should have a single responsibility, that is encapsulated by its context. Once a factory begins to exceed that singular purpose, a new factory should be created.

13.2 SINGLETONS

- Factories are singletons and return an object that contains the members of the service.

  Note: All Angular services are singletons.

13.3 ACCESSIBLE MEMBERS UP TOP

- Expose the callable members of the service (its interface) at the top, using a technique derived from the Revealing Module Pattern.

  Why?: Placing the callable members at the top makes it easy to read and helps you instantly identify which members of the service can be called and must be unit tested (and/or mocked).

  Why?: This is especially helpful when the file gets longer as it helps avoid the need to scroll to see what is exposed.

  Why?: Setting functions as you go can be easy, but when those functions are more than 1 line of code they can reduce the readability and cause more scrolling. Defining the callable interface via the returned service moves the implementation details down, keeps the callable interface up top, and makes it easier to read.

```
/* avoid */
function dataService() {
  var someValue = '';
  function save() {
    /* */
```

```
  };
  function validate() {
    /* */
  };

  return {
      save: save,
      someValue: someValue,
      validate: validate
  };
}

/* recommended */
function dataService() {
    var someValue = '';
    var service = {
        save: save,
        someValue: someValue,
        validate: validate
    };
    return service;

    ////////////

    function save() {
        /* */
    };

    function validate() {
        /* */
    };
}
```

This way bindings are mirrored across the host object, primitive values cannot update alone using the revealing module pattern.

```
(function() {
    'use strict';

    angular
        .module('blocks.logger')
        .factory('logger', logger);

    logger.$inject = ['$log', 'toastr'];

    function logger($log, toastr) {
        var service = {
            showToasts: true,

            error   : error,
            info    : info,
            success : success,
            warning : warning,

            // straight to console; bypass toastr
            log     : $log.log
        };

        return service;
        ////////////////////

        function error(message, data, title) {
            toastr.error(message, title);
            $log.error('Error: ' + message, data);
        }

        function info(message, data, title) {
```

14. FUNCTION DECLARATIONS TO HIDE IMPLEMENTATION DETAILS

- Use function declarations to hide implementation details. Keep your accessible members of the factory up top. Point those to function declarations that appears later in the file. For more details see this post.

  Why?: Placing accessible members at the top makes it easy to read and helps you instantly identify which functions of the factory you can access externally.

  Why?: Placing the implementation details of a function later in the file moves that complexity out of view so you can see the important stuff up top.

  Why?: Function declarations are hoisted so there are no concerns over using a function before it is defined (as there would be with function expressions).

  Why?: You never have to worry with function declarations that moving `var a` before `var b` will break your code because `a` depends on `b`.

  Why?: Order is critical with function expressions

```
/**
* avoid
* Using function expressions
*/
function dataservice($http, $location, $q, exception, logger) {
  var isPrimed = false;
  var primePromise;

  var getAvengers = function() {
      // implementation details go here
  };

  var getAvengerCount = function() {
      // implementation details go here
  };

  var getAvengersCast = function() {
     // implementation details go here
  };

  var prime = function() {
     // implementation details go here
  };
```

```
  var ready = function(nextPromises) {
      // implementation details go here
  };

  var service = {
      getAvengersCast: getAvengersCast,
      getAvengerCount: getAvengerCount,
      getAvengers: getAvengers,
      ready: ready
  };

  return service;
}

/**
* recommended
* Using function declarations
* and accessible members up top.
*/
function dataservice($http, $location, $q, exception, logger) {
    var isPrimed = false;
    var primePromise;

    var service = {
        getAvengersCast: getAvengersCast,
        getAvengerCount: getAvengerCount,
        getAvengers: getAvengers,
        ready: ready
    };

    return service;

    ////////////

    function getAvengers() {
        // implementation details go here
    }

    function getAvengerCount() {
        // implementation details go here
    }

    function getAvengersCast() {
        // implementation details go here
    }

    function prime() {
        // implementation details go here
    }
```

```
    function ready(nextPromises) {
        // implementation details go here
    }
}
```

15. DATA SERVICES

15.1 SEPARATE DATA CALLS

- Refactor logic for making data operations and interacting with data to a factory. Make data services responsible for XHR calls, local storage, stashing in memory, or any other data operations.

  Why?: The controller's responsibility is for the presentation and gathering of information for the view. It should not care how it gets the data, just that it knows who to ask for it. Separating the data services moves the logic on how to get it to the data service, and lets the controller be simpler and more focused on the view.

  Why?: This makes it easier to test (mock or real) the data calls when testing a controller that uses a data service.

  Why?: Data service implementation may have very specific code to handle the data repository. This may include headers, how to talk to the data, or other services such as `$http`. Separating the logic into a data service encapsulates this logic in a single place hiding the implementation from the outside consumers (perhaps a controller), also making it easier to change the implementation.

```
/* recommended */

// dataservice factory
angular
    .module('app.core')
    .factory('dataservice', dataservice);

dataservice.$inject = ['$http', 'logger'];

function dataservice($http, logger) {
    return {
        getAvengers: getAvengers
    };

    function getAvengers() {
        return $http.get('/api/maa')
            .then(getAvengersComplete)
            .catch(getAvengersFailed);

        function getAvengersComplete(response) {
```

```
            return response.data.results;
        }

        function getAvengersFailed(error) {
            logger.error('XHR Failed for getAvengers.' + error.data);
        }
    }
}
```

Note: The data service is called from consumers, such as a controller, hiding the implementation from the consumers, as shown below.

```
/* recommended */

// controller calling the dataservice factory
angular
    .module('app.avengers')
    .controller('AvengersController', AvengersController);

AvengersController.$inject = ['dataservice', 'logger'];

function AvengersController(dataservice, logger) {
    var vm = this;
    vm.avengers = [];

    activate();

    function activate() {
        return getAvengers().then(function() {
            logger.info('Activated Avengers View');
        });
    }

    function getAvengers() {
        return dataservice.getAvengers()
            .then(function(data) {
                vm.avengers = data;
                return vm.avengers;
            });
    }
}
```

15.2 RETURN A PROMISE FROM DATA CALLS

- When calling a data service that returns a promise such as `$http`, return a promise in your calling function too.

Why?: You can chain the promises 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;
      });
}
```

16. DIRECTIVES

16.1 LIMIT 1 PER FILE

- Create one directive per file. Name the file for the directive.

  Why?: 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.

  Why?: One directive per file is easy to maintain.

  Note: "**Best Practice**: Directives should clean up after themselves. You can use `element.on('$destroy', ...)` or `scope.$on('$destroy', ...)` to run a clean-up function when the directive is removed" ... from the Angular documentation.

```
/* 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)

    /* spinner directive that can be used anywhere across apps */
    .directive('sharedSpinner', sharedSpinner);

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

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

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

/* recommended */
/* calendar-range.directive.js */
```

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

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

```
/* recommended */
/* customer-info.directive.js */

/**
* @desc sales directive that can be used anywhere across the sales app at a
company named Acme
* @example <div acme-sales-customer-info></div>
*/
angular
    .module('sales.widgets')
    .directive('acmeSalesCustomerInfo', salesCustomerInfo);

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

```
/* recommended */
/* spinner.directive.js */

/**
* @desc spinner directive that can be used anywhere across apps at a company
named Acme
* @example <div acme-shared-spinner></div>
*/
angular
    .module('shared.widgets')
    .directive('acmeSharedSpinner', sharedSpinner);

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

Note: There are many naming options for directives, especially since they can be used in narrow or wide scopes. Choose one that makes the directive and its file name distinct and clear. Some examples are below, but see the Namingsection for more recommendations.

17. MANIPULATE DOM IN A DIRECTIVE

- When manipulating the DOM directly, use a directive. If alternative ways can be used such as using CSS to set styles or the animation services, Angular templating, `ngShow` or `ngHide`, then use those instead. For example, if the directive simply hides and shows, use ngHide/ngShow.

  Why?: DOM manipulation can be difficult to test, debug, and there are often better ways (e.g. CSS, animations, templates)

18. PROVIDE A UNIQUE DIRECTIVE PREFIX

- Provide a short, unique and descriptive directive prefix such as `acmeSalesCustomerInfo` which would be declared in HTML as `acme-sales-customer-info`.

  Why?: The unique short prefix identifies the directive's context and origin. For example a prefix of `cc-` may indicate that the directive is part of a CodeCamper app while `acme-` may indicate a directive for the Acme company.

  Note: Avoid `ng-` as these are reserved for Angular directives. Research widely used directives to avoid naming conflicts, such as `ion-` for the Ionic Framework.

19. RESTRICT TO ELEMENTS AND ATTRIBUTES

- When creating a directive that makes sense as a stand-alone element, allow restrict `E` (custom element) and optionally restrict `A` (custom attribute). Generally, if it could be its own control, `E` is appropriate. General guideline is allow `EA` but lean towards implementing as an element when it's stand-alone and as an attribute when it enhances its existing DOM element.

  Why?: It makes sense.

  Why?: While we can allow the directive to be used as a class, if the directive is truly acting as an element it makes more sense as an element or at least as an attribute.

  Note: EA is the default for Angular 1.3 +

```
<!-- avoid -->
<div class="my-calendar-range"></div>

/* avoid */
angular
    .module('app.widgets')
    .directive('myCalendarRange', myCalendarRange);

function myCalendarRange() {
    var directive = {
        link: link,
        templateUrl: '/template/is/located/here.html',
        restrict: 'C'
    };
    return directive;

    function link(scope, element, attrs) {
      /* */
    }
}

<!-- recommended -->
<my-calendar-range></my-calendar-range>
<div my-calendar-range></div>

/* recommended */
angular
    .module('app.widgets')
    .directive('myCalendarRange', myCalendarRange);
```

```
function myCalendarRange() {
    var directive = {
        link: link,
        templateUrl: '/template/is/located/here.html',
        restrict: 'EA'
    };
    return directive;

    function link(scope, element, attrs) {
      /* */
    }
}
```

20. DIRECTIVES AND CONTROLLERAS

- Use `controller as` syntax with a directive to be consistent with using `controller as` with view and controller pairings.

  Why?: It makes sense and it's not difficult.

  Note: The directive below demonstrates some of the ways you can use scope inside of link and directive controllers, using controllerAs. I in-lined the template just to keep it all in one place.

  Note: Regarding dependency injection, see Manually Identify Dependencies.

  Note: Note that the directive's controller is outside the directive's closure. This style eliminates issues where the injection gets created as unreachable code after a `return`.

```
<div my-example max="77"></div>

angular
    .module('app')
    .directive('myExample', myExample);

function myExample() {
    var directive = {
        restrict: 'EA',
        templateUrl: 'app/feature/example.directive.html',
        scope: {
            max: '='
        },
        link: linkFunc,
        controller: ExampleController,
        // note: This would be 'ExampleController' (the exported controller
name, as string)
        // if referring to a defined controller in its separate file.
        controllerAs: 'vm',
        bindToController: true // because the scope is isolated
    };

    return directive;

    function linkFunc(scope, el, attr, ctrl) {
        console.log('LINK: scope.min = %s *** should be undefined',
scope.min);
```

```
        console.log('LINK: scope.max = %s *** should be undefined',
scope.max);
        console.log('LINK: scope.vm.min = %s', scope.vm.min);
        console.log('LINK: scope.vm.max = %s', scope.vm.max);
    }
}

ExampleController.$inject = ['$scope'];

function ExampleController($scope) {
    // Injecting $scope just for comparison
    var vm = this;

    vm.min = 3;

    console.log('CTRL: $scope.vm.min = %s', $scope.vm.min);
    console.log('CTRL: $scope.vm.max = %s', $scope.vm.max);
    console.log('CTRL: vm.min = %s', vm.min);
    console.log('CTRL: vm.max = %s', vm.max);
}
```

```
<!-- example.directive.html -->
<div>hello world</div>
<div>max={{vm.max}}<input ng-model="vm.max"/></div>
<div>min={{vm.min}}<input ng-model="vm.min"/></div>
```

Note: You can also name the controller when you inject it into the link function and access directive attributes as properties of the controller.

```
// Alternative to above example
function linkFunc(scope, el, attr, vm) {
    console.log('LINK: scope.min = %s *** should be undefined', scope.min);
    console.log('LINK: scope.max = %s *** should be undefined', scope.max);
    console.log('LINK: vm.min = %s', vm.min);
    console.log('LINK: vm.max = %s', vm.max);
}
```

- Use `bindToController = true` when using `controller as` syntax with a directive when you want to bind the outer scope to the directive's controller's scope.

  Why?: It makes it easy to bind outer scope to the directive's controller scope.

  Note: `bindToController` was introduced in Angular 1.3.0.

```
<div my-example max="77"></div>

angular
    .module('app')
    .directive('myExample', myExample);
```

```
function myExample() {
    var directive = {
        restrict: 'EA',
        templateUrl: 'app/feature/example.directive.html',
        scope: {
            max: '='
        },
        controller: ExampleController,
        controllerAs: 'vm',
        bindToController: true
    };

    return directive;
}

function ExampleController() {
    var vm = this;
    vm.min = 3;
    console.log('CTRL: vm.min = %s', vm.min);
    console.log('CTRL: vm.max = %s', vm.max);
}
```

```
<!-- example.directive.html -->
<div>hello world</div>
<div>max={{vm.max}}<input ng-model="vm.max"/></div>
<div>min={{vm.min}}<input ng-model="vm.min"/></div>
```

21. RESOLVING PROMISES

21.1 CONTROLLER ACTIVATION PROMISES

- Resolve start-up logic for a controller in an `activate` function.

  Why?: Placing start-up logic in a consistent place in the controller makes it easier to locate, more consistent to test, and helps avoid spreading out the activation logic across the controller.

  Why?: The controller `activate` makes it convenient to re-use the logic for a refresh for the controller/View, keeps the logic together, gets the user to the View faster, makes animations easy on the `ng-view` or `ui-view`, and feels snappier to the user.

  Note: If you need to conditionally cancel the route before you start using the controller, use a route resolve instead.

```
/* avoid */
function AvengersController(dataservice) {
    var vm = this;
    vm.avengers = [];
    vm.title = 'Avengers';

    dataservice.getAvengers().then(function(data) {
        vm.avengers = data;
        return vm.avengers;
    });
}

/* recommended */
function AvengersController(dataservice) {
    var vm = this;
    vm.avengers = [];
    vm.title = 'Avengers';

    activate();

    ////////////

    function activate() {
        return dataservice.getAvengers().then(function(data) {
            vm.avengers = data;
            return vm.avengers;
```

```
        });
    }
}
```

21.2 ROUTE RESOLVE PROMISES

- When a controller depends on a promise to be resolved before the controller is activated, resolve those dependencies in the `$routeProvider` before the controller logic is executed. If you need to conditionally cancel a route before the controller is activated, use a route resolver.

- Use a route resolve when you want to decide to cancel the route before ever transitioning to the View.

  Why?: A controller may require data before it loads. That data may come from a promise via a custom factory or $http. Using a route resolve allows the promise to resolve before the controller logic executes, so it might take action based on that data from the promise.

  Why?: The code executes after the route and in the controller's activate function. The View starts to load right away. Data binding kicks in when the activate promise resolves. A "busy" animation can be shown during the view transition (via `ng-view` or `ui-view`)
  Note: The code executes before the route via a promise. Rejecting the promise cancels the route. Resolve makes the new view wait for the route to resolve. A "busy" animation can be shown before the resolve and through the view transition. If you want to get to the View faster and do not require a checkpoint to decide if you can get to the View, consider
  the controller `activate` technique instead.

```
/* avoid */
angular
    .module('app')
    .controller('AvengersController', AvengersController);

function AvengersController(movieService) {
    var vm = this;
    // unresolved
    vm.movies;
```

```
    // resolved asynchronously
    movieService.getMovies().then(function(response) {
        vm.movies = response.movies;
    });
}

/* better */

// route-config.js
angular
    .module('app')
    .config(config);

function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'AvengersController',
            controllerAs: 'vm',
            resolve: {
                moviesPrepService: function(movieService) {
                    return movieService.getMovies();
                }
            }
        });
}

// avengers.js
angular
    .module('app')
    .controller('AvengersController', AvengersController);

AvengersController.$inject = ['moviesPrepService'];
function AvengersController(moviesPrepService) {
    var vm = this;
    vm.movies = moviesPrepService.movies;
}
```

Note: The example below shows the route resolve points to a named function, which is easier to debug and easier to handle dependency injection.

```
/* even better */

// route-config.js
angular
    .module('app')
    .config(config);

function config($routeProvider) {
    $routeProvider
```

```
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'AvengersController',
            controllerAs: 'vm',
            resolve: {
                moviesPrepService: moviesPrepService
            }
        });
}

function moviesPrepService(movieService) {
    return movieService.getMovies();
}

// avengers.js
angular
    .module('app')
    .controller('AvengersController', AvengersController);

AvengersController.$inject = ['moviesPrepService'];
function AvengersController(moviesPrepService) {
      var vm = this;
      vm.movies = moviesPrepService.movies;
}
```

Note: The code example's dependency on `movieService` is not minification safe on its own. For details on how to make this code minification safe, see the sections on dependency injection and on minification and annotation.

21.3 HANDLING EXCEPTIONS WITH PROMISES

- The `catch` block of a promise must return a rejected promise to maintain the exception in the promise chain.
- Always handle exceptions in services/factories.

  Why?: If the `catch` block does not return a rejected promise, the caller of the promise will not know an exception occurred. The caller's `then` will execute. Thus, the user may never know what happened.

  Why?: To avoid swallowing errors and misinforming the user.

Note: Consider putting any exception handling in a function in a shared module and service.

```
/* avoid */

function getCustomer(id) {
    return $http.get('/api/customer/' + id)
        .then(getCustomerComplete)
        .catch(getCustomerFailed);

    function getCustomerComplete(data, status, headers, config) {
        return data.data;
    }

    function getCustomerFailed(e) {
        var newMessage = 'XHR Failed for getCustomer'
        if (e.data && e.data.description) {
          newMessage = newMessage + '\n' + e.data.description;
        }
        e.data.description = newMessage;
        logger.error(newMessage);
        // ***
        // Notice there is no return of the rejected promise
        // ***
    }
}

/* recommended */
function getCustomer(id) {
    return $http.get('/api/customer/' + id)
        .then(getCustomerComplete)
        .catch(getCustomerFailed);

    function getCustomerComplete(data, status, headers, config) {
        return data.data;
    }

    function getCustomerFailed(e) {
        var newMessage = 'XHR Failed for getCustomer'
        if (e.data && e.data.description) {
          newMessage = newMessage + '\n' + e.data.description;
        }
        e.data.description = newMessage;
        logger.error(newMessage);
        return $q.reject(e);
    }
}
```

22. MANUAL ANNOTATING FOR DEPENDENCY INJECTION

22.1 UNSAFE FROM MINIFICATION

- Avoid using the shortcut syntax of declaring dependencies without using a minification-safe approach.

  Why?: The parameters to the component (e.g. controller, factory, etc) will be converted to mangled variables. For example, `common` and `dataservice` may become `a` or `b` and not be found by Angular.

```
/* avoid - not minification-safe*/
angular
    .module('app')
    .controller('DashboardController', DashboardController);

function DashboardController(common, dataservice) {
}
```

  This code may produce mangled variables when minified and thus cause runtime errors.

```
/* avoid - not minification-safe*/
angular.module('app').controller('DashboardController', d);function d(a, b) {
}
```

22.2 MANUALLY IDENTIFY DEPENDENCIES

- Use `$inject` to manually identify your dependencies for Angular components.
  Why?: This technique mirrors the technique used by `ng-annotate`, which I recommend for automating the creation of minification safe dependencies. If `ng-annotate` detects injection has already been made, it will not duplicate it.
  Why?: This safeguards your dependencies from being vulnerable to minification issues when parameters may be mangled. For example, `common` and `dataservice` may become `a` or `b` and not be found by Angular.

Why?: Avoid creating in-line dependencies as long lists can be difficult to read in the array. Also it can be confusing that the array is a series of strings while the last item is the component's function.

```
/* avoid */
angular
    .module('app')
    .controller('DashboardController',
        ['$location', '$routeParams', 'common', 'dataservice',
            function Dashboard($location, $routeParams, common, dataservice)
{}
        ]);

/* avoid */
angular
  .module('app')
  .controller('DashboardController',
      ['$location', '$routeParams', 'common', 'dataservice', Dashboard]);

function Dashboard($location, $routeParams, common, dataservice) {
}

/* recommended */
angular
    .module('app')
    .controller('DashboardController', DashboardController);

DashboardController.$inject = ['$location', '$routeParams', 'common',
'dataservice'];

function DashboardController($location, $routeParams, common, dataservice) {
}
```

Note: When your function is below a return statement the `$inject` may be unreachable (this may happen in a directive). You can solve this by moving the Controller outside of the directive.

```
/* avoid */
// inside a directive definition
function outer() {
    var ddo = {
        controller: DashboardPanelController,
        controllerAs: 'vm'
    };
    return ddo;

    DashboardPanelController.$inject = ['logger']; // Unreachable
    function DashboardPanelController(logger) {
    }
```

```
}

/* recommended */
// outside a directive definition
function outer() {
    var ddo = {
        controller: DashboardPanelController,
        controllerAs: 'vm'
    };
    return ddo;
}

DashboardPanelController.$inject = ['logger'];
function DashboardPanelController(logger) {
}
```

23. MANUALLY IDENTIFY ROUTE RESOLVER DEPENDENCIES

- Use `$inject` to manually identify your route resolver dependencies for Angular components.

  Why?: This technique breaks out the anonymous function for the route resolver, making it easier to read.

  Why?: An `$inject` statement can easily precede the resolver to handle making any dependencies minification safe.

```
/* recommended */
function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'AvengersController',
            controllerAs: 'vm',
            resolve: {
                moviesPrepService: moviesPrepService
            }
        });
}

moviesPrepService.$inject = ['movieService'];
function moviesPrepService(movieService) {
    return movieService.getMovies();
}
```

24. MINIFICATION AND ANNOTATION

24.1 NG-ANNOTATE

- Use ng-annotate for Gulp or Grunt and comment functions that need automated dependency injection using `/* @ngInject */`

  Why?: This safeguards your code from any dependencies that may not be using minification-safe practices.

  Why?: `ng-min` is deprecated
  I prefer Gulp as I feel it is easier to write, to read, and to debug.

  The following code is not using minification safe dependencies.

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

/* @ngInject */
function AvengersController(storage, avengerService) {
    var vm = this;
    vm.heroSearch = '';
    vm.storeHero = storeHero;

    function storeHero() {
        var hero = avengerService.find(vm.heroSearch);
        storage.save(hero.name, hero);
    }
}
```

  When the above code is run through ng-annotate it will produce the following output with the `$inject` annotation and become minification-safe.

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

/* @ngInject */
function AvengersController(storage, avengerService) {
    var vm = this;
    vm.heroSearch = '';
    vm.storeHero = storeHero;

    function storeHero() {
        var hero = avengerService.find(vm.heroSearch);
```

```
        storage.save(hero.name, hero);
    }
}

AvengersController.$inject = ['storage', 'avengerService'];
```

Note: If `ng-annotate` detects injection has already been made (e.g. `@ngInject` was detected), it will not duplicate the`$inject` code.

Note: When using a route resolver you can prefix the resolver's function with `/* @ngInject */` and it will produce properly annotated code, keeping any injected dependencies minification safe.

```
// Using @ngInject annotations
function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'AvengersController',
            controllerAs: 'vm',
            resolve: { /* @ngInject */
                moviesPrepService: function(movieService) {
                    return movieService.getMovies();
                }
            }
        });
}
```

Note: Starting from Angular 1.3 you can use the `ngApp` directive's `ngStrictDi` parameter to detect any potentially missing minification safe dependencies. When present the injector will be created in "strict-di" mode causing the application to fail to invoke functions which do not use explicit function annotation (these may not be minification safe). Debugging info will be logged to the console to help track down the offending code. I prefer to only use `ng-strict-di` for debugging purposes only. `<body ng-app="APP" ng-strict-di>`

25. USE GULP OR GRUNT FOR NG-ANNOTATE

- Use gulp-ng-annotate or grunt-ng-annotate in an automated build task. Inject `/* @ngInject */` prior to any function that has dependencies.
  Why?: ng-annotate will catch most dependencies, but it sometimes requires hints using the `/* @ngInject */` syntax.

  The following code is an example of a gulp task using ngAnnotate

```
gulp.task('js', ['jshint'], function() {
    var source = pkg.paths.js;

    return gulp.src(source)
        .pipe(sourcemaps.init())
        .pipe(concat('all.min.js', {newLine: ';'}))
        // Annotate before uglify so the code get's min'd properly.
        .pipe(ngAnnotate({
            // true helps add where @ngInject is not used. It infers.
            // Doesn't work with resolve, so we must be explicit there
            add: true
        }))
        .pipe(bytediff.start())
        .pipe(uglify({mangle: true}))
        .pipe(bytediff.stop())
        .pipe(sourcemaps.write('./'))
        .pipe(gulp.dest(pkg.paths.dev));
});
```

26. EXCEPTION HANDLING

26.1 DECORATORS

- Use a decorator, at config time using the `$provide` service, on the `$exceptionHandler` service to perform custom actions when exceptions occur.

  Why?: Provides a consistent way to handle uncaught Angular exceptions for development-time or run-time.

  Note: Another option is to override the service instead of using a decorator. This is a fine option, but if you want to keep the default behavior and extend it a decorator is recommended.

```
/* recommended */
angular
    .module('blocks.exception')
    .config(exceptionConfig);

exceptionConfig.$inject = ['$provide'];

function exceptionConfig($provide) {
    $provide.decorator('$exceptionHandler', extendExceptionHandler);
}

extendExceptionHandler.$inject = ['$delegate', 'toastr'];

function extendExceptionHandler($delegate, toastr) {
    return function(exception, cause) {
        $delegate(exception, cause);
        var errorData = {
            exception: exception,
            cause: cause
        };
        /**
         * Could add the error to a service's collection,
         * add errors to $rootScope, log errors to remote web server,
         * or log locally. Or throw hard. It is entirely up to you.
         * throw exception;
         */
        toastr.error(exception.msg, errorData);
    };
}
```

26.2 Exception Catchers

- Create a factory that exposes an interface to catch and gracefully handle exceptions.

  Why?: Provides a consistent way to catch exceptions that may be thrown in your code (e.g. during XHR calls or promise failures).

  Note: The exception catcher is good for catching and reacting to specific exceptions from calls that you know may throw one. For example, when making an XHR call to retrieve data from a remote web service and you want to catch any exceptions from that service and react uniquely.

```
/* recommended */
angular
    .module('blocks.exception')
    .factory('exception', exception);

exception.$inject = ['logger'];

function exception(logger) {
    var service = {
        catcher: catcher
    };
    return service;

    function catcher(message) {
        return function(reason) {
            logger.error(message, reason);
        };
    }
}
```

26.3 Route Errors

- Handle and log all routing errors using `$routeChangeError`.

  Why?: Provides a consistent way to handle all routing errors.

  Why?: Potentially provides a better user experience if a routing error occurs and you route them to a friendly screen with more details or recovery options.

```
/* recommended */
```

```
var handlingRouteChangeError = false;

function handleRoutingErrors() {
    /**
     * Route cancellation:
     * On routing error, go to the dashboard.
     * Provide an exit clause if it tries to do it twice.
     */
    $rootScope.$on('$routeChangeError',
        function(event, current, previous, rejection) {
            if (handlingRouteChangeError) { return; }
            handlingRouteChangeError = true;
            var destination = (current && (current.title ||
                current.name || current.loadedTemplateUrl)) ||
                'unknown target';
            var msg = 'Error routing to ' + destination + '. ' +
                (rejection.msg || '');

            /**
             * Optionally log using a custom service or $log.
             * (Don't forget to inject custom service)
             */
            logger.warning(msg, [current]);

            /**
             * On routing error, go to another route/state.
             */
            $location.path('/');

        }
    );
}
```

27. NAMING

27.1 NAMING GUIDELINES

- Use consistent names for all components following a pattern that describes the component's feature then (optionally) its type. My recommended pattern is `feature.type.js`. There are 2 names for most assets:
    - the file name (`avengers.controller.js`)
    - the registered component name with Angular (`AvengersController`)

  Why?: Naming conventions help provide a consistent way to find content at a glance. Consistency within the project is vital. Consistency with a team is important. Consistency across a company provides tremendous efficiency.

  Why?: The naming conventions should simply help you find your code faster and make it easier to understand.

27.2 FEATURE FILE NAMES

- Use consistent names for all components following a pattern that describes the component's feature then (optionally) its type. My recommended pattern is `feature.type.js`.

  Why?: Provides a consistent way to quickly identify components.

  Why?: Provides pattern matching for any automated tasks.

```
/**
 * common options
 */

// Controllers
avengers.js
avengers.controller.js
avengersController.js

// Services/Factories
logger.js
logger.service.js
loggerService.js
```

```
/**
 * recommended
 */

// controllers
avengers.controller.js
avengers.controller.spec.js

// services/factories
logger.service.js
logger.service.spec.js

// constants
constants.js

// module definition
avengers.module.js

// routes
avengers.routes.js
avengers.routes.spec.js

// configuration
avengers.config.js

// directives
avenger-profile.directive.js
avenger-profile.directive.spec.js
```

Note: Another common convention is naming controller files without the word `controller` in the file name such as `avengers.js` instead of `avengers.controller.js`. All other conventions still hold using a suffix of the type. Controllers are the most common type of component so this just saves typing and is still easily identifiable. I recommend you choose 1 convention and be consistent for your team. My preference is `avengers.controller.js` identifying the `AvengersController`.

```
  /**
   * recommended
   */
  // Controllers
  avengers.js
  avengers.spec.js
```

27.3 TEST FILE NAMES

- Name test specifications similar to the component they test with a suffix of `spec`.

Why?: Provides a consistent way to quickly identify components.

Why?: Provides pattern matching for karma or other test runners.

```
/**
 * recommended
 */
avengers.controller.spec.js
logger.service.spec.js
avengers.routes.spec.js
avenger-profile.directive.spec.js
```

27.4 CONTROLLER NAMES

- Use consistent names for all controllers named after their feature. Use UpperCamelCase for controllers, as they are constructors.

  Why?: Provides a consistent way to quickly identify and reference controllers.

  Why?: UpperCamelCase is conventional for identifying object that can be instantiated using a constructor.

```
/**
 * recommended
 */

// avengers.controller.js
angular
    .module
    .controller('HeroAvengersController', HeroAvengersController);

function HeroAvengersController() { }
```

27.5 CONTROLLER NAME SUFFIX

- Append the controller name with the suffix `Controller`.
  Why?: The `Controller` suffix is more commonly used and is more explicitly descriptive.

```
/**
 * recommended
 */

// avengers.controller.js
angular
```

```
    .module
    .controller('AvengersController', AvengersController);

function AvengersController() { }
```

27.6 FACTORY AND SERVICE NAMES

- Use consistent names for all factories and services named after their feature. Use camel-casing for services and factories. Avoid prefixing factories and services with `$`. Only suffix service and factories with `Service` when it is not clear what they are (i.e. when they are nouns).

  Why?: Provides a consistent way to quickly identify and reference factories.

  Why?: Avoids name collisions with built-in factories and services that use the `$` prefix.
  Why?: Clear service names such as `logger` do not require a suffix.
  Why?: Service names such as `avengers` are nouns and require a suffix and should be named `avengersService`.

```
/**
 * recommended
 */

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

function logger() { }

/**
 * recommended
 */

// credit.service.js
angular
    .module
    .factory('creditService', creditService);

function creditService() { }

// customer.service.js
angular
    .module
```

```
    .service('customerService', customerService);

function customerService() { }
```

27.7 DIRECTIVE COMPONENT NAMES

- Use consistent names for all directives using camel-case. Use a short prefix to describe the area that the directives belong (some example are company prefix or project prefix).

  Why?: Provides a consistent way to quickly identify and reference components.

```
/**
 * recommended
 */

// avenger-profile.directive.js
angular
    .module
    .directive('xxAvengerProfile', xxAvengerProfile);

// usage is <xx-avenger-profile> </xx-avenger-profile>

function xxAvengerProfile() { }
```

28. MODULES

- When there are multiple modules, the main module file is named `app.module.js` while other dependent modules are named after what they represent. For example, an admin module is named `admin.module.js`. The respective registered module names would be `app` and `admin`.

  Why?: Provides consistency for multiple module apps, and for expanding to large applications.

  Why?: Provides easy way to use task automation to load all module definitions first, then all other angular files (for bundling).

28.1 CONFIGURATION

- Separate configuration for a module into its own file named after the module. A configuration file for the main `app`module is named `app.config.js` (or simply `config.js`). A configuration for a module named `admin.module.js` is named`admin.config.js`.

  Why?: Separates configuration from module definition, components, and active code.

  Why?: Provides an identifiable place to set configuration for a module.

28.2 ROUTES

- Separate route configuration into its own file. Examples might be `app.route.js` for the main module and `admin.route.js`for the `admin` module. Even in smaller apps I prefer this separation from the rest of the configuration.

29. APPLICATION STRUCTURE LIFT PRINCIPLE

29.1 LIFT

- Structure your app such that you can Locate your code quickly, Identify the code at a glance, keep the Flattest structure you can, and Try to stay DRY. The structure should follow these 4 basic guidelines.

  Why LIFT?: Provides a consistent structure that scales well, is modular, and makes it easier to increase developer efficiency by finding code quickly. Another way to check your app structure is to ask yourself: How quickly can you open and work in all of the related files for a feature?

  When I find my structure is not feeling comfortable, I go back and revisit these LIFT guidelines

  i. Locating our code is easy
  ii. Identify code at a glance
  iii. Flat structure as long as we can
  iv. Try to stay DRY (Don't Repeat Yourself) or T-DRY

29.2 LOCATE

- Make locating your code intuitive, simple and fast.

  Why?: I find this to be super important for a project. If the team cannot find the files they need to work on quickly, they will not be able to work as efficiently as possible, and the structure needs to change. You may not know the file name or where its related files are, so putting them in the most intuitive locations and near each other saves a ton of time. A descriptive folder structure can help with this.

```
/bower_components
/client
  /app
    /avengers
    /blocks
      /exception
```

```
      /logger
    /core
    /dashboard
    /data
    /layout
    /widgets
  /content
  index.html
.bower.json
```

29.2 IDENTIFY

- When you look at a file you should instantly know what it contains and represents.

  Why?: You spend less time hunting and pecking for code, and become more efficient. If this means you want longer file names, then so be it. Be descriptive with file names and keeping the contents of the file to exactly 1 component. Avoid files with multiple controllers, multiple services, or a mixture. There are deviations of the 1 per file rule when I have a set of very small features that are all related to each other, they are still easily identifiable.

30. FLAT

- Keep a flat folder structure as long as possible. When you get to 7+ files, begin considering separation.

  Why?: Nobody wants to search 7 levels of folders to find a file. Think about menus on web sites … anything deeper than 2 should take serious consideration. In a folder structure there is no hard and fast number rule, but when a folder has 7-10 files, that may be time to create subfolders. Base it on your comfort level. Use a flatter structure until there is an obvious value (to help the rest of LIFT) in creating a new folder.

31. T-DRY (TRY TO STICK TO DRY)

- Be DRY, but don't go nuts and sacrifice readability.

  Why?: Being DRY is important, but not crucial if it sacrifices the others in LIFT, which is why I call it T-DRY. I don’t want to type session-view.html for a view because, well, it’s obviously a view. If it is not obvious or by convention, then I name it.

32. Application Structure

32.1 Overall Guidelines

- Have a near term view of implementation and a long term vision. In other words, start small but keep in mind on where the app is heading down the road. All of the app's code goes in a root folder named `app`. All content is 1 feature per file. Each controller, service, module, view is in its own file. All 3rd party vendor scripts are stored in another root folder and not in the `app` folder. I didn't write them and I don't want them cluttering my app (`bower_components`, `scripts`, `lib`).

  Note: Find more details and reasoning behind the structure at this original post on application structure.

32.2 Layout

- Place components that define the overall layout of the application in a folder named `layout`. These may include a shell view and controller may act as the container for the app, navigation, menus, content areas, and other regions.

  Why?: Organizes all layout in a single place re-used throughout the application.

32.3 Folders-by-Feature Structure

- Create folders named for the feature they represent. When a folder grows to contain more than 7 files, start to consider creating a folder for them. Your threshold may be different, so adjust as needed.

  Why?: A developer can locate the code, identify what each file represents at a glance, the structure is flat as can be, and there is no repetitive nor redundant names.

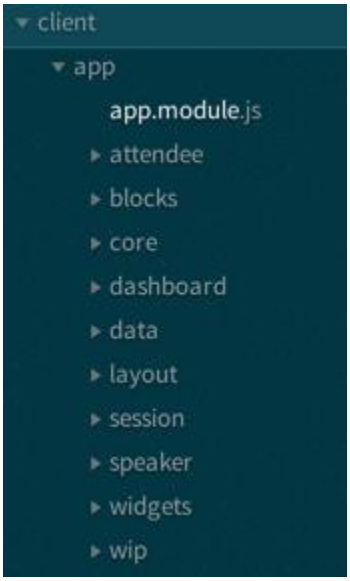
  Why?: The LIFT guidelines are all covered.

  Why?: Helps reduce the app from becoming cluttered through organizing the content and keeping them aligned with the LIFT guidelines.

Why?: When there are a lot of files (10+) locating them is easier with a consistent folder structures and more difficult in flat structures.

```
/**
 * recommended
 */

app/
    app.module.js
    app.config.js
    components/
        calendar.directive.js
        calendar.directive.html
        user-profile.directive.js
        user-profile.directive.html
    layout/
        shell.html
        shell.controller.js
        topnav.html
        topnav.controller.js
    people/
        attendees.html
        attendees.controller.js
        people.routes.js
        speakers.html
        speakers.controller.js
        speaker-detail.html
        speaker-detail.controller.js
    services/
        data.service.js
        localstorage.service.js
        logger.service.js
        spinner.service.js
    sessions/
        sessions.html
        sessions.controller.js
        sessions.routes.js
        session-detail.html
        session-detail.controller.js
```



Note: Do not structure your app using folders-by-type. This requires moving to multiple folders when working on a feature and gets unwieldy quickly as the app grows to 5, 10 or 25+ views and controllers (and other features), which makes it more difficult than folder-by-feature to locate files.

```
/*
* avoid
* Alternative folders-by-type.
* I recommend "folders-by-feature", instead.
*/

app/
    app.module.js
    app.config.js
    app.routes.js
    directives.js
    controllers/
        attendees.js
        session-detail.js
        sessions.js
        shell.js
        speakers.js
        speaker-detail.js
        topnav.js
    directives/
        calendar.directive.js
        calendar.directive.html
        user-profile.directive.js
        user-profile.directive.html
```

```
    services/
        dataservice.js
        localstorage.js
        logger.js
        spinner.js
    views/
        attendees.html
        session-detail.html
        sessions.html
        shell.html
        speakers.html
        speaker-detail.html
        topnav.html
```

33. MODULARITY

33.1 MANY SMALL, SELF CONTAINED MODULES

- Create small modules that encapsulate one responsibility.

  Why?: Modular applications make it easy to plug and go as they allow the development teams to build vertical slices of the applications and roll out incrementally. This means we can plug in new features as we develop them.

33.2 CREATE AN APP MODULE

- Create an application root module whose role is to pull together all of the modules and features of your application. Name this for your application.

  Why?: Angular encourages modularity and separation patterns. Creating an application root module whose role is to tie your other modules together provides a very straightforward way to add or remove modules from your application.

33.3 KEEP THE APP MODULE THIN

- Only put logic for pulling together the app in the application module. Leave features in their own modules.

  Why?: Adding additional roles to the application root to get remote data, display views, or other logic not related to pulling the app together muddies the app module and make both sets of features harder to reuse or turn off.

  Why?: The app module becomes a manifest that describes which modules help define the application.

33.4 FEATURE AREAS ARE MODULES

- Create modules that represent feature areas, such as layout, reusable and shared services, dashboards, and app specific features (e.g. customers, admin, sales).

Why?: Self contained modules can be added to the application with little or no friction.

Why?: Sprints or iterations can focus on feature areas and turn them on at the end of the sprint or iteration.

Why?: Separating feature areas into modules makes it easier to test the modules in isolation and reuse code.

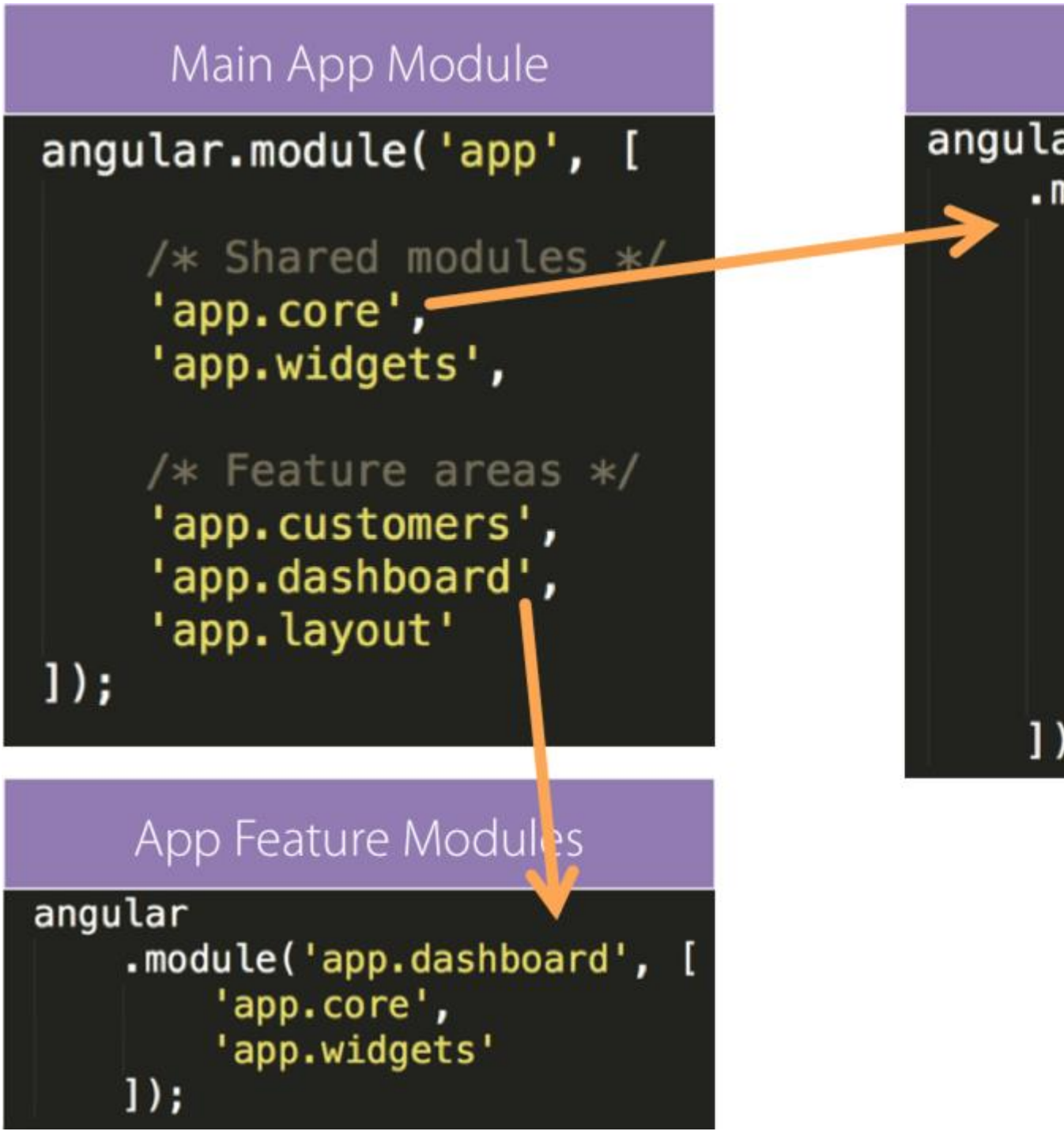
33.5 REUSABLE BLOCKS ARE MODULES

- Create modules that represent reusable application blocks for common services such as exception handling, logging, diagnostics, security, and local data stashing.

  Why?: These types of features are needed in many applications, so by keeping them separated in their own modules they can be application generic and be reused across applications.

34. MODULE DEPENDENCIES

- The application root module depends on the app specific feature modules and any shared or reusable modules.



Why?: The main app module contains a quickly identifiable manifest of the application's features.

Why?: Each feature area contains a manifest of what it depends on, so it can be pulled in as a dependency in other applications and still work.

Why?: Intra-App features such as shared data services become easy to locate and share from within `app.core` (choose your favorite name for this module).

Note: This is a strategy for consistency. There are many good options here. Choose one that is consistent, follows Angular's dependency rules, and is easy to maintain and scale.

My structures vary slightly between projects but they all follow these guidelines for structure and modularity. The implementation may vary depending on the features and the team. In other words, don't get hung up on an exact like-for-like structure but do justify your structure using consistency, maintainability, and efficiency in mind.

In a small app, you can also consider putting all the shared dependencies in the app module where the feature modules have no direct dependencies. This makes it easier to maintain the smaller application, but makes it harder to reuse modules outside of this application.

35. STARTUP LOGIC

35.1 CONFIGURATION

- Inject code into module configuration that must be configured before running the angular app. Ideal candidates include providers and constants.

  Why?: This makes it easier to have less places for configuration.

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

configure.$inject =
    ['routerHelperProvider', 'exceptionHandlerProvider', 'toastr'];

function configure (routerHelperProvider, exceptionHandlerProvider, toastr) {
    exceptionHandlerProvider.configure(config.appErrorPrefix);
    configureStateHelper();

    toastr.options.timeOut = 4000;
    toastr.options.positionClass = 'toast-bottom-right';

    ////////////////

    function configureStateHelper() {
        routerHelperProvider.configure({
            docTitle: 'NG-Modular: '
        });
    }
}
```

36.2 RUN BLOCKS

- Any code that needs to run when an application starts should be declared in a factory, exposed via a function, and injected into the run block.

  Why?: Code directly in a run block can be difficult to test. Placing in a factory makes it easier to abstract and mock.

```
angular
    .module('app')
    .run(runBlock);

runBlock.$inject = ['authenticator', 'translator'];
```

```
function runBlock(authenticator, translator) {
    authenticator.initialize();
    translator.initialize();
}
```

37. ANGULAR $ WRAPPER SERVICES

37.1 $DOCUMENT AND $WINDOW

- Use `$document` and `$window` instead of `document` and `window`.

  Why?: These services are wrapped by Angular and more easily testable than using document and window in tests. This helps you avoid having to mock document and window yourself.

37.2 $TIMEOUT AND $INTERVAL

- Use `$timeout` and `$interval` instead of `setTimeout` and `setInterval` .

  Why?: These services are wrapped by Angular and more easily testable and handle Angular's digest cycle thus keeping data binding in sync.