ANGULARJS
by Google

MAYFLOWER

Controllers

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>
```

controllerAs controller syntax

Use the controllerAs syntax over the classic 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 consider moving these uses to a factory and invoke from the controller.

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

```
/* recommended - but see next section */
function CustomerController() {
    this.name = {};
    this.sendMessage = function() { };
}
```

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 used with UpperCasing, as this convention means it is a constructor function, which is what a controller is in AngularJS.

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

Note: When creating watches in a controller using controllerAs, 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);
        });
}
```

Bindable members up top

Place bindable members up 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() {
        /* */
    }
}
```

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

```
/* recommended */
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';
}
```

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.

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.

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 declaration 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.

```
/*
* recommended
* 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;
            });
    }
}
```

Assigning controllers

When a controller must be paired with a view and either component may be reused 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>
```

```
/* 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.avengers = data;
                return vm.avengers;
            });
    }

    vm.getAvengers = getAvengers;

    activate();
}
```

```
/*
* recommended
* Using function declarations
* and bindable members up top.
*/
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);
    };
}
```

Directives

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 */
/* calendarRange.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 */
/* customerInfo.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 */
}
```

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)

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 content 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.

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 attribute when it's stand-alone 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) {
        /* */
    }
}
```

Directives and controllerAs

Use controllerAs syntax with a directive to be consistent with using controllerAs 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.

```
<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,
        controllerAs: 'vm',
        bindToController: true
    };
    return directive;

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

Note: You can also watch the bindToController syntax with a directive when you want to bind the outer scope to the directive's controller 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.

```
/* Alternative to above example */
function linkFunc(scope, el, attrs, 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);
}
```

```
<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>
    <h2>hello world</h2>
    <div>max={{vm.max}}<input ng-model="vm.max"/></div>
    <div>min={{vm.min}}<input ng-model="vm.min"/></div>
</div>
```

Note: Regarding dependency injection, see Manually identify dependencies.

Note: 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.

Single responsibility

Rule of 1

Define 1 component per file.

The following example defines the app module and its dependencies, defines a controller, and defines a factory all in the same file.

```
/* avoid */
angular
    .module('app', ['ngRoute'])
    .controller('SomeController', SomeController)
    .factory('someFactory', someFactory);

function SomeController() { }

function someFactory() { }
```

The same components are now separated into their own files.

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

```
/* recommended */
// someController.js
angular
    .module('app')
    .controller('SomeController', SomeController);

function SomeController() { }
```

```
/* recommended */
// someFactory.js
angular
    .module('app')
    .factory('someFactory', someFactory);

function someFactory() { }
```

Need support for your AngularJS demands?

AngularJS Development
We provide first class agile development based on Angular and Node.js.

AngularJS Training
Our training combines the knowledge of our well known conference speakers with real life knowhow from man years of practical angular development. We go beyond full frontal talks and offer coding dojos, pair programming and code reviews with the team to train your team hands on using your own project.

AngularJS Consulting
We provide fast hands-on problem solution for your Angular development. Performance, quality, security – ask us.

mayflower.de
mail kontakt@mayflower.de

Services

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

Modularity

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.

Create an app module

Create an app module whose role is 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.

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.

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.

Reusable blocks are modules

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

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.

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.

Our 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.

Modules

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.

Definitions (aka setters)

Declare modules without a variable using the setter syntax. 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'
    ]);
```

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() { }
```

Setting vs. getting

Only set once and get for all other instances. A module should only be created once, then retrieved from that point and after.

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

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

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() { }
```

Factories

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.

Singletons

Factories are singletons and return an object that contains the members of the service. Note: All Angular services are singletons.

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 reads better.

```
/* 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() {
        /* */
    }
}
```

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.

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 declaration 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.

```
/* 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
    }

    return service;
}
```

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

Routing

Client-side routing is important for creating a navigation flow between views and composing views that are made of many smaller templates and directives.

Use the AngularUI router for client-side routing.

Why? UI router offers all the features of the Angular router plus a few additional ones including nested routes and states.

Why? The syntax is quite similar to the Angular router and is easy to migrate to UI router.

Note: You can use a provider such as the routerHelperProvider shown below to help configure states across files, during the run phase.

```
// customers.routes.js
angular
    .module('app.customers')
    .run(appRun);
```

```
/* @ngInject */
function appRun(routerHelper) {
    routerHelper.configureStates(getStates());
}

function getStates() {
    return [
        {
            state: 'customer',
            config: {
                abstract: true,
                template: '<ui-view class="shuffle-animation"/>',
                url: '/customer'
            }
        }
    ];
}
```

```
// routerHelperProvider.js
angular
    .module('blocks.router')
    .provider('routerHelper', routerHelperProvider);

routerHelperProvider.$inject =
    ['$locationProvider', '$stateProvider',
    '$urlRouterProvider'];
```

```
/* @ngInject */
function routerHelperProvider($locationProvider,
    $stateProvider, $urlRouterProvider) {
    /* jshint validthis:true */
    this.$get = RouterHelper;

    $locationProvider.html5Mode(true);

    RouterHelper.$inject = ['$state'];
    /* @ngInject */
    function RouterHelper($state) {
        var hasOtherwise = false;

        var service = {
            configureStates: configureStates,
            getStates: getStates
        };

        return service;

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

        function configureStates(states,
            otherwisePath) {
            states.forEach(function(state) {
                $stateProvider.state(state.state,
                    state.config);
            });
            if (otherwisePath && !hasOtherwise) {
                hasOtherwise = true;
                $urlRouterProvider.otherwise(
                    otherwisePath);
            }
        }

        function getStates() {
            return $state.get();
        }
    }
}
```

Define routes for views in the module where they exist. Each module should contain the routes for the views in it.

Why? Each module should be able to stand on its own.

Why? When removing a module or adding a module, the app will only contain routes that point to existing views.

Why? This makes it easy to enable or disable portions of an application without concern over orphaned routes.

Naming

Naming guidelines

Use consistent names for all components following a pattern that describes the component's feature then its (optional) 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 a project is vital. Consistency across a company is tremendously efficient.

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

Feature file names

Use consistent names for all components following a pattern that describes the component's feature then its type. Our recommended pattern is feature.type.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
```

Test file names

Name test specifications similar to the component they test with a suffix of spec. This will help to quickly identify components and 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
```

Controller names

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

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() { }
```

Controller name suffix

Append the controller name with the suffix Controller. This behaviour is more commonly used and more explicitly descriptive.

```
/* recommended */
// avengers.controller.js
angular
    .module
    .controller('AvengersController', AvengersController);

function AvengersController() { }
```

Factory and service names

Use consistent names for all factories and services named after their feature. Use camel-casing for services and factories. With $ Only suffix service and factories with $ prefix, while avoid suffixing services and factories that use the $ prefix.

Why? Clear service names such as logger do not require a suffix.

```
/* 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() { }
```

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 use company prefix or project prefix). This 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() { }
```

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. This provides consistency for multiple module apps, and for expanding to large applications. And it's an easy way to use task automation to load all module definitions first, then all other angular files (for bundling).

Configuration

Separate configuration for a module into its own file named after the module. A configuration file for the main app module app.config.js (or simply config.js). A configuration for a module named admin.module.js is named admin.config.js. This separates configuration resource definition, component, and active code and provides an identifiable place to set configuration for a module.

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

function logger() { }
```

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 we prefer this separation from the rest of the configuration.