NETE Software Design Style Guide, Best Practices and Patterns

JavaScript Technology Stack

Table of Contents

Client-Side: AngularJS Style Guide 5

Single Responsibility 5

Rule of 1 5

IIFE 5

JavaScript Closures 5

Modules 7

Avoid Naming Collisions 7

Definitions (aka Setters) 7

Getters 7

Setting vs Getting 7

Named vs Anonymous Functions 8

Controllers 8

controllerAs View Syntax 8

controllerAs Controller Syntax 9

controllerAs with vm 9

Bindable Members Up Top 10

Function Declarations to Hide Implementation Details 12

Defer Controller Logic to Services 13

Keep Controllers Focused 14

Assigning Controllers 14

Services 15

Singletons 15

Factories 16

Single Responsibility 16

Singletons 16

Accessible Members Up Top 16

Function Declarations to Hide Implementation Details 17

Data Services 19

Separate Data Calls 19

Return a Promise from Data Calls 20

Directives 21

Limit 1 Per File 21

Manipulate DOM in a Directive 22

Provide a Unique Directive Prefix 22

Restrict to Elements and Attributes 23

Directives and ControllerAs 24

Resolving Promises for a Controller 26

Controller Activation Promises 26

Route Resolve Promises 26

Manual Annotating for Dependency Injection 28

UnSafe from Minification 28

Manually Identify Dependencies 28

Manually Identify Route Resolver Dependencies 30

Minification and Annotation 31

ng-annotate 31

Use Gulp or Grunt for ng-annotate 32

Exception Handling 33

decorators 33

Exception Catchers 34

Route Errors 35

Naming 36

Naming Guidelines 36

Feature File Names 36

Test File Names 37

Controller Names 38

Controller Name Suffix 38

Factory Names 39

Directive Component Names 39

Modules 39

Configuration 40

Routes 40

Application Structure LIFT Principle 40

LIFT 40

Locate 41

Identify 41

Flat 41

T-DRY (Try to Stick to DRY) 42

Application Structure 42

Overall Guidelines 42

Layout 42

Folders-by-Feature Structure 43

Modularity 45

Many Small, Self Contained Modules 45

Create an App Module 45

Keep the App Module Thin 46

Feature Areas are Modules 46

Reusable Blocks are Modules 46

Module Dependencies 46

Startup Logic 48

Configuration 48

Run Blocks 48

Angular $ Wrapper Services 49

$document and $window 49

$timeout and $interval 49

Testing 49

Write Tests with Stories 49

Testing Library 50

Test Runner 50

Stubbing and Spying 50

Headless Browser 50

Code Analysis 51

Alleviate Globals for JSHint Rules on Tests 51

Organizing Tests 51

Animations 52

Usage 52

Sub Second 53

animate.css 53

Comments 53

jsDoc 53

JS Hint 54

Use an Options File 54

JSCS 56

Use an Options File 56

Constants 57

Vendor Globals 57

File Templates and Snippets 59

Sublime Text 59

Visual Studio 59

WebStorm 59

Atom 60

Brackets 60

vim 61

Yeoman Generator 61

Routing 61

Task Automation 63

Filters 64

Server-side: Node.js Best Practices 64

Always Use Asynchronous Methods 64

Never require Modules Inside of Functions 65

Save a reference to this Because it Changes Based on Context 66

Always "use strict" 67

Validate that Callbacks are Callable 67

Callbacks Always Pass Error Parameter First 68

Always Check for "error" in Callbacks 69

Use Exception Handling When Errors Can Be Thrown 69

Use module.exports not exports 70

Use JSDoc 71

Use a Process Manager like upstart, forever, or pm2 72

Follow CommonJS Standard 72

References 72

# Client-Side: AngularJS Style Guide

## Single Responsibility

### Rule of 1

[Style [Y001](#style-y001)]

* 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.

```javascript /\* 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.

```javascript /\* recommended \*/

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

```javascript /\* recommended \*/

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

function SomeController() { } ```

```javascript /\* recommended \*/

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

function someFactory() { } ```

[**Back to top**](#table-of-contents)

## IIFE

### JavaScript Closures

[Style [Y010](#style-y010)]

* 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.

```javascript /\* 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() { } ```

```javascript /\*\* \* 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.

[**Back to top**](#table-of-contents)

## Modules

### Avoid Naming Collisions

[Style [Y020](#style-y020)]

* 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)

[Style [Y021](#style-y021)]

* 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.

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

Instead use the simple setter syntax.

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

### Getters

[Style [Y022](#style-y022)]

* 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.

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

function SomeController() { } ```

```javascript /\* recommended \*/ angular .module('app') .controller('SomeController', SomeController);

function SomeController() { } ```

### Setting vs Getting

[Style [Y023](#style-y023)]

* Only set once and get for all other instances.

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

- Use `angular.module('app', []);` to set a module.  
- Use `angular.module('app');` to get a module.

### Named vs Anonymous Functions

[Style [Y024](#style-y024)]

* 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.

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

```javascript /\* recommended \*/

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

function Dashboard() { } ```

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

function logger() { } ```

[**Back to top**](#table-of-contents)

## Controllers

### controllerAs View Syntax

[Style [Y030](#style-y030)]

* Use the [controllerAs](http://www.johnpapa.net/do-you-like-your-angular-controllers-with-or-without-sugar/) 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.

html <!-- avoid --> <div ng-controller="Customer"> {{ name }} </div>

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

### controllerAs Controller Syntax

[Style [Y031](#style-y031)]

* 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 them to a factory. Consider using $scope in a factory, or if in a controller just when needed. For example when publishing and subscribing events using [$emit](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$emit), [$broadcast](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$broadcast), or [$on](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on) consider moving these uses to a factory and invoke from the controller.

javascript /\* avoid \*/ function Customer($scope) { $scope.name = {}; $scope.sendMessage = function() { }; }

javascript /\* recommended - but see next section \*/ function Customer() { this.name = {}; this.sendMessage = function() { }; }

### controllerAs with vm

[Style [Y032](#style-y032)]

* 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.

javascript /\* avoid \*/ function Customer() { this.name = {}; this.sendMessage = function() { }; }

javascript /\* recommended \*/ function Customer() { var vm = this; vm.name = {}; vm.sendMessage = function() { }; }

Note: You can avoid any [jshint](http://www.jshint.com/) 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.

javascript /\* 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.)

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

```javascript 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

[Style [Y033](#style-y033)]

* 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.

```javascript /\* avoid \*/ function Sessions() { var vm = this;

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

```

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

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

```

![Controller Using "Above the Fold"](https://raw.githubusercontent.com/johnpapa/angular-styleguide/master/assets/above-the-fold-1.png)

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

```javascript /\* avoid \*/ function Sessions(data) { var vm = this;

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

```

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

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

```

### Function Declarations to Hide Implementation Details

[Style [Y034](#style-y034)]

* 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](http://www.johnpapa.net/angular-function-declarations-function-expressions-and-readable-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. (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

```javascript /\*\* \* avoid \* Using function expressions. \*/ function Avengers(dataservice, 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 dataservice.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.

```javascript / recommend \* Using function declarations \* and bindable members up top. \*/ function Avengers(dataservice, 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 dataservice.getAvengers().then(function(data) {  
 vm.avengers = data;  
 return vm.avengers;  
 });  
 }

} ```

### Defer Controller Logic to Services

[Style [Y035](#style-y035)]

* 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.

```javascript

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

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

} ```

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

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

} ```

### Keep Controllers Focused

[Style [Y037](#style-y037)]

* 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.

### Assigning Controllers

[Style [Y038](#style-y038)]

* 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](https://docs.angularjs.org/api/ng/directive/ngController), that view is always associated with the same controller.

```javascript /\* 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' }); } ```

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

```javascript /\* recommended \*/

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

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

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

[**Back to top**](#table-of-contents)

## Services

### Singletons

[Style [Y040](#style-y040)]

* 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](https://docs.angularjs.org/guide/services). This means that there is only one instance of a given service per injector.

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

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

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

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

[**Back to top**](#table-of-contents)

## Factories

### Single Responsibility

[Style [Y050](#style-y050)]

* Factories should have a [single responsibility](http://en.wikipedia.org/wiki/Single_responsibility_principle), that is encapsulated by its context. Once a factory begins to exceed that singular purpose, a new factory should be created.

### Singletons

[Style [Y051](#style-y051)]

* Factories are singletons and return an object that contains the members of the service.
* Note: [All Angular services are singletons](https://docs.angularjs.org/guide/services).

### Accessible Members Up Top

[Style [Y052](#style-y052)]

* Expose the callable members of the service (its interface) at the top, using a technique derived from the [Revealing Module Pattern](http://addyosmani.com/resources/essentialjsdesignpatterns/book/#revealingmodulepatternjavascript).
* *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.

```javascript /\* avoid */ function dataService() { var someValue = ''; function save() { /* */ }; function validate() { /* \*/ };

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

} ```

```javascript /\* 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.

![Factories Using "Above the Fold"](https://raw.githubusercontent.com/johnpapa/angular-styleguide/master/assets/above-the-fold-2.png)

### Function Declarations to Hide Implementation Details

[Style [Y053](#style-y053)]

* 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](http://www.johnpapa.net/angular-function-declarations-function-expressions-and-readable-code).
* *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.
* *Why?*: Order is critical with function expressions

```javascript /\*\* \* 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;

} ```

```javascript /\*\* \* 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  
 }

} ```

[**Back to top**](#table-of-contents)

## Data Services

### Separate Data Calls

[Style [Y060](#style-y060)]

* 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.

```javascript /\* recommended \*/

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

dataservice.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.

```javascript /\* recommended \*/

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

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

function Avengers(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;  
 });  
 }

} ```

### Return a Promise from Data Calls

[Style [Y061](#style-y061)]

* 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.

```javascript /\* 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; }); } ```

[**Back to top**](#table-of-contents)

## Directives

### Limit 1 Per File

[Style [Y070](#style-y070)]

* 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.

```javascript /\* 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 \*/ } ```

```javascript /\* recommended */ /* calendarRange.directive.js \*/

/\*\* \* @desc order directive that is specific to the order module at a company named Acme \* @example

\*/ angular .module('sales.order') .directive('acmeOrderCalendarRange', orderCalendarRange);

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

```javascript /\* recommended */ /* customerInfo.directive.js \*/

/\*\* \* @desc sales directive that can be used anywhere across the sales app at a company named Acme \* @example

\*/ angular .module('sales.widgets') .directive('acmeSalesCustomerInfo', salesCustomerInfo);

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

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

/\*\* \* @desc spinner directive that can be used anywhere across apps at a company named Acme \* @example

\*/ 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 [Naming](#naming) section for more recommendations.

### Manipulate DOM in a Directive

[Style [Y072](#style-y072)]

* 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](https://docs.angularjs.org/api/ngAnimate), Angular templating, [ngShow](https://docs.angularjs.org/api/ng/directive/ngShow) or [ngHide](https://docs.angularjs.org/api/ng/directive/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

[Style [Y073](#style-y073)]

* 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](http://ionicframework.com/).

### Restrict to Elements and Attributes

[Style [Y074](#style-y074)]

* 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 +

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

```javascript /\* 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) {  
 /\* \*/  
 }

} ```

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

```javascript /\* 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

[Style [Y075](#style-y075)]

* 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](#manual-annotating-for-dependency-injection).
* 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.

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

```javascript 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 // 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.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);

} ```

html <!-- 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.

javascript // 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); }

[Style [Y076](#style-y076)]

* 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.

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

```javascript 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); } ```

html <!-- 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>

[**Back to top**](#table-of-contents)

## Resolving Promises for a Controller

### Controller Activation Promises

[Style [Y080](#style-y080)]

* 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 use the controller, use a [route resolve](#style-y081) instead.

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

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

} ```

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

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

} ```

### Route Resolve Promises

[Style [Y081](#style-y081)]

* 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](https://docs.angularjs.org/api/ng/service/$http). Using a [route resolve](https://docs.angularjs.org/api/ngRoute/provider/$routeProvider) 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](#style-y080) instead.

```javascript /\* avoid \*/ angular .module('app') .controller('Avengers', Avengers);

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

```javascript /\* better \*/

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

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

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

Avengers.$inject = ['moviesPrepService']; function Avengers(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.

```javascript /\* even better \*/

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

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

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

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

Avengers.$inject = ['moviesPrepService']; function Avengers(moviesPrepService) { var vm = this; vm.movies = moviesPrepService.movies; } ``Note: The code example's dependency onmovieService` is not minification safe on its own. For details on how to make this code minification safe, see the sections on [dependency injection](#manual-annotating-for-dependency-injection) and on [minification and annotation](#minification-and-annotation).

[**Back to top**](#table-of-contents)

## Manual Annotating for Dependency Injection

### UnSafe from Minification

[Style [Y090](#style-y090)]

* 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('Dashboard', Dashboard);  
    
  function Dashboard(common, dataservice) {  
  }
* This code may produce mangled variables when minified and thus cause runtime errors.
* /\* avoid - not minification-safe\*/  
  angular.module('app').controller('Dashboard', d);function d(a, b) { }

### Manually Identify Dependencies

[Style [Y091](#style-y091)]

* Use $inject to manually identify your dependencies for Angular components.
* *Why?*: This technique mirrors the technique used by [ng-annotate](https://github.com/olov/ng-annotate), which I recommend for automating the creation of minification safe dependencies. If ng-annotate detects injection has already been made, it will not duplicate it.
* *Why?*: This safeguards your dependencies from being vulnerable to minification issues when parameters may be mangled. For example, common and dataservice may become a or b and not be found by Angular.
* *Why?*: Avoid creating in-line dependencies as long lists can be difficult to read in the array. Also it can be confusing that the array is a series of strings while the last item is the component's function.
* /\* avoid \*/  
  angular  
   .module('app')  
   .controller('Dashboard',  
   ['$location', '$routeParams', 'common', 'dataservice',  
   function Dashboard($location, $routeParams, common, dataservice) {}  
   ]);
* /\* avoid \*/  
  angular  
   .module('app')  
   .controller('Dashboard',  
   ['$location', '$routeParams', 'common', 'dataservice', Dashboard]);  
    
  function Dashboard($location, $routeParams, common, dataservice) {  
  }
* /\* recommended \*/  
  angular  
   .module('app')  
   .controller('Dashboard', Dashboard);  
    
  Dashboard.$inject = ['$location', '$routeParams', 'common', 'dataservice'];  
    
  function Dashboard($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) {  
  }

### Manually Identify Route Resolver Dependencies

[Style [Y092](#style-y092)]

* 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();  
  }

[**Back to top**](#table-of-contents)

## Minification and Annotation

### ng-annotate

[Style [Y100](#style-y100)]

* Use [ng-annotate](file://localhost//github.com/olov/ng-annotate) for [Gulp](http://gulpjs.com) or [Grunt](http://gruntjs.com) 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](https://github.com/btford/ngmin) 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('Avengers', Avengers);  
    
  /\* @ngInject \*/  
  function Avengers(storageService, avengerService) {  
   var vm = this;  
   vm.heroSearch = '';  
   vm.storeHero = storeHero;  
    
   function storeHero() {  
   var hero = avengerService.find(vm.heroSearch);  
   storageService.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('Avengers', Avengers);  
    
  /\* @ngInject \*/  
  function Avengers(storageService, avengerService) {  
   var vm = this;  
   vm.heroSearch = '';  
   vm.storeHero = storeHero;  
    
   function storeHero() {  
   var hero = avengerService.find(vm.heroSearch);  
   storageService.save(hero.name, hero);  
   }  
  }  
    
  Avengers.$inject = ['storageService', '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: 'Avengers',  
   controllerAs: 'vm',  
   resolve: { /\* @ngInject \*/  
   moviesPrepService: function(movieService) {  
   return movieService.getMovies();  
   }  
   }  
   });  
  }
* Note: Starting from Angular 1.3 you can use the [ngApp](https://docs.angularjs.org/api/ng/directive/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>

### Use Gulp or Grunt for ng-annotate

[Style [Y101](#style-y101)]

* Use [gulp-ng-annotate](https://www.npmjs.org/package/gulp-ng-annotate) or [grunt-ng-annotate](https://www.npmjs.org/package/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));  
  });

[**Back to top**](#table-of-contents)

## Exception Handling

### decorators

[Style [Y110](#style-y110)]

* Use a [decorator](https://docs.angularjs.org/api/auto/service/$provide#decorator), at config time using the [$provide](https://docs.angularjs.org/api/auto/service/$provide) service, on the [$exceptionHandler](https://docs.angularjs.org/api/ng/service/$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);  
   };  
  }

### Exception Catchers

[Style [Y111](#style-y111)]

* 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);  
   };  
   }  
  }

### Route Errors

[Style [Y112](#style-y112)]

* Handle and log all routing errors using [$routeChangeError](https://docs.angularjs.org/api/ngRoute/service/$route#$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('/');  
    
   }  
   );  
  }

[**Back to top**](#table-of-contents)

## Naming

### Naming Guidelines

[Style [Y120](#style-y120)]

* 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.

### Feature File Names

[Style [Y121](#style-y121)]

* 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.

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

### Test File Names

[Style [Y122](#style-y122)]

* 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](http://karma-runner.github.io/) or other test runners.
* /\*\*  
   \* recommended  
   \*/  
  avengers.controller.spec.js  
  logger.service.spec.js  
  avengers.routes.spec.js  
  avenger-profile.directive.spec.js

### Controller Names

[Style [Y123](#style-y123)]

* 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() { }

### Controller Name Suffix

[Style [Y124](#style-y124)]

* 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() { }

### Factory Names

[Style [Y125](#style-y125)]

* Use consistent names for all factories named after their feature. Use camel-casing for services and factories. Avoid prefixing factories and services with $.
* *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.
* /\*\*  
   \* recommended  
   \*/  
    
  // logger.service.js  
  angular  
   .module  
   .factory('logger', logger);  
    
  function logger() { }

### Directive Component Names

[Style [Y126](#style-y126)]

* 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() { }

### Modules

[Style [Y127](#style-y127)]

* 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).

### Configuration

[Style [Y128](#style-y128)]

* 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.

### Routes

[Style [Y129](#style-y129)]

* 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.

[**Back to top**](#table-of-contents)

## Application Structure LIFT Principle

### LIFT

[Style [Y140](#style-y140)]

* 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
  1. Locating our code is easy
  2. Identify code at a glance
  3. Flat structure as long as we can
  4. Try to stay DRY (Don’t Repeat Yourself) or T-DRY

### Locate

[Style [Y141](#style-y141)]

* 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

### Identify

[Style [Y142](#style-y142)]

* 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.

### Flat

[Style [Y143](#style-y143)]

* 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.

### T-DRY (Try to Stick to DRY)

[Style [Y144](#style-y144)]

* 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.

[**Back to top**](#table-of-contents)

## Application Structure

### Overall Guidelines

[Style [Y150](#style-y150)]

* 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](http://www.johnpapa.net/angular-app-structuring-guidelines/).

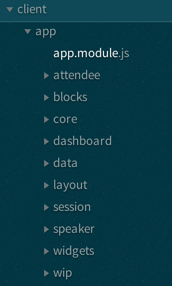
### Layout

[Style [Y151](#style-y151)]

* 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.

### Folders-by-Feature Structure

[Style [Y152](#style-y152)]

* 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
* 
* Sample App Structure
* 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

[**Back to top**](#table-of-contents)

## Modularity

### Many Small, Self Contained Modules

[Style [Y160](#style-y160)]

* 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

[Style [Y161](#style-y161)]

* Create an application root 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

[Style [Y162](#style-y162)]

* 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

[Style [Y163](#style-y163)]

* 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.

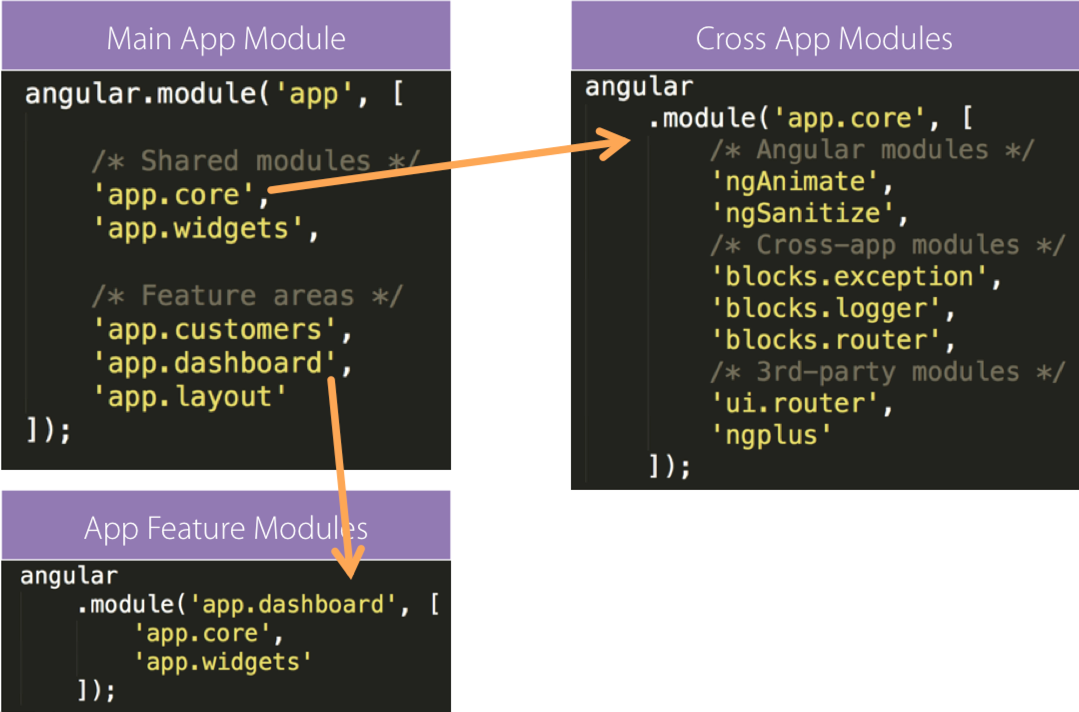
### Reusable Blocks are Modules

[Style [Y164](#style-y164)]

* 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.

### Module Dependencies

[Style [Y165](#style-y165)]

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

[**Back to top**](#table-of-contents)

## Startup Logic

### Configuration

[Style [Y170](#style-y170)]

* Inject code into [module configuration](https://docs.angularjs.org/guide/module#module-loading-dependencies) that must be configured before running the angular app. Ideal candidates include providers and constants.
* *Why?*: This makes it easier to have a less places for configuration.

```javascript 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: '  
 });  
 }

} ```

### Run Blocks

[Style [Y171](#style-y171)]

* 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](https://docs.angularjs.org/guide/module#module-loading-dependencies).
* *Why?*: Code directly in a run block can be difficult to test. Placing in a factory makes it easier to abstract and mock.

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

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

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

[**Back to top**](#table-of-contents)

## Angular $ Wrapper Services

### $document and $window

[Style [Y180](#style-y180)]

* Use [$document](https://docs.angularjs.org/api/ng/service/$document) and [$window](https://docs.angularjs.org/api/ng/service/$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.

### $timeout and $interval

[Style [Y181](#style-y181)]

* Use [$timeout](https://docs.angularjs.org/api/ng/service/$timeout) and [$interval](https://docs.angularjs.org/api/ng/service/$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.

[**Back to top**](#table-of-contents)

## Testing

Unit testing helps maintain clean code, as such I included some of my recommendations for unit testing foundations with links for more information.

### Write Tests with Stories

[Style [Y190](#style-y190)]

* Write a set of tests for every story. Start with an empty test and fill them in as you write the code for the story.
* *Why?*: Writing the test descriptions helps clearly define what your story will do, will not do, and how you can measure success.
* it('should have Avengers controller', function() {  
   // TODO  
  });  
    
  it('should find 1 Avenger when filtered by name', function() {  
   // TODO  
  });  
    
  it('should have 10 Avengers', function() {  
   // TODO (mock data?)  
  });  
    
  it('should return Avengers via XHR', function() {  
   // TODO ($httpBackend?)  
  });  
    
  // and so on

### Testing Library

[Style [Y191](#style-y191)]

* Use [Jasmine](http://jasmine.github.io/) or [Mocha](http://mochajs.org) for unit testing.
* *Why?*: Both Jasmine and Mocha are widely used in the Angular community. Both are stable, well maintained, and provide robust testing features.
* Note: When using Mocha, also consider choosing an assert library such as [Chai](http://chaijs.com). I prefer Mocha.

### Test Runner

[Style [Y192](#style-y192)]

* Use [Karma](http://karma-runner.github.io) as a test runner.
* *Why?*: Karma is easy to configure to run once or automatically when you change your code.
* *Why?*: Karma hooks into your Continuous Integration process easily on its own or through Grunt or Gulp.
* *Why?*: Some IDE's are beginning to integrate with Karma, such as [WebStorm](http://www.jetbrains.com/webstorm/) and [Visual Studio](http://visualstudiogallery.msdn.microsoft.com/02f47876-0e7a-4f6c-93f8-1af5d5189225).
* *Why?*: Karma works well with task automation leaders such as [Grunt](http://www.gruntjs.com) (with [grunt-karma](https://github.com/karma-runner/grunt-karma)) and [Gulp](http://www.gulpjs.com) (with [gulp-karma](https://github.com/lazd/gulp-karma)).

### Stubbing and Spying

[Style [Y193](#style-y193)]

* Use [Sinon](http://sinonjs.org/) for stubbing and spying.
* *Why?*: Sinon works well with both Jasmine and Mocha and extends the stubbing and spying features they offer.
* *Why?*: Sinon makes it easier to toggle between Jasmine and Mocha, if you want to try both.
* *Why?*: Sinon has descriptive messages when tests fail the assertions.

### Headless Browser

[Style [Y194](#style-y194)]

* Use [PhantomJS](http://phantomjs.org/) to run your tests on a server.
* *Why?*: PhantomJS is a headless browser that helps run your tests without needing a "visual" browser. So you do not have to install Chrome, Safari, IE, or other browsers on your server.
* Note: You should still test on all browsers in your environment, as appropriate for your target audience.

### Code Analysis

[Style [Y195](#style-y195)]

* Run JSHint on your tests.
* *Why?*: Tests are code. JSHint can help identify code quality issues that may cause the test to work improperly.

### Alleviate Globals for JSHint Rules on Tests

[Style [Y196](#style-y196)]

* Relax the rules on your test code to allow for common globals such as describe and expect. Relax the rules for expressions, as Mocha uses these.
* *Why?*: Your tests are code and require the same attention and code quality rules as all of your production code. However, global variables used by the testing framework, for example, can be relaxed by including this in your test specs.
* /\* jshint -W117, -W030 \*/
* Or you can add the following to your JSHint Options file.
* "jasmine": true,  
  "mocha": true,



Testing Tools

### Organizing Tests

[Style [Y197](#style-y197)]

* Place unit test files (specs) side-by-side with your client code. Place specs that cover server integration or test multiple components in a separate tests folder.
* *Why?*: Unit tests have a direct correlation to a specific component and file in source code.
* *Why?*: It is easier to keep them up to date since they are always in sight. When coding whether you do TDD or test during development or test after development, the specs are side-by-side and never out of sight nor mind, and thus more likely to be maintained which also helps maintain code coverage.
* *Why?*: When you update source code it is easier to go update the tests at the same time.
* *Why?*: Placing them side-by-side makes it easy to find them and easy to move them with the source code if you move the source.
* *Why?*: Having the spec nearby makes it easier for the source code reader to learn how the component is supposed to be used and to discover its known limitations.
* *Why?*: Separating specs so they are not in a distributed build is easy with grunt or gulp.
* /src/client/app/customers/customer-detail.controller.js  
   /customer-detail.controller.spec.js  
   /customers.controller.js  
   /customers.controller.spec.js  
   /customers.module.js  
   /customers.route.js  
   /customers.route.spec.js

[**Back to top**](#table-of-contents)

## Animations

### Usage

[Style [Y210](#style-y210)]

* Use subtle [animations with Angular](https://docs.angularjs.org/guide/animations) to transition between states for views and primary visual elements. Include the [ngAnimate module](https://docs.angularjs.org/api/ngAnimate). The 3 keys are subtle, smooth, seamless.
* *Why?*: Subtle animations can improve User Experience when used appropriately.
* *Why?*: Subtle animations can improve perceived performance as views transition.

### Sub Second

[Style [Y211](#style-y211)]

* Use short durations for animations. I generally start with 300ms and adjust until appropriate.
* *Why?*: Long animations can have the reverse affect on User Experience and perceived performance by giving the appearance of a slow application.

### animate.css

[Style [Y212](#style-y212)]

* Use [animate.css](http://daneden.github.io/animate.css/) for conventional animations.
* *Why?*: The animations that animate.css provides are fast, smooth, and easy to add to your application.
* *Why?*: Provides consistency in your animations.
* *Why?*: animate.css is widely used and tested.
* Note: See this [great post by Matias Niemelä on Angular animations](http://www.yearofmoo.com/2013/08/remastered-animation-in-angularjs-1-2.html)

[**Back to top**](#table-of-contents)

## Comments

### jsDoc

[Style [Y220](#style-y220)]

* If planning to produce documentation, use [jsDoc](http://usejsdoc.org/) syntax to document function names, description, params and returns. Use @namespace and @memberOf to match your app structure.
* *Why?*: You can generate (and regenerate) documentation from your code, instead of writing it from scratch.
* *Why?*: Provides consistency using a common industry tool.
* /\*\*  
   \* Logger Factory  
   \* @namespace Factories  
   \*/  
  (function() {  
   angular  
   .module('app')  
   .factory('logger', logger);  
    
   /\*\*  
   \* @namespace Logger  
   \* @desc Application wide logger  
   \* @memberOf Factories  
   \*/  
   function logger($log) {  
   var service = {  
   logError: logError  
   };  
   return service;  
    
   ////////////  
    
   /\*\*  
   \* @name logError  
   \* @desc Logs errors  
   \* @param {String} msg Message to log  
   \* @returns {String}  
   \* @memberOf Factories.Logger  
   \*/  
   function logError(msg) {  
   var loggedMsg = 'Error: ' + msg;  
   $log.error(loggedMsg);  
   return loggedMsg;  
   };  
   }  
  })();

[**Back to top**](#table-of-contents)

## JS Hint

### Use an Options File

[Style [Y230](#style-y230)]

* Use JS Hint for linting your JavaScript and be sure to customize the JS Hint options file and include in source control. See the [JS Hint docs](http://www.jshint.com/docs/) for details on the options.
* *Why?*: Provides a first alert prior to committing any code to source control.
* *Why?*: Provides consistency across your team.
* {  
   "bitwise": true,  
   "camelcase": true,  
   "curly": true,  
   "eqeqeq": true,  
   "es3": false,  
   "forin": true,  
   "freeze": true,  
   "immed": true,  
   "indent": 4,  
   "latedef": "nofunc",  
   "newcap": true,  
   "noarg": true,  
   "noempty": true,  
   "nonbsp": true,  
   "nonew": true,  
   "plusplus": false,  
   "quotmark": "single",  
   "undef": true,  
   "unused": false,  
   "strict": false,  
   "maxparams": 10,  
   "maxdepth": 5,  
   "maxstatements": 40,  
   "maxcomplexity": 8,  
   "maxlen": 120,  
    
   "asi": false,  
   "boss": false,  
   "debug": false,  
   "eqnull": true,  
   "esnext": false,  
   "evil": false,  
   "expr": false,  
   "funcscope": false,  
   "globalstrict": false,  
   "iterator": false,  
   "lastsemic": false,  
   "laxbreak": false,  
   "laxcomma": false,  
   "loopfunc": true,  
   "maxerr": false,  
   "moz": false,  
   "multistr": false,  
   "notypeof": false,  
   "proto": false,  
   "scripturl": false,  
   "shadow": false,  
   "sub": true,  
   "supernew": false,  
   "validthis": false,  
   "noyield": false,  
    
   "browser": true,  
   "node": true,  
    
   "globals": {  
   "angular": false,  
   "$": false  
   }  
  }

[**Back to top**](#table-of-contents)

## JSCS

### Use an Options File

[Style [Y235](#style-y235)]

* Use JSCS for checking your coding styles your JavaScript and be sure to customize the JSCS options file and include in source control. See the [JSCS docs](http://www.jscs.info) for details on the options.
* *Why?*: Provides a first alert prior to committing any code to source control.
* *Why?*: Provides consistency across your team.
* {  
   "excludeFiles": ["node\_modules/\*\*", "bower\_components/\*\*"],  
    
   "requireCurlyBraces": [  
   "if",  
   "else",  
   "for",  
   "while",  
   "do",  
   "try",  
   "catch"  
   ],  
   "requireOperatorBeforeLineBreak": true,  
   "requireCamelCaseOrUpperCaseIdentifiers": true,  
   "maximumLineLength": {  
   "value": 100,  
   "allowComments": true,  
   "allowRegex": true  
   },  
   "validateIndentation": 4,  
   "validateQuoteMarks": "'",  
    
   "disallowMultipleLineStrings": true,  
   "disallowMixedSpacesAndTabs": true,  
   "disallowTrailingWhitespace": true,  
   "disallowSpaceAfterPrefixUnaryOperators": true,  
   "disallowMultipleVarDecl": null,  
    
   "requireSpaceAfterKeywords": [  
   "if",  
   "else",  
   "for",  
   "while",  
   "do",  
   "switch",  
   "return",  
   "try",  
   "catch"  
   ],  
   "requireSpaceBeforeBinaryOperators": [  
   "=", "+=", "-=", "\*=", "/=", "%=", "<<=", ">>=", ">>>=",  
   "&=", "|=", "^=", "+=",  
    
   "+", "-", "\*", "/", "%", "<<", ">>", ">>>", "&",  
   "|", "^", "&&", "||", "===", "==", ">=",  
   "<=", "<", ">", "!=", "!=="  
   ],  
   "requireSpaceAfterBinaryOperators": true,  
   "requireSpacesInConditionalExpression": true,  
   "requireSpaceBeforeBlockStatements": true,  
   "requireLineFeedAtFileEnd": true,  
   "disallowSpacesInsideObjectBrackets": "all",  
   "disallowSpacesInsideArrayBrackets": "all",  
   "disallowSpacesInsideParentheses": true,  
    
   "validateJSDoc": {  
   "checkParamNames": true,  
   "requireParamTypes": true  
   },  
    
   "disallowMultipleLineBreaks": true,  
    
   "disallowCommaBeforeLineBreak": null,  
   "disallowDanglingUnderscores": null,  
   "disallowEmptyBlocks": null,  
   "disallowTrailingComma": null,  
   "requireCommaBeforeLineBreak": null,  
   "requireDotNotation": null,  
   "requireMultipleVarDecl": null,  
   "requireParenthesesAroundIIFE": true  
  }

[**Back to top**](#table-of-contents)

## Constants

### Vendor Globals

[Style [Y240](#style-y240)]

* Create an Angular Constant for vendor libraries' global variables.
* *Why?*: Provides a way to inject vendor libraries that otherwise are globals. This improves code testability by allowing you to more easily know what the dependencies of your components are (avoids leaky abstractions). It also allows you to mock these dependencies, where it makes sense.
* // constants.js  
    
  /\* global toastr:false, moment:false \*/  
  (function() {  
   'use strict';  
    
   angular  
   .module('app.core')  
   .constant('toastr', toastr)  
   .constant('moment', moment);  
  })();

[Style [Y241](#style-y241)]

* Use constants for values that do not change and do not come from another service. When constants are used only for a module that may be reused in multiple applications, place constants in a file per module named after the module. Until this is required, keep constants in the main module in a constants.js file.
* *Why?*: A value that may change, even infrequently, should be retrieved from a service so you do not have to change the source code. For example, a url for a data service could be placed in a constants but a better place would be to load it from a web service.
* *Why?*: Constants can be injected into any angular component, including providers.
* *Why?*: When an application is separated into modules that may be reused in other applications, each stand-alone module should be able to operate on its own including any dependent constants.
* // Constants used by the entire app  
  angular  
   .module('app.core')  
   .constant('moment', moment);  
    
  // Constants used only by the sales module  
  angular  
   .module('app.sales')  
   .constant('events', {  
   ORDER\_CREATED: 'event\_order\_created',  
   INVENTORY\_DEPLETED: 'event\_inventory\_depleted'  
   });

[**Back to top**](#table-of-contents)

## File Templates and Snippets

Use file templates or snippets to help follow consistent styles and patterns. Here are templates and/or snippets for some of the web development editors and IDEs.

### Sublime Text

[Style [Y250](#style-y250)]

* Angular snippets that follow these styles and guidelines.
  + Download the [Sublime Angular snippets](assets/sublime-angular-snippets?raw=true)
  + Place it in your Packages folder
  + Restart Sublime
  + In a JavaScript file type these commands followed by a TAB
* ngcontroller // creates an Angular controller  
  ngdirective // creates an Angular directive  
  ngfactory // creates an Angular factory  
  ngmodule // creates an Angular module  
  ngservice // creates an Angular service  
  ngfilter // creates an Angular filter

### Visual Studio

[Style [Y251](#style-y251)]

* Angular file templates that follow these styles and guidelines can be found at [SideWaffle](http://www.sidewaffle.com)
  + Download the [SideWaffle](http://www.sidewaffle.com) Visual Studio extension (vsix file)
  + Run the vsix file
  + Restart Visual Studio

### WebStorm

[Style [Y252](#style-y252)]

* Angular snippets and file templates that follow these styles and guidelines. You can import them into your WebStorm settings:
  + Download the [WebStorm Angular file templates and snippets](assets/webstorm-angular-file-template.settings.jar?raw=true)
  + Open WebStorm and go to the File menu
  + Choose the Import Settings menu option
  + Select the file and click OK
  + In a JavaScript file type these commands followed by a TAB:
* ng-c // creates an Angular controller  
  ng-f // creates an Angular factory  
  ng-m // creates an Angular module

### Atom

[Style [Y253](#style-y253)]

* Angular snippets that follow these styles and guidelines.
* apm install angularjs-styleguide-snippets
* or
  + Open Atom, then open the Package Manager (Packages -> Settings View -> Install Packages/Themes)
  + Search for the package 'angularjs-styleguide-snippets'
  + Click 'Install' to install the package
* In a JavaScript file type these commands followed by a TAB
* ngcontroller // creates an Angular controller  
  ngdirective // creates an Angular directive  
  ngfactory // creates an Angular factory  
  ngmodule // creates an Angular module  
  ngservice // creates an Angular service  
  ngfilter // creates an Angular filter

### Brackets

[Style [Y254](#style-y254)]

* Angular snippets that follow these styles and guidelines.
  + Download the [Brackets Angular snippets](assets/brackets-angular-snippets.yaml?raw=true)
  + Brackets Extension manager ( File > Extension manager )
  + Install ['Brackets Snippets (by edc)'](https://github.com/chuyik/brackets-snippets)
  + Click the light bulb in brackets' right gutter
  + Click Settings and then Import
  + Choose the file and select to skip or override
  + Click Start Import
* In a JavaScript file type these commands followed by a TAB
* // These are full file snippets containing an IIFE  
  ngcontroller // creates an Angular controller  
  ngdirective // creates an Angular directive  
  ngfactory // creates an Angular factory  
  ngapp // creates an Angular module setter  
  ngservice // creates an Angular service  
  ngfilter // creates an Angular filter  
    
  // These are partial snippets intended to chained  
  ngmodule // creates an Angular module getter  
  ngstate // creates an Angular UI Router state defintion  
  ngconfig // defines a configuration phase function  
  ngrun // defines a run phase function  
  ngroute // defines an Angular ngRoute 'when' definition  
  ngtranslate // uses $translate service with its promise

### vim

[Style [Y255](#style-y255)]

* vim snippets that follow these styles and guidelines.
  + Download the [vim Angular snippets](assets/vim-angular-snippets?raw=true)
  + set [neosnippet.vim](https://github.com/Shougo/neosnippet.vim)
  + copy snippets to snippet directory
* ngcontroller // creates an Angular controller  
  ngdirective // creates an Angular directive  
  ngfactory // creates an Angular factory  
  ngmodule // creates an Angular module  
  ngservice // creates an Angular service  
  ngfilter // creates an Angular filter
* [**Back to top**](#table-of-contents)

## Yeoman Generator

[Style [Y260](#style-y260)]

You can use the [HotTowel yeoman generator](http://jpapa.me/yohottowel) to create an app that serves as a starting point for Angular that follows this style guide.

1. Install generator-hottowel

npm install -g generator-hottowel

1. Create a new folder and change directory to it

mkdir myapp cd myapp

1. Run the generator

yo hottowel helloWorld

[**Back to top**](#table-of-contents)

## 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.

[Style [Y270](#style-y270)]

* Use the [AngularUI Router](http://angular-ui.github.io/ui-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(); }  
   }  
  }

[Style [Y271](#style-y271)]

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

[**Back to top**](#table-of-contents)

## Task Automation

Use [Gulp](http://gulpjs.com) or [Grunt](http://gruntjs.com) for creating automated tasks. Gulp leans to code over configuration while Grunt leans to configuration over code. I personally prefer Gulp as I feel it is easier to read and write, but both are excellent.

Learn more about gulp and patterns for task automation in my [Gulp Pluralsight course](http://jpapa.me/gulpps)

[Style [Y400](#style-y400)]

* Use task automation to list module definition files \*.module.js before all other application JavaScript files.
* *Why?*: Angular needs the module definitions to be registered before they are used.
* *Why?*: Naming modules with a specific pattern such as \*.module.js makes it easy to grab them with a glob and list them first.
* var clientApp = './src/client/app/';  
    
  // Always grab module files first  
  var files = [  
   clientApp + '\*\*/\*.module.js',  
   clientApp + '\*\*/\*.js'  
  ];

[**Back to top**](#table-of-contents)

## Filters

[Style [Y420](#style-y420)]

* Avoid using filters for scanning all properties of a complex object graph. Use filters for select properties.
* *Why?*: Filters can easily be abused and negatively affect performance if not used wisely, for example when a filter hits a large and deep object graph.

# Server-side: Node.js Best Practices

## Always Use Asynchronous Methods

The two most powerful aspect of Node.js are it's non-blocking IO and asynchronous runtime. Both of these aspects of Node.js are what give it the speed and robustness to serve more requests faster than other languages.

In order to take advantage of these features you have to always use asynchronous methods in your code. Below is an example showing the good and bad way to read files from a system.

The Bad Way reads a file from disk synchronously.

var data = fs.readFileSync('/path/to/file');  
console.log(data);  
// use the data from the file

The Good Way reads a file from disk asynchronously.

fs.readFile('/path/to/file', function (err, data) {  
 // err will be an error object if an error occurred  
 // data will be the file that was read  
 console.log(data);  
});

When a synchronous function is invoked the entire runtime halts. For example, above The Bad Way halts the execution of any other code that could be running while the file is read into memory. This means no users get served during this time. If your file takes five minutes to read into memory no users get served for five minutes.

By contrast The Good Way reads the file into memory without halting the runtime by using an asynchronous method. This means that if your file takes five minutes to read into memory all your users continue to get served.

## Never require Modules Inside of Functions

As stated above you should always use asynchronous methods and function calls in Node. The one exception is the require function, which imports external modules.

Node.js always runs require synchronously. This is so the module being required can require other needed modules. The Node.js developers realize that importing modules is an expensive process and so they intend for it to happen only once, when you start your Node.js application. They even cache the required modules so they won't be required again.

However, if you require an external module from within functions your module will be synchronously loaded when those functions, not when your application starts. This can cause two problems.

To explain one of the problems imagine you had a module that took 30 minutes to load, which is unreasonable, but just imagine. If that module is only needed in one route handler function it might take some time before someone triggers that route and Node.js has to require that module. When this happens the server would effectively be inaccessible for 30 minutes as that module is loaded. If this happens at peak hours several users would be unable to get any access to your server and requests will queue up.

The second problem is a bigger problem but builds on the first. If the module you require causes an error and crashes the server you may not know about the error for several days, especially is you use this module in a rarely used route handler. No one wants a call from a client at 4AM telling them the server is down.

The solution to both of these problems is to always require modules at the top of your file, outside of any function call. Save the required module to a variable and use the variable instead of re-requiring the module. Node.js will save the module to that variable and your code will run much faster.

var \_ = require('underscore');  
  
function myFunction(someArray){  
  
 // use underscore without the need  
 // to require it again  
 \_.sort(someArray, function(item){  
 // do something with item  
 });  
  
}  
  
module.exports.myFunction = myFunction;

## Save a reference to this Because it Changes Based on Context

If you have a background with Java, ActionScript, PHP, or basically any language that uses the this keyword you might think you understand how JavaScript treats the same keyword. Unfortunately you would be wrong.

Let me tell you how this is determined [officially by ECMAScript](http://www.ecma262-5.com/ELS5_Section_11.htm).

The this keyword evaluates to the value of the ThisBinding of the current execution context.

Basically that means that the value of the this variable is determined based on context, not encapsulation, as it is in other languages.

For example, if this is used inside a function, this references the object that invoked the function. That means that if you create a constructor function (basically a class in JavaScript) which then has methods attached to it, the this variable in those methods may not refer to the constructor function (class) they are inside of.

The above happens a lot in Node, but it might be hard to understand without seeing code.

In the code below this has two different values.

function MyClass() {  
 this.myMethod = function() {  
 console.log(this);  
 };  
}  
  
var myClass = new MyClass();  
myClass.myMethod(); // this resolves as the instance of MyClass  
  
var someFunction = myClass.myMethod;  
someFunction(); // this resolves as the window in a browser and the global object in Node

The best way to solve this is to preserve this as another variable and then use that other variable instead. The most common variable names to use are \_this, that, self, or root.

I personally like \_this or self best because \_this is easy to understand and self will be understood by anyone with Python or Ruby experience as both languages use self instead of this.

After making the changes your code should look like this.

function MyClass() {  
 var self = this;  
 this.myMethod = function() {  
 console.log(self);  
 };  
}  
  
var myClass = new MyClass();  
myClass.myMethod(); // self resolves as the instance of MyClass  
  
var someFunction = myClass.myMethod;  
someFunction(); // self also resolves as the instance of MyClass

self now always refers to the MyClass instance.

You could also call useful binding method, but it is less readable in some cases.

function MyClass() {  
  
 this.myMethod = function() {  
 console.log(this);  
 };  
}  
  
var myClass = new MyClass();  
myClass.myMethod(); // self resolves as the instance of MyClass  
  
var someFunction = myClass.myMethod.bind(myClass);  
someFunction(); // self also resolves as the instance of MyClass

## Always "use strict"

"use strict" is a behavior flag you can add to the first line of any JavaScript file or function. It causes errors when certain bad practices are use in your code, and disallows the use of certain functions, such as with.

Believe it or not but the best place I found to describe what JavaScript's strict mode changes is [Microsoft's JavaScript documentation](http://msdn.microsoft.com/en-us/library/ie/br230269(v=vs.94).aspx#rest).

## Validate that Callbacks are Callable

As stated before Node.js uses a lot of callbacks. Node.js is also weakly typed. The compiler allows any variable to be converted to any other data type. This lack of strict typing can become a big problem when passing parameters into functions for one reason.

Only functions are callable.

This means that if you pass a string to a function, that needed a callback function, your application will crash when it tries to execute that string.

This is obviously bad, but upon first blush there is no simple way to solve it. You could wrap the execution of all callbacks in try catch statements, or you could use if statements to determine if a callback has been passed in.

However, there is a simple way of validating that callbacks are callable which requires only one line of code, and it accounts for optional callbacks as well as checking data type.

callback = (typeof callback === 'function') ? callback : function() {};

This determines if the callback is a function. If it's not a function for any reason it creates an empty function and sets the callback to be that function. This way all callbacks are callable and optional.

Place that line at the top of each function that receives a callback and you will never crash due to uncallable callbacks again.

## Callbacks Always Pass Error Parameter First

Node.js is asynchronous, which means you usually have to use callback functions to determine when your code completes.

After writing Node.js code for a while you will want to start writing your own modules, which need callback functions to be passed in by the user of your module. If an error occurs in your module, how do you communicate that to the user of the module? If you're a Java developer you might think you should throw an exception, but throwing an exception in Node.js could potentially shutdown the server. Instead you should package the error into an object, and pass it to the callback function as the first parameter. If no error occurred you should pass null.

By [convention](http://nodeguide.com/style.html#callbacks) all callback functions are passed an error as the first parameter.

function myFunction(someArray, callback){  
  
 // an example of an error that could occur  
 // if the passed in object is  
 // not the right data type  
 if( !Array.isArray(someArray) ){  
 var err = new TypeError('someArray must be an array');  
 callback(err, null);  
 return;  
 }  
  
 // ... do other stuff  
  
 callback(null, someData);  
  
}  
  
module.export.myFunction = myFunction;

## Always Check for "error" in Callbacks

As stated above, by convention an error is always the first parameter passed to any callback function. This is great for making sure your site doesn't crash and that you can detect errors when they happen.

Now that you know what they are you should start using them. If your database query errors out you need to check for that before using the results. I'll give you an example.

myAsyncFunction({  
 some: 'data'  
 }, function(err, someReturnedData) {  
  
 if(err){  
 // don't use someReturnedData  
 // it's not populated  
 return;  
 }  
  
 // do something with someReturnedData  
 // we know there was no error  
  
 }  
});

## Use Exception Handling When Errors Can Be Thrown

Most methods in Node.js will follow the "error first" convention, but some functions don't. These functions are not Node.js specific function, they instead come from JavaScript. There are lots of functions that can cause exceptions. One of these functions is JSON.parse which throws an error if it can't parse a string into JSON.

How do we detect this error without crashing our server?

This is a perfect time to use a classic JavaScript try catch.

var parsedJSON;  
  
try {  
 parsedJSON = JSON.parse('some invalid JSON');  
} catch (err) {  
 // do something with your error  
}  
  
if (parsedJSON) {  
 // use parsedJSON  
}

You can now be sure that the JSON was parsed correctly before using it.

This can be even more useful when using it in modules.

function parseJSON(stringToParse, callback) {  
  
 callback = (typeof callback === 'function') ? callback : function() {};  
  
 try {  
  
 var parsedJSON = JSON.parse(stringToParse);  
  
 callback(null, parsedJSON);  
  
 } catch (err) {  
  
 callback(err, null);  
  
 }  
  
}

Of course the above example is slightly contrived, however, the idea of using try catch is a very good practice.

## Use module.exports not exports

You might have used module.exports and exports interchangeably thinking they are the same thing and in may cases they are. However, exports is more of a helper method that collects properties and attaches them to module.exports.

So what the problem? That sounds great.

Well don't get too excited. exports only collects properties and attaches them if module.exports doesn't already have existing properties. If module.exports has any properties, everything attached to exports is ignored and not attached to module.exports.

module.exports = {};  
  
exports.someProperty = 'someValue';

someProperty won't be exported as part of the module.

var exportedObject = require('./mod');  
  
console.log(exportedObject); // {}

The solution is simple. Don't use exports because it can create confusing, hard to track down bugs.

module.exports = {};  
  
module.exports.someProperty = 'someValue';

someProperty will be exported as part of the module.

var exportedObject = require('./mod');  
  
console.log(exportedObject); // { someProperty: 'someValue' }

## Use JSDoc

JavaScript is a weakly typed language. Any variable can be passed to any function without conflict, until you try to use that function.

function multiply(num1, num2) {  
  
 return num1 \* num2;  
  
}  
  
var value = multiply('Some String', 2);  
  
console.log(value) // NaN

Obviously this is a problem above that could easily be fixed by looking at the code. But what if the code was written by someone else and uses complex parameters that you don't really understand. You could spend several minutes tracking down the expected data type. Worst yet it might accept multiple data types, in which case it may take you longer to track it down.

The best thing to do is use [JSDoc](http://usejsdoc.org/). If you're a Java developer you will have heard of Javadoc. JSDoc is similar and at it's simplest adds comments above functions to describe how the function works, but it [can do a lot more](http://en.wikipedia.org/wiki/JSDoc#JSDoc_tags).

Some IDEs will even use JSDoc to make code suggestions.

## Use a Process Manager like upstart, forever, or pm2

Keeping a Node.js process running can be daunting. Simply using the node command is dangerous. If your Node.js server crashes the node command will not automatically restart the process.

However, programs like [upstart](https://en.wikipedia.org/wiki/Upstart), [forever](https://www.npmjs.org/package/forever), or [pm2](https://www.npmjs.org/package/pm2) will.

While upstart is a general purpose init daemon forever and pm2 are specific to Node.

## Follow CommonJS Standard

Node.js follows a standard for writing code that varies slightly from the standards that govern writing browser based JavaScript.

This standard is called [CommonJS](http://wiki.commonjs.org/wiki/CommonJS).

While CommonJS is far too large to cover here it's worth knowing about and learning. The most important points are that it mandates certain file organization and behavior that should be expected from the CommonJS module loader (require). It also describes how internals of the Node.js system should work.

Check it out.

## References

1. John Papa's Angular Style Guide. Copyright (c) 2014-2015 John Papa
2. Alan James: Node.js Best Practices