# Angular 1.5 Style Guide (Components Oriented)

## Purpose

> The purpose of this style guide is to provide guidance and best practices on building Angular applications using and thinking in components by showing conventions and examples in the proccess. Build our application with components allow us to upgrade to Angular 2 easily.
>
> **Note**: This style guide makes some references to old but valid style guides.
>
> **Note2**: Angular 1.5 is a transition to Angular 2. Introduces a new player called **.component()**. This is only a sugar syntax for directives, but simplifies the build process of a web components based.

## Index



## AngularJS Application Structure



### Introduction

As we build *AngularJS* applications, the number of files grows rapidly and this can impact the organization and structure of the applications directory. The aim of this document is to define a directory structure, following best practices and conventions, that will allow us to build scalable and maintainable applications.

## Basic Structure

The first approximation is to organize the application by file types. That is *components* and *views* each have their own folder etc. This structure is OK for small applications, but when the number of components, services etc. starts to grow, adding or changing functionality means having to locate the files to edit, identify the code to change, and then making sure no other functionality is affected.

## The LIFT guidelines

To improve on the standard structure we can apply the LIFT guidelines when defining our desired directory structure:

1. **L**ocating our code is easy
2. **I**dentify code at a glance
3. **F**lat structure as long as we can
4. **T**ry to stay DRY (Don't Repeat Yourself) or *T-DRY*

**Locate**: Locating code needs to be untuitive, simple and fast.

**Identify**: When we look at a file we expect to know what it contains and represents.

**Flat**: Nobody wants to search 7 levels of folders to find a file. Keep it as flat as possible.

**T-DRY**: This has more to do with conventions. E.g. Don't include *view* or *controller* in the name of a module if it's obvious what it does.

## Proposed Structure

The key to a clear structure is to have a short term view of implementation and a long term vision. In other words, start small but keep in mind on where the app is heading.

With this is mind and applying the LIFT guidelines, the directory structure proposal is shown below:

```
    app/
        features/
            home/
                home.component.js
                home.css
                home.html
                home.component.spec.js
                home.route.js
            about/
                about.component.js
                about.css
                about.html
                about.component.spec.js
                about.route.js
            ...
        components/
            mycomponent/
                mycomponent.component.js
                mycomponent.component.spec.js
                mycomponent.html
                mycomponent.css
                mycomponent.service.js
                mycomponent.service.spec.js
            mycomponent2/
                mycomponent2.component.js
                mycomponent2.component.spec.js
```

```
                mycomponent2.html
                mycomponent2.css
                mycomponent2.service.js
                mycomponent2.service.spec.js
            ...
        services/
            data.service.js
            logger.service.js
            ...
        app.module.js
        app.config.js
        app.constants.js
        app.routes.js
        app.run.js
    assets/
        img/   // Images and icons for your application
        css/   // All styles related files
        js/    // Non-angular js files
        libs/  // 3rd-party libs E.g. jQuery.
        fonts/ // 3rd-party fonts
    index.html
```

Where:

| Folder/File | Description |
|---|---|
| **index.html** | The *index.html* lives at the root of the front-end structure. It will primarily handle loading in all the libraries and Angular elements. |
| **assets** | The assets folder is also pretty standard. It will contain all the assets needed for your app that are not related to your AngularJS code. |
| **app** | This is where the application lives!<br>The *app.module.js* file will handle the setup of your app, load in AngularJS dependencies and so on.<br>The *app.route.js* file will handle the main route.<br>The *app.config.js* file can be used to establish configuration globally for the app |

The app folder contains the following sub-directories:

| Folder/File | Description |
|---|---|
| **components** | Reusable components for the application. |
| **services** | Services provide some service to the app for shareable features such as remote data access, data caching, local storage, and logging for example. |
| **features1..N** | These named folders represent features of the application. E.g. *login*, *session*, or *gallery* for example.<br>Each component, view etc. is in his own file.<br>The folder also contains test scripts, and css particular to the view if necessary.<br>In Reference #1 John Papa talks about the **3-7 file guideline**. If a feature contains more than 7 files, it may be time to divide that feature into smaller features. |

## Benefits of the Modularized Approach

**Code maintainability**

- Following the approach above will logically compartmentalize your apps and you will easily be able to locate and edit code.

**Scalable**

- Your code will be much easier to scale. Adding new directives and pages will not bloat existing folders. On-boarding new

developers should also be much easier once the structure is explained. Additionally, with this approach, you will be able to drop features in and out of your app with relative ease, so testing new functionality or removing it should be a breeze.

**Debugging**

- Debugging your code will be much easier with this modularized approach to app development. It will be easier to find the offending pieces of code and fix them.

**Testing**

- Writing test scripts and testing modernized apps is a whole lot easier then non-modularized ones.

# Best Practices

## Components

> Components are the new features of Angular 1.5. The method .component() replaces our old controllers and directives in this new way to build applications. Think in components as a functional complex element, which you can integrate inside others components and coexist with other to give to the application an unique sense.
>
> The .component() method helper is just syntactic sugar of the good old .directive() method. There is practically nothing you can do with .component() that you can't do with .directive().

- Components have **isolated scopes** by default.
- They automatically use *controllerAs* syntax therefore we can use **$ctrl** to access component data.
- They use **controllers** instead of *link* functions.
- The *bindToController* option is on **by default**.

```
// before
app.directive(name, fn)

// after
app.component(name, options)
```

```
.component('counter', {
    templateUrl: 'app/components/counter/counter.html',
    controller: CounterController,
    controllerAs: 'vm',
     // isolated scope binding
    bindings: {
        count: '='
    }
});

function CounterController() {
    var vm = this;
    vm.increment = increment;
    vm.decrement = decrement;

    function increment() {
```

```
        vm.count++;
      }
      function decrement() {
        vm.count--;
      }
  }
```

Inline template which is binded to message variable in the component controller:

```
<div>Counter {{vm.count}}</div>
```

**Bindings**

- The **< symbol** denotes one-way bindings. read.
- The **= symbol** denotes two-way bindings. read & write.
- The **@ symbol** can be used when the input is a string, especially when the value of the binding doesn't change.

```
bindings: {
    hero: '<',
    comment: '@'
}
```

- The **& symbol** Outputs are realized with & bindings, which function as callbacks to component events.

```
bindings: {
    onDelete: '&',
    onUpdate: '&'
}
```

```
<editable-field on-update="vm.update('location', value)"></editable-field><br>
<button ng-click="vm.onDelete({hero: vm.hero})">Delete</button>
```

## Routing components

Components are also useful as route templates (e.g. when using ngRoute). In a component-based application, every view is a component:

```
$routeProvider.when('/', {
    template: '<home></home>'
});
```

## Intercomponent Communication

Directives can require the controllers of other directives to enable communication between each other. This can be achieved in a component by providing an object mapping for the require property. The object keys specify the property names under which the required controllers (object values) will be bound to the requiring component's controller.

> Note that the required controllers will not be available during the instantiation of the controller, but they are guaranteed to be available just before the **$onInit** method is executed!

```
// Component 1
.component('myTabs', {
  transclude: true,
  controller: MyTabsController,
  controllerAs: 'vm',
  templateUrl: 'my-tabs.html'
});

function MyTabsController() {
    var vm = this;
    vm.panes = [];

    vm.addPane = function(pane) {
      // Code
    };
}

// Component 2
.component('myPane', {
  transclude: true,
  require: {
    tabsCtrl: '^myTabs'
  },
  bindings: {
    title: '@'
  },
  controller: MyPaneController,
  controllerAs: 'vm',
  templateUrl: 'my-pane.html'
});

function MyPaneController() {
    var vm = this;
    vm.$onInit = function() {
        vm.tabsCtrl.addPane(vm); // adding pane into addPane function of myTabs
    };
}
```

## Unit-testing Component Controllers

> The easiest way to unit-test a component controller is by using the $componentController that is included in ngMock. The advantage of this method is that you do not have to create any DOM elements.

```
describe('component: heroDetail', function() {
  var $componentController;

  beforeEach(module('heroApp'));
  beforeEach(inject(function(_$componentController_) {
    $componentController = _$componentController_;
  }));

  it('should expose a `hero` object', function() {
    // Here we are passing actual bindings to the component
    var bindings = {hero: {name: 'Wolverine'}};
    var vm = $componentController('heroDetail', null, bindings);
```

```
    expect(vm.hero).toBeDefined();
    expect(vm.hero.name).toBe('Wolverine');
  });
```

**Bindable Members Up Top**

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

More Info +

```
function SessionsController() {
    var vm = this;

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

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

    function gotoSession() {
      /* */
    }

    function refresh() {
      /* */
    }

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

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

```
function SessionsController(sessionDataService) {
    var vm = this;

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

**Function Declarations to Hide Implementation Details**

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

More Info +

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

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

    activate();

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

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

**Defer Controller Logic to Services**

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

More Info +

```
function OrderController(creditService) {
    var vm = this;
    vm.checkCredit = checkCredit;
    vm.isCreditOk;
    vm.total = 0;

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

## Services

**Singletons**

- Services are instantiated with the new keyword, use this for public methods and variables. Since these are so similar to factories, the use of factories is recommended instead for consistency.

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

```
// service
angular
    .module('app')
```

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

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

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

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

## Factories

**Single Responsibility**

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

**Singletons**

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

  Note: All Angular services are singletons.

**Accessible Members Up Top**

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

More Info +

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

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

    function save() {
        /* */
    };

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

```
    };
}
```

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

**Function Declarations to Hide Implementation Details**

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

More Info +

```
/**
 * 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
    }
}
```

## Data Services

**Separate Data Calls**

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

More Info +

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

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

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

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

        function getAvengersComplete(response) {
            return response.data.results;
        }

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

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

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

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

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

    activate();

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

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

```
            });
    }
}
```

**Return a Promise from Data Calls**

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

More Info +

```
activate();

function activate() {
    /**
     * Step 1
     * Ask the getAvengers function for the
     * avenger data and wait for the promise
     */
    return getAvengers().then(function() {
        /**
         * Step 4
         * Perform an action on resolve of final promise
         */
        logger.info('Activated Avengers View');
    });
}

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

## Directives

> Directives and templates/controllers are not necessary anymore, however the API remains consistent for backwards compatibility. Use a Directive for binding custom behaviour to existing DOM.

**What is the role of .directive()?**

- A Directive decorates the DOM, it adds behaviour and extends existing DOM. When you use a .component(), you create DOM, when you use .directive() you decorate DOM, that is the mindset.
- You already know ng-click, it's a Directive, therefore it decorates and adds existing behaviour to an existing element, it is not as an Element.

**How can I use .directive()?**

- Directives should be used when you need to conduct DOM manipulation outside of the Angular event loop and core
- Use the compile and/or link function to create the custom functionality you require
- Ensure you unbind custom events or DOM APIs such as element.addEventListener(); inside the $destroy event
- Obtain access to attributes and use $observe depending on what you need to access (readonly values/etc)
- If using a Controller, use as fourth link: fn argument, and only update view logic inside the Controller
- Never pass $scope for manipulating your data, always use fourth argument $ctrl (if using a Controller, depends on use case)
- Require and manipulation other Directives using require property as String or Array syntax

**What should I restrict my Components/Directives to?**

- .component() is restricted to 'E' by default, meaning custom element, you cannot change this.
- .directive() should decorate, therefore should be an attribute only, meaning restrict: 'A' always.

**Limit 1 Per File**

- Create one directive per file. Name the file for the directive. More info +

```
> 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.
```

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

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

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

```
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**

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

**Provide a Unique Directive Prefix**

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

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

**Restrict to Attributes**

- When creating a directive restrict A (custom attribute).

```
<my-calendar-range></my-calendar-range>
<div my-calendar-range></div>
```

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

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

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

**Directives and ControllerAs**

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

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

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

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

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

```
        controllerAs: 'vm',
        bindToController: true // because the scope is isolated
    };

    return directive;

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

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

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

    vm.min = 3;

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

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

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

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

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

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

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

function myExample() {
```

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

    return directive;
}

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

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

# Upgrading to Angular 2

> Writing components in this style will allow you to upgrade your Components using .component() into Angular 2 very easily, it'd look something like this in ECMAScript 5 and new template syntax:

```
import {Component} from '@angular/core';

@Component({
  selector: 'counter',
  template: `
    <div class="todo">
      <input type="text" [(ng-model)]="count">
      <button type="button" (click)="decrement();">-</button>
      <button type="button" (click)="increment();">+</button>
    </div>
  `
})
export default class CounterComponent {
  constructor() {

  }
  increment() {
    this.count++;
  }
  decrement() {
    this.count--;
  }
}
```

As you can see, it's no a trivial change. Change to Angular 2 is necesary changes to typescript and some learn some syntax, but the new orientation to components makes that change easier.

## Resources

1. Component versus Directive
2. From ng-controller to components with Angular 1.5. Juri Strumpflohner
3. Exploring component method. Todd Motto
4. App structuring guidelines. John Papa
5. Application Structure. John Papa - GitHub