Programming Standards for Senior Design Project

Regarding Angular

Reference: https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md

# **AngularJS styleguide (ES2015)**

### **Up-to-date with AngularJS 1.6 best practices. Architecture, file structure, components, one-way dataflow, lifecycle hooks.**

Want an example structure as reference? Check out my [component based architecture 1.5 app](https://github.com/toddmotto/angular-1-5-components-app).

*A sensible styleguide for teams by [@toddmotto](https://twitter.com/toddmotto)*

This architecture and styleguide has been rewritten from the ground up for ES2015, the changes in AngularJS 1.5+ for future-upgrading your application to Angular. This guide includes new best practices for one-way dataflow, event delegation, component architecture and component routing.

You can find the old styleguide [here](https://github.com/toddmotto/angular-styleguide/tree/angular-old-es5), and the reasoning behind the new one [here](https://toddmotto.com/rewriting-angular-styleguide-angular-2).

Join the Ultimate AngularJS learning experience to fully master beginner and advanced AngularJS features to build real-world apps that are fast, and scale.



## **Table of Contents**

1. [Modular architecture](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#modular-architecture)
   1. [Theory](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#module-theory)
   2. [Root module](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#root-module)
   3. [Component module](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#component-module)
   4. [Common module](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#common-module)
   5. [Low-level modules](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#low-level-modules)
   6. [File naming conventions](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#file-naming-conventions)
   7. [Scalable file structure](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#scalable-file-structure)
2. [Components](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#components)
   1. [Theory](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#component-theory)
   2. [Supported properties](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#supported-properties)
   3. [Controllers](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#controllers)
   4. [One-way dataflow and Events](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#one-way-dataflow-and-events)
   5. [Stateful Components](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#stateful-components)
   6. [Stateless Components](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#stateless-components)
   7. [Routed Components](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#routed-components)
3. [Directives](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#directives)
   1. [Theory](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#directive-theory)
   2. [Recommended properties](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#recommended-properties)
   3. [Constants or Classes](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#constants-or-classes)
4. [Services](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#services)
   1. [Theory](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#service-theory)
   2. [Classes for Service](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#classes-for-service)
5. [Styles](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#styles)
6. [ES2015 and Tooling](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#es2015-and-tooling)
7. [State management](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#state-management)
8. [Resources](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#resources)
9. [Documentation](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#documentation)
10. [Contributing](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#contributing)

# **Modular architecture**

Each module in an Angular app is a module component. A module component is the root definition for that module that encapsulates the logic, templates, routing and child components.

### **Module theory**

The design in the modules maps directly to our folder structure, which keeps things maintainable and predictable. We should ideally have three high-level modules: root, component and common. The root module defines the base module that bootstraps our app, and the corresponding template. We then import our component and common modules into the root module to include our dependencies. The component and common modules then require lower-level component modules, which contain our components, controllers, services, directives, filters and tests for each reusable feature.

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Root module**

A root module begins with a root component that defines the base element for the entire application, with a routing outlet defined, example shown using ui-view from ui-router.

// app.component.js  
export const AppComponent = {  
 template: `  
 <header>  
 Hello world  
 </header>  
 <div>  
 <div ui-view></div>  
 </div>  
 <footer>  
 Copyright MyApp 2016.  
 </footer>  
 `  
};

A root module is then created, with AppComponent imported and registered with .component('app', AppComponent). Further imports for submodules (component and common modules) are made to include all components relevant for the application. You'll notice styles are also being imported here, we'll come onto this in later chapters in this guide.

// app.module.js  
import angular from 'angular';  
import uiRouter from 'angular-ui-router';  
import { AppComponent } from './app.component';  
import { ComponentsModule } from './components/components.module';  
import { CommonModule } from './common/common.module';  
import './app.scss';  
  
export const AppModule = angular  
 .module('app', [  
 ComponentsModule,  
 CommonModule,  
 uiRouter  
 ])  
 .component('app', AppComponent)  
 .name;

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Component module**

A Component module is the container reference for all reusable components. See above how we import ComponentsModuleand inject them into the Root module, this gives us a single place to import all components for the app. These modules we require are decoupled from all other modules and thus can be moved into any other application with ease.

import angular from 'angular';  
import { CalendarModule } from './calendar/calendar.module';  
import { EventsModule } from './events/events.module';  
  
export const ComponentsModule = angular  
 .module('app.components', [  
 CalendarModule,  
 EventsModule  
 ])  
 .name;

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Common module**

The Common module is the container reference for all application specific components, that we don't want to use in another application. This can be things like layout, navigation and footers. See above how we import CommonModule and inject them into the Root module, this gives us a single place to import all common components for the app.

import angular from 'angular';  
import { NavModule } from './nav/nav.module';  
import { FooterModule } from './footer/footer.module';  
  
export const CommonModule = angular  
 .module('app.common', [  
 NavModule,  
 FooterModule  
 ])  
 .name;

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Low-level modules**

Low-level modules are individual component modules that contain the logic for each feature block. These will each define a module, to be imported to a higher-level module, such as a component or common module, an example below. Always remember to add the .name suffix to each export when creating a *new* module, not when referencing one. You'll noticed routing definitions also exist here, we'll come onto this in later chapters in this guide.

import angular from 'angular';  
import uiRouter from 'angular-ui-router';  
import { CalendarComponent } from './calendar.component';  
import './calendar.scss';  
  
export const CalendarModule = angular  
 .module('calendar', [  
 uiRouter  
 ])  
 .component('calendar', CalendarComponent)  
 .config(($stateProvider, $urlRouterProvider) => {  
 'ngInject';  
 $stateProvider  
 .state('calendar', {  
 url: '/calendar',  
 component: 'calendar'  
 });  
 $urlRouterProvider.otherwise('/');  
 })  
 .name;

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **File naming conventions**

Keep it simple and lowercase, use the component name, e.g. calendar.\*.js\*, calendar-grid.\*.js - with the name of the type of file in the middle. Use \*.module.js for the module definition file, as it keeps it verbose and consistent with Angular.

calendar.module.js  
calendar.component.js  
calendar.service.js  
calendar.directive.js  
calendar.filter.js  
calendar.spec.js  
calendar.html  
calendar.scss

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Scalable file structure**

File structure is extremely important, this describes a scalable and predictable structure. An example file structure to illustrate a modular component architecture.

├── app/  
│ ├── components/  
│ │ ├── calendar/  
│ │ │ ├── calendar.module.js  
│ │ │ ├── calendar.component.js  
│ │ │ ├── calendar.service.js  
│ │ │ ├── calendar.spec.js  
│ │ │ ├── calendar.html  
│ │ │ ├── calendar.scss  
│ │ │ └── calendar-grid/  
│ │ │ ├── calendar-grid.module.js  
│ │ │ ├── calendar-grid.component.js  
│ │ │ ├── calendar-grid.directive.js  
│ │ │ ├── calendar-grid.filter.js  
│ │ │ ├── calendar-grid.spec.js  
│ │ │ ├── calendar-grid.html  
│ │ │ └── calendar-grid.scss  
│ │ ├── events/  
│ │ │ ├── events.module.js  
│ │ │ ├── events.component.js  
│ │ │ ├── events.directive.js  
│ │ │ ├── events.service.js  
│ │ │ ├── events.spec.js  
│ │ │ ├── events.html  
│ │ │ ├── events.scss  
│ │ │ └── events-signup/  
│ │ │ ├── events-signup.module.js  
│ │ │ ├── events-signup.component.js  
│ │ │ ├── events-signup.service.js  
│ │ │ ├── events-signup.spec.js  
│ │ │ ├── events-signup.html  
│ │ │ └── events-signup.scss  
│ │ └── components.module.js  
│ ├── common/  
│ │ ├── nav/  
│ │ │ ├── nav.module.js  
│ │ │ ├── nav.component.js  
│ │ │ ├── nav.service.js  
│ │ │ ├── nav.spec.js  
│ │ │ ├── nav.html  
│ │ │ └── nav.scss  
│ │ ├── footer/  
│ │ │ ├── footer.module.js  
│ │ │ ├── footer.component.js  
│ │ │ ├── footer.service.js  
│ │ │ ├── footer.spec.js  
│ │ │ ├── footer.html  
│ │ │ └── footer.scss  
│ │ └── common.module.js  
│ ├── app.module.js  
│ ├── app.component.js  
│ └── app.scss  
└── index.html

The high level folder structure simply contains index.html and app/, a directory in which all our root, component, common and low-level modules live along with the markup and styles for each component.

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

# **Components**

### **Component theory**

Components are essentially templates with a controller. They are *not* Directives, nor should you replace Directives with Components, unless you are upgrading "template Directives" with controllers, which are best suited as a component. Components also contain bindings that define inputs and outputs for data and events, lifecycle hooks and the ability to use one-way data flow and event Objects to get data back up to a parent component. These are the new defacto standard in AngularJS 1.5 and above. Everything template and controller driven that we create will likely be a component, which may be a stateful, stateless or routed component. You can think of a "component" as a complete piece of code, not just the .component() definition Object. Let's explore some best practices and advisories for components, then dive into how you should be structuring them via stateful, stateless and routed component concepts.

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Supported properties**

These are the supported properties for .component() that you can/should use:

|  |  |
| --- | --- |
| **Property** | **Support** |
| bindings | Yes, use '@', '<', '&' only |
| controller | Yes |
| controllerAs | Yes, default is $ctrl |
| require | Yes (new Object syntax) |
| template | Yes |
| templateUrl | Yes |
| transclude | Yes |

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Controllers**

Controllers should only be used alongside components, never anywhere else. If you feel you need a controller, what you really need is likely a stateless component to manage that particular piece of behaviour.

Here are some advisories for using Class for controllers:

* Drop the name "Controller", i.e. use controller: class TodoComponent {...} to aid future Angular migration
* Always use the constructor for dependency injection purposes
* Use [ng-annotate](https://github.com/olov/ng-annotate)'s 'ngInject'; syntax for $inject annotations
* If you need to access the lexical scope, use arrow functions
* Alternatively to arrow functions, let ctrl = this; is also acceptable and may make more sense depending on the use case
* Bind all public functions directly to the Class
* Make use of the appropriate lifecycle hooks, $onInit, $onChanges, $postLink and $onDestroy
  + Note: $onChanges is called before $onInit, see [resources](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#resources) section for articles detailing this in more depth
* Use require alongside $onInit to reference any inherited logic
* Do not override the default $ctrl alias for the controllerAs syntax, therefore do not use controllerAs anywhere

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **One-way dataflow and Events**

One-way dataflow was introduced in AngularJS 1.5, and redefines component communication.

Here are some advisories for using one-way dataflow:

* In components that receive data, always use one-way databinding syntax '<'
* *Do not* use '=' two-way databinding syntax anymore, anywhere
* Components that have bindings should use $onChanges to clone the one-way binding data to break Objects passing by reference and updating the parent data
* Use $event as a function argument in the parent method (see stateful example below $ctrl.addTodo($event))
* Pass an $event: {} Object back up from a stateless component (see stateless example below this.onAddTodo).
  + Bonus: Use an EventEmitter wrapper with .value() to mirror Angular, avoids manual $event Object creation
* Why? This mirrors Angular and keeps consistency inside every component. It also makes state predictable.

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Stateful components**

Let's define what we'd call a "stateful component".

* Fetches state, essentially communicating to a backend API through a service
* Does not directly mutate state
* Renders child components that mutate state
* Also referred to as smart/container components

An example of a stateful component, complete with its low-level module definition (this is only for demonstration, so some code has been omitted for brevity):

/\* ----- todo/todo.component.js ----- \*/  
import templateUrl from './todo.html';  
  
export const TodoComponent = {  
 templateUrl,  
 controller: class TodoComponent {  
 constructor(TodoService) {  
 'ngInject';  
 this.todoService = TodoService;  
 }  
 $onInit() {  
 this.newTodo = {  
 title: '',  
 selected: false  
 };  
 this.todos = [];  
 this.todoService.getTodos().then(response => this.todos = response);  
 }  
 addTodo({ todo }) {  
 if (!todo) return;  
 this.todos.unshift(todo);  
 this.newTodo = {  
 title: '',  
 selected: false  
 };  
 }  
 }  
};  
  
/\* ----- todo/todo.html ----- \*/  
<div class="todo">  
 <todo-form  
 todo="$ctrl.newTodo"  
 on-add-todo="$ctrl.addTodo($event);"></todo-form>  
 <todo-list  
 todos="$ctrl.todos"></todo-list>  
</div>  
  
/\* ----- todo/todo.module.js ----- \*/  
import angular from 'angular';  
import { TodoComponent } from './todo.component';  
import './todo.scss';  
  
export const TodoModule = angular  
 .module('todo', [])  
 .component('todo', TodoComponent)  
 .name;

This example shows a stateful component, that fetches state inside the controller, through a service, and then passes it down into stateless child components. Notice how there are no Directives being used such as ng-repeat and friends inside the template. Instead, data and functions are delegated into <todo-form> and <todo-list> stateless components.

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Stateless components**

Let's define what we'd call a "stateless component".

* Has defined inputs and outputs using bindings: {}
* Data enters the component through attribute bindings (inputs)
* Data leaves the component through events (outputs)
* Mutates state, passes data back up on-demand (such as a click or submit event)
* Doesn't care where data comes from - it's stateless
* Are highly reusable components
* Also referred to as dumb/presentational components

An example of a stateless component (let's use <todo-form> as an example), complete with its low-level module definition (this is only for demonstration, so some code has been omitted for brevity):

/\* ----- todo/todo-form/todo-form.component.js ----- \*/  
import templateUrl from './todo-form.html';  
  
export const TodoFormComponent = {  
 bindings: {  
 todo: '<',  
 onAddTodo: '&'  
 },  
 templateUrl,  
 controller: class TodoFormComponent {  
 constructor(EventEmitter) {  
 'ngInject';  
 this.EventEmitter = EventEmitter;  
 }  
 $onChanges(changes) {  
 if (changes.todo) {  
 this.todo = Object.assign({}, this.todo);  
 }  
 }  
 onSubmit() {  
 if (!this.todo.title) return;  
 // with EventEmitter wrapper  
 this.onAddTodo(  
 this.EventEmitter({  
 todo: this.todo  
 })  
 );  
 // without EventEmitter wrapper  
 this.onAddTodo({  
 $event: {  
 todo: this.todo  
 }  
 });  
 }  
 }  
};  
  
/\* ----- todo/todo-form/todo-form.html ----- \*/  
<form name="todoForm" ng-submit="$ctrl.onSubmit();">  
 <input type="text" ng-model="$ctrl.todo.title">  
 <button type="submit">Submit</button>  
</form>  
  
/\* ----- todo/todo-form/todo-form.module.js ----- \*/  
import angular from 'angular';  
import { TodoFormComponent } from './todo-form.component';  
import './todo-form.scss';  
  
export const TodoFormModule = angular  
 .module('todo.form', [])  
 .component('todoForm', TodoFormComponent)  
 .value('EventEmitter', payload => ({ $event: payload }))  
 .name;

Note how the <todo-form> component fetches no state, it simply receives it, mutates an Object via the controller logic associated with it, and passes it back to the parent component through the property bindings. In this example, the $onChanges lifecycle hook makes a clone of the initial this.todo binding Object and reassigns it, which means the parent data is not affected until we submit the form, alongside one-way data flow new binding syntax '<'.

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Routed components**

Let's define what we'd call a "routed component".

* It's essentially a stateful component, with routing definitions
* No more router.js files
* We use Routed components to define their own routing logic
* Data "input" for the component is done via the route resolve (optional, still available in the controller with service calls)

For this example, we're going to take the existing <todo> component, refactor it to use a route definition and bindings on the component which receives data (the secret here with ui-router is the resolve properties we create, in this case todoData directly map across to bindings for us). We treat it as a routed component because it's essentially a "view":

/\* ----- todo/todo.component.js ----- \*/  
import templateUrl from './todo.html';  
  
export const TodoComponent = {  
 bindings: {  
 todoData: '<'  
 },  
 templateUrl,  
 controller: class TodoComponent {  
 constructor() {  
 'ngInject'; // Not actually needed but best practice to keep here incase dependencies needed in the future  
 }  
 $onInit() {  
 this.newTodo = {  
 title: '',  
 selected: false  
 };  
 }  
 $onChanges(changes) {  
 if (changes.todoData) {  
 this.todos = Object.assign({}, this.todoData);  
 }  
 }  
 addTodo({ todo }) {  
 if (!todo) return;  
 this.todos.unshift(todo);  
 this.newTodo = {  
 title: '',  
 selected: false  
 };  
 }  
 }  
};  
  
/\* ----- todo/todo.html ----- \*/  
<div class="todo">  
 <todo-form  
 todo="$ctrl.newTodo"  
 on-add-todo="$ctrl.addTodo($event);"></todo-form>  
 <todo-list  
 todos="$ctrl.todos"></todo-list>  
</div>  
  
/\* ----- todo/todo.service.js ----- \*/  
export class TodoService {  
 constructor($http) {  
 'ngInject';  
 this.$http = $http;  
 }  
 getTodos() {  
 return this.$http.get('/api/todos').then(response => response.data);  
 }  
}  
  
/\* ----- todo/todo.module.js ----- \*/  
import angular from 'angular';  
import uiRouter from 'angular-ui-router';  
import { TodoComponent } from './todo.component';  
import { TodoService } from './todo.service';  
import './todo.scss';  
  
export const TodoModule = angular  
 .module('todo', [  
 uiRouter  
 ])  
 .component('todo', TodoComponent)  
 .service('TodoService', TodoService)  
 .config(($stateProvider, $urlRouterProvider) => {  
 'ngInject';  
 $stateProvider  
 .state('todos', {  
 url: '/todos',  
 component: 'todo',  
 resolve: {  
 todoData: TodoService => TodoService.getTodos()  
 }  
 });  
 $urlRouterProvider.otherwise('/');  
 })  
 .name;

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

# **Directives**

### **Directive theory**

Directives gives us template, scope bindings, bindToController, link and many other things. The usage of these should be carefully considered now that .component() exists. Directives should not declare templates and controllers anymore, or receive data through bindings. Directives should be used solely for decorating the DOM. By this, it means extending existing HTML - created with .component(). In a simple sense, if you need custom DOM events/APIs and logic, use a Directive and bind it to a template inside a component. If you need a sensible amount of DOM manipulation, there is also the $postLinklifecycle hook to consider, however this is not a place to migrate all your DOM manipulation to, use a Directive if you can for non-Angular things.

Here are some advisories for using Directives:

* Never use templates, scope, bindToController or controllers
* Always restrict: 'A' with Directives
* Use compile and link where necessary
* Remember to destroy and unbind event handlers inside $scope.$on('$destroy', fn);

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Recommended properties**

Due to the fact directives support most of what .component() does (template directives were the original component), I'm recommending limiting your directive Object definitions to only these properties, to avoid using directives incorrectly:

|  |  |  |
| --- | --- | --- |
| **Property** | **Use it?** | **Why** |
| bindToController | No | Use bindings in components |
| compile | Yes | For pre-compile DOM manipulation/events |
| controller | No | Use a component |
| controllerAs | No | Use a component |
| link functions | Yes | For pre/post DOM manipulation/events |
| multiElement | Yes | [See docs](https://docs.angularjs.org/api/ng/service/$compile#-multielement-) |
| priority | Yes | [See docs](https://docs.angularjs.org/api/ng/service/$compile#-priority-) |
| require | No | Use a component |
| restrict | Yes | Defines directive usage, always use 'A' |
| scope | No | Use a component |
| template | No | Use a component |
| templateNamespace | Yes (if you must) | [See docs](https://docs.angularjs.org/api/ng/service/$compile#-templatenamespace-) |
| templateUrl | No | Use a component |
| transclude | No | Use a component |

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Constants or Classes**

There are a few ways to approach using ES2015 and directives, either with an arrow function and easier assignment, or using an ES2015 Class. Choose what's best for you or your team, keep in mind Angular uses Class.

Here's an example using a constant with an Arrow function an expression wrapper () => ({}) returning an Object literal (note the usage differences inside .directive()):

/\* ----- todo/todo-autofocus.directive.js ----- \*/  
import angular from 'angular';  
  
export const TodoAutoFocus = ($timeout) => {  
 'ngInject';  
 return {  
 restrict: 'A',  
 link($scope, $element, $attrs) {  
 $scope.$watch($attrs.todoAutofocus, (newValue, oldValue) => {  
 if (!newValue) {  
 return;  
 }  
 $timeout(() => $element[0].focus());  
 });  
 }  
 }  
};  
  
/\* ----- todo/todo.module.js ----- \*/  
import angular from 'angular';  
import { TodoComponent } from './todo.component';  
import { TodoAutofocus } from './todo-autofocus.directive';  
import './todo.scss';  
  
export const TodoModule = angular  
 .module('todo', [])  
 .component('todo', TodoComponent)  
 .directive('todoAutofocus', TodoAutoFocus)  
 .name;

Or using ES2015 Class (note manually calling new TodoAutoFocus when registering the directive) to create the Object:

/\* ----- todo/todo-autofocus.directive.js ----- \*/  
import angular from 'angular';  
  
export class TodoAutoFocus {  
 constructor($timeout) {  
 'ngInject';  
 this.restrict = 'A';  
 this.$timeout = $timeout;  
 }  
 link($scope, $element, $attrs) {  
 $scope.$watch($attrs.todoAutofocus, (newValue, oldValue) => {  
 if (!newValue) {  
 return;  
 }  
 this.$timeout(() => $element[0].focus());  
 });  
 }  
}  
  
/\* ----- todo/todo.module.js ----- \*/  
import angular from 'angular';  
import { TodoComponent } from './todo.component';  
import { TodoAutofocus } from './todo-autofocus.directive';  
import './todo.scss';  
  
export const TodoModule = angular  
 .module('todo', [])  
 .component('todo', TodoComponent)  
 .directive('todoAutofocus', ($timeout) => new TodoAutoFocus($timeout))  
 .name;

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

# **Services**

### **Service theory**

Services are essentially containers for business logic that our components shouldn't request directly. Services contain other built-in or external services such as $http, that we can then inject into component controllers elsewhere in our app. We have two ways of doing services, using .service() or .factory(). With ES2015 Class, we should only use .service(), complete with dependency injection annotation using $inject.

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

### **Classes for Service**

Here's an example implementation for our <todo> app using ES2015 Class:

/\* ----- todo/todo.service.js ----- \*/  
export class TodoService {  
 constructor($http) {  
 'ngInject';  
 this.$http = $http;  
 }  
 getTodos() {  
 return this.$http.get('/api/todos').then(response => response.data);  
 }  
}  
  
/\* ----- todo/todo.module.js ----- \*/  
import angular from 'angular';  
import { TodoComponent } from './todo.component';  
import { TodoService } from './todo.service';  
import './todo.scss';  
  
export const TodoModule = angular  
 .module('todo', [])  
 .component('todo', TodoComponent)  
 .service('TodoService', TodoService)  
 .name;

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

# **Styles**

Using [Webpack](https://webpack.github.io/) we can now use import statements on our .scss files in our \*.module.js to let Webpack know to include that file in our stylesheet. Doing this lets us keep our components isolated for both functionality and style; it also aligns more closely to how stylesheets are declared for use in Angular. Doing this won't isolate our styles to just that component like it does with Angular; the styles will still be usable application wide but it is more manageable and makes our applications structure easier to reason about.

If you have some variables or globally used styles like form input elements then these files should still be placed into the root scss folder. e.g. scss/\_forms.scss. These global styles can then be @imported into your root module (app.module.js) stylesheet like you would normally do.

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

# **ES2015 and Tooling**

##### **ES2015**

* Use [Babel](https://babeljs.io/) to compile your ES2015+ code and any polyfills
* Consider using [TypeScript](http://www.typescriptlang.org/) to make way for any Angular upgrades

##### **Tooling**

* Use ui-router [latest alpha](https://github.com/angular-ui/ui-router) (see the Readme) if you want to support component-routing
  + Otherwise you're stuck with template: '<component>' and no bindings/resolve mapping
* Consider preloading templates into $templateCache with angular-templates or ngtemplate-loader
  + [Gulp version](https://www.npmjs.com/package/gulp-angular-templatecache)
  + [Grunt version](https://www.npmjs.com/package/grunt-angular-templates)
  + [Webpack version](https://github.com/WearyMonkey/ngtemplate-loader)
* Consider using [Webpack](https://webpack.github.io/) for compiling your ES2015 code and styles
* Use [ngAnnotate](https://github.com/olov/ng-annotate) to automatically annotate $inject properties
* How to use [ngAnnotate with ES6](https://www.timroes.de/2015/07/29/using-ecmascript-6-es6-with-angularjs-1-x/#ng-annotate)

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

# **State management**

Consider using Redux with AngularJS 1.5 for data management.

* [Angular Redux](https://github.com/angular-redux/ng-redux)

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

# **Resources**

* [Stateful and stateless components, the missing manual](https://toddmotto.com/stateful-stateless-components)
* [Understanding the .component() method](https://toddmotto.com/exploring-the-angular-1-5-component-method/)
* [Using "require" with $onInit](https://toddmotto.com/on-init-require-object-syntax-angular-component/)
* [Understanding all the lifecycle hooks, $onInit, $onChanges, $postLink, $onDestroy](https://toddmotto.com/angular-1-5-lifecycle-hooks)
* [Using "resolve" in routes](https://toddmotto.com/resolve-promises-in-angular-routes/)
* [Redux and Angular state management](http://blog.rangle.io/managing-state-redux-angular/)
* [Sample Application from Community](https://github.com/chihab/angular-styleguide-sample)

[Back to top](https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md#table-of-contents)

# **Documentation**

For anything else, including API reference, check the [AngularJS documentation](https://docs.angularjs.org/api).

# **Contributing**

Open an issue first to discuss potential changes/additions. Please don't open issues for questions.

## **License**

#### **(The MIT License)**

Copyright (c) 2016-2018 Todd Motto

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.