AngularJs Clean Code

5 June 2015 | **Product Engineering** | **Mate Balo Bordas**, **Marvin Minder**
Reading time: 13 minutes

Getting started with AngularJS can be hard, especially if it's your first JavaScript framework. Once you got the basics down, you still need to find a way to write a maintainable code. Since I'm new to AngularJS, I decided to do some research to improve my skills and also try and help out others. I created a very basic tourist guide application and I will walk through the implementations of these practices using this application. With this application I will heavily rely on John Papa's best practices. [1]

Separating concerns

Angular is built around separation of concerns. Separation of concerns is a design pattern that makes our code easier to maintain and extend, it also makes our code more reusable and testable. It helps us encapsulate logic, and makes sure that the components do only one thing and do that one thing well. Separation of concerns is the core of writing clean code with Angular, and it uses the rule of one i.e. 1 component, 1 role, 1 per file. For example the controller should handle logic for a view, but shouldn't be concerned about how XHR requests are handled or how the logging is handled. To achieve this you should always take advantage of the dependency injection, and the modules that Angular gives you out of the box. Separation of concerns has many benefits and makes it easier

- to maintain and extend our codebase,
- to reuse code, we won't repeat ourselves
- to test, as each component will do one thing only
- to isolate and fix bugs.

If your component does everything, it does nothing well. So first you should make sure that one component has only one job. A good way to achieve this is by grouping our modules into features. Like this:

```
angular.module('app', [
    …
   'app.core','app.routes','app.pubs','app.parks',’app.transport’
]);
```

Core would provide all the core features shared through the application. Routes would handle

the routing functionalities. Pubs, parks and transport are more vertically separated. By breaking the application into modules, we can change the code in one place without affecting the others, of course as long as we don't break the contract of the interface.

Folder structure

As the app grows it becomes more and more important to have a structure that allows easy management and maintenance. No matter how you decide on your structure, it's important to keep your application consistent. When descending on practices like this, you should also always make sure that it works not only for you, but for your team as well. Your folder structure should allow you to immediately know where you can locate your code, without having to search through a complex structure. The code we write should also be kept separately from third party libraries. For our application this will look like this:

```
/app /assets /scripts /tests
```

App will contain our angular code, assets will contain the static assets such as images and styles, unit tests will be inside tests and scripts will contain our third party codes. Another common question in Angular is: Should the application be structured by type or by feature? By type is pretty common in tutorials, and looks like this:

```
App/ Controllers/ Services/ Views/
```

But as your application grows larger, this quickly becomes very hard to maintain. While, if we organize by feature, it will look like this:

```
App/ /core /pubs /parks /transport
```

This works well for larger apps and you won't spend as much time looking for modules, because the controller will be right next to the view, and this structure will form modules more easily. To help us out with the structure we can use the LIFT principles, introduced by John Papa, which also help us think about the why. LIFT stands for:

- **L** – **Locating** our code is easy
- **I** – **Identify** the code at a glance, when I look at file I should know what it contains.
- **F** – **Flat** structure as long as we can. Don't make the file structure too deep, and between 3 and 7 files consider a new folder.
- **T** – (**T-DRY**) **Try** to stay DRY (do not repeat yourself).

Readable Code

If we want the refactoring to be as efficient as possible, it's important to have readable code.

What's inside our code matters just as much as the application structure.

Naming conventions

One possibility, recommended by John Papa, for naming controllers would be with Pascal casing, i.e.: Pubs. This way you always know that a controller starts with big letters. The component names should be descriptive, so we don't need to think about what they are and void generic names like utility. In this case we could make a distinction between controllers and everything else by using Camel case for everything that's not a controller: i.e. appRouting, appLogging, dataService Another possibility would be using suffixes, this way the names are more descriptive but they also become longer. So we would add Controller for controllers and Factory for factories e.g. PubsController, TransportController etc.

Declaring the module dependencies

Modular applications are easy to extend and maintain, they allow the development team to build vertical slices of applications and roll out incrementally. They are feature areas, and help us break our features into modules. They also help us follow the Separation of Concerns pattern. A module can be many things: an app, widgets, services, factories or any of these. Modules can be dependent on other modules, and they usually are. They promote continuous development, so we can iterate through and add features as we go. Modules are containers of angular components. We group our modules in three categories:

- AngularJS modules (ngRoute, ngAnimate)
- 3rd party modules (kendo, breeze etc.) – modules that someone else wrote
- Custom modules – the ones we wrote
  Because modules often rely on other modules, e.g. on helper modules, categorizing then can help us focus on the ones we need. Module Dependency strategies – there are two ways we can declare our dependencies:

- Define dependencies for each module in each module – this will always work and it's fairly common, but it may be difficult to follow, so this one is not recommended.
- Define dependencies at specific levels – this requires some forethought about organizing, but it's easier to maintain
  We should also keep in mind that feature modules shouldn't depend on each other. If there are shared services/directives/filters that are used by multiple modules, they should be extracted into a service module.

Defining Modules

There are many ways to define modules in Angular.

- Module variables: defining our modules as variable can be helpful if we want to use it in other places, but this will be creating global variables and that's not good, because it can easily lead to conflicts.

```
var app = angular.module('app');
app.controller('pubs', function() {
    var vm = this;
    vm.title = "Pubs";
});
```

- Modules chaining: defining modules with chaining can help us avoid global variables. E.g. app, app.services, app.log etc.

```
angular
.module('app')
.controller('Pubs', function() {
   var vm = this;
   var.title = 'Pubs';
});
```

- IFFE (Immediately Invoked Function Expression) – this is a commonly used convention to avoid global variables. This way we wrap our code into a function, and execute it immediately, while keeping variables out of the global scope.

```
(function() {
   angular
      .module('app')
          .controller('Pubs', Pubs);

              function Pubs() {
                   var vm = this;
                   var.title = 'Pubs';
             }
})();
```

Functions Naming

Anonymous functions – these are perfectly fine in small examples and are easy read, and if the function is going to be used in one place Only. Named functions – should be used for more complex functions, because they are easier to read and reuse.

Safe code minifiactions

Minification tools like **uglify** use different techniques to make our code smaller. One of these techniques is shortening parameter names, e.g. the controller and the factory names will be rammed to some letter like: *a* or *b.* This can lead to Angular not finding the dependencies with these names. A common way to avoid this is by adding our dependencies into a string array. So the first string will be matched with the first parameter, the second with the second etc. The code below shows the string array to make our code minifaction safe.

```
 angular.module('app.pubs', [])
.controller('Pubs',['dataservice',Pubs]);
     function Pubs(dataservice) {
…
}
```

Patterns for registering, injecting and defining components

Again there are many ways to do this, you should choose the one that works best for you.

1) Register and Inject, then Function
One of the most common. This is ok, but the array can get long and hard to read.

```
angular
 .module('app.pubs', [])
 .controller('Pubs',['core’,’dataservice',Pubs]);
       function Pubs(core, dataservice) {
 //controller goes here
}
```

2) Function, Inject, Register
Another option. $inject keeps the injection separate and helps avoid ugly array of injectables. But if the function is long, you’ll have to scroll to see the injections, and this may lead to dependencies being out of sequence.

```
function Pubs(core, dataservice) {
 //controller goes here
}
Pubs.$inject = ['core’,’dataservice']);
angular
 .module('app.pubs', [])
 .controller('Pubs',Pubs]);
```

For controllers and services it helps if we can see what the code uses on top, and the implementation below that. This can be achieved easily with the following patterns:

3) Register, Inject, Function
This is the third option, this way we can see the injection list right next to the controller’s Parameters.

```
angular
 .module('app.pubs', [])
 .controller('Pubs',Pubs]);
       Pubs.$inject = ['core’,’dataservice'];
       function Pubs(core, dataservice) {
 //controller goes here
}
```

4) Inject Later with ngAnnotate:
We can use this with automation tools like *Gulp*. This injects the dependencies just by adding the @ngInject Annotation.

```
angular
 .module('app.pubs', [])
 .controller('Pubs',Pubs]);
       /* @ngInject */
       function Pubs(dataservice) {
 //controller goes here
}
```

Nested Controllers

Nested controllers can cause problems if we don’t know how to reference our variables. There are a couple of ways to deal with this.

1) Dot Notations

It’s a good practice to use dots to provide context for nested controllers to provide clarity. It’s easier to provide the parent scope and removes unexpected behavior. It’s a good practice to have a base model (in this case the *vm*), to reference our variables.

```
<div ng-controller="City">
    <label>City name: {{vm.name}}</label>

    <div ng-controller="Hotel">
        <label>Hotel name: {{vm.name}}</label>
    </div>
 </div>

City.$inject = ['$scope'];
    function City($scope) {
           $scope.vm = {
               name: 'Belgrade'
           };
    }

Hotel.$inject = ['$scope'];
      function Hotel($scope) {
         $scope.vm = {
             name: 'Hotel Moskva'
        };
      }
```

2) Controller-as notation

If we don’t want to inject the $scope explicitly in to the controller, we can also use *controller-as* in the router, to add variables to the current scope.

```
<div ng-controller="City as vm">
    <label>City name: {{vm.name}}</label>

    <div ng-controller="Hotel as vm">
         <label>Hotel name: {{vm.name}}</label>
    </div>
 </div>

 function City() {
    var vm = this;
    vm.name = 'Belgrade';
 }

 function Hotel() {
     var vm = this;
     vm.name = 'Hotel Moskva';
 }
```

Common mistakes

1) Confusing Services and Factory

In order to have good understanding of this, we need to know how AngularJS works. In Angular apps most of the objects are instantiated and wired together automatically by the injector service. The injector creates two types of objects: **services** and **specialized objects**. Services are objects whose API is defined by the developer writing the service. Specialized objects conform to a specific Angular framework API, that is: controller, directive, filter or animation. In order for the injector to know how to create and wire together all of these objects, it needs a registry of “recipes”. Each recipe has an identifier of the object and the description of how to create this object. There are 5 types of recipes an injector can use: Provider, Value, Factory, Service and Constant. Now, to stick to the point, I will only explain the Service and the Factory.

Service

Service is suitable for cases when we have a ready function that we need to share throughout the app without any “preprocessing”. The Service recipe produces a service just like the Value or Factory recipes, but it does so by *invoking a constructor with the new operator*.

Factory

Factory allows us to add some logic before creating the object we require The Factory recipe

constructs a new service using a function with zero or more arguments. The return value of this function is the service instance created by this recipe. We should keep in mind that the Factory recipe can create a service of any type, whether it be a primitive, object literal, function, or even an instance of a custom type. Note: All services in Angular are singletons. That means that the injector uses each recipe at most once to create the object. The injector then caches the reference for all future needs.

2) Too many watchers

If we don't pay attention to the data binding, the page can run into some performance issues, when we reach around 2000 watchers can cause noticeable performance issues. Sometimes we really need to bind our data using watchers, especially for SPA because the data is updated in real time. In other cases we can have data that needs to be rendered only once because they are immutable so you shouldn't keep watching them for changes. There are several ways to solve this: One way would be is just to pay attention to the binding and use the controller's link function to update the UI. But there is still one problem that remains, and that is *the existence of the data when the directive renders the content.* Usually the directives, unless you use watchers or bind their attributes to the scope (still a watcher), render the content when they are loaded into the markup, but if at that given time your data is not available, the directive can't render it. As of AngularJS 1.3 we can use the Angular's native *bind once* to handle this. To use it we just need to prefix our bound variable with :

```
<tr ng-repeat="pub in ::vm.pubs">
   <td>{{pub.name}}</td>
 </tr>
```

3) Using jQuery

jQuery has many great features, but the way it should be used does not align with AngularJS. AngularJS is a framework for building applications, while jQuery is a library for simplifying the use of JavaScript. AngularJS is a about the architecture of the applications. The DOM manipulations should be done using directives, and always consider what features Angular provides before reaching for a jQuery library. If there is no way around using a jQuery plugin in your application, create a directive and put your code inside the *link* function. Do not put the code inside a Controller.

Next steps

This was just a brief overview to get you started; for further readings I recommend John Papa's best practices (https://github.com/johnpapa/angular-styleguide) and Todd Motto's style guide (http://toddmotto.com/opinionated-angular-js-styleguide-for-teams/). Also keep in mind that none of the above will get you far if you don't have a strong JavaScript foundation,

so if you are not sure about how certain things work, build on your foundations to get a better understanding of what's happening underneath, make sure you understand how the scope works in vanilla JavaScript, its different from classic OOP languages, behavior of the *this* property, the event queue, hoisting, all of this will help you a lot.