Angular.js Recepies: viewing around.

> *This practical guide will help you avoid mistakes in construction of building complex architecture and more readable code.*

> *Here you can find the often used tricks, which compliance certainly would help you in real projects.*

> *Document is periodically updated, so you can always read and download the latest version here: https://github.com/asduser/angularjs-recepies.*

> *Made by @asduser - https://github.com/asduser.*

---

1. Decrease the HTML confusion.

Description:

As you know, HTML is a hypertext markup language, not a programming and not a some application logical core. It was invented to solve an issues of UI representation.

But with the release the Angular.js, many beginner developers have written a lot of critical logic inside html-blocks. Why it can be

dangerous?

1. You can’t to debug that code properly. Variables, expressions, functional methods and the sets of data - all of that may be ignored in debugger mode.
2. Each new JavaScript code inside HTML significantly makes worse the common code readability. Over time you will find it difficult to support that “view”.
3. In most cases this approach prevents the code reuse, because each block has been closely related with a corresponding application logic.

Solution:

Try decrease the javascript uses inside html blocks. Do it so often as you can. It may cause some discomfort (you have to take care about application architecture), but after a time you reach a success.

2. Dont forget about ng-include.

Angular.js provides a powerful methodology of using html-templates and ignoring them - a real offence. If html page has a lot of code, the good practice is a dividing it to the smaller functional views. As a consequence, code editing will be a much easier and readable.

```
<!-- Instead of: -->
```

```
<div id="tabs">
  <div id="tab-1"> ... </div>
  <div id="tab-2"> ... </div>
</div>

<!-- Use: -->

<div id="tabs">
  <div ng-include=" 'tabs/my-tab1.html' "> ... </div>
  <div ng-include=" 'tabs/my-tab2.html' "> ... </div>
</div>
```

Now, to make changes enough modify *my-tab1.html* or *my-tab2.html* which are located in appropriate directory.

3. Explicit named variables.

Dont try mislead your teammates or another developers, who will support you code, they have enough problems. On the contrary, help them to read each line of code like a sentences of adventure book.

For example:

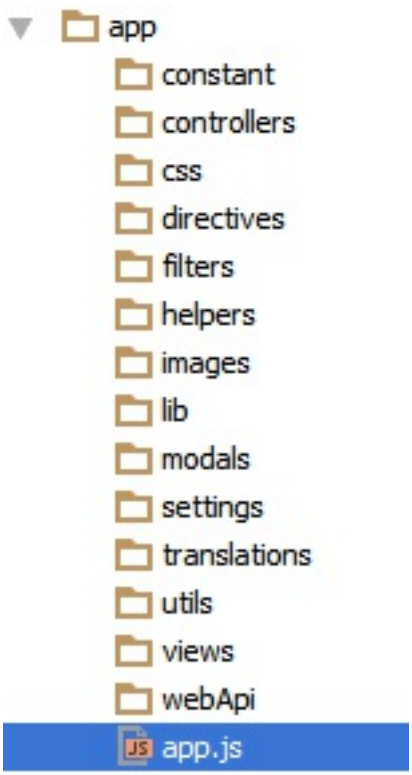
```
<!-- #1 -->
<div ng-hide="!isVisible" class="container"></div>
```

```
<!-- #2 -->
<div ng-show="isVisible" class="container"></div>
```

At the second example you clearly understand what does the code. First example provides you a logic of double negation. It’s so easy sample, but the each complication in code will make understanding more difficult (e.g ternary operator with a multiple sequence).

4. Project architecture.

There are a lot of different cases how to develop an internal application levels, but I would like to share you my own solution which allows to create a flexible and scalable structure in the future.



In app.js declare the common modules, dependencies and the master module. It’s looks like this:

```
angular.module("app", [

  "app.constant",
  "app.filters",
  "app.helpers",
  "app.modals",
  "_webApi_",
  "app.utils",

  // another dependencies

]);
```

Now let’s talk about project architecture above. First of all, a main module was declared that means all another components have to be injected into it.

There is a naming methodology for native application modules such as “filters, helpers, utils” etc. According to this principle we certainly prevent the presence of suspicious or undesirable components in project.

Explanation:

- **Constant**: contains all static variables for the common entities.
- **Controllers**: project controllers store with 2-level nesting, for example: *controllers -> login -> LoginController.js*, *controllers -> login -> ForgotPasswordController.js*.
- **Css**: styles for different .html representation.
- **Directives**: angular.js directives store with 2-level nesting, for example -> *directives -> form-validator -> formValidator.js*.
- **Filters**: set of filters.
- **Helpers**: a special components to work with controllers (see below).
- **Translations**: different languages for the current application.
- **Images**: graphic objects, .svg, .gif etc.
- **Lib**: external libraries (see below).
- **Modals**: interactive user windows to expose some additional functionality.
- **Repositories**: internal entities to interact with webApi component.
- **Settings**: project configuration file (.js).
- **Utils**: set of specific services, factories which are independent from the current project structure and may bi included into another project.
- **Views**: .html views. Desirable to use an analogical level nesting which is described in controllers directory. For example: we have *controller -> food -> FoodController*. So, a good solution is creating a directory *views -> food -> main.html*, where will be

displayed all existing items. To add a new .html page, which is responsible for some Food actions, just add it *into views -> food -> editFood.html*.

- **webApi**: $http requests manager.

You may be sure you will save a lot of time and there is no need for finding a concrete method not knowing where the parent file is exactly located.

5. Designation Components.

It is a good approach to use the single modules for each functional set of entities due to code modularity, structuration, a single component responsibility.

As for declaring component dependencies, so I recommend you work with “IoC container” like this:

```
// Module.js file in repositories directory.
angular
  .module("app.repositories", []);

// Some repository is here.
angular
  .module("app.repositories")
  .factory("repositories.Food", foodRepository);
```

```
function foodRepository(appSettings){

  // logic is here...

}

// IoC container.
foodRepository.$inject = [
  "constant.APP_SETTINGS"
];
```

“IoC container” is responsible for external dependencies, functional components and another entities inside a current item.

6. Use filters.

If your application uses a lot of different conversions, expression, values translation or smth else - just use a filters. It makes your code clearly, shorter and easy to support. Any time a specific block of code could be changed, modified or absolutely replaced by another logic.

```
<div ng-app="myApp">

<div ng-controller="MyCtrl">
  <ul>
```

```
  <li ng-repeat="u in users"> {{ u | userinfo }}
  </ul>
</div>

</div>
```

```
var myApp = angular.module('myApp',[]);

myApp.controller("MyCtrl", function($scope){

$scope.users = [
    {name: "Bob", age: 20},
  {name: "John", age: 25}
];

});

myApp.filter("userinfo", function(){
    return function(item) {
    return "My name is " + item.name + ". I am " + item.age
;
  }
});
```

Result is:

```
My name is Bob. I am 20
My name is John. I am 25
```

7. Controller logic division.

Controller is your place where you may manage your data, invoke an appropriate methods and use different factories, services etc. One of the benefits of using multiple controllers is a lack of code duplication. Each repeating block of logic recommended to transfer into a suitable entity.

```
  angular
    .module("app")
    .service("dateConverter", function(){
      // some code...
    });
```

You have to think out your further actions immediately before creating a new service or factory, because there are independent controls, which may be injected into a special controller to manage the data. Each service or factory responds only for one goal. That principle helps to create more powerful and scalable entities.

```
// Example #1

```

```
app.controller("UserController",  ["$scope", "imageService"
,
  function($scope, imageService){

    // ....

    $scope.getAvatar = function(url){
      return imageService.get(url);
    };

    // Returns default image url.
    var image1 = $scope.getAvatar(null);

  }]);
```

We dont need the details what the imageService does, which actions and operations are exist there. You have to understand a one thing - the imageService will return a valid link or default image url (which is determined by a constant type).

But dont rush to write a code and clone a bunch of services, which won't be a unique entity or responds only for one JavaScript method. Service or factory is a set of methods, which have a similar behaviour and exist as a different plain levels of big complicated task.

8. Controller naming.

Try use postfix Controller instead a famous Ctrl. It does titles more readable and looks like a completed word.

```
app.controller("PlayerController", ["$scope", function($sco
pe){

//some code ...

}]
```

9. Helpers.

Description:

If two or more controllers have a resembling blocks of code, methods here is a solution: create a service\factory which performs within itself all of that logic. Helpers aren’t like a utils, services a factories like you used to see them. There are a special entities, which implement only a specific set of methods for one or more controllers.

For example, here is a FoodController where user may order food, see the current food directory, rank them or just browse a chemical composition of famous items.

Quite possibly will be the case when it will be necessary to covnert one

value into another (ex. *kilograms* (kg) to *pounds* (lb), *liters* (l) to *milliliters* (mL) etc.). Create a single factory or service isn't a best decision, because service is responsible for a specific functionality and may be injected into another Angular.js entity.

What do you think, is it up to our expectations? There are a several disadvantages of using such services:

1. The service will be very common and has a lot of unnecessary functions.
2. Inability division of existing functionality into single entities, because there are a lot of codependent JavaScript blocks.
3. With increasing the size of application will increase the number of methods.

Solution:

Create a single service\factory which will depend on a specific controller(s), but implement some logic inside without duplicating throughout the application.

```
app.service("foodConversionService", function(){

//some conversion methods...

});
```

And here is a good place to work with weight transformation, changing the liquid values and other physical and chemical food properties.

10. Useful directives.

Often we need to pass some data into a specific entity to gain a some functional html-code. And here is a solution - Angular.js directives. But remember, each component consumes a certain amount operating memory, moreover it exponentially loads the browser.

There is a reason why we need to use a directives deliberately. Use derectives only for use - it’s a main mistake of beginners Angular.js developers, cause in many times we can do the usual template without a specific logic and postrenderring.

- Think twice, before you’ll use some component.
- Use directives in places repetition code blocks.
- Be careful when you work with graphic elements, try simplify an existing code as much as it possible.

11. Constants.

For a qualitative interation a several components we may take care about the process of data representation. Don’t forget that a truly scalable application shouldn’t depend on the variables. As a result, use a special Angular.js component - constant.

```
app.constant("userDetails", {

    "GENDER": {
      "0": "Female",
      "1": "Male"
    },

    "MIN_AGE": 18

});
```

Here we can see a special *“userDetails”* constant, which contains the additional information about our application visitors. To add a new property or modify an existing - just enough to change it in one location, there is no need search for it throughout the whole application.

A good practive to use constants for the complicated block of logic, repeating methods, codependent components etc.

Initial application settings, options for different components should be included into suitable constant files too. It prevents a code duplication and significantly simplifies an ability to support current application in the future.

```
app.constant("APP_SETTINGS", {
```

```
    "START_PAGE": "app.dashboard",
    "REDIRECT_UNAUTHORIZED": "app.login",
    // another variables ...

});
```

Thereafter quite enough refer to the existing constant variable from code for calling some action. For example:

```
if (user.role != "Anonymous") {
  $state.go(APP_SETTINGS.START_PAGE);
} else {
  $state.go(APP_SETTINGS.REDIRECT_UNAUTHORIZED);
}
```

The process of variable value changing includes:

1. Openning a specific constant file.
2. Setting a new value to variable.

There is no need to find all existing variables within application, which significantly saves the time and simplifies a common work.

Better to spend more time thinking about the additional constant type, than have only wasting time to find and fix values in the future.

12. Utils.

Description:

As mentioned above, the repeating code should be involved into a service\factory to it reusing. It a good practice create such a functional modules, which may be transferred into another project without any difficulties.

Supposably, there is a service to wrk with dates: create, filter, convert into appropriate formats, use a special mask etc. But here is a one often repeated question - some block of our service depends on the current application structure.

```
app.service("dateManager", ["constant.countries", "someFact
ory", function(countries, someFactory){

// Don't do this inside a service!
$("#some-field").datepicker();

// some code...

}]);
```

Static jQuery #id handling, using another modules o factories to exapose some additional functionality, injection of constant files and many other reasons due to which the service cann’t be called a “reusable”.

Solution:

To solve that issue try modify an existing service as much as it possible:

- Get rid of extra dependencies.
- No need to use an internal constant files inside a “utils services\factories”.
- Don’t use the static jQuery element handling.

But there are cases where the use of additional files to ensure correct operation of the module is necessary. Anyway, try make “utils” service universal.

13. Libraries.

External libraries, modules and minified files should be located into a special application directory - lib. There are only libraries, which are not associated with our app.

For example: jQuery, D3.js, Angular.js etc.

14. Modals.

Often a modern Angular.js applications use modal windows for user interaction, which is especially useful when exist a lot of different dynamic interfaces. *For example:* content filters, confirmations, alerts, run-time installers etc.

To simplify the main logic and prevent code duplication inside each of that modal window, include a special “ModalManager” where exist a common functional methods.

```
// here is some actions within ModalManager controller.

$scope.openFoodDetailsModal = function(size) {

    var modalInstance = $modal.open({
        templateUrl: 'source to .html',
        controller: 'FoodDetailsController',
        size: size,
        resolve: {
            numberCollection: function () {
                return [1, 2, 3, 4, 5];
            }
        }
    });
```

```
    modalInstance.result.then(function (response) {
        // some actions with response...
    });

};
```

As you see, we used a special component to manage our modals. At this point it doesn't matter, but it is important to understand how controllers may interact between themselves.

"openFoodDetailsModal" method render a new window, which will retrieve specified *"FoodDetailsController"* and some data when it initializes. The last one contains in "resolve" field and it is a plain object, each field of which is a specific data type.

To inject more different values enough add a new files, ex.:

```
resolve: {
            numberCollection: function () {
                return [1, 2, 3, 4, 5];
            },
            user: $scope.currentUser
        }
```

Moreover, you may transmit into it a current controller variables, collections, methods or even $rootScope parameters. Each new dependency injection passes throughout ModalManager and there is no need declare many different controllers, services only for delivering some data into specific modal windows.

15. Application Settings.

A lot of Angular.js project have internal data, which declares in many places, complicating the support and modules expansion. It is not just about common constant files (gender, countries, colors). For example, we have an application “TITLE” parameter.

```
app.constant("APP_SETTINGS", {
  "TITLE": "Food Recepies"
});
```

Inject that constant in controller, service, factory is very simple and trivial task, but what about variables definition for using them within html-code? Try do this:

1. Include *“APP_SETTINGS”* constant into $rootScope controller.
2. Declare a suitable variable, which will describe internal

application settings like this: $rootScope.appConfig = “APP_SETTINGS”.

And declare it inside “Header”, “Login Page” and “Forgot Password Page”:

```
<span> {{ appConfig.TITLE }} </span>
```

Now we may change the title without any worrying. Furthermore, each new developer who will work with your code, dont need where is exactly uses that parameter, he have to know just a location for “APP_SETTINGS” constant. Usually, it is a directory “settigns” in main “app” folder.

16. webApi module.

Nowadays virtually all Angular.js project have a close relationship with server-side to manage the data. Authorization, getting some data for tables creating, update something or even administrate mobile apps - all of this depend on a stable and well thought out functionality.

It should be a clear division of logic for web Api module, because some block would be changing very often in the early stages of development, but another - not.

Unlikely we need often change the basic types of database queries ($http.put, $http.get, $http.post, $http.delete etc.), thus all of these components should be located into a “core” module. It rarely changes, because the fundamental logic described there and it is no need to make changes every time when you add a new http-request.

At the section *“Useful links”* you can **read and download** the RESTful webApi module with settings, core functionality, a set of special query formatters and filters.

Notice: webApi module shouldn’t know about your internal data, relationships or smth else, because it has been included only for sending queries to server-side and getting an appropriate responses. It is a really big trouble when somebody declares $http methods inside controllers and specifies a lot of data there.

17. Repositories.

In continuation of the previous paragraph, now we’ll talk about entities which handle $http requests and do some actions after successful or failure response. But, this components don’t know nothing about internal data too.

Repository is responsible for:

- Sending http-requests with specified params.
- Implementation response functionality and notify controller.

- Manage all internal error requests and different exceptions.

```
// repository declaration above...
function getAll(){
  return webApi.getAllFoodItems();
}
```

In this example showed how repository may interact with webApi module without any data or logic disclosing. This approach helps to save the components encapsulation and allows to work at the higher abstraction level.

Recommended to use in repositories for methods without returning a flexible exception handlers, which certainly allow code much easier to read.

```
function update(id){
  webApi.updateFoodById({
    "url": { "id": id }
  }).success(function(response){
    checkResponse(response, "Updated!");
  });
}
```

```
// Exception handler.
function checkResponse(response, message) {
  if (response && response.data) {
    // Case #1
    // Example: alert(message);
  } else {
    // Case #2
    // Example: display default message with text from resp
onse or constant file.
  }
}
```

Explanation:

1. In first case enough to manage an existing response and show on UI a specific message. It doesn't matter which service obtains and processes a derived message at the moment.
2. In next one we have to write some logic for catching the exceptions and show information from internal constant file if necessary.

To improve an existing code and implementation the encapsulation principles let's use a special *"Message Service",* because many repositories may contain similar method "checkReponse()".

```
// Code in repository.
```

```
function update(id){
  webApi.updateFoodById({
    "url": { "id": id }
  }).success(function(response){
    messageService.checkResponse(response, "Updated!");
  });
}

// Code in some service.
this.checkResponse = function(response, message) {
  // Internal implementation.
}
```

18. Useful links.

- https://github.com/asduser/webApi-angularjs - webApi RESTful module. *Powerful and flexible tool to work with $http*.
- https://github.com/asduser/form-validator - html forms validator. *A special directive which provides functionality to work with html forms*.
- https://github.com/asduser/ui-notifications - notifcication service. *Handles a different UI messages*.
- https://github.com/asduser/storage-manager-anguarljs - storage manager. *Service to manage data storages on UI-side*.
- https://github.com/asduser/grunt-automation-example - grunt automation tool. *How to set up task manager to simplify the*

process of development Angular.js project.