Una única responsabilidad: código ravioli vs. código espagueti

Cada archivo del proyecto tiene que tener una única responsabilidad, sin importar el número de líneas que pueda tener. Cada responsabilidad hay que encapsularla y aislarla para conseguir código ravioli.

Aunque, en un principio, tengamos la tentación de unificar varias funcionalidades con la excusa de que son pocas líneas, es muy importante empezar a separar el código desde el principio. A medida de que el proyecto vaya creciendo, aumentarán las  líneas de código y si no seguimos esta buena práctica, nos encontraremos con código espagueti.

La diferencia entre un código espagueti y uno ravioli se ve al hacer cambios en el código. En el espagueti, esa parte del código puede estar “enredado” con otras funcionalidades y provocar mal funcionamientos o conflictos indeseados en otras partes del código. Sin embargo, en un código ravioli al estar todo empaquetado individualmente, cada cambio afectaría únicamente a esa parte; sin provocar daños colaterales.

Por ello, evitaremos el siguiente código:

*angular*

*.module('app', ['ui.router'])*

*.controller('SomeController', SomeController)*

*.factory('someFactory', someFactory);*

*function SomeController() { }*

*function someFactory() { }*

Y lo organizaremos de este modo:

|  |  |
| --- | --- |
| *1*  *2* | *angular*  *.module('app', ['ui.router']);* |

*angular*

*.module('app')*

*.controller('SomeController', SomeController);*

*function SomeController() { }*

*angular*

*.module('app')*

*.factory('someFactory', someFactory);*

*function someFactory() { }*

### IIFE (Immediately Invoked Function Expression)

**Es una buena práctica empaquetar los componentes en funciones que se invocan inmediatamente**. De ese modo, evitamos que las variables y funciones estén más tiempo de lo esperado en el global scope, al mismo tiempo que evitamos posibles colisiones entre variables.

Por lo tanto evitaremos este código:

*angular*

*.module('app')*

*.factory('logger', logger);*

*// la función logger es añadida como variable global*

*function logger() { }*

Y usaremos el siguiente código:

*(function() {*

*'use strict';*

*angular*

*.module('app')*

*.factory('logger', logger);*

*function logger() {}*

*})();*

Para no alargar las líneas de código, en los siguientes puntos se omitirá esta buena práctica.

### Evitar colisiones de nombres

Para evitar la colisión de nombres, utilizaremos una convención de nombres única con separadores para los sub-módulos. Además el uso de separadores ayuda a definir los módulos y sus jerarquías.

Por ejemplo, **app** puede ser nuestro módulo raíz y **app.dashboard** y **app.users** pueden ser módulos que dependen de él.

### Módulos

Declararemos los módulos sin asignarlos a una variable y lo haremos utilizando la sintaxis de los setters.

Por lo tanto, evitaremos esto:

*var app = angular.module('app', [*

*'ngAnimate',*

*'ui.router',*

*'app.shared'*

*]);*

Y utilizaremos lo siguiente:

*angular*

*.module('app', [*

*'ngAnimate',*

*'ui.router',*

*'app.shared'*

*]);*

Usaremos **angular.module(‘app’, [])** para declarar un módulo y especificar sus dependencias (sintaxis setter) y **angular.module(‘app’)** para recuperar un módulo (sintaxis getter).

### Controller as vm

Utilizaremos la sintaxis controller as en vez del clásico controller con $scope y capturaremos el this en una variable llamada vm (ViewModel).

Este método nos ayuda a prevenir el uso excesivo del $scope, ya que en algunos casos suele ser muy tentador utilizar dicha variable para declarar funciones que deberían de estar delegados a un servicio o factoría. El $scope se debe utilizar únicamente cuando es necesario; por ejemplo, al llamar a eventos utilizando $emit, $broadcast o $on.

Por lo tanto evitaremos este código:

*function Customer() {*

*this.name = 'Irontec';*

*this.sendMessage = function() {};*

*}*

Sustituyéndolo por el siguiente código:

*function Customer() {*

*var vm = this;*

*vm.name = 'Irontec';*

*vm.sendMessage = function() {};*

*}*

*<input ng-model="vm.name"/>*

### Funciones anónimas vs funciones declaradas

En la medida de lo posible utilizaremos funciones declaradas, ya que el código será más limpio, más fácil de debuggear y reducirá la cantidad de llamadas anidadas.

Por lo tanto, en vez de utilizar este código:

*angular*

*.module('app')*

*.controller('DashboardController', function() { })*

*.factory('logger', function() { });*

*Y lo organizaremos de la siguiente manera:*

*angular*

*.module('app')*

*.controller('DashboardController', DashboardController);*

*function DashboardController() { }*

*angular*

*.module('app')*

*.factory('logger', logger);*

*function logger() { }*

### Asociaciones en la parte superior del controlador

### Colocaremos las asociaciones en la parte superior del controlador; facilita la lectura y ayuda a identificar las variables asociadas a la vista.

Evitaremos este tipo de orden:

*function Sessions() {*

*var vm = this;*

*vm.gotoSession = function() {*

*/\* ... \*/*

*};*

*vm.sessions = [];*

*vm.title = 'Sessions';*

Y ordenaremos los controladores de la siguiente manera. A poder ser agrupando primero todas las variables y a continuación todas las funciones.

*function Sessions() {*

*var vm = this;*

*// Variables*

*vm.sessions = [];*

*vm.title = 'Sessions';*

*// Funciones*

*vm.gotoSession = gotoSession;*

*////////////*

*function gotoSession() {*

*/\* ... \*/*

*}*

### Delegar la lógica dentro de los controladores a servicios o factorías

Es recomendable delegar la lógica de los controladores a servicios o factorías para reutilizar la lógica empleada. Además se puede aislar en un test unitario y elimina dependencias a la vez que esconde detalles de la implementación.

Por lo tanto, en vez de programar la lógica en el controller.

*function OrderController($http, $q, config, userInfo) {*

*var vm = this;*

*vm.isCreditOk;*

*vm.checkCredit = checkCredit;*

*/////////////////*

*function checkCredit() {*

*var settings = {/\* ... \*/};*

*return $http.get(settings)*

*.then(function(data) {*

*vm.isCreditOk = vm.total <= maxRemainingAmount*

*})*

*.catch(function(error) {*

*// Gestionar el error*

*});*

*};*

*}*

Delegaremos esa función a un servicio:

*function OrderController(creditService) {*

*var vm = this;*

*vm.isCreditOk;*

*vm.total = 0;*

*vm.checkCredit = checkCredit;*

*///////////////////*

*function checkCredit() {*

*return creditService.isOrderTotalOk(vm.total)*

*.then(function(isOk) { vm.isCreditOk = isOk; })*

*.catch(showError);*

*};*

*}*

### Utiliza snippets

Por último, cabe destacar que existen snippets**basados en estas buenas prácticas** que agilizan muchísimo el desarrollo de aplicaciones . Para utilizar dichos snippets en Atom, basta con ejecutar el siguiente comando:

|  |  |
| --- | --- |
| *1* | *apm install angularjs-styleguide-snippets* |

Los *snippets* incluidos en ese paquete son los siguientes:

* ngcontroller // crea un controlador de Angular
* ngdirective // crea una directiva de Angular
* ngfactory // crea una factoría de Angular
* ngmodule // crea un módulode Angular
* ngservice // crea un servicio de Angular
* ngfilter // crea un filtro de Angular