Buenas Prácticas de AngularJS

Single Responsibility

Definir 1 componente por archivo.

El siguiente ejemplo define el modulo app y sus dependencias, un controlador y un servicio en el mismo archivo.

```
/* avoid */
angular
    .module('app', ['ngRoute'])
    .controller('SomeController' , SomeController)
    .factory('someFactory' , someFactory);

function SomeController() { }

function someFactory() { }
```

Single Responsibility

Los mismos componentes son separados ahora cada uno en su propio archivo.

```
/* recommended */
// app.module.js
angular
    .module('app', ['ngRoute']);

/* recommended */
// someController.js
angular
    .module('app')
    .controller('SomeController' , SomeController);

function SomeController() { }

/* recommended */
 // someFactory.js
angular
    .module('app')
    .factory('someFactory' , someFactory);

function someFactory() { }
```

IIFE – JavaScript Closures

Definir nuestros componentes AngularJS dentro de una Immediately Invoked Function Expression (IIFE).

Por qué?: Las IIFE remueven variables del scope global. Esto evita que funciones y variables vivan más allá de lo necesario o fuera del ámbito donde fueron definidas, como así también se previenen colisiones.

Por qué?: Cuando nuestro código es minificado y ofuscado para ser subido a producción, podemos tener colisiones de variables locales con variables globales. IIFE nos protege de este problema definendo un scope específico para cada archivo.

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

// logger function is added as a global variable
function logger() { }

// storage.js
angular
    .module('app')
    .factory('storage', storage);

// storage function is added as a global variable
function storage() { }
```

IIFE – JavaScript Closures

```
/**
* recommended
*
* no globals are left behind
*/

// logger.js
(function() {
    'use strict';

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

    function logger() { }
})();

// storage.js
(function() {
    'use strict';

    angular
        .module('app')
        .factory('storage', storage);

    function storage() { }
})();
```

Modules

Avoid Naming Collisions

Usar convenciones de nombre con separadores para sub módulos.

Por qué?: Nombres únicos evitan colisiones de módulos por nombre. Separadores ayudan a definir la jerarquía de nuestros módulos y sus submódulos. Por ejemplo app puede ser nuestro modulo principal, luego podemos tener app.dashboard y app.users como submódulos y dependencias de nuestro modulo app.

Modules

Declare módulos sin utilizar una variable sino el método setter.

Por qué?: Con 1 componente por archivo va a ser raramente necesario utilizar una variable para contener nuestro modulo.

```
/* avoid */
var app = angular.module('app', [
    'ngAnimate',
    'ngRoute',
    'app.shared',
    'app.dashboard'
]);
```

Modules

En su lugar usar la sintaxis simple de setter.

```
/* recommended */
angular
      .module('app', [
        'ngAnimate',
        'ngRoute',
        'app.shared',
        'app.dashboard'
    ]);
```

Modules

Getters

Al usar un modulo, evitar usar variables para guardarlos en ellas, en su lugar usar el encadenamiento o sintaxis puntuada provista por la sintaxis más sencilla de getter.

Por qué? : Esto produce código más legible y evita posibles colisiones.

```
/* avoid */
var app = angular.module('app');
app.controller('SomeController' , SomeController);

function SomeController() { }
```

Modules

```
/* recommended */
angular
    .module('app')
    .controller('SomeController' , SomeController);

function SomeController() { }
```

Modules

Setting vs Getting

Sólamente setee una vez y luego haga un get para el resto de las instancias.

Por qué?: Un modulo debe ser creado solo una vez, luego desde ahí ser obtenido la cantidad de veces que sea necesario.

Use angular.module('app', []); to set a module.
Use angular.module('app'); to get a module.

Modules

Named vs Anonymous Functions

Al momento de la creación de módulos use funciones nombradas en lugar de funciones anónimas como callbacks del método de creación.

Por qué?: Esto produce código más legible, es más fácil para debuggear y detectar errores y reduce la cantidad de callbacks anidados.

```
/* avoid */
angular
    .module('app')
    .controller('Dashboard', function() { });
    .factory('logger', function() { });
```

Modules

```
/* recommended */

// dashboard.js
angular
    .module('app')
    .controller('Dashboard', Dashboard);

function Dashboard() { }
// logger.js
angular
    .module('app')
    .factory('logger', logger);

function logger() { }
```