Guía de estilo AngularJS

> Esta es una guía colaborativa sobre sintaxis, convenciones y estructura de aplicaciones con AngularJS.

Tabla de contenidos

Single responsibility o Responsabilidad única

La regla del 1

> Define 1 componente por archivo.

¿Por qué?: Un componente por archivo promueve pruebas unitarias más fáciles.

¿Por qué?: Un componente por archivo hace que sea mucho más fácil de leer, mantener, y evita colisiones con los equipos en el control de código.

¿Por qué?: Un componente por archivo evita errores ocultos que a menudo surgen cuando se combinan componentes en un archivo donde pueden compartir variables, crear closures (clausuras) no deseadas, o acoplamiento indeseado de dependencias.

El siguiente ejemplo define el módulo app y sus dependencias, define un controlador, y defines una factoría todo en el mismo archivo.

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

function SomeController() { }

function someFactory() { }
```

Los mismos componentes están separados en su propio archivo.

```
/* recomendado */

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

```
/* recomendado */

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

function SomeController() { }
```

```
/* recomendado */

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

function someFactory() { }
```

IIFE

Closures de javaScript

> Envuelve los componentes Angular en una expresión de función que se invoca inmediatamente Immediately Invoked Function Expression (IIFE).

¿Por qué?: Una IIFE elimina las variables del scope global. Esto ayuda a prevenir que las variables y las declaraciones de funciones vivan más de lo esperado en el scope global, evitando así colisión de variables.

¿Por qué?: Cuando tu código se minimiza y se empaqueta en un archivo único para desplegar al servidor de producción, podrías tener colisión de variables y muchas variables globales. Una IIFE te protege contra ambos, creando una scope por cada archivo.

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

// La función logger es añadida como variable global
function logger() { }

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

// la función storage es añadida como variable global
function storage() { }
```

```
/**
 * recomendado
 *
 * así no dejamos ninguna variable global
 */

// 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() { }
})();
```

- Nota: Para acortar únicamente, el resto de los ejemplos de esta guía podrían omitir la sintaxis IIFE.
- Nota: IIFE previente que el código de los tests llegue a sus variables privadas, como expresiones regulares o funciones de ayuda que normalmente vienen bien para hacer pruebas por sí solas. Sin embargo, puedes acceder a ellas creando accesorios o accediendo a través de sus componentes. Por ejemplo, poniendo las funciones de ayuda, expresiones regulares o constantes en su propia factoría.

Módulos

Evitando la colisión de nombres

> Usa una convención de nombres única con separadores para los sub-módulos.

¿Por qué?: Nombres únicos ayudan a evitar colisiones en los nombres de módulos. Los separadores ayudan a definir los módulos y la jerarquía de sus sub-módulos. Por ejemplo app puede ser tu módulo raíz y app.dashboard y app.users pueden ser módulos que dependen de app.

Definiciones (aka Setters)

> Declara los módulos sin usar una variable, usando la sintaxis de los setters.

¿Por qué?: Con un componente por archivo, es raro que necesitemos introducir una variable para el módulo.

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

En su lugar usa la sintaxis de los setters

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

Getters

> Al usar un módulo, evita usar una variable y en su lugar usa encadenamiento con la sintaxis de los getter.

¿Por qué?: Esto hace más legible el código y evita que las variables colisionen.

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

function SomeController() { }
```

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

function SomeController() { }
```

Setting vs Getting

> Setea sólo una vez y usa get para el resto de instancias.

¿Por qué?: Un módulo debe ser creado sólo una vez y recuperado desde ese punto.

- Usa 'angular.module('app', []);' para setear un módulo.
- Usa 'angular.module('app');' para recuperar un módulo.

Funciones anónimas vs funciones con nombre

> Usa funciones con nombre en lugar de pasar una función anónima en el callback.

¿Por qué?: Así el código es más legible, es más fácil de debugear, y reduce la cantidad de código anidado en los callbacks.

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

```
/* recomendado */

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

```
function Dashboard() { }
```

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

function logger() { }
```

Controladores

controllerAs sintaxis en la vista

> Usa la sintaxis controllerAs en lugar del clásico controlador con $scope.

¿Por qué?: Los Controladores se construyen, renuevan y proporcionan una nueva instancia única, y la sintaxis controllerAs se acerca más a eso que la sintaxis clásica de $scope.

¿Por qué?: Promueves el uso de binding usando el "." en el objeto dentro de la Vista (ej. customer.name en lugar de name), así es más contextual, fácil de leer y evitas problemas de referencia que pueden aparecer con el "punto".

¿Por qué?: Ayuda a evitar usar $parent en las Vistas con controladores anidados.

```
<!-- evitar -->
<div ng-controller="Customer">
    {{ name }}
</div>
```

```
<!-- recomendado -->
<div ng-controller="Customer as customer">
    {{ customer.name }}
</div>
```

controllerAs sintaxis en el controlador

> Usa la sintaxis controllerAs en lugar del clásico controlador con $scope.

- La sintaxis controllerAs usa this dentro de los controladores que se asocian al $scope.

¿Por qué?: controllerAs es azúcar sintáctico sobre el $scope. Puedes enlazar a la vista y acceder a los métodos del $scope.

¿Por qué?: Ayuda a evitar la tentación de usar los métodos del $scope dentro de un controller cuando debería ser mejor evitar usarlos o moverlos a una factoría. Considera usar $scope en una factory, o en un controlador sólo cuando sea necesario. Por ejemplo cuando publicas y te suscribes a eventos usando $emit, $broadcast, o $on considera mover estos usos a una factoría e invocarlos desde el controlador.

```
/* evitar */
function Customer($scope) {
    $scope.name = {};
    $scope.sendMessage = function() { };
}
```

```
/* recomendado - pero mira la sección siguiente */
function Customer() {
    this.name = {};
    this.sendMessage = function() { };
}
```

controllerAs con vm

> Usa una variable para capturar this cuando uses la sintaxis controllerAs. Elige un nombre de variable consistente como vm, de ViewModel.

¿Por qué?: La palabra this es contextual y cuando es usada dentro de una función en un controlador puede cambiar su contexto. Capturando el contexto de this te evita encontrarte este problema.

```
/* evitar */
function Customer() {
    this.name = {};
    this.sendMessage = function() { };
}
```

```
/* recomendado */
function Customer() {
    var vm = this;
    vm.name = {};
    vm.sendMessage = function() { };
}
```

Nota: Puedes evitar los warnings de jshint escribiendo un comentario encima de la línea de código. Sin embargo no hace falta si el nombre de la función empieza con mayúsculas, ya que esa es la convención para las funciones de los constructores, que es lo que un controller en Angular es.

```
/* jshint validthis: true */
var vm = this;
```

Nota: Cuando crees watchers en un controlador usando controller as, puedes observar la variable vm.* usando la siguiente sintaxis.(Crea los watchers con precaución ya que añaden mucha carga al ciclo de digest)

```
<input ng-model="vm.title"/>
```

```
function SomeController($scope, $log) {
```

```
        var vm = this;
        vm.title = 'Some Title';

        $scope.$watch('vm.title', function(current, original) {
            $log.info('vm.title was %s', original);
            $log.info('vm.title is now %s', current);
        });
    }
```

Miembros bindeables arriba

> Coloca las asociaciones en la parte superior del controlador, ordenalas alfabéticamente y no las distribuyas a lo largo del código del controlador.

¿Por qué?: Colocar las variables asignables arriba hace más fácil la lectura y te ayuda a identificar qué variables del controlador pueden ser asociadas y usadas en la Vista.

¿Por qué?: Setear funciones anónimas puede ser fácil, pero cuando esas funciones tienen más de una línea de código se hace menos legible. Definir las funciones bajo las variables bindeables (las declaraciones de las funciones serán movidas hacia arriba en el proceso de hoisting), hace que los detalles de implementación estén abajo, deja las variables arriba y más sencilla la lectura.

```
    /* evitar */
    function Sessions() {
        var vm = this;

        vm.gotoSession = function() {
          /* ... */
        };
        vm.refresh = function() {
          /* ... */
        };
        vm.search = function() {
          /* ... */
        };
        vm.sessions = [];
        vm.title = 'Sessions';
```

```
    /* recomendado */
    function Sessions() {
        var vm = this;

        vm.gotoSession = gotoSession;
        vm.refresh = refresh;
        vm.search = search;
        vm.sessions = [];
        vm.title = 'Sessions';

        ////////////

        function gotoSession() {
          /* */
        }

        function refresh() {
          /* */
        }
```

```
    function search() {
      /* */
    }
```

Nota: Si la función es de una línea, déjala arriba, siempre y cuando no afecte en la legibilidad.

```
/* evitar */
function Sessions(data) {
    var vm = this;

    vm.gotoSession = gotoSession;
    vm.refresh = function() {
        /**
         * líneas
         * de
         * código
         * que afectan a
         * la legibilidad
         */
    };
    vm.search = search;
    vm.sessions = [];
    vm.title = 'Sessions';
```

```
/* recomendado */
function Sessions(dataservice) {
    var vm = this;

    vm.gotoSession = gotoSession;
    vm.refresh = dataservice.refresh; // 1 liner is OK
    vm.search = search;
    vm.sessions = [];
    vm.title = 'Sessions';
```

Declaraciones de funciones para ocultar los detalles de implementación

> Declara funciones para ocultar detalles de implementación. Mantén las variables bindeables arriba. Cuando necesites bindear una función a un controlador referencia una función que aparezca después en el archivo. Esto está directamente relacionado con la sección: Miembros Bindeables Arriba. Para más detalles mira este post.

¿Por qué?: Colocar las variables bindeables arriba hace más fácil la lectura y te ayuda a identificar qué variables del controlador pueden ser asociadas y usadas en la Vista.

¿Por qué?: Colocar los detalles de implementación de una función al final del archivo deja la complejidad fuera de vista así puedes ver las cosas importantes arriba.

¿Por qué?: La declaración de las funciones son movidas arriba por el proceso de hoisting así que no tenemos que preocuparnos por usar una función antes de que sea definida (como la habría si fueran funciones en forma de expresión).

¿Por qué?: No tendrás que preocuparte de que si pones var a antes de var b se rompa el código porque a dependa de b.

¿Por qué?: El orden es crítico para las funciones en forma de expresión.

```
/**
 * evitar
 * Using function expressions.
 */
function Avengers(dataservice, logger) {
    var vm = this;
    vm.avengers = [];
    vm.title = 'Avengers';

    var activate = function() {
        return getAvengers().then(function() {
            logger.info('Activated Avengers View');
        });
    }

    var getAvengers = function() {
        return dataservice.getAvengers().then(function(data) {
            vm.avengers = data;
            return vm.avengers;
        });
    }

    vm.getAvengers = getAvengers;

    activate();
}
```

Nótese que las cosas importantes están dispersas en el ejemplo anterior. En el siguiente ejemplo, lo importante está arriba. Por ejemplo, las variables asociadas al controlador como vm.avengers y vm.title. Los detalles de implementación están debajo. Así es más fácil de leer.

```
/*
 * recomendado
 * Usando declaraciones de funciones y
 * miembros bindeables arriba.
 */
function Avengers(dataservice, logger) {
    var vm = this;
    vm.avengers = [];
    vm.getAvengers = getAvengers;
    vm.title = 'Avengers';

    activate();

    function activate() {
        return getAvengers().then(function() {
            logger.info('Activated Avengers View');
        });
    }

    function getAvengers() {
        return dataservice.getAvengers().then(function(data) {
            vm.avengers = data;
            return vm.avengers;
        });
    }
}
```

Diferir la lógica del controlador

> Difiere la lógica dentro de un controlador delegándola a servicios y factorías.

¿Por qué?: La lógica podría ser reutilizada por varios controladores cuando la colocas en un servicio y la expones como una función.

¿Por qué?: La lógica en un servicio puede ser aislada en un test unitario, mientras que la lógica de llamadas en un controlador se puede mockear fácilmente.

¿Por qué?: Elimina dependencias y esconde detalles de implementación del controlador.

```
/* evitar */
function Order($http, $q, config, userInfo) {
    var vm = this;
    vm.checkCredit = checkCredit;
    vm.isCreditOk;
    vm.total = 0;

    function checkCredit() {
        var settings = {};
        // Get the credit service base URL from config
        // Set credit service required headers
        // Prepare URL query string or data object with request data
        // Add user-identifying info so service gets the right credit limit for this user.
        // Use JSONP for this browser if it doesn't support CORS
        return $http.get(settings)
            .then(function(data) {
             // Unpack JSON data in the response object
               // to find maxRemainingAmount
               vm.isCreditOk = vm.total <= maxRemainingAmount
            })
            .catch(function(error) {
               // Interpret error
               // Cope w/ timeout? retry? try alternate service?
               // Re-reject with appropriate error for a user to see
            });
    };
}
```

```
/* recomendado */
function Order(creditService) {
    var vm = this;
    vm.checkCredit = checkCredit;
    vm.isCreditOk;
    vm.total = 0;

    function checkCredit() {
       return creditService.isOrderTotalOk(vm.total)
    .then(function(isOk) { vm.isCreditOk = isOk; })
          .catch(showServiceError);
    };
}
```

Mantén tus controladores enfocados

> Define un controlador para una vista, no intentes reutilizar el controlador para otras vistas. En lugar de eso, mueve la lógica que se pueda reutilizar a factorías y deja el controlador simple y enfocado en su vista.

¿Por qué?: Reutilizar controladores con varias vistas es arriesgado y necesitarías buena cobertura de tests end to end (e2e) para asegurar que todo funciona bien en la aplicación.

Asignando controladores

> Cuando un controlador debe ser asociado a una vista y cada componente puede ser reutilizado por otros controladores o vistas, define controladores con sus rutas.

Nota: Si una Vista es cargada por otra además de por la ruta, entonces usa la sintaxis ng-controller="Avengers as vm".

¿Por qué?: Emparejar el controlador en la ruta permite a diferentes rutas invocar diferentes pares de controladores y vistas. Cuando los controladores son asignados en la vista usando ng-controller, esa vista siempre estará asociada al mismo controlador.

```
/* evitar - cuando se use con una ruta y queramos asociarlo dinámicamente */

// route-config.js
angular
    .module('app')
    .config(config);

function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
          templateUrl: 'avengers.html'
        });
}
```

```
<!-- avengers.html -->
<div ng-controller="Avengers as vm">
</div>
```

```
/* recomendado */

// route-config.js
angular
    .module('app')
    .config(config);

function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'Avengers',
            controllerAs: 'vm'
        });
}
```

```
<!-- avengers.html -->
<div>
</div>
```

Servicios

Singletons

> Los Servicios son instanciados con un new, usan this para los métodos públicos y las variables. Ya que son muy similares a las factories, usa una factory en su lugar por consistencia.

Nota: Todos los servicios Angular son singletons. Esto significa que sólo hay una instancia de un servicio por inyector.

```
// service
angular
    .module('app')
    .service('logger', logger);

function logger() {
  this.logError = function(msg) {
    /* */
  };
}
```

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

function logger() {
    return {
        logError: function(msg) {
          /* */
        }
   };
}
```

Factorías

Responsabilidad única

> Las factorías deben tener una responsabilidad única, que es encapsulada por su contexto. Cuando una factoría empiece a exceder el principio de responsabilidad única, una nueva factoría debe ser creada.

Singletons

> Las Factorías son singleton y devuelven un objeto que contiene las variables del servicio.

Nota: Todos los servicios Angular son singletons.

Miembros accesiblesarriba

> Expón las variables que se llaman del servicio (su interfaz) arriba, usando la técnica derivada de Revealing Module Pattern.

¿Por qué?: Colocar los elementos que se llaman arriba hace más fácil la lectura y te ayuda a identificar los elementos del servicio que se pueden llamar y se deben testear (y/o mockear).

¿Por qué?: Es especialmente útil cuando el archivo se hace más largo, ya que ayuda a evitar el scroll para ver qué se expone.

¿Por qué?: Setear las funciones puede ser fácil, pero cuando tienen más de una línea ser reduce la legibilidad. Definiendo la interfaz mueve los detalles de implementación abajo, mantiene la interfaz que va a ser llamada arriba y lo hace más fácil de leer.

```
/* evitar */
function dataService() {
  var someValue = '';
  function save() {
    /* */
  };
  function validate() {
    /* */
  };

  return {
      save: save,
      someValue: someValue,
      validate: validate
  };
}
```

```
/* recomendado */
function dataService() {
    var someValue = '';
    var service = {
        save: save,
        someValue: someValue,
        validate: validate
    };
    return service;

    ////////////

    function save() {
        /* */
    };

    function validate() {
        /* */
    };
}
```

De esta forma se asocian los bindeos desde el objeto que lo mantiene, los valores primitivos no se pueden modificar por si solos usando este patrón

```
(function() {
    'use strict';

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

    logger.$inject = ['$log', 'toastr'];

    function logger($log, toastr) {
        var service = {
            showToasts: true,

            error   : error,
            info    : info,
            success : success,
            warning : warning,

            // straight to console; bypass toastr
            log     : $log.log
        };

        return service;
        ////////////////////

        function error(message, data, title) {
            toastr.error(message, title);
            $log.error('Error: ' + message, data);
        }

        function info(message, data, title) {
```

Declaración de funciones para esconder los detalles de implementación

> Declara funciones para esconder detalles de implementación. Mantén los elementos accesibles en la parte superior de la factoría. Referencia a los que aparezcan después en el archivo. Para más detalles visita este post.

¿Por qué?: Coloca los elementos accesibles en la parte superior para hacerlo más fácil de leer y ayudarte a identificar instantáneamente qué funciones de la factoría se pueden accesar externamente.

¿Por qué?: Colocar los detalles de implementación de una función al final del archivo mueve esa complejidad fuera de la vista, de esta forma puedes dejar lo importante arriba.

¿Por qué?: Las declaraciones de las funciones son "elevadas" de esta forma no hay problemas en usar una función antes de su definición (como la habría si fueran funciones en forma de expresión).

¿Por qué?: No tendrás que preocuparte de que si pones var a antes de var b se rompa el código porque a dependa de b.

¿Por qué?: El orden es crítico para las funciones en forma de expresión

```
/**
 * evitar
 * Usar función como expresión
 */
function dataservice($http, $location, $q, exception, logger) {
    var isPrimed = false;
    var primePromise;

    var getAvengers = function() {
        // detalles de implementación van aquí
    };

    var getAvengerCount = function() {
        // detalles de implementación van aquí
    };

    var getAvengersCast = function() {
        // detalles de implementación van aquí
    };
```

```
    var prime = function() {
        // detalles de implementación van aquí
    };

    var ready = function(nextPromises) {
        // detalles de implementación van aquí
    };

    var service = {
        getAvengersCast: getAvengersCast,
        getAvengerCount: getAvengerCount,
        getAvengers: getAvengers,
        ready: ready
    };

    return service;
}
```

```
/**
 * recomendado
 * Usar declaración de funciones
 * y miembros accesibles arriba
 */
function dataservice($http, $location, $q, exception, logger) {
    var isPrimed = false;
    var primePromise;

    var service = {
        getAvengersCast: getAvengersCast,
        getAvengerCount: getAvengerCount,
        getAvengers: getAvengers,
        ready: ready
    };

    return service;

    ////////////

    function getAvengers() {
        // detalles de implementación van aquí
    }

    function getAvengerCount() {
        // detalles de implementación van aquí
    }

    function getAvengersCast() {
        // detalles de implementación van aquí
    }

    function prime() {
        // detalles de implementación van aquí
    }

    function ready(nextPromises) {
        // detalles de implementación van aquí
    }
}
```

Servicios de datos

Separate data calls

> Refactoriza la lógica para hacer operaciones e interaciones con datos en una factory. Crear data services responsables de las peticiones XHR, local storage, memoria o cualquier otra operación con datos.

¿Por qué?: La responsabilidad del controlador es la de presentar y recoger información para la vista. No debe importarle cómo se consiguen los datos, sólo saber cómo conseguirlos. Separando los datos de servicios movemos la lógica de cómo conseguirlos al servicio de datos, y deja el controlador simple, enfocándose en la vista.

¿Por qué?: Hace más fácil testear (mock o real) las llamadas de datos cuando testeamos un controlador que usa un data service.

¿Por qué?: La implementación del servicio de datos puede tener código muy específico para usar el repositorio de datos. Podría incluir cabeceras, cómo hablar a los datos, u otros servicios como $http. Separando la lógica en servicios de datos encapsulamos la lógica en un único lugar, escondiendo la implementación de sus consumidores externos (quizá un controlador), de esta forma es más fácil cambiar la implementación.

```
/* recomendado */

// dataservice factory
angular
    .module('app.core')
    .factory('dataservice', dataservice);

dataservice.$inject = ['$http', 'logger'];

function dataservice($http, logger) {
    return {
        getAvengers: getAvengers
    };

    function getAvengers() {
        return $http.get('/api/maa')
            .then(getAvengersComplete)
            .catch(getAvengersFailed);

        function getAvengersComplete(response) {
            return response.data.results;
        }

        function getAvengersFailed(error) {
            logger.error('XHR Failed for getAvengers.' + error.data);
        }
    }
}
```

Nota: El servicio de datos es llamado desde los consumidores, como el controlador, escondiendo la implementación del consumidor como se muestra a continuación.

```
/* recomendado */
```

```
// controller llamando a la factory del data service
angular
    .module('app.avengers')
    .controller('Avengers', Avengers);

Avengers.$inject = ['dataservice', 'logger'];

function Avengers(dataservice, logger) {
    var vm = this;
    vm.avengers = [];

    activate();

    function activate() {
        return getAvengers().then(function() {
            logger.info('Activated Avengers View');
        });
    }

    function getAvengers() {
        return dataservice.getAvengers()
            .then(function(data) {
                vm.avengers = data;
                return vm.avengers;
            });
    }
}
```

Devuelve una promesa desde las llamadas a datos

> Cuando llamamos a servicios de datos que devuelven una promesa como $http, devuelve una promesa en la llamada de tu función también.

¿Por qué?: Puedes encadenar promesas y hacer algo cuando la llamada se complete y resuelva o rechace la promesa.

```
/* recomendado */

activate();

function activate() {
    /**
     * Step 1
     * Pide a la función getAvengers por los datos
     * de los vengadores y espera la promesa
     */
    return getAvengers().then(function() {
        /**
         * Step 4
         * Ejecuta una acción cuando se resuelva la promesa final
         */
        logger.info('Activated Avengers View');
    });
}

function getAvengers() {
      /**
       * Step 2
       * Pide al servicio de datos los datos y espera
       * por la promesa
```

```
         */
        return dataservice.getAvengers()
            .then(function(data) {
                /**
                 * Step 3
                 * setea los datos y resuelve la promesa
                 */
                vm.avengers = data;
                return vm.avengers;
        });
    }
```

Directivas

Limitalas a una por archivo

> Crea una directiva por archivo. Llama al archivo como la directiva.

¿Por qué?: Es muy fácil colocar todas las directivas en un archivo, pero será más difícil de partir para ser compartida entre aplicaciones, módulos o para un simple módulo.

¿Por qué?: Una directiva por archivo es fácil de mantener.

```
/* evitar */
/* directives.js */

angular
    .module('app.widgets')

    /* directiva de órdenes que es específica del módulo de órdenes*/
    .directive('orderCalendarRange', orderCalendarRange)

    /* directiva de ventas que puede ser usada en algún otro lado a lo
    largo de la aplicación de ventas */
    .directive('salesCustomerInfo', salesCustomerInfo)

    /* directiva de spinner que puede ser usada a lo largo de las
    aplicaciones */
    .directive('sharedSpinner', sharedSpinner);

function orderCalendarRange() {
    /* detalles de implementación */
}

function salesCustomerInfo() {
    /* detalles de implementación */
}

function sharedSpinner() {
    /* detalles de implementación */
}
```

```
/* recomendado */
/* calendarRange.directive.js */
```

```
  /**
   * @desc directiva de órdenes que es específica al módulo de órdenes en la compañía Acme
   * @example <div acme-order-calendar-range></div>
   */
  angular
      .module('sales.order')
      .directive('acmeOrderCalendarRange', orderCalendarRange);

  function orderCalendarRange() {
      /* detalles de implementación */
  }
```

```
  /* recomendado */
  /* customerInfo.directive.js */

  /**
   * @desc directiva de ventas que puede ser usada a lo largo de la aplicación de ventas en la
compañía Acme
   * @example <div acme-sales-customer-info></div>
   */
  angular
      .module('sales.widgets')
      .directive('acmeSalesCustomerInfo', salesCustomerInfo);

  function salesCustomerInfo() {
      /* detalles de implementación */
  }
```

```
  /* recomendado */
  /* spinner.directive.js */

  /**
   * @desc directiva de spinner que puede ser usada a lo largo de las aplicaciones en la compañía
Acme
   * @example <div acme-shared-spinner></div>
   */
  angular
      .module('shared.widgets')
      .directive('acmeSharedSpinner', sharedSpinner);

  function sharedSpinner() {
      /* detalles de implementación */
  }
```

Nota: Hay muchas formas de llamar a las directivas, especialmente cuando pueden ser usadas en ámbitos específicos. Elige un nombre que tenga sentido para la directiva y que su archivo sea distintivo y claro. Hemos visto algunos ejemplos antes, pero veremos más en la sección de cómo nombrar.

Manipula el DOM en una directiva

> Cuando manipules DOM directamente, usa una directiva. Si hay alguna alternativa como usando CSS para cambiar los estilos o los animation services, Angular templating, ngShow o ngHide, entonces úsalos en su lugar. Por ejemplo, si la directiva sólo muestra o esconde elementos, usa ngHide/ngShow.

¿Por qué?: Manipular el DOM puede ser difícil de testear, debugear y normalmente hay mejores maneras (e.g. CSS, animations, templates)

Provee un prefijo único de Directiva

> Proporciona un prefijo corto, único y descriptivo como acmeSalesCustomerInfo que se declare en el HTML como acme-sales-customer-info.

¿Por qué?: El prefijo corto y único identifica el contexto de la directiva y el origen. Por ejemplo el prefijo cc- puede indicar que la directiva en particular es parte de la aplicación CodeCamper, mientras que acme- pudiera indicar que la directiva es de la compañía Acme.

Nota: Evita ng- ya que está reservado para las directivas AngularJS. Estudia sabiamente las directivas usadas para evitar conflictos de nombres, como ion- de Ionic Framework.

Limitate a elementos y atributos

> Cuando crees directivas que tengan sentido como elemento, restringe E (elemento personalizado) y opcionalmente restringe A (atributo personalizado). Generalmente, si puede ser su control propio, E es apropiado, La pauta general es permitir EA pero intenta implementarlo como un elemento cuando sea un elemento único y como un atributo cuando añada mejoras a su propio elemento existente en el DOM.

¿Por qué?: Tiene sentido.

¿Por qué?: Mientras permitamos que una directiva sea usada como una clase, si esa directiva realmente está actuando como un elemento, tiene sentido que sea un elemento, o al menos un atributo.

Nota: En Angular 1.3+ EA es el valor por defecto

```
<!-- evitar -->
<div class="my-calendar-range"></div>
```

```
/* evitar */
angular
    .module('app.widgets')
    .directive('myCalendarRange', myCalendarRange);

function myCalendarRange() {
    var directive = {
        link: link,
        templateUrl: '/template/is/located/here.html',
        restrict: 'C'
    };
    return directive;

    function link(scope, element, attrs) {
      /* */
    }
}
```

```
<!-- recomendado -->
<my-calendar-range></my-calendar-range>
<div my-calendar-range></div>
```

```
/* recomendado */
angular
    .module('app.widgets')
    .directive('myCalendarRange', myCalendarRange);

function myCalendarRange() {
    var directive = {
        link: link,
        templateUrl: '/template/is/located/here.html',
        restrict: 'EA'
    };
    return directive;

    function link(scope, element, attrs) {
      /* */
    }
}
```

Directivas y ControllerAs

> Usa la sintaxis controller as con una directiva para ser consistente con el uso de controller as con los pares de vista y controlador.

¿Por qué?: Tiene sentido y no es difícil.

Nota: La siguiente directiva demuestra algunas de las formas en las que puedes usar el scope dentro del link y el controlador de una directiva, usando controllerAs. He puesto la template para dejarlo todo en un lugar.

Nota: En cuanto a la inyección de dependencias, mira Identificar Dependencias Manualmente.

Nota: Nótese que el controlador de la directiva está fuera del closure de la directiva. Este estilo elimina los problemas que genera la inyección de dependencias donde la inyección es creada en un código no alcanzable después del return.

```
<div my-example max="77"></div>
```

```
angular
    .module('app')
    .directive('myExample', myExample);

function myExample() {
    var directive = {
        restrict: 'EA',
        templateUrl: 'app/feature/example.directive.html',
        scope: {
            max: '='
        },
        link: linkFunc,
        controller: ExampleController,
          controllerAs: 'vm',
          bindToController: true // porque el scope is aislado
      };

    return directive;

    function linkFunc(scope, el, attr, ctrl) {
        console.log('LINK: scope.min = %s *** should be undefined', scope.min);
```

```
        console.log('LINK: scope.max = %s *** should be undefined', scope.max);
        console.log('LINK: scope.vm.min = %s', scope.vm.min);
        console.log('LINK: scope.vm.max = %s', scope.vm.max);
    }
}

ExampleController.$inject = ['$scope'];

function ExampleController($scope) {
    // Inyectando el $scope solo para comparación
    var vm = this;

    vm.min = 3;

    console.log('CTRL: $scope.vm.min = %s', $scope.vm.min);
    console.log('CTRL: $scope.vm.max = %s', $scope.vm.max);
    console.log('CTRL: vm.min = %s', vm.min);
    console.log('CTRL: vm.max = %s', vm.max);
}
```

```
<!-- example.directive.html -->
<div>hello world</div>
<div>max={{vm.max}}<input ng-model="vm.max"/></div>
<div>min={{vm.min}}<input ng-model="vm.min"/></div>
```

> Usa bindToController = true cuando uses controller as con una directiva cuando quieras asociar el scope exterior al scope del controller de la directiva.

¿Por qué?: Lo hace más fácil a la hora de asociar el scope exterior al scope del controlador de la directiva.

Nota: bindToController fue introducido en Angular 1.3.0.

```
<div my-example max="77"></div>
```

```
angular
    .module('app')
    .directive('myExample', myExample);

function myExample() {
    var directive = {
        restrict: 'EA',
        templateUrl: 'app/feature/example.directive.html',
        scope: {
            max: '='
        },
        controller: ExampleController,
          controllerAs: 'vm',
          bindToController: true
      };

    return directive;
}

function ExampleController() {
    var vm = this;
    vm.min = 3;
```

```
        console.log('CTRL: vm.min = %s', vm.min);
        console.log('CTRL: vm.max = %s', vm.max);
    }
```

```
<!-- example.directive.html -->
<div>hello world</div>
<div>max={{vm.max}}<input ng-model="vm.max"/></div>
<div>min={{vm.min}}<input ng-model="vm.min"/></div>
```

Resolviendo promesas en un controlador

Promesas de activación de un controlador

> Resuelve la lógica de inicialización de un controlador en una función activate.

¿Por qué?: Colocar la lógica de inicialización en un lugar consistente del controlador lo hace más fácil de localizar, más consistente de testear, y ayuda a evitar que la lógica de activación se propage a lo largo del controlador.

¿Por qué?: El activate del controlador hace que la lógica para refrescar el controlador/Vista sea reutilizable, mantiene la lógica junta, lleva el usuario a la Vista más rápido, hace las animaciones más fáciles en ng-view o ui-view y lo hace más rápido a la vista del usuario.

Nota: Si necesitas condicionalmente cancelar la ruta antes de empezar el controller, usa en su lugar route resolve.

```
/* evitar */
function Avengers(dataservice) {
    var vm = this;
    vm.avengers = [];
    vm.title = 'Avengers';

    dataservice.getAvengers().then(function(data) {
        vm.avengers = data;
        return vm.avengers;
    });
}
```

```
/* recomendado */
function Avengers(dataservice) {
    var vm = this;
    vm.avengers = [];
    vm.title = 'Avengers';

    activate();

    ////////////

    function activate() {
        return dataservice.getAvengers().then(function(data) {
            vm.avengers = data;
            return vm.avengers;
        });
    }
}
```

Resolución de promesas en la ruta

> Cuando un controlador depende en una promesa a ser resuelta antes de que el controlador se active, resuelve esas dependencias en el $routeProvider antes de que la lógica del controlador sea ejecutada. Si necesitas condicionalmente cancelar una ruta antes de que el controlador sea activado, usa un route resolver.
>
> Usa un route resolver cuando decidas cancelar la ruta antes de hacer la transición a la Vista.

¿Por qué?: Un controlador puede requerir datos antes de que se cargue. Esos datos deben venir desde una promesa a través de una factoría o de $http. Usando un route resolve permite que la promesa se resuelva antes de que la lógica del controlador se ejecute, así puedes tomar decisiones basándote en los datos de la promesa.

¿Por qué?: El código se ejecuta después de la ruta y la función activate del controlador. La Vista empieza a cargar al instante. El bindeo de los datos se ejecutan cuando la promesa del activate se resuelva. Una animación de "Cargando" se puede mostrar durante la transición de la vista (via ng-view o ui-view)

Nota: El código se ejecuta antes que la ruta mediante una promesa. Rechazar la promesa cancela la ruta. Resolverla hace que la nueva vista espere a que la ruta sea resuelta. Una animación de "Cargando" puede ser mostrada antes de que se resuelva. Si quieres que la Vista aparezca más rápido y no necesitas un checkpoint para decidir si puedes mostrar o no la view, considera la técnica controller activate.

```
/* evitar */
angular
    .module('app')
    .controller('Avengers', Avengers);

function Avengers(movieService) {
    var vm = this;
    // sin resolver
    vm.movies;
    // resulta asincronamente
    movieService.getMovies().then(function(response) {
        vm.movies = response.movies;
    });
}
```

```
/* better */

// route-config.js
angular
    .module('app')
    .config(config);

function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'Avengers',
            controllerAs: 'vm',
            resolve: {
                moviesPrepService: function(movieService) {
                    return movieService.getMovies();
                }
            }
```

```
        });
}

// avengers.js
angular
    .module('app')
    .controller('Avengers', Avengers);

Avengers.$inject = ['moviesPrepService'];
function Avengers(moviesPrepService) {
      var vm = this;
      vm.movies = moviesPrepService.movies;
}
```

Nota: El siguiente ejemplo muestra una ruta que cuando se resuelve apunta a una función, haciéndolo más fácil de debugear y más fácil de manejar la inyección de dependencias.

```
/* even better */

// route-config.js
angular
    .module('app')
    .config(config);

function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'Avengers',
            controllerAs: 'vm',
            resolve: {
                moviesPrepService: moviesPrepService
            }
        });
}

function moviePrepService(movieService) {
    return movieService.getMovies();
}

// avengers.js
angular
    .module('app')
    .controller('Avengers', Avengers);

Avengers.$inject = ['moviesPrepService'];
function Avengers(moviesPrepService) {
      var vm = this;
      vm.movies = moviesPrepService.movies;
}
```

Nota: El código del ejemplo de dependencia en movieService no se puede minimizar tal cual. Para detalles en cómo hacer este código sea minimizable, mira la sección en inyección de dependencias y en minimización y anotación.

Anotación manual para la inyección de dependencias

Insegura después de la Minificación

> Evita usar la sintaxis acortada para declarar dependencias sin usar algún método que permita minificación.

¿Por qué?: Los parámetros al componente (e.g. controller, factory, etc) se convertirán en variables acortadas. Por ejemplo, common y dataservice se convertirán a o b y no serán encontradas por AngularJS.

```
/* evitar - not minification-safe*/
angular
    .module('app')
    .controller('Dashboard', Dashboard);

function Dashboard(common, dataservice) {
}
```

```
Este código acortará las variables cuando se minimice y causará errores en tiempo de ejecución.
```

```
/* evitar - not minification-safe*/
angular.module('app').controller('Dashboard', d);function d(a, b) { }
```

Identifica dependencias manualmente

> Usa $inject Para identificar manualmente las dependencias de tus componentes AngularJS.

¿Por qué?: Esta técnica es la misma que se usa con ng-annotate, la cuál recomiendo para automatizar la creación de dependencias minificadas de forma segura. Si ng-annotate detecta que la inyección ha sido hecha, no la duplicará.

¿Por qué?: Esto salvaguarda tus dependencias de ser vulnerables de problemas a la hora de minimizar cuando los parámetros se acorten. Por ejemplo, common y dataservice se convertirán a o b y no serán encontradas por AngularJS.

¿Por qué?: Evita crear dependencias en línea, ya que las listas largas pueden ser difícil de leer en el arreglo. También puede ser confuso que el arreglo sea una serie de cadenas mientras que el último componente es una función.

```
/* evitar */
angular
    .module('app')
    .controller('Dashboard',
        ['$location', '$routeParams', 'common', 'dataservice',
            function Dashboard($location, $routeParams, common, dataservice) {}
        ]);
```

```
/* evitar */
angular
  .module('app')
  .controller('Dashboard',
      ['$location', '$routeParams', 'common', 'dataservice', Dashboard]);

function Dashboard($location, $routeParams, common, dataservice) {
}
```

```
/* recomendado */
angular
    .module('app')
    .controller('Dashboard', Dashboard);

Dashboard.$inject = ['$location', '$routeParams', 'common', 'dataservice'];

function Dashboard($location, $routeParams, common, dataservice) {
}
```

Nota: Cuando tu función está debajo de un return, $inject puede ser inalcanzable (esto puede pasar en una directiva). Puedes solucionarlo moviendo el $inject encima del return o usando la sintaxis de arreglo para inyectar.

Nota: ng-annotate 0.10.0 introduce una funcionalidad donde mueve $inject donde es alcanzable.

```
// dentro de la definición de una directiva
function outer() {
    return {
        controller: DashboardPanel,
    };

    DashboardPanel.$inject = ['logger']; // Inalcanzable
    function DashboardPanel(logger) {
    }
}
```

```
// dentro de la definición de una directiva
function outer() {
    DashboardPanel.$inject = ['logger']; // alcanzable
    return {
        controller: DashboardPanel,
    };

    function DashboardPanel(logger) {
    }
}
```

Identifica manualmente dependencias del route resolver

> Usa $inject para identificar manualmente las dependencias de tu route resolver para componentes de AngularJS.

¿Por qué?: Esta técnica separa la función anónima para el route resolver, haciendola más fácil de leer.

¿Por qué?: Una declaración $inject puede ser fácilmente preceder al route resolver para hacer cualquier minificación de dependencias segura.

```
/* recomendado */
function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'Avengers',
            controllerAs: 'vm',
            resolve: {
                moviesPrepService: moviePrepService
```

```
                }
            });
    }

    moviePrepService.$inject = ['movieService'];
    function moviePrepService(movieService) {
        return movieService.getMovies();
    }
```

>

Minificación y Anotación

ng-annotate

> Usa ng-annotate para Gulp or Grunt y comenta funciones que necesiten inyección de dependencias automatizadas usando /** @ngInject */

¿Por qué?: Salvaguarda tu código de cualquier dependencia que pueda no estar usando prácticas de minificación segura.

¿Por qué?: ng-min está obsoleto

> Yo prefiero Gulp porque siento que es más fácil de escribir, leer, y debugear.

El siguiente código no está usando minificación de dependencias segura.

```
    angular
        .module('app')
        .controller('Avengers', Avengers);

    /* @ngInject */
    function Avengers(storageService, avengerService) {
        var vm = this;
        vm.heroSearch = '';
        vm.storeHero = storeHero;

        function storeHero() {
            var hero = avengerService.find(vm.heroSearch);
            storageService.save(hero.name, hero);
        }
    }
```

Cuando el código de arriba es ejecutado a través de ng-annotate producirá la siguiente salida con la anotación $inject y será seguro para ser minificado.

```
    angular
        .module('app')
        .controller('Avengers', Avengers);

    /* @ngInject */
```

```
function Avengers(storageService, avengerService) {
    var vm = this;
    vm.heroSearch = '';
    vm.storeHero = storeHero;

    function storeHero() {
        var hero = avengerService.find(vm.heroSearch);
        storageService.save(hero.name, hero);
    }
}

Avengers.$inject = ['storageService', 'avengerService'];
```

Nota: Si ng-annotate detecta que la inyección ya ha sido hecha (e.g. @ngInject fué detectado), no duplicará el código de $inject.

Nota: Al usar un route resolver puedes prefijar a la función del resolver con /* @ngInject */ y producirá código propiamente anotado, manteniendo cualquier inyección de dependencias segura para ser minificada.

```
// Using @ngInject annotations
function config($routeProvider) {
    $routeProvider
        .when('/avengers', {
            templateUrl: 'avengers.html',
            controller: 'Avengers',
            controllerAs: 'vm',
            resolve: { /* @ngInject */
                moviesPrepService: function(movieService) {
                    return movieService.getMovies();
                }
            }
        });
}
```

Nota: A partir de Angular 1.3 usa el párametro ngStrictDi de la directiva ngApp. Al presentarse el injector será creado en modo "strict-di" causando que la aplicación falle al invocar funciones que no usan explícitamente anotación de funciones (éstas podrían no estar minificadas en forma segura). Información para debugear será mostrada en la consola para ayudar a rastrear el código infractor.

```
<body ng-app="APP" ng-strict-di>
```

Usa Gulp o Grunt para ng-annotate

> Usa gulp-ng-annotate o grunt-ng-annotate en una tarea de construcción automática. Inyecta /* @ngInject */ antes de cualquier función que tenga dependecias.

¿Por qué?: ng-annotate atrapará la mayoría de las dependencias, pero algunas veces requiere indicios usando la sintaxis /* @ngInject */.

El código siguiente es un ejemplo de una tarea de Gulp que usa ngAnnotate.

```
gulp.task('js', ['jshint'], function() {
    var source = pkg.paths.js;
    return gulp.src(source)
        .pipe(sourcemaps.init())
        .pipe(concat('all.min.js', {newLine: ';'}))
        // Agrega la notación antes de ofuscar para que el código sea minificicado
```

```
apropiadamente.
            .pipe(ngAnnotate({
                // true ayuda a añadir @ngInject donde no es usado. Infiere.
                // No funciona con resolve, así que tenemos que ser explícitos en ese caso
                add: true
            }))
            .pipe(bytediff.start())
            .pipe(uglify({mangle: true}))
            .pipe(bytediff.stop())
            .pipe(sourcemaps.write('./'))
            .pipe(gulp.dest(pkg.paths.dev));
    });
```

Manejo de excepciones

Decoradores

> Usa un decorador o decorator, en tiempo de configuración usando el servicio $provide, en el servicio $exceptionHandler para realizar acciones personalizadas cuando una excepción ocurra.

¿Por qué?: Provee una manera consistente de manejar excepciones de Angular que no están siendo capturadas en tiempo de desarrollo o en tiempo de ejecución.

Nota: Otra opción es sobreescribir el servicio en lugar de usar un decorador. Esto está bien, pero si quiere mantener el comportamiento por default y extenderlo se recomienda usar un decorador.

```
    /* recomendado */
    angular
        .module('blocks.exception')
        .config(exceptionConfig);

    exceptionConfig.$inject = ['$provide'];

    function exceptionConfig($provide) {
        $provide.decorator('$exceptionHandler', extendExceptionHandler);
    }

    extendExceptionHandler.$inject = ['$delegate', 'toastr'];

    function extendExceptionHandler($delegate, toastr) {
        return function(exception, cause) {
            $delegate(exception, cause);
            var errorData = {
                exception: exception,
                cause: cause
            };
            /**
             * Pudieramos agregar el error a la colección de un servicio,
             * agregar los errores en el $rootScope,
             * logear los errores a un servidor remoto,
             * o logear localmente. O arrojarlos llanamente. Dependende totalmente de tí.
             * arrojar excepción;
             */
            toastr.error(exception.msg, errorData);
        };
```

```
}
```

Capturadores de excepciones

> Crea una factoría que exponga una interfaz para capturar y manejar excepciones elegantemente.

¿Por qué?: Provee de una manera consistente de capturar excepciones que puedan ser arrojadas en tu código (e.g. durante llamadas XHR o promesas que fallaron).

Nota: El capturadoor de excepciones es bueno para capturar y reaccionar a excepciones específicas de llamadas que tu sabes van a arrojar una. Por ejemplo, al hacer una llamada XHR para obtener datos desde un servicio web remoto y quieres capturar cualquier excepción de ese servicio y reaccionar únicamente.

```
/* recomendado */
angular
    .module('blocks.exception')
    .factory('exception', exception);

exception.$inject = ['logger'];

function exception(logger) {
    var service = {
        catcher: catcher
    };
    return service;

    function catcher(message) {
        return function(reason) {
            logger.error(message, reason);
        };
    }
}
```

Errores de ruta

> Maneja y logea todos los errores de enrutamiento usando $routeChangeError.

¿Por qué?: Provee una manera consistente de manejar todos los errores de enrutamiento.

¿Por qué?: Potencialmente provee una mejor experiencia de usuario si un error de enrutamiento ocurre y tu los rediriges a una pantalla amigable con más detalles u opciones de recuperación.

```
/* recomendado */
function handleRoutingErrors() {
    /**
     * Route cancellation:
     * Cancelación de la Ruta:
     * En un error de ruteo, ir al dashboard.
     * Proveer una cláusula de salida si trata de hacerlo dos veces.
     */
    $rootScope.$on('$routeChangeError',
        function(event, current, previous, rejection) {
            var destination = (current && (current.title || current.name ||
```

```
    current.loadedTemplateUrl)) ||
                        'unknown target';
                    var msg = 'Error routing to ' + destination + '. ' + (rejection.msg || '');
                    /**
                     * Optionally log using a custom service or $log.
                     * Opcionalmente logear usando un servicio personalizado o $log.
                     * (Don't forget to inject custom service)
                     */
                    logger.warning(msg, [current]);
                }
            );
        }
```

Cómo Nombrar

Pautas para nombrar

> Usa nombres consistentes para todos los componentes siguiendo un patrón que describa las características del componente y después (opcionalmente) su tipo. Mi patrón recomendado es feature.type.js. Hay dos nombres para la mayoría de los assets: * el nombre del archivo (avengers.controller.js) * el nombre del componente registrado en Angular (AvengersController)

¿Por qué?: Las pautas de como nombrar nos ayudan a proveer una manera consistente para encontrar contenido en un vistazo. La Consistencia es vital dentro del proyecto. La Consistencia es importante dentro de un equipo. La Consistencia a lo largo de una compañía provee de una tremenda eficacia.

¿Por qué?: Las pautas para nombrar deberían simplemente ayudarte a encontrar tu código rápidamente y hacerlo más fácil de entender.

Nombres de archivo según característica

> Usa nombres consistentes para todos los componentes siguiendo un patrón que describa la característica o feature del componente y después (opcionalmente) su tipo. Mi patrón recomendado es feature.type.js.

¿Por qué?: Provee de una manera consistente para identificar componentes rápidamente.

¿Por qué?: Provee un patrón de coincidencia para tareas automatizadas.

```
    /**
     * opciones comunes
     */

    // Controladores
    avengers.js
    avengers.controller.js
    avengersController.js

    // Servicios/Factories
    logger.js
    logger.service.js
    loggerService.js
```

```
/**
 * recomendado
 */

// controllers
avengers.controller.js
avengers.controller.spec.js

// servicios/factories
logger.service.js
logger.service.spec.js

// constantes
constants.js

// definición de módulos
avengers.module.js

// rutas
avengers.routes.js
avengers.routes.spec.js

// configuración
avengers.config.js

// directivas
avenger-profile.directive.js
avenger-profile.directive.spec.js
```

Nota: Otra convención común es nombrar archivos de controladores sin la palabra controller en el archivo tal como avengers.js en lugar de avengers.controller.js. Todas las demás convenciones todavía usan un sufijo del tipo. Los Controladores son el tipo más común de componente así que esto nos ahorra escribir y aún es fácilmente identificable. Yo recomiendo que elijas 1 convención y seas consistente dentro de tu equipo.

```
/**
 * recomendado
 */
// Controladores
avengers.js
avengers.spec.js
```

Nombres de archivos de prueba

> Nombra especificaciones de pruebas de manera similar a la del componente que están probando con un sufijo de spec.

¿Por qué?: Provee de una manera consistente de identificar componentes rápidamente.

¿Por qué?: Provee de un patrón de coincidencia para karma u otros test runners.

```
/**
 * recomendado
 */
avengers.controller.spec.js
logger.service.spec.js
avengers.routes.spec.js
```

```
avenger-profile.directive.spec.js
```

Nombres de controladores

> Usa nombres consistentes para todos los controladores nombrados a partir de lo que hacen. Usa UpperCamelCase para controladores, ya que son constructores.

¿Por qué?: Provee de una manera consistente de identificar y referenciar controladores rápidamente.

¿Por qué?: UpperCamelCase es convencional para identificar objetos que pueden ser instanciados usando un constructor.

```
/**
 * recomendado
 */

// avengers.controller.js
angular
    .module
    .controller('HeroAvengers', HeroAvengers);

function HeroAvengers() { }
```

Sufijo para el nombre del controlador

> Agrega el sufijo Controller al nombre del controlador o déjalo sin sufijo. Escoge uno, no uses ambos.

¿Por qué?: El sufijo Controller es usado más comúnmente y es más descriptivo explícitamente.

¿Por qué?: Omitir el sufijo es más breve y el controlador es fácilmente identificable más seguido incluso sin el sufijo.

```
/**
 * recomendado: Opción 1
 */

// avengers.controller.js
angular
    .module
    .controller('Avengers', Avengers);

function Avengers() { }
```

```
/**
 * recomendado: Opción 2
 */

// avengers.controller.js
angular
    .module
    .controller('AvengersController', AvengersController);

function AvengersController() { }
```

Nombres de factorías

> Usa nombres consistentes para todas las factorías nombradas a partir de lo que hacen. Usa camel-casing para los servicios y las factorías.

¿Por qué?: Provee una manera consistente de identificar y referenciar factorías rápidamente.

```
/**
 * recomendado
 */

// logger.service.js
angular
    .module
    .factory('logger', logger);

function logger() { }
```

Nombres para directivas

> Usa nombres consistentes para todas las directivas usando camel-case. Usa un prefijo corto para describir el área a la que la directiva pertenece (algunos ejemplos son un prefijo según la compañía o un prefijo según el proyecto).

¿Por qué?: Provee una manera consistente de identificar y referenciar componentes rápidamente.

```
/**
 * recomendado
 */

// avenger-profile.directive.js
angular
    .module
    .directive('xxAvengerProfile', xxAvengerProfile);

// el uso es <xx-avenger-profile> </xx-avenger-profile>

function xxAvengerProfile() { }
```

Módulos

> Cuando haya múltiples módulos, el archivo del módulo principal es nombrado app.module.js mientras que otros módulos que dependan de él son nombrados a partir de lo que ellos representan. Por ejemplo, un módulo de admin es nombrado admin.module.js. Los nombres de los módulos registrados serán respectivamente app y admin.

¿Por qué?: Provee consistencia para múltiples módulos de aplicación, y para poder expandirse a aplicaciones más grandes.

¿Por qué?: Provee una manera fácil de usar tareas automatizadas para cargar todas las definiciones de módulos primero, y luego todos los

otros archivos de angular (agrupación).

Configuración

> Separa la configuración de un módulo en un archivo propio nombrado a partir del nombre del módulo. Un archivo de configuración para el módulo principal de app es nombrado app.config.js (o simplemente config.js). La configuración para un módulo llamado admin.module.js es nombrada admin.config.js.

¿Por qué?: Separa la configuración de la definición del módulo, componentes y código activo.

¿Por qué?: Provee un lugar identificable para establecer configuración para un módulo.

Rutas

> Separa la configuración de la ruta en un archivo propio. Algunos ejemplos pueden ser app.route.js para el módulo principal y admin.route.js para el módulo admin admin. Incluso en aplicaciones pequeñas prefiero esta separación del resto de la configuración.

Estructura de la aplicación. El principio LIFT

LIFT

> Estructura tu aplicación de tal manera que puedas Localizar (Locate) tu código rápidamente, Identificar (Identify) el código de un vistazo, mantener la estructura más plana (flat, en ingles) (Flattest) que puedas, y Trata (Try) de mantenerte DRY. La estructura debe de seguir estas 4 pautas básicas.

```
*¿Por qué LIFT?*: Provee una estructura consistente que escala bien, es modular, y hace más fácil
incrementar la eficiencia de los desarrolladores al encontrar código rápidamente. Otra manera de
chequear la estructura de tu aplicación es preguntarte a ti mismo: ¿Qué tan rápido puede abrir y
trabajar en todos los archivos relacionados a una caracteristica?

Cuando encuentro que mi estructura no se siente cómoda, regreso y reviso estas pautas LIFT

1. `L`ocating - Localizar nuestro código es fácil
2. `I`dentify - Identificar código de un vistazo
3. `F`lat - Estructura plana tanto como sea posible
4. `T`ry - Tratar de mantenerse DRY (Don't Repeat Yourself) or T-DRY
```

Localizar

> Haz que la localización tu código sea intuitivo, simple y rápido.

¿Por qué?: Encuentro que esto es super importante para un proyecto. Si el equipo no puede encontrar los archivos en los que necesita trabajar rápidamente, no podrán trabajar tan eficientemente como sea posible, y la estructura necesita cambiar. Puede que no conozcas el nombre del archivo o donde están sus archivos relacionados, así que poniéndolos en las locaciones más intuitivas y cerca de los otros ahorra

mucho tiempo. Una estructura de directorios descriptiva puede ayudar con esto.

```
/bower_components
/client
  /app
    /avengers
    /blocks
      /exception
      /logger
    /core
    /dashboard
    /data
    /layout
    /widgets
  /content
  index.html
.bower.json
```

Identificar

> Cuando miras en un archivo deberías saber instantáneamente qué contiene y qué representa.

¿Por qué?: Gastas menos tiempo buscando y urgando por código, y es más eficiente. Si esto significa que quieres nombres de archivos más largos, entonces que así sea. Se descriptivo con los nombres de los archivos y mantén el contenido del archivo a exactamente un componente. Evita archivos con múltiples controladores, o una mezcla. Hay excepciones a la regla de un por archivo cuando tengo un conjunto de pequeñas features que están relacionadas unas con otras, aún así son fácilmente identificables.

Estructura plana

> Mantén una estructura de directorios plana tanto como sea posible. Cuando llegues a un total de mas de siete archivos, comienza a considerar separación.

¿Por qué?: Nadie quiere buscar en siete niveles de directorios por un arhivo. Piensa en los menús de los sitios web ... cualquiera más profundo que dos debería ser seriamente considerado. En una estructura de directorios no hay una regla dura o rápida en cuanto a un número, pero cuando un directorio tiene de siete a diez archivos, tal vez ese sea el momento para empezar a crear subdirectorios. Básate en tu nivel de confort. Usa una estructura más plana hasta que haya un valor obvio (para ayudar al resto de LIFT) en crear un nuevo directorio.

T-DRY (Try to Stick to DRY - Trata de apegarte a DRY)

> Se DRY, pero no te vuelvas loco y sacrifiques legibilidad.

¿Por qué?: Ser DRY es importante, pero no crucial si sacrifica otras partes de LIFT, es por eso que lo llamo T-DRY. No quiero escribir session-view.html por una vista porque, obviamente es una vista. Si no es obvio o por convención, entonces la nombro así.

Estructura de la aplicación

Pautas universales

> Ten una visión de implementación de corto y largo plazo. En otras palabras, empieza con poco pero ten en mente hacia donde se dirige la aplicación. Todo el código de la aplicación va en el directorio raíz llamado app. Todo el contenido es separado en una característica por archivo. Cada controlador, servicio, módulo, vista tiene su propio archivo. Todos los vendor scripts de terceros son almacenados en otro directorio raíz y no en el directorio app. Si yo no lo escribí no los quiero saturando mi aplicación (bower_components, scripts, lib).

Nota: Encuentra más detalles y el razonamiento detrás de esta estructura en este post original sobre la estructura de una aplicación.

Layout

> Coloca los componentes que definen el layout universal de la aplicación en un directorio llamado layout. Estos pueden incluir una vista caparazón y un controlador que actúen como un contenedor para la aplicación, navegación, menús, áreas de contenido, y otras regiones.

¿Por qué?: Organiza todo el layout en un lugar único reusado a lo largo de la aplicación.

Estructura de carpetas-por-característica

> Crea carpetas llamadas de acuerdo al característica que representan. Cuando una carpeta crezca para contener más de siete archivos, comienza a considerar la creación de una carpeta para ellos. Tu límite puede ser diferente, así que ajusta de acuerdo a tus necesidades.

¿Por qué?: Un desarrollador puede localizar el código, identificar cada qué representa cada archivo de un vistazo, la estructura es tan plana como puede ser, y no hay nombres repetidos o redundantes.

¿Por qué?: Las pautas LIFT estarán cubiertas.

¿Por qué?: Ayuda a evitar que la aplicación se sature a través de organizar el contenido y conservarlo alineado con las pautas LIFT.

¿Por qué?: Cuando hay demasiados archivos (10+) localizarlos es más fácil con una estructura de directorios consistente y más difíciles en una estructura plana.

```
/**
 * recomendado
 */

app/
    app.module.js
    app.config.js
    app.routes.js
    components/
        calendar.directive.js
        calendar.directive.html
        user-profile.directive.js
        user-profile.directive.html
    layout/
        shell.html
        shell.controller.js
        topnav.html
        topnav.controller.js
    people/
        attendees.html
        attendees.controller.js
        speakers.html
```

```
            speakers.controller.js
            speaker-detail.html
            speaker-detail.controller.js
        services/
            data.service.js
            localstorage.service.js
            logger.service.js
            spinner.service.js
        sessions/
            sessions.html
            sessions.controller.js
            session-detail.html
            session-detail.controller.js
```

```
  ![Sample App Structure](https://raw.githubusercontent.com/johnpapa/angular-
styleguide/master/a1/assets/modularity-2.png)
```

Nota: No estructures tu aplicación usando directorios-por-tipo. Esto requiere mover múltiples directorios cuando se está trabajando en una característica y se vuelve difícil de manejar conforme la aplicación crece a 5, 10 o 25+ vistas y controladores (y otras características), lo que lo hace más difícil que localizar archivos en una aplicación estructura en directorios-por-característica.

```
    /*
    * evita
    * Alternativa directorios-por-tipo
    * Yo recomiendo "directorios-por-característica", en su lugar.
    */

    app/
        app.module.js
        app.config.js
        app.routes.js
        controllers/
            attendees.js
            session-detail.js
            sessions.js
            shell.js
            speakers.js
            speaker-detail.js
            topnav.js
        directives/
            calendar.directive.js
            calendar.directive.html
            user-profile.directive.js
            user-profile.directive.html
        services/
            dataservice.j
            localstorage.js
            logger.js
            spinner.js
        views/
            attendees.html
            session-detail.html
            sessions.html
            shell.html
            speakers.html
            speaker-detail.html
            topnav.html
```

Modularidad

Muy pequeños, Módulos autocontenidos

> Crea módulos pequeños que encapsulen una responsabilidad.

¿Por qué?: Aplicaciones modulares hace más fácil el plug and go ya que permiten a los equipos de desarrollo construir porciones verticales de la aplicación y lanzarlas incrementalmente. Esto significa que podemos conectar nuevas características conforme las desarrollamos.

Crea un Módulo App

> Crea una módulo raíz de aplicación cuyo rol sea unir todos los módulos y características de tu aplicación. Nombra éste de acuerdo a tu aplicación.

¿Por qué?: Angular incentiva la modularidad y patrones de separación. Crear un módulo raíz de aplicación cuyo rol es atar otros módulos juntos provee una manera muy directa de agregar o remover módulos de tu aplicación.

Mantén el módulo App delgado

> Solo coloca lógica para unir la aplicación en el módulo app. Deja las características en sus propios módulos.

¿Por qué?: Agregar roles adicionales a la aplicación raíz para obtener datos remotos, mostrar vistas, u otra lógica no relaciona a la unión de la aplicación enturbia el módulo app y hace ambos conjuntos de características difíciles de reusar y apagar.

¿Por qué?: El módulo app se convierte en el manifiesto que describe qué módulos definen la aplicación.

Áreas de características son módulos

> Crea módulos que representen áreas de características, como el layout, servicios reusables y compartidos, dashboards, y características específicás de la aplicación (e.g. customers, admin, sales).

¿Por qué?: Módulos autocontenidos pueden ser agregados a la aplicación con poca o sin ninguna fricción.

¿Por qué?: Sprints o iteraciones pueden enfocarse en áreas de características y encendarlas al final del sprint o iteración.

¿Por qué?: Separar áreas de características en módulos hace más fácil testear módulos en aislamiento y reusar código.

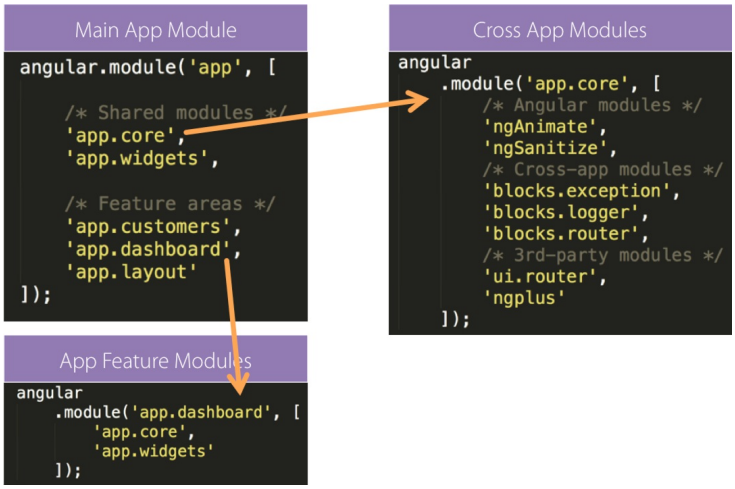
Bloques reusables son módulos

> Crea módulos que representen bloques de la aplicación reusables para servicios cómunes como manejo de excepciones, logeo, diagnóstico, seguridad, y almacenamiento local de datos.

¿Por qué?: Este tipo de características son necesarias en muchas aplicaciones, así que mantenerlas separadas en sus propios módulos pueden ser genéricas de aplicación y pueden ser reusadas a lo largo de varias aplicaciones.

Dependencias de módulos

> El módulo raiz de la aplicación depende de módulos de características específicas y cualquier módulo compartido o reusable.



¿Por qué?: El módulo principal de la aplicación contiene un manifiesto rápidamente identificable de las características de la aplicación.

¿Por qué?: Cada área de características contiene un manifiesto de lo que depende, así que puede ser extraído como dependencia en otras aplicaciones y seguir funcionando.

¿Por qué?: Características internas de la aplicación como servicios de datos compartidos se hacen fácil de localizar y compartir desde app.core (elije tu nombre favorito para este módulo).

Nota: Esta es una estrategia para consistencia. Hay muy buenas opciones aquí. Escoge una que sea consistente, que siga las reglas de dependencias de AngularJS, y que sea fácil de mantener y escalar.

> Mis estructuras varían ligeramente entre proyectos pero todas ellas siguen estas pautas para estructuras y modularidad. La implementación puede variar dependiendo de las características y el equipo. En otras palabras, no te quedes colgado en una estructura igual pero justifica tu estructura usando consistencia, mantenibilidad, y eficacia en mente.
>
> En una aplicación pequeña, también puedes considerar poner todas las dependencias compartidas en el módulo principal dónde los módulos de características no tienen dependencias directas. Esto hace más fácil mantener aplicaciones pequeñas, pero hace más difícil el reusar módulos fuera de esta aplicación.

Lógica de arranque

Configuración

> Inyecta código dentro de module configuration que necesite ser configurado antes de correr la aplicación angular. Candidatos ideales incluyen providers y constantes.

¿Por qué?: Esto hace más fácil tener menos lugares para la configuración.

```
angular
    .module('app')
    .config(configure);

configure.$inject =
    ['routerHelperProvider', 'exceptionHandlerProvider', 'toastr'];

function configure (routerHelperProvider, exceptionHandlerProvider, toastr) {
    exceptionHandlerProvider.configure(config.appErrorPrefix);
    configureStateHelper();

    toastr.options.timeOut = 4000;
    toastr.options.positionClass = 'toast-bottom-right';

    ////////////////

    function configureStateHelper() {
        routerHelperProvider.configure({
            docTitle: 'NG-Modular: '
        });
    }
}
```

Bloques run

> Cualquier código que necesite ser ejecutado cuando una aplicación arranca debe ser declarado en una factoría, ser expuesto a través de una función, o inyectado en el bloque run.

¿Por qué?: Código que está directamente en un bloque run puede ser difícil de testear. Colocarlo en una factoría lo hace fácil de abstraer y mockear.

```
angular
    .module('app')
    .run(runBlock);

  runBlock.$inject = ['authenticator', 'translator'];

  function runBlock(authenticator, translator) {
      authenticator.initialize();
      translator.initialize();
  }
```

Servicios envoltorios $ de Angular

$document y $window

> Usa $document y $window en lugar de document y window.

¿Por qué?: Estos servicios son envueltos por Angular y son más fáciles de testear en lugar de usar document y window en las pruebas. Esto te ayuda a evitar que tener que mockear document y window tu mismo.

$timeout y $interval

> Usa $timeout y $interval en lugar de setTimeout y setInterval

¿Por qué?: Estos servicios están envueltos por Angular y son más fáciles de testear y manejar el ciclo digest de Angular así que mantienen el bindeo de los datos en sincronización.

Pruebas

Las pruebas unitarias ayudan a mantener el código limpio, así que incluyo algunas de mis recomendaciones en los fundamentos del testeo unitario con links para mayor información.

Escribe pruebas con historias

> Escribe un conjunto de pruebas para cada historia. Comienza con un test vacío y llénalo conforme escribas el código para la historia.

¿Por qué?: Escribir descripciones para la prueba ayuda a definir claramente qué es lo que tu historia hará, qué no hará, y cómo puedes medir el éxito.

```
    it('should have Avengers controller', function() {
        // TODO
    });

    it('should find 1 Avenger when filtered by name', function() {
        // TODO
    });

    it('should have 10 Avengers', function() {
        // TODO (mock data?)
    });

    it('should return Avengers via XHR', function() {
        // TODO ($httpBackend?)
    });

    // y así
```

Librería para las pruebas

> Usa Jasmine o Mocha para las pruebas unitarias.

¿Por qué?: Ambas Jasmine y Mocha son usadas ampliamente por la comunidad de AngularJS. Ambas son estables, bien mantenidas, y proveen de características de pruebas robustas.

Nota: Cuando uses Mocha, también considera elegir una librería como Chai.

Test runner

> Usa Karma como test runner.

¿Por qué?: Karma es fácil de configurar para correr una vez o automáticamente cuando cambias tu código.

¿Por qué?: Karma encaja en tu proceso de Integración Continua fácilmente por sí sola o a través de Grunt o Gulp.

¿Por qué?: Algunos IDE's están comenzando a integrarse con Karma, tal como WebStorm y Visual Studio.

¿Por qué?: Karma funciona bien con líderes de automatización de tareas tales como Grunt (con grunt-karma) y Gulp (con gulp-karma).

Stubear y espíar

> Usa Sinon para el stubeo y espíar.

¿Por qué?: Sinon funciona bien con ambos Jasmine y Mocha y extiende las características de stubeo y espíonaje que ellos ofrecen.

¿Por qué?: Sinon hace más fácil cambiar entre Jasmine y Mocha, si quieres probar ambos.

Headless browser

> Usa PhantomJS para correr tus pruebas en un servidor.

¿Por qué?: PhantomJS es un navegador headless que ayuda a correr las pruebas necesitar una navegador "visual". Así que no necesitas instalar Chrom, Safari u otros navegadores en tu servidor.

Nota: Aún debes testear en todos los navegadores de tu entorno, así como sea apropiado para tu audiencia meta.

Ánalisis de código

> Corre JSHint en tus pruebas.

¿Por qué?: Las pruebas son código. JSHint puede ayudar a identificar problemas en la calidad del código que pueden causar que tus pruebas funcionen inapropiadamente.

Mitiga palabras globales dentro de las reglas de JSHint en las pruebas

> Relaja las reglas en tu código de prueba para permitir palabras globales comúnes como describe y expect.

¿Por qué?: Tus pruebas son código y requieren la misma atención y reglas de calidad de código que todo tu código de producción. Sin embargo, variables globales usadas por el framework para pruebas, por ejemplo, puede ser relajado al incluir esto en tus specs de prueba.

```
/* global sinon, describe, it, afterEach, beforeEach, expect, inject */
```

Organizando las pruebas

> Coloca archivos de pruebas unitarias (specs) lado a lado con tu código del cliente. Coloca tus specs que cubren la integración con el servidor o que prueban múltiples componentes en un directorio tests separado.

¿Por qué?: Las Pruebas Unitarias tiene una correlación directa con un componente y archivo específico en tu código fuente.

¿Por qué?: Es más fácil mantenerlas actualizadas ya que siempre están a la vista. Al escribir código ya sea que realices TDD o pruebes durante el desarrollo o después del desarrollo, los specs están lado a lado y nunca fuera de la vista o de la mente, así es más probable que sean mantenidas lo cual ayuda a mantener la cobertura de pruebas.

¿Por qué?: Cuando actualices código fuente es más fácil ir y actualizar las pruebas al mismo tiempo.

¿Por qué?: Colocarlas lado a lado hace más fácil encontrarlas y fácil de moverlas con el código fuente si mueves la fuente.

¿Por qué?: Tener el spec cerca hace más fácil al lector del código fuente aprender cómo se supone que el componente es usado y descubrir sus propias limitaciones.

¿Por qué?: Separar specs para que no estén un build de distribución es fácil con grunt o gulp.

```
/src/client/app/customers/customer-detail.controller.js
                         /customer-detail.controller.spec.js
                         /customers.controller.spec.js
                         /customers.controller-detail.spec.js
                         /customers.module.js
                         /customers.route.js
                         /customers.route.spec.js
```

Animaciones

Uso

> Usa sutiles animaciones con AngularJS para hacer transiciones entre estados en vistas y elementos visuales primarios. Incluye el módulo ngAnimate. Las 3 claves son sutil, fluido, transparente.

¿Por qué?: Animaciones sutiles pueden mejorar la Experiencia de Usuario cuando son usadas apropiadamente.

¿Por qué?: Animaciones sutiles pueden mejorar el rendimiento percibido como una transición de vista.

Sub Segundos

> Usa duraciones cortas para las animaciones. Yo generalmente empiezo con 300ms y ajusto hasta que es apropiado.

¿Por qué?: Animaciones largas pueden tener el efecto contrario en la Experiencia de Usuario y el rendimiento percibido al dar la apariencia de una aplicación lenta.

animate.css

> Usa animate.css para animaciones convencionales.

¿Por qué?: Las animaciones que animate.css provee son rápidas, fluidas, y fáciles de agregar en tu aplicación.

¿Por qué?: Provee consistencia en tus animaciones.

¿Por qué?: animate.css está ampliamente usado y testeado.

Nota: Ve este excelente post de Matias Niemelä sobre animaciones AngularJS

Comentarios

jsDoc

> Si planeas producir documentación, usa la sintaxis jsDoc para documentar nombres de funciones, descripción, parámetros y devoluciones. Usa @namespace y @memberOf para igualar la estructura de tu aplicación.

¿Por qué?: Puedes generar (y regenerar) documentación desde tu código, en lugar de escribirla desde cero.

¿Por qué?: Provee consistencia al usar una herramienta industrial común.

```
    /**
     * Logger Factory
     * @namespace Factories
     */
    (function() {
      angular
          .module('app')
          .factory('logger', logger);

      /**
       * @namespace Logger
       * @desc Application wide logger
       * @memberOf Factories
       */
      function logger($log) {
          var service = {
```

```
            logError: logError
        };
        return service;

        ////////////

        /**
         * @name logError
         * @desc Logs errors
         * @param {String} msg Message to log
         * @returns {String}
         * @memberOf Factories.Logger
         */
        function logError(msg) {
            var loggedMsg = 'Error: ' + msg;
            $log.error(loggedMsg);
            return loggedMsg;
        };
    }
})();
```

JS Hint

Usa un archivo de opciones

> Usa JS Hint para resaltar problemas en tu JavaScript y asegurate de personalizar el arhivo de opciones de JS Hint e incluirlo en el control de versiones. Ve los JS Hint docs para detalles sobre estas opciones.

¿Por qué?: Provee una primera alerta antes de hacer commit de cualquier código al control de versiones.

¿Por qué?: Provee consistencia a lo largo de tu equipo.

```
{
    "bitwise": true,
    "camelcase": true,
    "curly": true,
    "eqeqeq": true,
    "es3": false,
    "forin": true,
    "freeze": true,
    "immed": true,
    "indent": 4,
    "latedef": "nofunc",
    "newcap": true,
    "noarg": true,
    "noempty": true,
    "nonbsp": true,
    "nonew": true,
    "plusplus": false,
    "quotmark": "single",
    "undef": true,
    "unused": false,
    "strict": false,
    "maxparams": 10,
    "maxdepth": 5,
    "maxstatements": 40,
```

```
        "maxcomplexity": 8,
        "maxlen": 120,

        "asi": false,
        "boss": false,
        "debug": false,
        "eqnull": true,
        "esnext": false,
        "evil": false,
        "expr": false,
        "funcscope": false,
        "globalstrict": false,
        "iterator": false,
        "lastsemic": false,
        "laxbreak": false,
        "laxcomma": false,
        "loopfunc": true,
        "maxerr": false,
        "moz": false,
        "multistr": false,
        "notypeof": false,
        "proto": false,
        "scripturl": false,
        "shadow": false,
        "sub": true,
        "supernew": false,
        "validthis": false,
        "noyield": false,

        "browser": true,
        "node": true,

        "globals": {
            "angular": false,
            "$": false
        }
    }
```

Constantes

Globales de vendor

> Crea una Constante de Angular para variables globales en librerías vendor.

¿Por qué?: Provee una manera de inyectar librerías vendor que de otra manera son globales. Esto mejora la testeabilidad al permitirte saber más fácilmente cuáles son las dependencias de tus componentes (evita abstracciones malformadas). También te permite mockear estas dependencias, cuando tiene sentido.

```
    // constants.js

    /* global toastr:false, moment:false */
    (function() {
        'use strict';

        angular
            .module('app.core')
```

```
        .constant('toastr', toastr)
        .constant('moment', moment);
})();
```

> Usa constantes para valores que no cambian y no vienen de otro servicio. Cuando las constantes son usadas solo por para un módulo que pueda ser reutilizado en múltiples aplicaciones, coloca las constantes en un archivo por módulo nombrado a partir del módulo. Hasta que esto sea requerido, mantén las constantes en el módulo principal en un archivo constants.js.

¿Por qué?: Un valor que puede cambiar, incluso infrecuentemente, debería ser obtenido desde un servicio así no tendrás que cambiar el código fuente. Por ejemplo, una url para un servicio de datos puede ser colocada en una constante pero un mejor lugar sería cargarla desde un servicio web.

¿Por qué?: Las Constantes pueden ser inyectadas en cualquier componente de angular, incluyendo providers.

¿Por qué?: Cuando una aplicación es separada en módulos que pueden ser reutilizados en otras aplicaciones, cada módulo autónomo debería ser capaz de operar por sí mismo incluyendo cualquier constante de la cual dependa.

```
// Constantes usadas por la aplicación entera
angular
    .module('app.core')
    .constant('moment', moment);

// Constantes usadas solo por el módulo de ventas
angular
    .module('app.sales')
    .constant('events', {
        ORDER_CREATED: 'event_order_created',
        INVENTORY_DEPLETED: 'event_inventory_depleted'
    });
```

Plantillas y snippets

Usa Plantillas o snippets para ayudarte a seguir estilos consistentes o patrones. Aquí hay plantillas y/o snippets para algunos de los editores de desarrollo web e IDEs.

Sublime Text

> Snippets de Angular que siguen estos estilos y directrices.

```
1. Descarga los [snippets de Angular para Sublime](../assets/sublime-angular-snippets?raw=true)
1. Colócalos en tu directorio de Packages
1. Reinicia Sublime
1. En un archivo de JavaScript escibe estos comandos seguidos de un `TAB`
```

```
ngcontroller // crea un controlador de Angular
ngdirective // crea una directiva de Angular
ngfactory // crea una factory de Angular
ngmodule // crea un módulo de Angular
```

Visual Studio

> Plantillas de Angular que siguen estos estilos y directrices pueden ser encontrados en SideWaffle

```
1. Descarga la extensión [SideWaffle](http://www.sidewaffle.com) de Visual Studio (archivo vsix)
1. Corre el archivo vsix
1. Reinicia Visual Studio
```

WebStorm

> Snippets y arhivos de Angular que siguen estos estilos y directrices. Puedes importarlos en tus configuraciones de WebStorm:

```
1. Descarga los [snippets y plantillas de Angular para WebStorm](assets/webstorm-angular-file-
template.settings.jar?raw=true)
1. Abre WebStorm y ve al menú `File`
1. Elije la opción `Import Settings`
1. Selecciona el archivo y da click en `OK`
1. En un archivo de JavaScript escribe estos comandos seguidos de un `TAB`:
```

```
    ng-c // crea un controlador de Angular
    ng-f // crea una factory de Angular
    ng-m // crea un módulo de Angular
```

Generador de Yeoman

Puedes usar el generador de yeoman HotTowel para crear una aplicación que te sirve como punto de inicio en Angular que sigue esta guía de estilos.

1. Instala generator-hottowel

```
  npm install -g generator-hottowel
```

2. Crea un nuevo directorio y entra en el

```
  mkdir myapp
  cd myapp
```

3. Corre el generador

```
  yo hottowel helloWorld
```

Enrutamiento

Enrutamiento del lado del Cliente es importante para crear un flujo de navegación entre vistas y vistas de composición que están hechas de muchas pequeñas plantillas y directivas.

> Usa el AngularUI Router para ruteo del lado del cliente.

¿Por qué?: UI Router ofrece todas las características del router de Angular mas algunas adicionales incluyendo rutas anidadas y estados.

¿Por qué?: La sintaxis es bastante similar al router de Angular y es fácil de migrar al UI Router.

> Define rutas para vistas en el módulo dónde éstas existen. Cada módulo debería contener las rutas para las vistas en ese módulo.

¿Por qué?: Cada módulo debe ser capaz de funcionar por sí mismo.

¿Por qué?: Al remover un módulo o al agregar un módulo, la aplicación solo contendrá rutas que apunten a las vistas existentes.

¿Por qué?: Esto hace más fácil habilitar o deshabilitar porciones de una aplicación sin preocuparse de rutas huérfanas.

Automatización de tareas

Usa Gulp o Grunt para crear tareas automatizadas. Gulp deriva a código sobre configuración mientras que Grunt deriva a configuración sobre código. Personalmente yo prefiero Gulp ya que se siente más fácil de leer y escribir, pero ambos son excelentes.

> Usa automatización de tareas para listar archivos que definan módulos *.module.js antes que otros archivos de JavaScript en la aplicación.

¿Por qué?: Angular necesita la definición de módulos para ser registrados antes de que sean usados.

¿Por qué?: Nombra módulos con un patrón específico como *.module.js hace más fácil tomarlos con un glob y listarlos primero.

```
var clientApp = './src/client/app/';

// Siempre toma archivos de módulos primero
var files = [
  clientApp + '**/*.module.js',
  clientApp + '**/*.js'
];
```

Guía de estilos colaborativa de Angular para equipos por @john_papa

f