Angular 1.5 Style Guide (Components Oriented)

Propósito

> El propósito de esta guía de estilo es proporcionar orientación y mejores prácticas sobre la construcción de aplicaciones de AngularJS que usen y piensen en componentes mostrando convenciones y ejemplos en el proceso. Construir nuestra aplicación con componentes nos permite actualizar fácilmente a Angular 2.

Nota: Esta guía de estilo hace algunas referencias a viejas pero válidas guías de estilo.

Nota2: Angular 1.5 es una transición a Angular 2. Introduce un nuevo jugador llamado **.Component()**. Esto es sólo una sintaxis de azúcar para las directivas, pero simplifica el proceso de construcción de un componente web.

Índice

Estructura de la aplicación AngularJS

Introducción

A medida que construimos *aplicaciones AngularJS* , el número de archivos crece rápidamente y esto puede afectar a la organización y a la estructura del directorio de aplicaciones. El objetivo de este documento es definir una estructura de directorios, siguiendo las mejores prácticas y convenciones, que nos permita construir aplicaciones escalables y mantenibles.

Estructura basica

> La primera aproximación es organizar la aplicación por tipos de archivo.

Esta estructura está bien para aplicaciones pequeñas, pero cuando el número de componentes, servicios, etc comienza a crecer, la adición o el cambio de funcionalidad significa tener que localizar los archivos para editar , identificar el código a cambiar y, a continuación, asegurarse de que no se vea afectada ninguna otra funcionalidad.

Las directrices de LIFT

Para mejorar la estructura estándar podemos aplicar las pautas LIFT al definir nuestra estructura de directorio deseada:

1. Locating - Localizar nuestro código es fácil
2. Identify - Identificar código de un vistazo
3. Flat - Estructura plana tanto como sea posible
4. Try - Tratar de mantenerse DRY (Don't Repeat Yourself) or T-DRY

Localizar: El código de localización necesita ser intuitivo, simple y rápido.

Identificar: Cuando miramos un archivo esperamos saber qué contiene y representa.

Flat: Nadie quiere buscar 7 niveles de carpetas para encontrar un archivo. Manténgalo lo más plano posible.

T-DRY: Esto tiene más que ver con convenciones. P.ej. No incluya *view* o *controller* en el nombre de un módulo si es obvio lo que hace.

Estructura propuesta

> La clave para una estructura clara es tener una visión a corto plazo y otra a largo plazo de la implementación.

En otras palabras, comienza pequeño, pero ten en cuenta hacia dónde se dirige la aplicación.

Con esto en mente y la aplicación de las directrices LIFT, la propuesta de estructura de directorios se muestra a continuación:

```
app/
    features/
        home/
            home.component.js
            home.css
            home.html
            home.component.spec.js
            home.route.js
        about/
            about.component.js
            about.css
            about.html
            about.component.spec.js
            about.route.js
        ...
    components/
        mycomponent/
            mycomponent.component.js
            mycomponent.component.spec.js
            mycomponent.html
            mycomponent.css
            mycomponent.service.js
            mycomponent.service.spec.js
```

```
            mycomponent2/
                mycomponent2.component.js
                mycomponent2.component.spec.js
                mycomponent2.html
                mycomponent2.css
                mycomponent2.service.js
                mycomponent2.service.spec.js
            ...
        services/
            data.service.js
            logger.service.js
            ...
        app.module.js
        app.config.js
        app.constants.js
        app.routes.js
        app.run.js
    assets/
        img/   // Images and icons for your application
        css/   // All styles related files
        js/    // Non-angular js files
        libs/  // 3rd-party libs E.g. jQuery.
        fonts/ // 3rd-party fonts
    index.html
```

Dónde:

Carpeta / Archivo	Descripción
index.html	El *index.html* vive en la raíz de la estructura de front-end. Se encargará principalmente de la carga de todas las bibliotecas y elementos de AngularJS.
assets	La carpeta de assets también es bastante estándar. Contendrá todos los recursos necesarios para su aplicación que no estén relacionados con el código AngularJS.
app	Aquí es donde vive la aplicación. El archivo *app.module.js* manejará la configuración de su aplicación, carga las dependencias AngularJS y así sucesivamente. El archivo *app.route.js* manejará la ruta principal. El archivo *app.config.js* se puede utilizar para establecer la configuración globalmente para la aplicación.

La carpeta de aplicaciones contiene los siguientes subdirectorios:

Carpeta / Archivo	Descripción
components	Componentes reutilizables para la aplicación.
services	Los servicios proporcionan algún servicio a la aplicación para funciones compartidas, como acceso a datos remotos, almacenamiento en caché de datos, almacenamiento local y registro, por ejemplo.
features1..N	Estas carpetas con nombre representan funciones de la aplicación. P.ej. *login*, *session* o *gallery* por ejemplo. Cada componente, vista, etc., está en su propio archivo. La carpeta también contiene scripts de prueba y css particular a la vista si es necesario. En [Referencia # 1] (# recursos) John Papa habla de la ** 3-7 guía de archivo **. Si una característica contiene más de 7 archivos, puede ser hora de dividir esa característica en características más pequeñas.

Beneficios del enfoque modularizado

Mantenimiento del código

> Siguiendo el enfoque anterior, compartiendo lógicamente sus aplicaciones y fácilmente podrá localizar y editar código.

Escalable

> El código será mucho más fácil de escalar. Agregar nuevas directivas y páginas no hinchará las carpetas existentes.

El embarque de nuevos desarrolladores también debería ser mucho más fácil una vez que se explica la estructura. Además, con este enfoque, podrás soltar características de entrada y salida de tu aplicación con relativa facilidad, por lo que probar nuevas funcionalidades o eliminarlas debería ser una brisa.

Depuración

> La depuración del código será mucho más fácil con este enfoque modular para el desarrollo de aplicaciones.

Será más fácil encontrar las piezas ofensivas del código y arreglarlas.

Pruebas

> Escribir scripts de prueba y probar aplicaciones modularizadas es mucho más fácil que hacerlo en proyectos no modularizados.

Mejores prácticas

Componentes

> Los componentes son las nuevas características de Angular 1.5.

El método .component() sustituye a nuestros viejos controladores y directivas de esta nueva forma de crear aplicaciones. Piensa en los componentes como un elemento complejo funcional, que puede integrar dentro de otros componentes y coexistir con otros para dar a la aplicación un sentido único.

> El ayudante de método .component() es simplemente azúcar sintáctico del buen método .directive() antiguo. No hay prácticamente nada que puedas hacer con .component() que no puedas hacer con .directive().

- Los componentes tienen **ámbitos aislados** de forma predeterminada.
- Utilizan automáticamente la sintaxis *controllerAs*, por lo tanto podemos usar **$ctrl** para acceder a los datos de los componentes.
- Usan **controladores** en lugar de *link* funciones.
- La opción *bindToController* está activada **de forma predeterminada**.

```
// before
app.directive(name, fn)

// after
app.component(name, options)
```

```
.component('counter', {
    templateUrl: 'app/components/counter/counter.html',
    controller: CounterController,
    controllerAs: 'vm',
     // isolated scope binding
    bindings: {
        count: '='
    }
});

function CounterController() {
    var vm = this;
    vm.increment = increment;
    vm.decrement = decrement;

    function increment() {
      vm.count++;
    }
    function decrement() {
      vm.count--;
    }
}
```

Plantilla en línea que está enlazada a una variable de mensaje en el controlador del componente:

```
<div>Counter {{vm.count}}</div>
```

Bindings

- El **< símbolo** indica enlaces unidireccionales. Lectura.
- El **= símbolo** indica enlaces de dos vías. Lectura y escritura.
- El **@ símbolo** se puede utilizar cuando la entrada es una cadena, especialmente cuando el valor de la vinculación no cambia.

  ```
  bindings: {
      hero: '<',
      comment: '@'
  }
  ```

- El **& symbol** Las salidas se realizan con & bindings, que funcionan como callbacks a los acontecimientos del componente.

```
bindings: {
    onDelete: '&',
    onUpdate: '&'
}
```

```
<editable-field on-update="vm.update('location', value)"></editable-field><br>
<button ng-click="vm.onDelete({hero: vm.hero})">Delete</button>
```

Componentes de enrutamiento

Los componentes también son útiles como plantillas de ruta (por ejemplo, cuando se utiliza ngRoute). En una aplicación basada en componentes, cada vista es un componente:

```
$routeProvider.when('/', {
    template: '<home></home>'
});
```

Comunicación intercomponente

Las directivas pueden exigir a los controladores de otras directivas que permitan la comunicación entre sí. Esto se puede lograr en un componente proporcionando una asignación de objetos para la propiedad require. Las claves de objeto especifican los nombres de propiedad bajo los cuales los controladores requeridos (valores de objeto) se enlazarán al controlador del componente que requiere.

> Ten en cuenta que los controladores requeridos no estarán disponibles durante la instancia del controlador, pero se garantiza que estarán disponibles justo antes de ejecutarse el método **$onInit**.

```
// Component 1
.component('myTabs', {
  transclude: true,
  controller: MyTabsController,
  controllerAs: 'vm',
  templateUrl: 'my-tabs.html'
});

function MyTabsController() {
    var vm = this;
    vm.panes = [];

    vm.addPane = function(pane) {
      // Code
    };
}

// Component 2
.component('myPane', {
  transclude: true,
  require: {
    tabsCtrl: '^myTabs'
  },
  bindings: {
    title: '@'
  },
  controller: MyPaneController,
```

```
  controllerAs: 'vm',
  templateUrl: 'my-pane.html'
});

function MyPaneController() {
    var vm = this;
    vm.$onInit = function() {
        vm.tabsCtrl.addPane(vm); // adding pane into addPane function of myTabs
    };
}
```

Controladores de componentes de prueba unitarias

> La forma más sencilla de probar un controlador de componente es mediante $componentController que se incluye en ngMock. La ventaja de este método es que no es necesario crear ningún elemento DOM.

```
describe('component: heroDetail', function() {
  var $componentController;

  beforeEach(module('heroApp'));
  beforeEach(inject(function(_$componentController_) {
    $componentController = _$componentController_;
  }));

  it('should expose a `hero` object', function() {
    // Here we are passing actual bindings to the component
    var bindings = {hero: {name: 'Wolverine'}};
    var vm = $componentController('heroDetail', null, bindings);

    expect(vm.hero).toBeDefined();
    expect(vm.hero.name).toBe('Wolverine');
  });
```

Miembros enlazables arriba

> Coloca los miembros vinculables en la parte superior del controlador, ordenados alfabéticamente y no se distribuyan a través del código del controlador.

Mas Info +

```
function SessionsController() {
    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 una línea, considere mantenerla arriba, siempre y cuando la legibilidad no se vea afectada.

```
function SessionsController(sessionDataService) {
    var vm = this;

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

Declaraciones de función para ocultar detalles de implementación

> Utiliza declaraciones de función para ocultar los detalles de implementación.

Mantén tus miembros ligables arriba. Cuando necesites enlazar una función en un controlador, apúntala a una declaración de función que aparece más adelante en el archivo. Esto está vinculado directamente a la sección "Miembros enlazables arriba". Para más detalles ver [esta publicación] (http://www.johnpapa.net/angular-function-declarations-function-expressions-and-readable-code/).

```
/*
 * recommend
 * Using function declarations
 * and bindable members up top.
 */

function AvengersController(avengersService, 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 avengersService.getAvengers().then(function(data) {
            vm.avengers = data;
            return vm.avengers;
        });
    }
```

```
}
```

Difiere la lógica del controlador a los servicios

> Difiere la lógica del controlador delegándola a servicios y factorias.

Mas Info +

```
function OrderController(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(showError);
    };
}
```

Servicios

Singletons

> Los servicios se instancian con la palabra clave new, usa this para métodos públicos y variables. Puesto que éstos son tan similares a las factorias, se recomienda el uso de factorías por consistencia.

Nota: Todos los servicios en Angular son singleton. Esto significa que sólo hay una instancia de un servicio dado 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) {
          /* */
        }
    };
}
```

Factorias

Responsabilidad única

> Las factorías deberian tener una única responsabilidad, que está encapsulada por su contexto. Una vez que una factoría comienza a exceder ese propósito singular, una nueva factoría debe ser creada.

Singleton

> Las factorías son singletons y devuelven un objeto que contiene los miembros del servicio.

Nota: Todos las factorías en Angular son singletons.

Accessible Members Up Top

> Expón los miembros del servicio llamados (su interfaz) en la parte superior, utilizando una técnica derivada del Patrón Modulo Revelación.

Mas Info +

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

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

    function save() {
        /* */
    };

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

De esta manera los enlaces se reflejan en el objeto host, los valores primitivos no pueden actualizarse solo usando el patrón de módulo revelación.

Declaración de funciones para ocultar detalles de implementación

> Usa declaración de funciones para ocultar detalles de implementación. Mantén a tus miembros accesibles al principio de la factoría. Apunta a aquellas declaraciones de función que aparecen más adelante en el fichero. Para mas detalles mira este post.

Mas Info +

```
/**
 * recommended
 * Using function declarations
 * and accessible members up top.
 */
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() {
        // implementation details go here
    }

    function getAvengerCount() {
        // implementation details go here
    }

    function getAvengersCast() {
        // implementation details go here
    }

    function prime() {
        // implementation details go here
    }

    function ready(nextPromises) {
        // implementation details go here
    }
}
```

Servicios de datos

Llamadas de datos independientes

> Refactoriza la lógica para realizar operaciones de datos e interactuar con datos en una factoría. Haz que los servicios de datos sean responsables de llamadas XHR, almacenamiento local, almacenamiento en memoria o cualquier otra operación de datos.

Mas Info +

```
// 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 por los consumidores, como un controlador, ocultando la implementación de los consumidores, como se muestra a continuación.

```
```javascript
```

// controller calling the dataservice factory angular .module('app.avengers') .controller('AvengersController', AvengersController);

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

function AvengersController(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 de llamadas de datos

> Cuando llames a un servicio de datos que devuelve una promesa como `$http`, devuelve también una promesa en la llamada a la función.

[Mas Info+](https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md#return-a-promise-from-data-calls)

```javascript

activate();

function activate() {
    /**
     * Step 1
     * Ask the getAvengers function for the
     * avenger data and wait for the promise
     */
    return getAvengers().then(function() {
        /**
         * Step 4
         * Perform an action on resolve of final promise
         */
        logger.info('Activated Avengers View');
    });
}

function getAvengers() {
      /**
       * Step 2
       * Ask the data service for the data and wait
       * for the promise
       */
      return dataservice.getAvengers()
          .then(function(data) {
              /**
               * Step 3
               * set the data and resolve the promise
               */
              vm.avengers = data;
              return vm.avengers;
      });
}
```

Directivas

> Directivas y plantillas / controladores ya no son necesarios, sin embargo, la API sigue siendo coherente para la compatibilidad hacia atrás. Utilice una directiva para vincular el comportamiento personalizado al DOM existente.
```

¿Cuál es el papel de .directive()?

- Una directiva decora el DOM, agrega comportamiento y extiende el DOM existente. Cuando se utiliza un .component(), se crea en el DOM, cuando se utiliza .directive() se decora el DOM, que es la mentalidad.
- Ya sabes ng-click, es una directiva, por lo que decora y añade el comportamiento existente a un elemento existente, no es como elemento.

¿Cómo puedo utilizar .directive()?

- Las directivas deben utilizarse cuando se requiera realizar manipulación DOM fuera del ciclo de eventos de angular y del núcleo.
- Utiliza la función de compilación y/o enlace para crear la funcionalidad personalizada que necesita.
- Asegúrate de desvincular eventos personalizados o API de DOM como element.addEventListener(); dentro del evento $destroy.
- Obtén acceso a los atributos y usa $observe dependiendo de lo que necesites acceder (valores readonly / etc).
- Si utilizas un controlador, úsalo como cuarto enlace: argumento fn, y actualiza solo la lógica de la vista dentro del controlador.
- Nunca pases $scope para manipular los datos, usa siempre el cuarto argumento $ctrl.
- Requiere y manipula otras directivas que usan la propiedad require como String o sintaxis de Array.

¿A qué debo restringir mis Componentes / Directivas?

- .component() está restringido a 'E' por defecto, lo que significa elemento personalizado, no se puede cambiar esto.
- .directive() debe decorar, por lo tanto debe ser un atributo solamente, lo que significa restringir: 'A' siempre.

Limítala a una por archivo

> Crea una directiva por archivo. Nombra el archivo para la directiva. [Más info +] (https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md#directives)

Nota: "**Práctica óptima**: Las directivas deben limpiarse después. Puedes utilizar element.on('$ destroy', ...) o scope.$On('$ destroy', ...) para ejecutar una función de limpieza cuando se elimina la directiva" ... de la documentación angular.

```
/* calendar-range.directive.js */

/**
 * @desc order directive that is specific to the order module at a company named Acme
 * @example <div acme-order-calendar-range></div>
 */
angular
    .module('sales.order')
    .directive('acmeOrderCalendarRange', orderCalendarRange);

function orderCalendarRange() {
    /* implementation details */
}
```

Nota: Existen muchas opciones de nomenclatura para las directivas, especialmente porque pueden utilizarse en ámbitos estrechos o amplios. Elija uno que haga que la directiva y su nombre de archivo sean claros y distintos. Algunos ejemplos están a continuación, pero vea la sección Naming para obtener más recomendaciones.

Manipulación del DOM en una directiva

> Para manipular el DOM directamente, utiliza una directiva. Si se pueden utilizar formas alternativas, como el uso de CSS para establecer estilos o el servicio de animación (https://docs.angularjs.org/api/ngAnimate), Angular templating, ngShow o ngHide, úsalos en su lugar. Por ejemplo, si la directiva simplemente oculta o muestra, usa ngHide/ngShow. Mas info +

Proporciona un prefijo único a la directiva

> Proporciona un prefijo corto, único y descriptivo a la directiva comoacmeSalesCustomerInfo el cual podria ser declarado en HTML como acme-sales-customer-info. Mas info +

Nota: Evita ng- ya que está reservado para las directivas de Angular. Investiga ampliamente las directivas utilizadas para evitar conflictos de denominación, como ion- para el Ionic Framework.

Restringir a los atributos

> Al crear una directiva, restringe A (atributo personalizado).

```
```html
```
```

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

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

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

Directivas y ControllerAs

> Usa la sintaxis controllerAs con una directiva para ser coherente con el uso de controller as con parejas de vista y controlador. [Más información +] (https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md#directives-and-controlleras)

Nota: La directiva a continuación muestra algunas de las formas en que puedes utilizar el ámbito dentro de controladores de enlace y directivas, utilizando controllerAs.

Nota: Ten en cuenta que el controlador de la directiva está fuera del cierre de la directiva. Este estilo elimina los problemas en los que la inyección se crea como código inaccesible después de un return.

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

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

function myExample() {
    var directive = {
        restrict: 'A',
        templateUrl: 'app/feature/example.directive.html',
        scope: {
            max: '='
        },
        link: linkFunc,
        controller: ExampleController,
        // note: This would be 'ExampleController' (the exported controller name, as string)
        // if referring to a defined controller in its separate file.
        controllerAs: 'vm',
        bindToController: true // because the scope is isolated
    };

    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) {
    // Injecting $scope just for comparison
    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>
```

Nota: También puedes asignar un nombre al controlador cuando lo inyectas en la función de enlace y acceder a los atributos de la directiva como propiedades del controlador.

```
```javascript
```
```

// Alternative to above example function linkFunc(scope, el, attr, vm) { 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: vm.min = %s', vm.min); console.log('LINK: vm.max = %s', vm.max); }

> Utiliza bindToController = true cuando utilices la sintaxiscontroller as con una directiva cuando desees enlazar el ámbito externo al ámbito del controlador de la directiva.

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

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

function myExample() {
    var directive = {
        restrict: 'A',
        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>
```

Actualizando a Angular2

> Escribir componentes en este estilo te permitirá actualizar tus componentes usando .component() en Angular2 muy fácilmente.

Se vería algo así en ECMAScript 5 y la nueva sintaxis de plantilla:

```
import {Component} from '@angular/core';

@Component({
  selector: 'counter',
  template: `
    <div class="todo">
      <input type="text" [(ng-model)]="count">
      <button type="button" (click)="decrement();">-</button>
      <button type="button" (click)="increment();">+</button>
    </div>
  `
})

export default class CounterComponent {
  constructor() {

  }
  increment() {
    this.count++;
  }
  decrement() {
    this.count--;
  }
}
```

> Como puedes ver, no es un cambio trivial. Para cambiar a Angular2 es necesario cambiar a typescript y aprender cierta sintaxis, pero este nuevo modelo orientado a componentes hace que el cambio sea más fácil.

Recursos

1. Component versus Directive
2. From ng-controller to components with Angular 1.5. Juri Strumpflohner
3. Exploring component method. Todd Motto
4. App structuring guidelines. John Papa
5. Application Structure. John Papa - GitHub