Guía de estilo AngularJS (ES2015)

Arquitectura, estructura de archivos, componentes, one-way dataflow y buenas prácticas.

Una guía de estilos sensata par equipos, por @toddmotto

Esta arquitectura y guía de estilo ha sido reescrita desde cero para ES2015, los cambios en AngularJS 1.5+ para la futura actualización de tu aplicación a Angular. Esta guía incluye nuevas buenas prácticas para flujos de datos en una dirección, delegación de eventos, arquitectura de componentes y enrutamiento de componentes.

Puedes encontrar la vieja guía de estilo aquí, y el razonamiento detrás de la nueva aquí.

> Unete a la experiencia Ultimate de AngularJS y domina completamente características básicas y avanzadas de AngularJS para desarrollar aplicaciones del mundo real que son rápidas y escalables.

Tabla de contenidos

Arquitectura modular

Cada modulo en una aplicación angular es un module component. Un module component es la definición principal de dicho modulo que encapsula la lógica, templates, enrutamiento y componentes hijo.

Teoría

El diseño de modulos se correlaciona directamente con nuestra estructura de directorios, lo que mantiene las cosas mantenibles y predecibles. Deberíamos tener idealmente tres modulos de alto nivel: root, component y common. El módulo root define la base que inicia nuestra aplicación, y la plantilla correspondiente. Después importamos nuestro componente y módulos comunes dentro del módulo root para incluir nuestras dependencias. El componente y common modules entonces requieren módulos de más bajo nivel, los cuales contienen nuestros componentes, controladores, servicios, directivas, filtros y pruebas para cada característica reutilizable.

Volver arriba

Root module

Un Root modulecomienza con un root component que define el elemento base para toda la aplicación, con una salida de enrutamiento definida, el ejemplo se muestra el uso de ui-view desde ui-router.

```
// app.component.js
const AppComponent = {
  template: `
    <header>
        Hello world
    </header>
    <div>
        <div ui-view></div>
    </div>
    <footer>
        Copyright MyApp 2016.
    </footer>
  `
};

export default AppComponent;
```

Un Root module es entonces creado, con el AppComponent importado y registrado con .component('app', AppComponent). Nuevas importaciones para submodulos(componentes y common modules) son hechas para incluir todos los componentes relevantes para la aplicación.

```
// app.module.js
import angular from 'angular';
import uiRouter from 'angular-ui-router';
import AppComponent from './app.component';
import ComponentsModule from './components/components.module';
import CommonModule from './common/common.module';

const AppModule = angular
  .module('app', [
    ComponentsModule,
    CommonModule,
    uiRouter
  ])
  .component('app', AppComponent)
  .name;

export default AppModule;
```

Volver arriba

Component module

Un Component module es el contenedor referencia para todos los componentes reusables. Ve más arriba como importamos Componentes y los inyectamos dentro del Root module, esto nos da un lugar único para importar todos los componentes para la aplicación. Estos módulos que necesitas estan desconectados de los demás modulos y por lo tanto se pueden mover dentro de cualqueir otra aplicación con facilidad.

```
import angular from 'angular';
import CalendarModule from './calendar/calendar.module';
import EventsModule from './events/events.module';

const ComponentsModule = angular
  .module('app.components', [
    CalendarModule,
    EventsModule
  ])
  .name;

export default ComponentsModule;
```

Volver arriba

Common module

El Common module es el contenedor referencia para todos los componentes específicos de la aplicación, que no queremos utilizar en ninguna otra aplicación. Estos pueden ser cosas como el layout, navegación y pie de página. Ve más arriba como importamos CommonModule y los inyectamos dentro del Root module, esto nos da un lugar único para importar todos los componentes comunes para la aplicación.

```
import angular from 'angular';
import NavModule from './nav/nav.module';
import FooterModule from './footer/footer.module';

const CommonModule = angular
  .module('app.common', [
    NavModule,
    FooterModule
  ])
```

```
  .name;

export default CommonModule;
```

Volver arriba

Low-level modules

Low-level modules son componentes individuales que contienen la lógica para cada bloque de función. Cada una de ellas definirá un modulo, para ser importado a un modulo de un nivel más alto, como un componente o common module, hay un ejemplo abajo. Siempre recuerda agregar el sufijo .name a cada export cuando se crea un moduloe *nuevo*, y no cuando se haga referencia a uno. Notarás que existen definiciones de enrutamiento aquí, los cuales se verán en un captítulo posterior de está guía.

```
import angular from 'angular';
import uiRouter from 'angular-ui-router';
import CalendarComponent from './calendar.component';

const CalendarModule = angular
  .module('calendar', [
    uiRouter
  ])
  .component('calendar', CalendarComponent)
  .config(($stateProvider, $urlRouterProvider) => {
    $stateProvider
      .state('calendar', {
        url: '/calendar',
        component: 'calendar'
      });
    $urlRouterProvider.otherwise('/');
  })
  .name;

export default CalendarModule;
```

Volver arriba

Convenciones de nombres de archivos

Mantenlo simple y en minúsculas, utiliza el nombre del componente, ejemplo calendar.*.js*, calendar-grid.*.js - con el nombre del tpo de archivo en el medio:

```
calendar.module.js
calendar.controller.js
calendar.component.js
calendar.service.js
calendar.directive.js
calendar.filter.js
calendar.spec.js
```

Volver arriba

Estructura de archivos escalable

La estructura de archivos es extremadamente importante, esto describe una estructura escalable y predecible. Un ejemplo de estructura de archivo para ilustrar una arquitectura de componentes modulares.

```
├── app/
│   ├── components/
│   │   ├── calendar/
│   │   │   ├── calendar.module.js
│   │   │   ├── calendar.controller.js
│   │   │   ├── calendar.component.js
│   │   │   ├── calendar.service.js
│   │   │   ├── calendar.spec.js
│   │   │   └── calendar-grid/
│   │   │       ├── calendar-grid.module.js
│   │   │       ├── calendar-grid.controller.js
│   │   │       ├── calendar-grid.component.js
│   │   │       ├── calendar-grid.directive.js
│   │   │       ├── calendar-grid.filter.js
│   │   │       └── calendar-grid.spec.js
│   │   ├── events/
│   │   │   ├── events.module.js
│   │   │   ├── events.controller.js
│   │   │   ├── events.component.js
│   │   │   ├── events.directive.js
│   │   │   ├── events.service.js
│   │   │   ├── events.spec.js
│   │   │   └── events-signup/
│   │   │       ├── events-signup.module.js
│   │   │       ├── events-signup.controller.js
│   │   │       ├── events-signup.component.js
│   │   │       ├── events-signup.service.js
│   │   │       └── events-signup.spec.js
│   │   └── components.module.js
│   ├── common/
│   │   ├── nav/
│   │   │       ├── nav.module.js
│   │   │       ├── nav.controller.js
│   │   │       ├── nav.component.js
│   │   │       ├── nav.service.js
│   │   │       └── nav.spec.js
│   │   ├── footer/
│   │   │       ├── footer.module.js
│   │   │       ├── footer.controller.js
│   │   │       ├── footer.component.js
│   │   │       ├── footer.service.js
│   │   │       └── footer.spec.js
│   │   └── common.module.js
│   ├── app.module.js
│   └── app.component.js
└── index.html
```

El nivel más alto de la estructura del directorio contiene simplemente index.html y app/, un directorio donde todos nuestro root, component, common y low-level modules viven.

Volver arriba

Componentes

Teoría

Los componentes son esencialmente plantillas con un controlador. *No* son directivas, ni debe sustituir las directivas con Componentes, a menos que estes actuliazando "template Directives" con controladores, que son los más adecuados para un componente. Los componentes tambien contienen bindings que definen los inputs y outputs de datos y eventos, lifecycle hooks y la habilidad de utilizar flujos de datos

unidireccionales y eventos de objetos para obtener copias de seguridad de un componente padre. Estos son el nuevo estándar de facto en AngularJS 1.5 y superior. Cualquier template y controlador que creemos será un componente, que puede ser stateful, stateless o routed component. Se puede pensar en un "component" como una pieza completa de código, no solo la defincion de objeto del .component(). Vamos a explorar algunas de las mejores prácticas y advertencias de los componentes y exploraremos la forma en que deben estructurarse a través de conceptos de componentes stateful, stateless y routed.

Volver arriba

Propiedades admitidas

Estas son las propiedades adminitidas para .component() que tu puedes/deberías utilizar:

Property	Support
bindings	Sí, usa solo '@', '<', '&'
controller	Sí
controllerAs	Sí, el valor por defecto es $ctrl
require	Sí (nueva sintaxis de objeto)
template	Sí
templateUrl	Sí
transclude	Sí

Volver arriba

Controllers

Los controladores solo deben ser utilizados junto con los componentes, nunca en otro lugar. Si sientes que necesitas un controlador, lo que realmente necesitas es problablemente un stateless component para manejar esa pieza particular de comportamiento.

Estas son algunas advertencias para usar Class en un controlador:

- Siempre use el constructor para propósitos de inyección de dependencias.
- No exportes la clasedirectamente, exporta su nombre para permitir anotaciones $inject
- Si necesitas acceder al lexical scope, utilizar arrow functions
- Una alternativa a las arrow functions, let ctrl = this; es también aceptable y puede tener más sentido según el caso de uso
- Liga todas las funciones publicas directamente a la Clase
- Haz uso del apropiado lifecycle hook, $onInit, $onChanges, $postLink y $onDestroy
    - Nota: $onChanges es llamado antes de $onInit, ve la sección de recursos para encontrar artículos que detallan esto en más profundidad.
- Utiliza require junto con $onInit para referencia una lógica heredada
- No sobreescribas el alias default $ctrl para la sintaxis controllerAs, por lo tanto no uses controllerAs en cualquier sitio.

Volver arriba

One-way dataflow y Eventos

One-way dataflow fue introducido en AngularJS 1.5, y redefine la comunicación de componentes.

Estas son algunas advertencias para usar one-way dataflow:

- En componentes que reciben datos, siempre utiliza la sintaxis one-way databindings '<'
- *No utilices* '=' la sintaxis two-way databiding nunca más, en nigún sitio

- Los componentes que tengan bindings deben itilizar $onChanges para clonar el one-way binding data para romper los objetos que para por referencia y actualizar la información de los padres.
- Utiliza $event como un argumento de la función en el método padre (mirá el ejemplo stateful abajo $ctrl.addTodo($event))
- Pasa un objeto de respaldo $event: {} de un stateless component (mirá el ejemplo stateless abajo this.onAddTodo)
    - Bonus: Utiliza una wrapper EventEmitter con .value() para reflejar Angular,evita la creación manual de objetos $event.
- ¿Por qué? Este refleja Angular y mantiene la consistencia dentro de cada componente. Si no que también hace un estado predecible.

Volver arriba

Stateful components

Vamos a definir a lo que llamaríamos un "stateful component".

- Obtiene el estado, esencialmente comunicando aun API de backend a través de un servicio
- No muta directamente de estado
- Renderea componentes hijo que mutan de estado
- También se denominan como un componente inteligente/contenedor

Un ejemplo de stateful component, completa con su definición de modulo low-level (esto es sólo para demostración, así que algún código ha sido omitido por razones de brevedad):

```
/* ----- todo/todo.component.js ----- */
import controller from './todo.controller';

const TodoComponent = {
  controller,
  template: `
    <div class="todo">
      <todo-form
        todo="$ctrl.newTodo"
        on-add-todo="$ctrl.addTodo($event);"></todo-form>
      <todo-list
        todos="$ctrl.todos"></todo-list>
    </div>
  `
};

export default TodoComponent;

/* ----- todo/todo.controller.js ----- */
class TodoController {
  constructor(TodoService) {
    this.todoService = TodoService;
  }
  $onInit() {
    this.newTodo = {
      title: '',
      selected: false
    };
    this.todos = [];
    this.todoService.getTodos().then(response => this.todos = response);
  }
  addTodo({ todo }) {
    if (!todo) return;
    this.todos.unshift(todo);
    this.newTodo = {
```

```
      title: '',
      selected: false
    };
  }
}

TodoController.$inject = ['TodoService'];

export default TodoController;

/* ----- todo/todo.module.js ----- */
import angular from 'angular';
import TodoComponent from './todo.component';

const TodoModule = angular
  .module('todo', [])
  .component('todo', TodoComponent)
  .name;

export default TodoModule;
```

Este ejemplo muestra un stateful component, que obtiene el estado dentro del controlador, a través de un servicios, y a continuación pasa hacia los componentes hijo de tipo stateless. Nota como no hay directivas siendo utilizadas como ng-repeat y relacionadas dentro del template. En cambio, los datos y funcionar son delegadas dentro de los stateless components <todo-form> y <todo-list>.

Volver arriba

Stateless components

Vamos a definir a lo que llamaríamos un "stateless component".

- Tiene inputs y outputs definidas utilizando bindings: {}
- Los datos ingresan al componente a través de atributos bindings (inputs)
- Los datos abandonan el componente a través de eventos (outputs)
- Mutación de estados, pasa a través del respaldo bajo demanda (como un click o envíando un evento)
- No le importa de donde vienen los datos, son stateless
- Son componentes altamente reusables
- También se denominan como un componente dumb/presentacional

Un ejemplo de stateless component (usemos <todo-form> como un ejemplo), completa con su definición de modulo low-level (esto es sólo para demostración, así que algún código ha sido omitido por razones de brevedad):

```
/* ----- todo/todo-form/todo-form.component.js ----- */
import controller from './todo-form.controller';

const TodoFormComponent = {
  bindings: {
    todo: '<',
    onAddTodo: '&'
  },
  controller,
  template: `
    <form name="todoForm" ng-submit="$ctrl.onSubmit();">
      <input type="text" ng-model="$ctrl.todo.title">
      <button type="submit">Submit</button>
    </form>
  `
};
```

```
export default TodoFormComponent;

/* ----- todo/todo-form/todo-form.controller.js ----- */
class TodoFormController {
  constructor(EventEmitter) {
      this.EventEmitter = EventEmitter;
  }
  $onChanges(changes) {
    if (changes.todo) {
      this.todo = Object.assign({}, this.todo);
    }
  }
  onSubmit() {
    if (!this.todo.title) return;
    // with EventEmitter wrapper
    this.onAddTodo(
      this.EventEmitter({
        newTodo: this.todo
      })
    );
    // without EventEmitter wrapper
    this.onAddTodo({
      $event: {
        newTodo: this.todo
      }
    });
  }
}

TodoFormController.$inject = ['EventEmitter'];

export default TodoFormController;

/* ----- todo/todo-form/todo-form.module.js ----- */
import angular from 'angular';
import TodoFormComponent from './todo-form.component';

const TodoFormModule = angular
  .module('todo.form', [])
  .component('todoForm', TodoFormComponent)
  .value('EventEmitter', payload => ({ $event: payload}))
  .name;

export default TodoFormModule;
```

Nota como el componente <todo-form> no obtiene ningún estado, simplemente lo recibe, muta un objeto a través de la logica del controlador asociada a él, y lo envía de regreso al componente padre a través de la propiedad bindings. En este ejemplo, el lifecycle hook $onChanges crear un clon del objeto binding inicial this.todo y lo reasigna, lo que significa que la información padre no es afectada hasta que se envía el formulario, junto a la nueva sintaxis one-way data flow '<'.

Volver arriba

Routed components

Vamos a definir a lo que llamaríamos un "routed component".

- Es esencialmente un componente stateful, con definición de rutas
- No más archivos router.js
- Utilizamos componentes routed para definir su propia lógica de ruteo
- Los datos de "ingreso" para el componente son realizados a través del route resolve(opcional, todavía disponible en el controlador a través de llamadas del service)

Para este ejemplo, vamos a tomar el componente existente <todo>, refactorizarlo para utilizar la definición route y bindings en el componente que recibe los datos (el secreto aquí con ui-router es la propiedad resolve que creamos, en este caso todoData mapea directamente a través de los bindings por nosotros). Lo tratamos como un routed component por que es esencialmente una "vista":

```
/* ----- todo/todo.component.js ----- */
import controller from './todo.controller';

const TodoComponent = {
  bindings: {
    todoData: '<'
  },
  controller,
  template: `
    <div class="todo">
      <todo-form
        todo="$ctrl.newTodo"
        on-add-todo="$ctrl.addTodo($event);"></todo-form>
      <todo-list
        todos="$ctrl.todos"></todo-list>
    </div>
  `
};

export default TodoComponent;

/* ----- todo/todo.controller.js ----- */
class TodoController {
  constructor() {}
  $onInit() {
    this.newTodo = {
      title: '',
      selected: false
    };
  }
  $onChanges(changes) {
    if (changes.todoData) {
      this.todos = Object.assign({}, this.todoData);
    }
  }
  addTodo({ todo }) {
    if (!todo) return;
    this.todos.unshift(todo);
    this.newTodo = {
      title: '',
      selected: false
    };
  }
}

export default TodoController;

/* ----- todo/todo.module.js ----- */
import angular from 'angular';
import TodoComponent from './todo.component';

const TodoModule = angular
  .module('todo', [])
  .component('todo', TodoComponent)
  .service('TodoService', TodoService)
  .config(($stateProvider, $urlRouterProvider) => {
    $stateProvider
      .state('todos', {
        url: '/todos',
        component: 'todo',
```

```
        resolve: {
          todoData: PeopleService => PeopleService.getAllPeople()
        }
      });
    $urlRouterProvider.otherwise('/');
  })
  .name;

export default TodoModule;
```

Volver arriba

Directivas

Teoría

Las directivas nos da template, scope bindings, bindToController, link y muchas otras cosas. El uso de estas debe ser cuidadosamente considerando ahora la existencia de .component(). Las directivas no deben declarar templates y controladores nunca más, o recibir información a través de bindings. Las directivas deben ser utilizadas solamente para decoración del DOM. Por esto, si necesitas eventos/APIS personalizadas y lógica, usa una directiva y ligalo al template dentro de un componente. Si necesitas una cantidad importante de manipulación del DOM, también está el lifecycle hoook $postLink para ser considerado, de cualquier manera este no es el lugar para migrar toda tu manipulación del DOM, usa una directiva si es posible para cosas no-Angular.

Estas son algunas de las advertencias para el uso de Directivas:

- Nunca utilizar templates, scope, bindToController o controladores
- Siempre usar restrict: 'A' en directivas
- Utiliza compile y link cuando sea necesario
- Recuerda destruir y desvincular event handlers dentro $scope.$on('$destroy', fn);

Volver arriba

Propiedades recomendades

Debido al hecho de que las directivas soportan más que lo hace .component() (template directives fueron el componente original), yo recomiendo limitar la definiciones de objeto de la directiva a solo estas propiedades, para evitar utilizar directivas incorrectamente:

Propiedad	Usarlo?	Razón
bindToController	No	Utiliza bindings en componentes
compile	Sí	Para pre-compilar manipulación/eventos DOM
controller	No	Utiliza un componente
controllerAs	No	Utiliza un componente
link functions	Sí	Para pre/post manipulación/eventos DOM
multiElement	Sí	Ver documentación
priority	Sí	Ver documentación
require	No	Utiliza un componente
restrict	Sí	Define el uso de la directive, utiliza siempre 'A'
scope	No	Utiliza un componente
template	No	Utiliza un componente

templateNamespace	Sí (si es necesario)	Ver documentación
templateUrl	No	Utiliza un componente
transclude	No	Utiliza un componente

Volver arriba

Constantes o Clases

Hay algunas maneras de abordar el uso de ES2015 y directivas, ya sea con una arrow function y la asignación más sencilla, o utilizando una Clase de ES2015. Selecciona lo mejor que sea para ti y tu equipo, manten en mente que Angular utiliza clases.

Aquí hay un ejemplo utilizando una constante con una Arrow function y expression wrapper, () => ({}) regresa un Objeto, (toma en cuenta las diferencias de uso en el interior de .directive()):

```
/* ----- todo/todo-autofocus.directive.js ----- */
import angular from 'angular';

const TodoAutoFocus = ($timeout) => ({
  restrict: 'A',
  link($scope, $element, $attrs) {
    $scope.$watch($attrs.todoAutofocus, (newValue, oldValue) => {
      if (!newValue) {
        return;
      }
      $timeout(() => $element[0].focus());
    });
  }
});

TodoAutoFocus.$inject = ['$timeout'];

export default TodoAutoFocus;

/* ----- todo/todo.module.js ----- */
import angular from 'angular';
import TodoComponent from './todo.component';
import TodoAutofocus from './todo-autofocus.directive';

const TodoModule = angular
  .module('todo', [])
  .component('todo', TodoComponent)
  .directive('todoAutofocus', TodoAutoFocus)
  .name;

export default TodoModule;
```

O utilizando una clases ES2015 (toma en cuenta la llamada manual de new TodoAutoFocus cuando se registra la directiva) para crear el objeto:

```
/* ----- todo/todo-autofocus.directive.js ----- */
import angular from 'angular';

class TodoAutoFocus {
  constructor($timeout) {
    this.restrict = 'A';
    this.$timeout = $timeout;
  }
  link($scope, $element, $attrs) {
```

```
    $scope.$watch($attrs.todoAutofocus, (newValue, oldValue) => {
      if (!newValue) {
        return;
      }
      this.$timeout(() => $element[0].focus());
    });
  }
}

TodoAutoFocus.$inject = ['$timeout'];

export default TodoAutoFocus;

/* ----- todo/todo.module.js ----- */
import angular from 'angular';
import TodoComponent from './todo.component';
import TodoAutofocus from './todo-autofocus.directive';

const TodoModule = angular
  .module('todo', [])
  .component('todo', TodoComponent)
  .directive('todoAutofocus', () => new TodoAutoFocus)
  .name;

export default TodoModule;
```

Volver arriba

Servicios

Teoría

Los servicios son esencialmente contenedores para la lógica de negocio que nuestros componentes no deben solicitar directamente. Los servicios contienen otros servicios incorporados o externos como lo es $http, que podemos inyectar dentro de los controladores de nuestro componente en otra parte de nuestra aplicación. Tenemos dos maneras de hacer servicios, utilizando .service() o .factory(). Con las Clases ES2015, solo debemos utilizar .services(), completa la anotación de inyección de dependiencias con $inject.

Volver arriba

Clases para Services

Aquí está un ejemplo de implementación para nuestro aplicación de todo utilizando una Clase ES2015:

```
/* ----- todo/todo.service.js ----- */
class TodoService {
  constructor($http) {
    this.$http = $http;
  }
  getTodos() {
    return this.$http.get('/api/todos').then(response => response.data);
  }
}

TodoService.$inject = ['$http'];

export default TodoService;

/* ----- todo/todo.module.js ----- */
```

```
import angular from 'angular';
import TodoComponent from './todo.component';
import TodoService from './todo.service';

const TodoModule = angular
  .module('todo', [])
  .component('todo', TodoComponent)
  .service('TodoService', TodoService)
  .name;

export default TodoModule;
```

Volver arriba

ES2015 y Herramientas

ES2015

- Utiliza Babel para compilar tu código ES2015+ code y cualquier polyfills
- Considera utilizar TypeScript para dar paso a cualquier actualización de Angular

Herramientas

- Utiliza ui-router latest alpha (ve el Readme) si tu quiere soporte de component-routing
    - De otra manera estarás atado a template: '<component>' y no bindings
- Considera utilizar Webpack para compilar tu código ES2015
- Utiliza ngAnnotate para anotar automáticamente propiedades en el $inject

Volver arriba

State management

Considera el uso de Redux con AngularJS 1.5 para la gestión de datos.

- Angular Redux

Volver arriba

Recursos

- Comprendiendo el método .component()
- Utilizando "require" con $onInit
- Comprendiendo todo el lifecycle hooks, $onInit, $onChanges, $postLink, $onDestroy
- Utilizando "resolve" en routes
- Redux y Angular state management
- Sample Application from Community

Volver arriba

Documentación

Para algo más, incluyendo referencia al API, revisa la documentación de AngularJS.

Contribuyendo

Abre un issue primero para discutir posibles cambios/adiciones. Por favor no abras issues para preguntas.

License