Contenido

Vocabulario de estilo:

Cada guía describe una buena o una mala práctica, y todas tienen una presentación coherente.

La redacción de cada directriz indica qué tan fuerte es la recomendación.

Esto es algo que siempre debe seguirse. Siempre puede ser una palabra demasiado fuerte. Las pautas que deben seguirse literalmente siempre son extremadamente raras. Por otro lado, necesita un caso realmente inusual para romper una guía de buenas prácticas.

Considere que se deben seguir las pautas. Si comprende completamente el significado detrás de la directriz y tiene una buena razón para desviarse, hágalo. Por favor, esfuércese por ser coherente.

Evitar indica algo que casi nunca debería hacer. Los ejemplos de código que se deben evitar tienen un encabezado rojo inconfundible.

Convenciones de estructura de archivos:

Algunos ejemplos de código muestran un archivo que tiene uno o más archivos complementarios con nombres similares. Por ejemplo, *hero.component.ts* y *hero.component.html*.

La guía utiliza el atajo hero.component.ts | html | css | spec para representar esos diversos archivos. El uso de este acceso directo hace que las estructuras de archivos de esta guía sean más fáciles de leer y concisas.

Responsabilidad Única:

Aplicar el principio de responsabilidad única (SRP) a todos los componentes, servicios y otros símbolos. Esto ayuda a que la aplicación sea más limpia, más fácil de leer y mantener y más comprobable.

Regla de uno

Estilo 01-01

Defina una cosa, como un servicio o componente, por archivo.

Considere limitar los archivos a 400 líneas de código.

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

¿Por qué? Un componente por archivo evita errores ocultos que a menudo surgen al combinar componentes en un archivo donde pueden compartir variables, crear cierres no deseados o acoplamientos no deseados con dependencias.

¿Por qué? Un solo componente puede ser la exportación predeterminada para su archivo, lo que facilita la carga diferida con el enrutador.

La clave es hacer que el código sea más reutilizable, más fácil de leer y menos propenso a errores.

El siguiente ejemplo negativo define el AppComponent, inicia la aplicación, define el objeto de modelo Hero y carga héroes desde el servidor, todo en el mismo archivo. No hagas esto.

app/heroes/hero.component.ts

```
/* avoid */
import { Component, NgModule, OnInit } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

interface Hero {
  id: number;
  name: string;
}

@Component({
  selector: 'app-root',
  template: `
      <h1>{{title}}</h1>
      <pre>{{heroes | json}}</pre>
    `,
  styleUrls: ['app/app.component.css']
})
class AppComponent implements OnInit {
  title = 'Tour of Heroes';

  heroes: Hero[] = [];

  ngOnInit() {
    getHeroes().then(heroes => (this.heroes = heroes));
  }
}
```

```
@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  exports: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule {}

platformBrowserDynamic().bootstrapModule(AppModule);

const HEROES: Hero[] = [
  { id: 1, name: 'Bombasto' },
  { id: 2, name: 'Tornado' },
  { id: 3, name: 'Magneta' }
];

function getHeroes(): Promise<Hero[]> {
  return Promise.resolve(HEROES); // TODO: get hero data from the server;
}
```

Pequeñas Funciones:

Estilo 01-02:

Defina pequeñas funciones

Considere limitar a no más de 75 líneas.

¿Por qué? Las funciones pequeñas son más fáciles de probar, especialmente cuando hacen una cosa y tienen un propósito.

¿Por qué? Las pequeñas funciones promueven la reutilización.

¿Por qué? Las funciones pequeñas son más fáciles de leer.

¿Por qué? Las funciones pequeñas son más fáciles de mantener.

¿Por qué? Las funciones pequeñas ayudan a evitar errores ocultos que vienen con funciones grandes que comparten variables con alcance externo, crean cierres no deseados o acoplamientos no deseados con dependencias.

Reglas de Nombrado:

Las convenciones de nomenclatura son muy importantes para la facilidad de mantenimiento y la legibilidad. Esta guía recomienda convenciones de nomenclatura para el nombre del archivo y el nombre del símbolo.

Pautas generales de nomenclatura

Estilo 02-01

Use nombres consistentes para todos los símbolos.

Siga un patrón que describa la característica del símbolo y luego su tipo. El patrón recomendado es feature.type.ts.

¿Por qué? Las convenciones de nomenclatura ayudan a proporcionar una forma coherente de buscar contenido de un vistazo. La coherencia dentro del proyecto es vital. La coherencia con un equipo es importante. La coherencia en una empresa proporciona una eficiencia tremenda.

¿Por qué? Las convenciones de nomenclatura deberían simplemente ayudar a encontrar el código deseado más rápido y hacerlo más fácil de entender.

¿Por qué? Los nombres de carpetas y archivos deben transmitir claramente su intención. Por ejemplo, app / heroes / hero-list.component.ts puede contener un componente que gestiona una lista de héroes.

Separación de los nombres de los archivos con puntos y guiones

Estilo 02-02

Utilice guiones para separar palabras en el nombre descriptivo.

Utilice puntos para separar el nombre descriptivo del tipo.

Utilice nombres de tipo coherentes para todos los componentes siguiendo un patrón que describa la característica del componente y luego su tipo. Un patrón recomendado es feature.type.ts.

Utilice nombres de tipos convencionales, incluidos .service, .component, .pipe, .module y .directive. Invente nombres de tipos adicionales si es necesario, pero tenga cuidado de no crear demasiados.

¿Por qué? Los nombres de tipo proporcionan una forma coherente de identificar rápidamente lo que hay en el archivo.

¿Por qué? Los nombres de tipo facilitan la búsqueda de un tipo de archivo específico mediante un editor o las técnicas de búsqueda difusa del IDE.

¿Por qué? Los nombres de tipos no abreviados, como .service, son descriptivos e inequívocos. Las abreviaturas como .srv, .svc y .serv pueden resultar confusas.

¿Por qué? Los nombres de tipo proporcionan una coincidencia de patrones para cualquier tarea automatizada.

Símbolos y nombres de archivos

Estilo 02-03

Use nombres consistentes para todos los activos nombrados después de lo que representan.

Utilice mayúsculas en camello para los nombres de las clases.

Haga coincidir el nombre del símbolo con el nombre del archivo.

Agregue el nombre del símbolo con el sufijo convencional (como Componente, Directiva, Módulo, Tubería o Servicio) para algo de ese tipo.

Dé al nombre de archivo el sufijo convencional (como .component.ts, .directive.ts, .module.ts, .pipe.ts o .service.ts) para un archivo de ese tipo.

¿Por qué? Las convenciones coherentes facilitan la identificación rápida y la referencia de activos de diferentes tipos.

Symbol Name	File Name
`@Component({ ... })` `export class AppComponent { }`	app.component.ts
`@Component({ ... })` `export class HeroesComponent { }`	heroes.component.ts
`@Component({ ... })` `export class HeroListComponent { }`	hero-list.component.ts
`@Component({ ... })` `export class HeroDetailComponent { }`	hero-detail.component.ts
`@Directive({ ... })` `export class ValidationDirective { }`	validation.directive.ts
`@NgModule({ ... })` `export class AppModule`	app.module.ts
`@Pipe({ name: 'initCaps' })` `export class InitCapsPipe implements PipeTransform { }`	init-caps.pipe.ts
`@Injectable()` `export class UserProfileService { }`	user-profile.service.ts

Nombres de servicios

Estilo 02-04

Utilice nombres coherentes para todos los servicios que tengan el nombre de su función.

Agregue el sufijo de un nombre de clase de servicio con Service. Por ejemplo, algo que obtiene datos o héroes debe llamarse DataService o HeroService.

Algunos términos son inequívocamente servicios. Por lo general, indican la agencia terminando en "-er". Es posible que prefiera nombrar un servicio que registra mensajes Logger en lugar de LoggerService. Decida si esta excepción es aceptable en su proyecto. Como siempre, esfuércese por la coherencia.

¿Por qué? Proporciona una forma coherente de identificar y hacer referencia a los servicios rápidamente.

¿Por qué? Los nombres de servicios claros, como Logger, no requieren un sufijo.

¿Por qué? Los nombres de servicios como Crédito son sustantivos y requieren un sufijo y se deben nombrar con un sufijo cuando no sea obvio si se trata de un servicio o de otra cosa.

Symbol Name	File Name
`@Injectable()` `export class HeroDataService { }`	hero-data.service.ts
`@Injectable()` `export class CreditService { }`	credit.service.ts
`@Injectable()` `export class Logger { }`	logger.service.ts

Bootstrapping

Estilo 02-05

Ponga Bootstrapping y lógica de plataforma para la aplicación en un archivo llamado main.ts.

Incluya el manejo de errores en la lógica de arranque.

Evite poner la lógica de la aplicación en *main.ts*. En su lugar, considere colocarlo en un componente o servicio.

¿Por qué? Sigue una convención coherente para la lógica de inicio de una aplicación.

¿Por qué? Sigue una convención familiar de otras plataformas tecnológicas.

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
  .then(success => console.log(`Bootstrap success`))
  .catch(err => console.error(err));
```

Selectores de componentes

Estilo 05-02

Utilice mayúsculas y minúsculas para nombrar los selectores de elementos de los componentes.

¿Por qué? Mantiene los nombres de los elementos coherentes con la especificación de Elementos personalizados.

app/heroes/shared/hero-button/hero-button.component.ts

```
/* avoid */

@Component({
  selector: 'tohHeroButton',
  templateUrl: './hero-button.component.html'
})
export class HeroButtonComponent {}
```

Prefijo personalizado de componente

Estilo 02-07

Utilice un valor selector de elementos en minúsculas y con guion; por ejemplo, admin-users.

Utilice un prefijo personalizado para un selector de componentes. Por ejemplo, el prefijo toh representa Tour of Heroes y el prefijo admin representa un área de funciones de administración.

Utilice un prefijo que identifique el área de funciones o la aplicación en sí.

¿Por qué? Evita las colisiones de nombres de elementos con componentes de otras aplicaciones y con elementos HTML nativos.

¿Por qué? Hace que sea más fácil promocionar y compartir el componente en otras aplicaciones.

¿Por qué? Los componentes son fáciles de identificar en el DOM.

app/heroes/hero.component.ts

```
/* avoid */

// HeroComponent is in the Tour of Heroes feature
@Component({
  selector: 'hero'
})
export class HeroComponent {}
```

app/users/users.component.ts

```
/* avoid */

// UsersComponent is in an Admin feature
@Component({
  selector: 'users'
})
export class UsersComponent {}
```

app/heroes/hero.component.ts

```
@Component({
  selector: 'toh-hero'
})
export class HeroComponent {}
```

app/users/users.component.ts

```
@Component({
  selector: 'admin-users'
})
export class UsersComponent {}
```

Selectores de directivas

Estilo 02-06

Utilice mayúsculas y minúsculas para nombrar a los selectores de directivas.

¿Por qué? Mantiene los nombres de las propiedades definidas en las directivas que están vinculadas a la vista de acuerdo con los nombres de los atributos.

¿Por qué? El analizador de HTML angular distingue entre mayúsculas y minúsculas y reconoce las minúsculas en camello.

Prefijo personalizado de directiva

Estilo 02-08

Utilice un prefijo personalizado para el selector de directivas (por ejemplo, el prefijo toh de Tour of Heroes).

Deletree los selectores de no elementos en minúsculas, a menos que el selector esté destinado a coincidir con un atributo HTML nativo.

¿Por qué? Evita colisiones de nombres.

¿Por qué? Las directivas se identifican fácilmente.

app/shared/validate.directive.ts

```
/* avoid */

@Directive({
  selector: '[validate]'
})
export class ValidateDirective {}
```

app/shared/validate.directive.ts

```
@Directive({
  selector: '[tohValidate]'
})
export class ValidateDirective {}
```

Nombres de tubería

Estilo 02-09

Utilice nombres coherentes para todas las tuberías, con el nombre de su función. El nombre de la clase de tubería debe usar UpperCamelCase (la convención general para los nombres de clases) y la cadena de nombre correspondiente debe usar lowerCamelCase. La cadena de nombre no puede utilizar guiones ("caso-guión" o "caso-kebab").

¿Por qué? Proporciona una forma coherente de identificar y hacer referencia a las tuberías rápidamente.

Symbol Name	File Name
`@Pipe({ name: 'ellipsis' })` `export class EllipsisPipe implements PipeTransform { }`	ellipsis.pipe.ts
`@Pipe({ name: 'initCaps' })` `export class InitCapsPipe implements PipeTransform { }`	init-caps.pipe.ts

Nombres de archivos de prueba unitaria

Estilo 02-10

Nombra los archivos de especificación de prueba del mismo modo que el componente que prueban.

Nombrar archivos de especificación de prueba con un sufijo de .spec.

¿Por qué? Proporciona una forma coherente de identificar pruebas rápidamente.

¿Por qué? Proporciona coincidencia de patrones para karma u otros corredores de prueba.

Test Type	File Names
Components	heroes.component.spec.ts hero-list.component.spec.ts hero-detail.component.spec.ts
Services	logger.service.spec.ts hero.service.spec.ts filter-text.service.spec.ts
Pipes	ellipsis.pipe.spec.ts init-caps.pipe.spec.ts

Nombres de archivos de prueba de extremo a extremo (E2E)

Estilo 02-11

Nombra los archivos de especificación de prueba de un extremo a otro después de la característica que prueban con un sufijo de .e2e-spec.

¿Por qué? Proporciona una forma coherente de identificar rápidamente las pruebas de un extremo a otro.

¿Por qué? Proporciona coincidencia de patrones para los corredores de pruebas y la automatización de compilaciones.

Test Type	File Names
End-to-End Tests	app.e2e-spec.ts heroes.e2e-spec.ts

Nombres de Angular NgModule

Estilo 02-12

Agregue el nombre del símbolo con el sufijo Módulo.

Déle al nombre del archivo la extensión .module.ts.

Nombra el módulo según la función y la carpeta en la que reside.

¿Por qué? Proporciona una forma coherente de identificar y hacer referencia a los módulos rápidamente.

¿Por qué? El caso de camello superior es convencional para identificar objetos que se pueden instanciar usando un constructor.

¿Por qué? Identifica fácilmente el módulo como la raíz de la misma función nombrada.

Agregue el sufijo de un nombre de clase RoutingModule con RoutingModule.

Termine el nombre de archivo de un RoutingModule con -routing.module.ts.

¿Por qué? Un RoutingModule es un módulo dedicado exclusivamente a configurar el enrutador angular. Una convención consistente de clases y nombres de archivos hace que estos módulos sean fáciles de detectar y verificar.

Symbol Name	File Name
`@NgModule({ ... })` `export class AppModule { }`	app.module.ts
`@NgModule({ ... })` `export class HeroesModule { }`	heroes.module.ts
`@NgModule({ ... })` `export class VillainsModule { }`	villains.module.ts
`@NgModule({ ... })` `export class AppRoutingModule { }`	app-routing.module.ts
`@NgModule({ ... })` `export class HeroesRoutingModule { }`	heroes-routing.module.ts

Estructura de la Aplicación y módulos NG

Todo el código de la app va en una carpeta llamada src. Todas las áreas características están en su propia carpeta, con su propio modulo NG

Todo el contenido es un activo por carpeta. Cada componente, servicio y tubería está en su propia carpeta. Todo otro script de tercera categoría es almacenado en otra carpeta y no en la carpeta src. Usa la convención de nombres para carpetas de esta guía.

LIFT

Estilo 04-01

Estructura la app para poder localizar el código rápidamente, identificar el código en un simple vistazo, mantener la estructura lo más plana posible.

Definir la estructura para poder seguir estas guías básicas, listadas en orden de importancia

¿Por qué? LIFT provee una consistente estructura escalable, es modular, y hace fácil incrementar la eficiencia de desarrollo encontrando código rápidamente. Para confirmar tu intuición sobre una estructura en particular, preguntante: ¿Puedo empezar a trabajar rápido en todas las carpetas relacionadas a esta característica?

ubicación

Estilo 04-02

Haz la ubicación del código intuitiva, simple y rapida

¿Por qué? Para trabajar eficientemente, debes poder encontrar las carpetas o archivos rápidamente, especialmente cuando no quieras (o recuerdas) el nombre de los archivos. Mantener archivos relacionados unos cerca de otros en una ubicación intuitiva ahorran tiempo. Una estructura de carpetas descriptivas hace una gran diferencia para vos y la gente que trabaja luego

Identificación

Estilo 04-03

El nombre del archivo debe permitirte saber automáticamente que contiene y representa

Se descriptivo con el nombre de archivos y carpetas, y mantener el contenido de los mismos en un solo componente

Evita archivos con múltiples componentes, múltiples servicios o mezclas de ambas.

¿Por qué? Se necesita menos tiempo para encontrar código, y se hace más eficiente dicha tarea. Nombres largos de archivos son mejores que nombres cortos y abreviados

Flat

Estilo 04-04

Mantener una estructura de carpetas plana el mayor tiempo posible

Considera crear subcarpetas cuando una carpeta archiva siete o más carpetas o archivos

Configura el IDE para ocultar distracciones, archivos irrelevantes como el generador .js y el .js.map

¿Por qué? Nadie quiere buscar un archivo por siete niveles de subcarpetas. Una estructura de carpetas plana es más fácil de escanear

Por otro lado, los psicólogos creen que los humanos empiezan a esforzarse cuando el número de objetos adyacentes excede los nueve. Entonces, cuando una carpeta tiene diez o más archivos, es hora de crear subcarpetas

Guías de estructura General

Estilo 04-06

Empieza de a poco, pero ten en mente hacia donde se dirige la app.

Ten una visión a corto y largo plazo de implementación

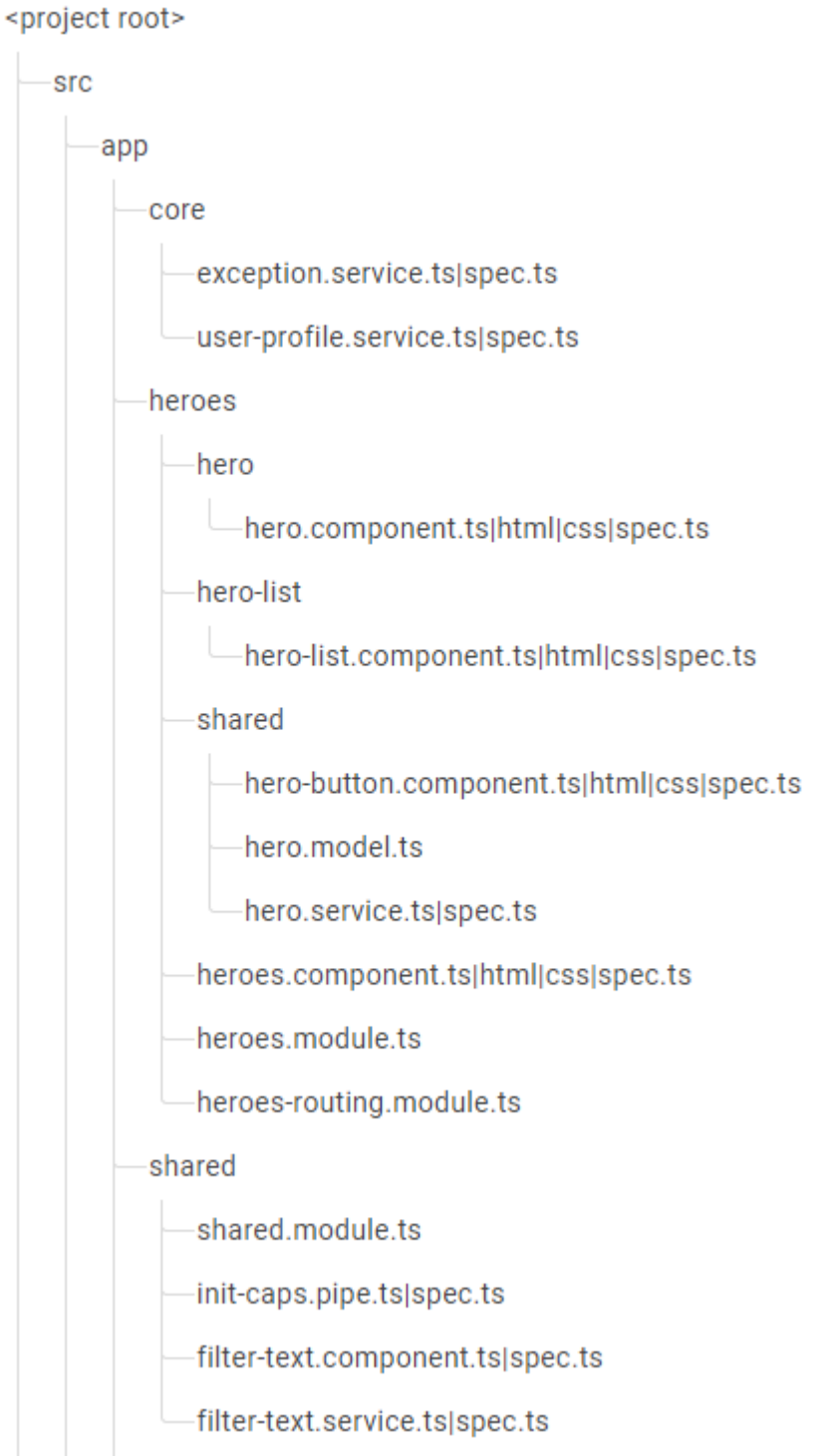
Pon todo el código de las apps en una carpeta llamada src.

Considera crear una carpeta para un componente cuando tiene múltiples archivos relacionados

¿Por qué? Ayuda a mantener la estructura de la app pequeña y fácil de mantener en las etapas tempranas, mientras resulta más fácil el crecimiento y evolución.

¿Por qué? Los componentes habitualmente tienen cuatro archivos .html, *.css, *.ts, and *.spec.ts por ejemplo y puede desordenar la carpeta fácilmente.

Aquí está la estructura de carpetas


<project root>
src
app
core
exception.service.ts|spec.ts
user-profile.service.ts|spec.ts
heroes
hero
hero.component.ts|html|css|spec.ts
hero-list
hero-list.component.ts|html|css|spec.ts
shared
hero-button.component.ts|html|css|spec.ts
hero.model.ts
hero.service.ts|spec.ts
heroes.component.ts|html|css|spec.ts
heroes.module.ts
heroes-routing.module.ts
shared
shared.module.ts
init-caps.pipe.ts|spec.ts
filter-text.component.ts|spec.ts
filter-text.service.ts|spec.ts

Estructura de Carpetas por características

Estilo 04-07

Crea carpetas llamadas según las áreas características que representan.

¿Por qué? Un desarrollador puede ubicar el código e identificar lo que cada archivo representa en un vistazo.

Cree un Módulo Ng para cada área de características.

¿Por qué? Los módulos NG facilitan la carga diferida de funciones enrutables.

¿Por qué? Los módulos NG facilitan el aislamiento, la prueba y la reutilización de funciones.

Módulo de la app raiz

Estilo 04-08

Cree un Módulo NG en la carpeta raíz de la aplicación, por ejemplo, en / src / app.

¿Por qué? Cada aplicación requiere al menos un Módulo NG raíz.

Considere nombrar el módulo raíz app.module.ts.

¿Por qué? Facilita la localización e identificación del módulo raíz.

app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { HeroesComponent } from './heroes/heroes.component';

@NgModule({
  imports: [
    BrowserModule,
  ],
  declarations: [
    AppComponent,
    HeroesComponent
  ],
  exports: [ AppComponent ],
  entryComponents: [ AppComponent ]
})
export class AppModule {}
```

Módulos de características

Cree un Módulo NG para todas las funciones distintas de una aplicación; por ejemplo, una función de héroes.

Coloque el módulo de funciones en la misma carpeta con el nombre que el área de funciones; por ejemplo, en app / héroes.

Nombra el archivo del módulo de funciones que refleje el nombre del área y la carpeta de funciones; por ejemplo, app / héroes / heroes.module.ts.

Nombra el símbolo del módulo de funciones que refleje el nombre del área, carpeta y archivo de funciones; por ejemplo, app / héroes / heroes.module.ts define HeroesModule.

¿Por qué? Un módulo de funciones puede exponer u ocultar su implementación de otros módulos.

¿Por qué? Un módulo de características identifica distintos conjuntos de componentes relacionados que comprenden el área de características.

¿Por qué? Un módulo de funciones puede enrutarse fácilmente tanto con entusiasmo como con pereza.

¿Por qué? Un módulo de funciones define límites claros entre funciones específicas y otras funciones de la aplicación.

¿Por qué? Un módulo de funciones ayuda a aclarar y facilitar la asignación de responsabilidades de desarrollo a diferentes equipos.

¿Por qué? Un módulo de funciones se puede aislar fácilmente para realizar pruebas.

Módulo de características Compartidas

cree un módulo de funciones llamado SharedModule en una carpeta compartida; por ejemplo, app / shared / shared.module.ts define SharedModule.

Declare componentes, directivas y canalizaciones en un módulo compartido cuando esos elementos serán reutilizados y referenciados por los componentes declarados en otros módulos de funciones.

Considere usar el nombre SharedModule cuando se hace referencia al contenido de un módulo compartido en toda la aplicación.

Considere no proporcionar servicios en módulos compartidos. Los servicios suelen ser singleton que se proporcionan una vez para toda la aplicación o en un módulo de funciones en particular. Sin embargo, existen excepciones. Por ejemplo, en el código de muestra que sigue, observe que SharedModule proporciona FilterTextService. Esto es aceptable aquí porque el servicio no tiene estado; es decir, los consumidores del servicio no se ven afectados por nuevas instancias.

Importe todos los módulos requeridos por los activos en SharedModule; por ejemplo, CommonModule y FormsModule.

¿Por qué? SharedModule contendrá componentes, directivas y canalizaciones que pueden necesitar características de otro módulo común; por ejemplo, ngFor en CommonModule.

Declare todos los componentes, directivas y canalizaciones en SharedModule

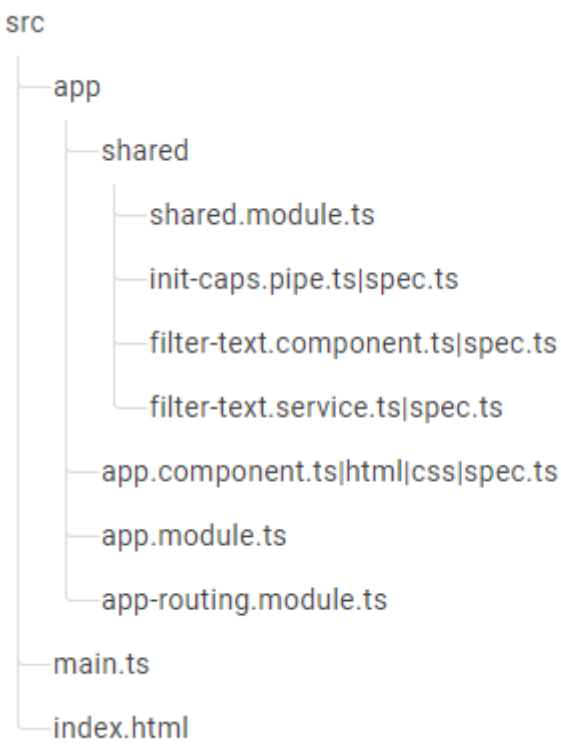
Exporte todos los símbolos del SharedModule que necesiten utilizar otros módulos de funciones.

¿Por qué? SharedModule existe para hacer que los componentes, directivas y conductos de uso común estén disponibles para su uso en las plantillas de componentes en muchos otros módulos.

Evite especificar proveedores de singleton para toda la aplicación en un SharedModule. Los singleton intencionales están bien. Cuídate.

¿Por qué? Un módulo de funciones de carga diferida que importa ese módulo compartido hará su propia copia del servicio y probablemente tenga resultados no deseados.

¿Por qué? No desea que cada módulo tenga su propia instancia separada de servicios singleton. Sin embargo, existe un peligro real de que eso suceda si SharedModule proporciona un servicio.



```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { FilterTextComponent } from './filter-text/filter-text.component';
import { FilterTextService } from './filter-text/filter-text.service';
import { InitCapsPipe } from './init-caps.pipe';

@NgModule({
  imports: [CommonModule, FormsModule],
  declarations: [
    FilterTextComponent,
    InitCapsPipe
  ],
  providers: [FilterTextService],
  exports: [
    CommonModule,
    FormsModule,
    FilterTextComponent,
    InitCapsPipe
  ]
})
export class SharedModule { }
```

COMPONENTES

Componentes como Elementos

Considere dar a los componentes un selector de elementos, en lugar de los selectores de atributos o clases.

¿Por qué? Los componentes tienen plantillas que contienen HTML y sintaxis de plantilla angular opcional. Muestran contenido. Los desarrolladores colocan componentes en la página como lo harían con elementos HTML nativos y componentes web.

¿Por qué? Es más fácil reconocer que un símbolo es un componente mirando el html de la plantilla.

app/heroes/hero-button/hero-button.component.ts

```
/* avoid */

@Component({
  selector: '[tohHeroButton]',
  templateUrl: './hero-button.component.html'
})
export class HeroButtonComponent {}
```

app/app.component.html

```
<!-- avoid -->

<div tohHeroButton></div>
```

app/heroes/shared/hero-button/hero-button.component.ts

```
@Component({
  selector: 'toh-hero-button',
  templateUrl: './hero-button.component.html'
})
export class HeroButtonComponent {}
```

Extrae moldes y estilos de tus propios archivos

Extraiga plantillas y estilos en un archivo separado, cuando haya más de 3 líneas.

Nombra el archivo de plantilla [nombre-componente]. component.html, donde [nombre-componente] es el nombre del componente.

Nombra el archivo de estilo [nombre-componente]. component.css, donde [nombre-componente] es el nombre del componente.

Especifique URL relativas al componente, con el prefijo./.

¿Por qué? Las plantillas y estilos grandes en línea oscurecen el propósito y la implementación del componente, lo que reduce la legibilidad y el mantenimiento.

¿Por qué? En la mayoría de los editores, las sugerencias de sintaxis y los fragmentos de código no están disponibles al desarrollar estilos y plantillas en línea. El servicio de lenguaje Angular TypeScript (de próxima aparición) promete superar esta deficiencia

para las plantillas HTML en aquellos editores que lo admitan; no ayudará con los estilos CSS.

¿Por qué? Una URL relativa a un componente no requiere cambios cuando mueve los archivos del componente, siempre que los archivos permanezcan juntos.

¿Por qué? El prefijo. / es una sintaxis estándar para URL relativas; no dependa de la capacidad actual de Angular para prescindir de ese prefijo.

app/heroes/heroes.component.ts

```
/* avoid */

@Component({
  selector: 'toh-heroes',
  template: `
    <div>
      <h2>My Heroes</h2>
      <ul class="heroes">
        <li *ngFor="let hero of heroes | async" (click)="selectedHero=hero">
          <span class="badge">{{hero.id}}</span> {{hero.name}}
        </li>
      </ul>
      <div *ngIf="selectedHero">
        <h2>{{selectedHero.name | uppercase}} is my hero</h2>
      </div>
    </div>
  `,
  styles: [`
    .heroes {
      margin: 0 0 2em 0;
      list-style-type: none;
      padding: 0;
      width: 15em;
    }
```

```
    .heroes li {
      cursor: pointer;
      position: relative;
      left: 0;
      background-color: #EEE;
      margin: .5em;
      padding: .3em 0;
      height: 1.6em;
      border-radius: 4px;
    }
    .heroes .badge {
      display: inline-block;
      font-size: small;
      color: white;
      padding: 0.8em 0.7em 0 0.7em;
      background-color: #607D8B;
      line-height: 1em;
      position: relative;
      left: -1px;
      top: -4px;
      height: 1.8em;
      margin-right: .8em;
      border-radius: 4px 0 0 4px;
    }
  `]
})
export class HeroesComponent implements OnInit {
  heroes: Observable<Hero[]>;
  selectedHero: Hero;

 constructor(private heroService: HeroService) { }
```

app/heroes/heroes.component.ts app/heroes/heroes.component.html

```
@Component({
  selector: 'toh-heroes',
  templateUrl: './heroes.component.html',
  styleUrls:  ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {
  heroes: Observable<Hero[]>;
  selectedHero: Hero;

 constructor(private heroService: HeroService) { }

  ngOnInit() {
    this.heroes = this.heroService.getHeroes();
  }
}
```

Decora propiedades de entrada y salida

Utilice los decoradores de clase @Input () y @Output () en lugar de las propiedades de entradas y salidas de los metadatos @Directive y @Component:

Considere colocar @Input () o @Output () en la misma línea que la propiedad que decora.

¿Por qué? Es más fácil y legible identificar qué propiedades de una clase son entradas o salidas.

¿Por qué? Si alguna vez necesita cambiar el nombre de la propiedad o del evento asociado con @Input () o @Output (), puede modificarlo en un solo lugar.

¿Por qué? La declaración de metadatos adjunta a la directiva es más corta y, por tanto, más legible.

¿Por qué? Colocar el decorador en la misma línea generalmente hace que el código sea más corto y aún identifica fácilmente la propiedad como una entrada o salida. Colóquelo en la línea de arriba cuando al hacerlo sea claramente más legible.

app/heroes/shared/hero-button/hero-button.component.ts

```
/* avoid */

@Component({
  selector: 'toh-hero-button',
  template: `<button></button>`,
  inputs: [
    'label'
  ],
  outputs: [
    'heroChange'
  ]
})
export class HeroButtonComponent {
  heroChange = new EventEmitter<any>();
  label: string;
}
```

app/heroes/shared/hero-button/hero-button.component.ts

```
@Component({
  selector: 'toh-hero-button',
  template: `<button>{{label}}</button>`
})
export class HeroButtonComponent {
  @Output() heroChange = new EventEmitter<any>();
  @Input() label: string;
}
```

Delegar lógica de componentes complejos a los servicios

Estilo 05-15

Limite la lógica en un componente a solo la requerida para la vista. Toda otra lógica debe delegarse a los servicios.

Mueva la lógica reutilizable a los servicios y mantenga los componentes simples y enfocados en su propósito previsto.

¿Por qué? La lógica puede ser reutilizada por múltiples componentes cuando se coloca dentro de un servicio y se expone a través de una función.

¿Por qué? La lógica de un servicio se puede aislar más fácilmente en una prueba unitaria, mientras que la lógica de llamada en el componente se puede burlar fácilmente.

¿Por qué? Elimina las dependencias y oculta los detalles de implementación del componente.

¿Por qué? Mantiene el componente delgado, recortado y enfocado.

```
/* avoid */

import { OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';
import { catchError, finalize } from 'rxjs/operators';

import { Hero } from '../shared/hero.model';

const heroesUrl = 'http://angular.io';

export class HeroListComponent implements OnInit {
  heroes: Hero[];
  constructor(private http: HttpClient) {}
  getHeroes() {
    this.heroes = [];
    this.http.get(heroesUrl).pipe(
      catchError(this.catchBadResponse),
      finalize(() => this.hideSpinner())
    ).subscribe((heroes: Hero[]) => this.heroes = heroes);
  }
  ngOnInit() {
    this.getHeroes();
  }

  private catchBadResponse(err: any, source: Observable<any>)
    // log and handle the exception
    return new Observable();
  }

  private hideSpinner() {
    // hide the spinner
```

app/heroes/hero-list/hero-list.component.ts

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

import { Hero, HeroService } from '../shared';

@Component({
  selector: 'toh-hero-list',
  template: `...`
})
export class HeroListComponent implements OnInit {
  heroes: Hero[];
  constructor(private heroService: HeroService) {}
  getHeroes() {
    this.heroes = [];
    this.heroService.getHeroes()
      .subscribe(heroes => this.heroes = heroes);
  }
  ngOnInit() {
    this.getHeroes();
  }
}
```

Poner lógica de presentación en la clase de componente

Estilo 05-17

Coloque la lógica de presentación en la clase de componente y no en la plantilla.

¿Por qué? La lógica estará contenida en un lugar (la clase de componente) en lugar de estar esparcida en dos lugares.

¿Por qué? Mantener la lógica de presentación del componente en la clase en lugar de la plantilla mejora la capacidad de prueba, el mantenimiento y la reutilización.

app/heroes/hero-list/hero-list.component.ts

```
/* avoid */

@Component({
  selector: 'toh-hero-list',
  template: `
    <section>
      Our list of heroes:
      <toh-hero *ngFor="let hero of heroes" [hero]="hero">
      </toh-hero>
      Total powers: {{totalPowers}}<br>
      Average power: {{totalPowers / heroes.length}}
    </section>
  `
})
export class HeroListComponent {
  heroes: Hero[];
  totalPowers: number;
}
```

Directivas

Usar directivas para mejorar un elemento

Style 06-01

Utilice directivas de atributos cuando tenga una lógica de presentación sin plantilla.

¿Por qué? Las directivas de atributos no tienen una plantilla asociada.

¿Por qué? Un elemento puede tener aplicada más de una directiva de atributo.

app/shared/highlight.directive.ts

```
@Directive({
  selector: '[tohHighlight]'
})
export class HighlightDirective {
  @HostListener('mouseover') onMouseEnter() {
    // do highlight work
  }
}
```

app/app.component.html

```
<div tohHighlight>Bombasta</div>
```

Decoradores de HostListener/HostBinding contra metadatos de host

Considere la posibilidad de preferir el @HostListener y el @HostBinding a la propiedad del host de los decoradores @Directive y @Component.

Ser consistente en su elección.

¿Por qué? La propiedad asociada a @HostBinding o el método asociado a @HostListener sólo se puede modificar en un único lugar, en la clase de la directiva. Si utiliza la propiedad de metadatos del host, debe modificar tanto la declaración de la propiedad/método en la clase de la directiva como los metadatos del decorador asociado a la directiva.

app/shared/validator.directive.ts

```
import { Directive, HostBinding, HostListener } from '@angular/core';

@Directive({
  selector: '[tohValidator]'
})
export class ValidatorDirective {
  @HostBinding('attr.role') role = 'button';
  @HostListener('mouseenter') onMouseEnter() {
    // do work
  }
}
```

Comparar con la alternativa de metadatos del host menos preferida.

¿Por qué? Los metadatos del host son sólo un término para recordar y no requieren importaciones adicionales de ES.

app/shared/validator2.directive.ts

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

@Directive({
  selector: '[tohValidator2]',
  host: {
    '[attr.role]': 'role',
    '(mouseenter)': 'onMouseEnter()'
  }
})
export class Validator2Directive {
  role = 'button';
  onMouseEnter() {
    // do work
  }
}
```

Services

Los servicios son monolitos (singletons)

Utilice los servicios como singleton dentro del mismo inyector. Úselos para compartir datos y funcionalidad.

¿Por qué? Los servicios son ideales para compartir métodos a través de un área de características o una aplicación.

app/heroes/shared/hero.service.ts

```
export class HeroService {
  constructor(private http: HttpClient) { }

  getHeroes() {
    return this.http.get<Hero[]>('api/heroes');
  }
}
```

¿Por qué? Los servicios son ideales para compartir el estado de los datos en memoria.

Servicios de Una sola responsabilidad

Crean servicios con una responsabilidad única que está encapsulada por su contexto.

Crear un nuevo servicio una vez que el servicio comienza a exceder ese propósito singular.

¿Por qué? Cuando un servicio tiene múltiples responsabilidades, se hace difícil de probar.

¿Por qué? Cuando un servicio tiene múltiples responsabilidades, cada componente o servicio que lo inyecta ahora lleva el peso de todas ellas.

Proporcionar un servicio

Proporcione un servicio con el inyector de la raíz de la aplicación en el @Injectable decorator del servicio.

¿Por qué? El inyector angular es jerárquico.

¿Por qué? Cuando se proporciona el servicio a un inyector de raíz, esa instancia del servicio se comparte y está disponible en cada clase que necesita el servicio. Esto es ideal cuando un servicio está compartiendo métodos o estados.

¿Por qué? Cuando registras un servicio en el decorador @Injectable del servicio, las herramientas de optimización como las utilizadas por los constructores de producción de la CLI angular pueden realizar "sacudidas de árbol" y eliminar servicios que no son utilizados por tu aplicación.

¿Por qué? Esto no es ideal cuando dos componentes diferentes necesitan diferentes instancias de un servicio. En este escenario sería mejor proporcionar el servicio al nivel del componente que necesita la nueva y separada instancia.

src/app/treeshaking/service.ts

```
@Injectable({
  providedIn: 'root',
})
export class Service {
}
```

Use the @Injectable() class decorator (Usa el decorador de la clase @Injectable())

Utiliza el decorador de clase @Injectable() en lugar del decorador de parámetros @Inject cuando utilices los tipos como fichas para las dependencias de un servicio.

¿Por qué? El mecanismo de Inyección de Dependencia Angular (DI) resuelve las propias dependencias de un servicio basándose en los tipos declarados de los parámetros constructores de ese servicio.

¿Por qué? Cuando un servicio acepta sólo las dependencias asociadas a los tipos de testigo, la sintaxis de @Injectable() es mucho menos verbosa en comparación con el uso de @Inject() en cada parámetro constructor individual.

app/heroes/shared/hero-arena.service.ts

```
/* avoid */

export class HeroArena {
  constructor(
      @Inject(HeroService) private heroService: HeroService,
      @Inject(HttpClient) private http: HttpClient) {}
}
```

app/heroes/shared/hero-arena.service.ts

```
@Injectable()
export class HeroArena {
  constructor(
    private heroService: HeroService,
    private http: HttpClient) {}
}
```

Data Services

Hablar con el servidor a través de un servicio

Hacer la lógica de refactorización para hacer operaciones de datos e interactuar con los datos a un servicio.

Hacer que los servicios de datos sean responsables de las llamadas XHR, el almacenamiento local, el almacenamiento en memoria o cualquier otra operación de datos.

¿Por qué? La responsabilidad del componente es la presentación y la recopilación de información para la vista. No debería importarle cómo obtiene los datos, sólo que sepa a quién pedírselos. Separar los servicios de datos mueve la lógica sobre cómo llevarlos al servicio de datos, y permite al componente ser más simple y estar más enfocado en la vista.

¿Por qué? Esto facilita la prueba (simulada o real) de las llamadas de datos cuando se prueba un componente que utiliza un servicio de datos.

¿Por qué? Los detalles de la gestión de datos, como los encabezados, los métodos HTTP, el almacenamiento en caché, el manejo de errores y la lógica de reintentos, son irrelevantes para los componentes y otros consumidores de datos.

Un servicio de datos encapsula estos detalles. Es más fácil evolucionar estos detalles dentro del servicio sin afectar a sus consumidores. Y es más fácil probar a los consumidores con implementaciones de servicio simuladas.

Lifecycle hooks

Usar los hooks del Ciclo de Vida para aprovechar los eventos importantes expuestos por Angular.

Implementar las interfaces de hook del cico de vida

¿Por qué? Las interfaces de hook de ciclo de vida prescriben firmas de métodos mecanografiadas. Usar esas firmas para marcar errores ortográficos y sintácticos.

app/heroes/shared/hero-button/hero-button.component.ts

```
/* avoid */

@Component({
  selector: 'toh-hero-button',
  template: `<button>OK<button>`
})
export class HeroButtonComponent {
  onInit() { // misspelled
    console.log('The component is initialized');
  }
}
```

app/heroes/shared/hero-button/hero-button.component.ts

```
@Component({
  selector: 'toh-hero-button',
  template: `<button>OK</button>`
})
export class HeroButtonComponent implements OnInit {
  ngOnInit() {
    console.log('The component is initialized');
  }
}
```

Anexo

Codelyzer

Un conjunto de reglas tslint para el análisis de código estático de proyectos de Angular con TypeScript. (https://www.npmjs.com/package/codelyzer)

File templates and snippets

Utiliza plantillas de archivos o snippets para ayudar a seguir estilos y patrones consistentes. Aquí hay plantillas y/o recortes para algunos de los editores de desarrollo web e IDEs.

Nosotros utilizamos Visual Studio Code con : https://marketplace.visualstudio.com/items?itemName=johnpapa.Angular2