

Guía de estilo de codificación de Angular

¿Busca una guía de opinión sobre la sintaxis angular, las convenciones y la estructura de aplicación? ¡Adelante! Esta guía de estilo presenta las convenciones preferidas y, lo que es más importante, explica por qué.

Vocabulario de estilo

Cada pauta describe una buena o mala práctica, y todas tienen una presentación consistente.

La redacción de cada directriz indica cuán fuerte es la recomendación.

Do es uno que siempre debe seguirse. *Siempre* puede ser una palabra demasiado fuerte. Las pautas que literalmente siempre deben seguirse son extremadamente raras. Por otro lado, necesita un caso realmente inusual para romper una directriz *Do*.

Considerar las pautas que generalmente se deben seguir. Si comprende completamente el significado detrás de la directriz y tiene una buena razón para desviarse, hágalo. Trate de ser coherente.

Evitar indica algo que casi nunca debes hacer. Los ejemplos de código para *evitar* tienen un encabezado rojo inconfundible.

¿Por qué? da razones para seguir las recomendaciones anteriores.

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 método abreviado `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 más concisas.

Una sola responsabilidad

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

No definir 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 leer, mantener y evitar colisiones con equipos en el 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 un acoplamiento no deseado 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 el código más reutilizable, más fácil de leer y menos propenso a errores.

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

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

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: obtener datos del héroe
}

```

Es una mejor práctica redistribuir el componente y sus clases de apoyo en sus propios archivos dedicados.

```

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

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

platformBrowserDynamic().bootstrapModule(AppModule);

```

```

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

```

```

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

```

```

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

```

```

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

import { HeroService } from './heroes';

```

```
@Component({
  selector: 'toh-app',
  template: `
    <toh-heroes></toh-heroes>
  `,
  styleUrls: ['./app.component.css'],
  providers: [HeroService]
})
export class AppComponent {}

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

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

@Component({
  selector: 'toh-heroes',
  template: `
    <pre>{{heroes | json}}</pre>
  `
})
export class HeroesComponent implements OnInit {
  heroes: Hero[] = [];

  constructor(private heroService: HeroService) {}

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

import { Injectable } from '@angular/core';

import { HEROES } from './mock-heroes';

@Injectable()
export class HeroService {
  getHeroes() {
    return Promise.resolve(HEROES);
  }
}

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

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

export const HEROES: Hero[] = [
  { id: 1, name: 'Bombasto' },
  { id: 2, name: 'Tornado' },
]
```

```
{ id: 3, name: 'Magneta' }  
];
```

A medida que crece la aplicación, esta regla se vuelve aún más importante. [Volver arriba](#)

Pequeñas funciones

Estilo 01-02

No definir funciones pequeños

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? 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 un alcance externo, crean cierres no deseados o un acoplamiento no deseado con dependencias.

[Volver al principio](#)

Naming

Las convenciones de denominación son muy importantes para la conservación y la legibilidad. En esta guía se recomiendan las convenciones de nomenclatura para el nombre del archivo y el nombre del símbolo.

Directrices generales de denominación

Estilo 02-01

Do utilizar nombres coherentes para todos los símbolos.

No seguir un patrón que describe función del símbolo entonces 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 consistencia dentro del proyecto es vital. La coherencia con un equipo es importante. La consistencia en una empresa proporciona una eficiencia tremenda.

¿Por qué? Las convenciones de nomenclatura deberían ayudar a encontrar el código deseado más rápido y facilitar su comprensión.

¿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.

[Volver al principio](#)

Separa los nombres de los archivos con puntos y guiones

Estilo 02-02

Hacer uso de guiones para separar las palabras en el nombre descriptivo.

¿Los puntos de uso para separar el nombre descriptivo del tipo.

No utilice nombres de tipos coherentes para todos los componentes siguiendo un patrón que describe función del componente entonces su tipo. Un patrón recomendado es `feature.type.ts`.

No utilice nombres de los tipos convencionales incluyendo `.service`, `.component`, `.pipe`, `.module` y `.directive`. Si es necesario, invente nombres de tipos adicionales, 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 utilizando un editor o las técnicas de búsqueda difusa de IDE.

¿Por qué? Los nombres de tipo no `.service` como `.service` son descriptivos y no ambiguos. Las abreviaturas como `.srv`, `.svc` y `.serv` pueden ser confusas.

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

[Volver al principio](#)

Símbolos y nombres de archivos

Estilo 02-03

Do utilizar nombres consistentes para todos los activos con nombres de lo que representan.

No utilizar el caso de camellos superior para los nombres de clase.

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

Do añadir el nombre de símbolo con el sufijo convencionales (tales como `Component` , `Directive` , `Module` , `Pipe` , o `Service`) para una cosa de ese tipo.

No dar el nombre de archivo del sufijo convencional (como `.component.ts` , `.directive.ts` , `.module.ts` , `.pipe.ts` , o `.service.ts`) para un archivo de ese tipo.

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

Nombre del símbolo

```
@Component({ ... })  
export class AppComponent { }
```

```
@Component({ ... })  
export class HeroesComponent { }
```

```
@Component({ ... })  
export class HeroListComponent { }
```

```
@Component({ ... })  
export class HeroDetailComponent { }
```

```
@Directive({ ... })  
export class ValidationDirective { }
```

```
@NgModule({ ... })  
export class AppModule
```

```
@Pipe({ name: 'initCaps' })  
export class InitCapsPipe implements PipeTransform { }
```

```
@Injectable()  
export class UserProfileService { }
```

[Volver al principio](#)

Nombres de servicio

Estilo 02-04

Do utilizar nombres consistentes para todos los servicios el nombre de su función.

Do sufijo un nombre de clase de servicio con `Service`. Por ejemplo, algo que obtiene datos o héroes debería llamarse `DataService` o `HeroService`.

Algunos términos son servicios inequívocos. Por lo general, indican agencia terminando en "-er". Es posible que prefiera nombrar un servicio que registra los mensajes `Logger` en lugar de `LoggerService`. Decide si esta excepción es aceptable en tu proyecto. Como siempre, luche por la consistencia.

¿Por qué? Proporciona una manera consistente de identificar y referenciar servicios rápidamente.

¿Por qué? Los nombres de servicio claros como `Logger` no requieren un sufijo.

¿Por qué? Los nombres de servicios como `Credit` son sustantivos y requieren un sufijo, y deben nombrarse con un sufijo cuando no es obvio si es un servicio u otra cosa.

Nombre del símbolo	Nombre del archivo
<pre>@Injectable() export class HeroDataService { }</pre>	hero-data.service.ts
<pre>@Injectable() export class CreditService { }</pre>	credit.service.ts
<pre>@Injectable() export class Logger { }</pre>	logger.service.ts

[Volver al principio](#)

Bootstrapping

Estilo 02-05

Hacer bootstrapping put y la lógica plataforma para la aplicación en un archivo llamado `main.ts`.

Do incluyen el tratamiento de errores en la lógica de programa previo.

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

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

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

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

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

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

[Volver al principio](#)

Los selectores de componentes

Estilo 05-02

Hacer uso de los casos discontinua o kebab de los casos para dar nombre a los selectores de elementos de componentes.

¿Por qué? Mantiene los nombres de los elementos coherentes con la especificación de los [elementos personalizados](#).

```
/* evitar */

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

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

[`<toh-hero-button></toh-hero-button>`](#)

[Volver al principio](#)

Prefijo personalizado del componente

Estilo 02-07

Hacer uso de un valor de selector guión, elemento minúsculas; por ejemplo, `admin-users`.

No usar un prefijo personalizado para un selector de componente. Por ejemplo, el prefijo `toh` representa **T** nuestro **o** f **H** eroes y el prefijo `admin` representa un área de características de administración.

No usar un prefijo que identifica el área de características o de la propia aplicación.

¿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.

```
/* evitar */

// HeroComponent está en la función Tour of Heroes
@Component({
  selector: 'hero'
})
export class HeroComponent {}

/* evitar */

// UsersComponent está en una función de administrador
@Component({
  selector: 'users'
})
export class UsersComponent {}

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

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

[Volver al principio](#)

Los selectores de la directiva

Estilo 02-06

Do menor uso camel case para nombrar los selectores de directivas.

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

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

[Volver al principio](#)

Prefijo personalizado de la directiva

Estilo 02-08

No utilizar un prefijo personalizado para el selector de directivas (por ejemplo, el prefijo `toh` de `T` nuestra `ofH eroes`).

No deletrear selectores no de elementos en minúsculas camello menos que el selector está destinado a coincidir con un atributo HTML nativo.

No coloque el prefijo `ng` en un nombre de directiva porque ese prefijo está reservado para Angular y su uso podría causar errores que son difíciles de diagnosticar.

¿Por qué? Previene colisiones de nombres.

¿Por qué? Las directivas se identifican fácilmente.

```
/* evitar */

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

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

[Volver al principio](#)

Los nombres de las tuberías

Estilo 02-09

Do utilizar nombres consistentes para todos los tubos, llamados así por su función. El nombre de la clase de tubería debe usar [UpperCamelCase](#) (la convención general para los nombres de clase), y la cadena de `name` correspondiente debe usar [lowerCamelCase](#). La cadena de `name` no puede usar guiones ("guion-caso" o "kebab-caso").

¿Por qué? Proporciona una forma consistente de identificar y referenciar rápidamente las tuberías.

Nombre del símbolo

```
@Pipe({ name: 'ellipsis' })
export class EllipsisPipe implements PipeTransform { }
```

```
@Pipe({ name: 'initCaps' })
export class InitCapsPipe implements PipeTransform { }
```

[Volver al principio](#)

Los nombres de los archivos de la unidad de pruebas

Estilo 02-10

Hacer archivos de especificación de prueba nombre el mismo que el componente que prueba.

Hacer archivos de especificación de prueba nombre con un sufijo de `.spec`.

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

¿Por qué? Proporciona coincidencia de patrones para [karma](#) u otros corredores de prueba.

Tipo de prueba	Nombres de archivos
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
-------	---

[Volver al principio](#)

End-to-End Nombres de archivo de prueba de extremo a extremo (E2E)

Estilo 02-11

Hacer denominar los archivos de especificación de prueba de extremo a extremo después de la función que ponen a prueba con un sufijo de `.e2e-spec`.

¿Por qué? Proporciona una manera consistente de identificar rápidamente las pruebas de extremo a extremo.

¿Por qué? Proporciona coincidencia de patrones para corredores de prueba y automatización de compilación.

Tipo de prueba	Nombres de archivos
Pruebas de extremo a extremo	app.e2e-spec.ts heroes.e2e-spec.ts

[Volver al principio](#)

Angularares NgModule nombres

Estilo 02-12

Hacer añadir el nombre de símbolo con el sufijo `Module`.

No dar el nombre de archivo del `.module.ts` extensión.

No nombrar el módulo después de la función y la carpeta en la que reside.

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

¿Por qué? La caja de camello superior es convencional para identificar objetos que pueden ser instanciados usando un constructor.

¿Por qué? Identifica fácilmente el módulo como la raíz de la característica con el mismo nombre.

Do sufijo A *RoutingModule* nombre de la clase con `RoutingModule`.

No terminará el nombre de archivo de un *RoutingModule* con `-routing.module.ts`.

¿Por qué? Un `RoutingModule` es un módulo dedicado exclusivamente a la configuración del enrutador angular. Una convención de clase y nombre de archivo consistente hace que estos módulos sean fáciles de detectar y verificar.

Nombre del símbolo	Nombre del archivo
<pre>@NgModule({ ... }) export class AppModule { }</pre>	app.module.ts
<pre>@NgModule({ ... }) export class HeroesModule { }</pre>	heroes.module.ts
<pre>@NgModule({ ... }) export class VillainsModule { }</pre>	villains.module.ts
<pre>@NgModule({ ... }) export class AppRoutingModule { }</pre>	app-routing.module.ts
<pre>@NgModule({ ... }) export class HeroesRoutingModule { }</pre>	heroes-routing.module.ts

[Volver al principio](#)

Estructura de la aplicación y módulos NgModules

Tenga una visión a corto plazo de la aplicación y una visión a largo plazo. Empiece con poco, pero tenga en cuenta hacia dónde se dirige la aplicación.

Todo el código de la aplicación va en una carpeta llamada `src`. Todas las áreas de características están en su propia carpeta, con su propio NgModule.

Todo el contenido es un activo por archivo. Cada componente, servicio y tubería está en su propio archivo. Todos los scripts de proveedores externos se almacenan en otra carpeta y no en la carpeta `src`. No los escribiste y no los quieres abarrotados de `src`. Use las convenciones de nomenclatura para archivos en esta guía. [Volver arriba](#)

LIFT

Estilo 04-01

Hacer estructura de la aplicación de este tipo que pueda **L** código Octate rápidamente, **me** identifí el código de un solo vistazo, mantenga el **F** estructura lattest que pueda, y **T** ry a ser seco.

Do definen la estructura de seguir estas cuatro pautas básicas, que se enumeran en orden de importancia.

¿Por qué? LIFT proporciona una estructura consistente que escala bien, es modular y facilita el aumento de la eficiencia del desarrollador al encontrar código rápidamente. Para confirmar su intuición sobre una estructura particular, pregunte: *¿puedo abrir rápidamente y comenzar a trabajar en todos los archivos relacionados para esta función?*

[Volver al principio](#)

Locate

Estilo 04-02

Haga que la localización del código sea intuitiva y rápida.

¿Por qué? Para trabajar de manera eficiente, debe poder encontrar archivos rápidamente, especialmente cuando no conoce (o no recuerda) los *nombres* de los archivos. Mantener archivos relacionados entre sí en una ubicación intuitiva ahorra tiempo. Una estructura de carpetas descriptiva hace una gran diferencia para usted y las personas que vienen después de usted.

[Volver al principio](#)

Identify

Estilo 04-03

Asigne un nombre al archivo de manera que sepa instantáneamente lo que contiene y representa.

Sea descriptivo con los nombres de archivo y mantenga el contenido del archivo exactamente en un componente.

Evite archivos con múltiples componentes, múltiples servicios o una mezcla.

¿Por qué? Pase menos tiempo buscando y picoteando códigos, y sea más eficiente. Los nombres de archivo más largos son mucho mejores que los nombres abreviados *cortos pero oscuros*.

Puede ser ventajoso desviarse de la regla de *una cosa por archivo* cuando tiene un conjunto de características pequeñas y estrechamente relacionadas que se descubren y entienden mejor en un solo archivo que como múltiples archivos. Ten cuidado con esta escapatoria.

[Volver al principio](#)

Flat

Estilo 04-04

No mantener una estructura de carpetas plana mayor tiempo posible.

Considere crear subcarpetas cuando una carpeta llegue a siete o más archivos.

Considere configurar el IDE para ocultar archivos irrelevantes y distractores, como los archivos `.js` y `.js.map` generados.

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

Por otro lado, los [psicólogos creen](#) que los humanos comienzan a luchar cuando el número de cosas interesantes adyacentes excede de nueve. Entonces, cuando una carpeta tiene diez o más archivos, puede ser el momento de crear subcarpetas.

Basa tu decisión en tu nivel de comodidad. Use una estructura más plana hasta que haya un valor obvio para crear una nueva carpeta.

[Volver al principio](#)

T-DRY (Intenta estar DRY)

Estilo 04-05

No será DRY (Do not Repeat Yourself).

Evite estar tan SECO que sacrifique la legibilidad.

¿Por qué? Estar SECO es importante, pero no crucial si sacrifica los otros elementos de LIFT. Por eso se llama *T-DRY*. Por ejemplo, es redundante nombrar una plantilla `hero-view.component.html` porque con la extensión `.html`, obviamente es una vista. Pero si algo no es obvio o se aleja de una convención, entonces explíquelo.

[Volver al principio](#)

Directrices estructurales generales

Estilo 04-06

No empezar poco a poco, pero tener en cuenta que la aplicación se dirige por el camino.

Tenga una visión a corto plazo de la implementación y una visión a largo plazo.

Do poner todo el código de la aplicación en una carpeta llamada `src`.

Considere crear una carpeta para un componente cuando tiene varios archivos acompañantes (`.ts`, `.html`, `.css` y `.spec`).

¿Por qué? Ayuda a mantener la estructura de la aplicación pequeña y fácil de mantener en las primeras etapas, mientras que es fácil de evolucionar a medida que la aplicación crece.

¿Por qué? Componentes suelen tener cuatro archivos (por ejemplo, `*.html`, `*.css`, `*.ts`, y `*.spec.ts`) y pueden estorbar una carpeta rápidamente.

Aquí hay una estructura de carpetas y archivos que cumple con los requisitos:

```
&lt;project root&gt;
  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
villains
  villain
  ...
  villain-list
  ...
  shared
    ...
  villains.component.ts|html|css|spec.ts
  villains.module.ts
  villains-routing.module.ts
  app.component.ts|html|css|spec.ts
  app.module.ts
  app-routing.module.ts
main.ts
index.html
...
node_modules/...
...
```

Aunque se prefiere ampliamente que los componentes estén en carpetas dedicadas, otra opción para las aplicaciones pequeñas es mantener los componentes planos (no en una carpeta dedicada). Esto añade hasta cuatro archivos a la carpeta existente, pero también reduce el anidamiento de la carpeta. Elija lo que elija, sé coherente.

[Volver al principio](#)

Folders-by-feature Estructura de carpetas por función

Estilo 04-07

Cree carpetas con el nombre del área de características que representan.

¿Por qué? Un desarrollador puede localizar el código e identificar lo que cada archivo representa de un vistazo. La estructura es tan plana como puede ser y no hay nombres repetitivos o redundantes.

¿Por qué? Las pautas de LIFT están todas cubiertas.

¿Por qué? Ayuda a evitar que la aplicación se abarrote al organizar el contenido y mantenerlo alineado con las pautas de LIFT.

¿Por qué? Cuando hay muchos archivos, por ejemplo 10+, localizarlos es más fácil con una estructura de carpetas consistente y más difícil en una estructura plana.

Cree un NgModule para cada área de características.

¿Por qué? NgModules facilita la carga diferida de funciones enrutables.

¿Por qué? NgModules facilita el aislamiento, la prueba y la reutilización de funciones.

Para obtener más información, consulte [este ejemplo de carpeta y estructura de archivos](#).

[Volver al principio](#)

root module aplicación

Estilo 04-08

Cree un NgModule en la carpeta raíz de la aplicación, por ejemplo, en `/src/app` .

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

Considere nombrar el módulo raíz `app.module.ts` .

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

```
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 ]
})
export class AppModule {}
```

[Volver al principio](#)

Módulos de características

Estilo 04-09

Cree un NgModule para todas las características distintas en una aplicación; por ejemplo, una característica de `Heroes` .

Hacer colocar el módulo de función en la misma carpeta con el nombre como el área de características; por ejemplo, en `app/heroes` .

Asigne un nombre al archivo del módulo de funciones que refleje el nombre del área de funciones y la carpeta; por ejemplo, `app/heroes/heroes.module.ts` .

Hacer nombrar el símbolo de módulo de función que refleja el nombre de la zona característica, carpeta y archivo; por ejemplo, `app/heroes/heroes.module.ts` define `HeroesModule` .

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

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

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

¿Por qué? Un módulo de características define límites claros entre la funcionalidad específica y otras características 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.

[Volver al principio](#)

Módulo de características compartidas

Estilo 04-10

Cree un módulo de funciones denominado `SharedModule` en una carpeta `shared`; por ejemplo, `app/shared/shared.module.ts` define `SharedModule` .

Hacer componentes de declarar, directivas y tuberías en un módulo compartido cuando esos artículos serán re-utilizado y referenciado por los componentes declarados en otros módulos de características.

Considere usar el nombre `SharedModule` cuando se hace referencia a los contenidos de un módulo compartido en toda la aplicación.

Considere no proporcionar servicios en módulos compartidos. Los servicios suelen ser singlettons que se proporcionan una vez para toda la aplicación o en un módulo de características en particular. Hay excepciones, sin embargo. 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 las nuevas instancias.

No importar todos los módulos requeridos por los activos en el `SharedModule`; por ejemplo, `CommonModule` y `FormsModule`.

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

Do declarar todos los componentes, directivas y tuberías en el `SharedModule`.

No exportar todos los símbolos de la `SharedModule` que otros módulos de características necesitan para su uso.

¿Por qué? `SharedModule` existe para hacer que los componentes, directivas y tuberías 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 singletons intencionales están bien. Cuídate.

¿Por qué? Un módulo de funciones con carga lenta que importa ese módulo compartido hará su propia copia del servicio y probablemente tendrá 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.

```
src
  app
    shared
      shared.module.ts
      init-caps.pipe.ts|spec.ts
      filter-text.component.ts|spec.ts
      filter-text.service.ts|spec.ts
      app.component.ts|html|css|spec.ts
      app.module.ts
      app-routing.module.ts
    main.ts
    index.html
  ...

```

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

import { FilterTextComponent } from './filter-text/filter-text.cor
import { FilterTextService } from './filter-text/filter-text.serv:
import { InitCapsPipe } from './init-caps.pipe';

@NgModule({
  imports: [CommonModule, FormsModule],
```

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

```
< >
```

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'initCaps' })
export class InitCapsPipe implements PipeTransform {
  transform = (value: string) => value;
}

import { Component, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'toh-filter-text',
  template: '<input type="text" id="filterText" [(ngModel)]="filter">',
})
export class FilterTextComponent {
  @Output() changed: EventEmitter<string>;
  filter = '';

  constructor() {
    this.changed = new EventEmitter<string>();
  }

  clear() {
    this.filter = '';
  }

  filterChanged(event: any) {
    event.preventDefault();
    console.log(`Filter Changed: ${this.filter}`);
    this.changed.emit(this.filter);
  }
}
```

```
< >
```

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

@Injectable()
export class FilterTextService {
  constructor() {
    console.log('Created an instance of FilterTextService');
  }
}
```

```
}

filter(data: string, props: Array<string>, originalList: Array<any>): any {
  let filteredList: any[] = [];
  if (data && props && originalList) {
    data = data.toLowerCase();
    const filtered = originalList.filter(item => {
      let match = false;
      for (const prop of props) {
        if (item[prop].toString().toLowerCase().indexOf(data) > -1) {
          match = true;
          break;
        }
      }
      return match;
    });
    filteredList = filtered;
  } else {
    filteredList = originalList;
  }
  return filteredList;
}
}
```



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

import { FilterTextService } from '../shared/filter-text/filter-text.service';

@Component({
  selector: 'toh-heroes',
  templateUrl: './heroes.component.html'
})
export class HeroesComponent {

  heroes = [
    { id: 1, name: 'Windstorm' },
    { id: 2, name: 'Bombasto' },
    { id: 3, name: 'Magneta' },
    { id: 4, name: 'Tornado' }
  ];

  filteredHeroes = this.heroes;

  constructor(private filterService: FilterTextService) { }

  filterChanged(searchText: string) {
    this.filteredHeroes = this.filterService.filter(searchText, [
    ])
  }
}
```



```
<div>This is heroes component</div>
<ul>
  <li *ngFor="let hero of filteredHeroes">
    {{hero.name}}
  </li>
</ul>
```

```
</li>
</ul>
<toh-filter-text (changed)="filterChanged($event)"></toh-filter-t
```

[Volver al principio](#)

Carpetas cargadas perezosamente

Estilo 04-11

Una característica de aplicación o flujo de trabajo distinto puede *cargarse de forma diferida* o *cargarse a pedido* en lugar de cuando se inicia la aplicación.

Do puso a los contenidos de características cargados perezosos en una *carpeta de carga perezosa*. Una *carpeta con carga diferida* típica contiene un *componente de enrutamiento*, sus componentes secundarios y sus activos y módulos relacionados.

¿Por qué? La carpeta facilita la identificación y el aislamiento del contenido de la función.

[Volver al principio](#)

Nunca importe directamente carpetas cargadas perezosamente

Estilo 04-12

Evite permitir que los módulos en las carpetas hermana y principal importen directamente un módulo en una *función de carga diferida*.

¿Por qué? La importación directa y el uso de un módulo lo cargarán inmediatamente cuando la intención sea cargarlo a pedido.

[Volver al principio](#)

No añadir lógica de filtrado y ordenación a las tuberías

Style 04-13

Evite agregar lógica de filtrado o clasificación en conductos personalizados.

Calcule previamente la lógica de filtrado y clasificación en componentes o servicios antes de vincular el modelo en plantillas.

¿Por qué? El filtrado y especialmente la clasificación son operaciones costosas. Como Angular puede llamar a métodos de canalización muchas veces por segundo, las operaciones de clasificación y filtrado pueden degradar gravemente la experiencia del usuario incluso para listas de tamaño moderado.

[Volver al principio](#)

Components

Los componentes como elementos

Estilo 05-03

Consideré dar a los componentes un selector de *elementos* , en oposición a *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.

Hay algunos casos en los que le da un atributo a un componente, como cuando desea aumentar un elemento incorporado. Por ejemplo, [Material Design](#) utiliza esta técnica con `<button mat-button>`. Sin embargo, no usaría esta técnica en un elemento personalizado.

```
/* evitar */

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

```
<! - evitar ->

<div tohHeroButton></div>
```

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

[Volver al principio](#)

Extraer plantillas y estilos a sus propios archivos

Estilo 05-04

Hacer plantillas y estilos de extracto en un archivo separado, cuando más de 3 líneas.

No nombrar el archivo de plantilla `[component-name].component.html`, donde [nombre-componente] es el nombre del componente.

No nombrar el archivo de estilo `[component-name].component.css`, donde [nombre-componente] es el nombre del componente.

No especifique *componentes pariente* URL, con el prefijo `./`.

¿Por qué? Las plantillas y estilos grandes en línea oscurecen el propósito y la implementación del componente, reduciendo la legibilidad y la facilidad de mantenimiento.

¿Por qué? En la mayoría de los editores, las sugerencias de sintaxis y los fragmentos de código no están disponibles al desarrollar plantillas y estilos en línea. Angular TypeScript Language Service (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? La URL *relativa de 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 las URL relativas; no dependa de la capacidad actual de Angular para prescindir de ese prefijo.

```
/* 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>
  `,
  styleUrls: ['./toh-heroes.component.css']
})
```

```

        </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 {
    heroes: Observable<Hero[]>;
    selectedHero!: Hero;

    constructor(private heroService: HeroService) {
        this.heroes = this.heroService.getHeroes();
    }
}

```



```

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

    constructor(private heroService: HeroService) {
        this.heroes = this.heroService.getHeroes();
    }
}

```

```

        }
    }

<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>

```



```

.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;
}

```

[Volver al principio](#)

Decorar propiedades de `input` y `output`

Estilo 05-12

Hacer uso de la `@Input()` y `@Output()` decoradores de clase en lugar de las `inputs` y `outputs` las propiedades de la `@Directive` y `@Component` de metadatos:

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

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

¿Por qué? Si alguna vez necesita cambiar el nombre de la propiedad o el nombre 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 lo tanto, más legible.

¿Por qué? Colocar el decorador en la misma línea *generalmente* genera un código más corto y aún identifica fácilmente la propiedad como entrada o salida. Póngalo en la línea de arriba cuando hacerlo sea claramente más legible.

```
/* evitar */

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

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

[Volver al principio](#)

Evite el alias de `inputs` y `outputs`

Estilo 05-13

Evite los alias de *entrada* y *salida*, excepto cuando tenga un propósito importante.

¿Por qué? Dos nombres para la misma propiedad (uno privado, uno público) es intrínsecamente confuso.

¿Por qué? Debe usar un alias cuando el nombre de la directiva también sea una propiedad de *entrada* y el nombre de la directiva no describa la propiedad.

```
/ * evitar aliasing sin sentido * /  
  
@Component({  
  selector: 'toh-hero-button',  
  template: `<button>{{label}}</button>`  
})  
export class HeroButtonComponent {  
  // Alias sin sentido  
  @Output('heroChangeEvent') heroChange = new EventEmitter<any>();  
  @Input('labelAttribute') label: string;  
}  
  
<!-- evitar --&gt;<br/>  
<toh-hero-button labelAttribute="OK" (changeEvent)="doSomething()">  
</toh-hero-button>  
  
@Component({  
  selector: 'toh-hero-button',  
  template: `<button>{{label}}</button>`  
})  
export class HeroButtonComponent {  
  // Sin alias  
  @Output() heroChange = new EventEmitter<any>();  
  @Input() label = '';  
}  
  
import { Directive, ElementRef, Input, OnChanges } from '@angular/  
@Directive({ selector: '[heroHighlight]' })  
export class HeroHighlightDirective implements OnChanges {  
  
  // Alias porque `color` es un nombre de propiedad mejor que `hero  
  @Input('heroHighlight') color = '';  
  
  constructor(private el: ElementRef) {}  
  
  ngOnChanges() {  
    this.el.nativeElement.style.backgroundColor = this.color || ''  
  }  
}
```

```
    }
}
```

```
<toh-hero-button label="OK" (change)="doSomething()">
</toh-hero-button>

<! - `heroHighlight` es tanto el nombre de la directiva como el n
<h3 heroHighlight="skyblue">The Great Bombasto</h3>
```

[Volver al principio](#)

Secuencia de miembros

Estilo 05-14

Hacer lugar propiedades hasta la parte superior seguido por métodos.

Hacer lugar después de que miembros miembros privados públicas, en orden alfabético.

¿Por qué? Colocar a los miembros en una secuencia consistente hace que sea fácil de leer y ayuda a identificar instantáneamente qué miembros del componente sirven para qué propósito.

```
/* evitar */

export class ToastComponent implements OnInit {

  private defaults = {
    title: '',
    message: 'May the Force be with you'
  };
  message: string;
  title: string;
  private toastElement: any;

  ngOnInit() {
    this.toastElement = document.getElementById('toh-toast');
  }

  // métodos privados
  private hide() {
    this.toastElement.style.opacity = 0;
    window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);
  }

  activate(message = this.defaults.message, title = this.defaults.title) {
    this.title = title;
    this.message = message;
    this.show();
  }
}
```

```

private show() {
  console.log(this.message);
  this.toastElement.style.opacity = 1;
  this.toastElement.style.zIndex = 9999;

  window.setTimeout(() => this.hide(), 2500);
}
}

export class ToastComponent implements OnInit {
  // propiedades públicas
  message = '';
  title = '';

  // campos privados
  private defaults = {
    title: '',
    message: 'May the Force be with you'
  };
  private toastElement: any;

  // métodos públicos
  activate(message = this.defaults.message, title = this.defaults.title) {
    this.title = title;
    this.message = message;
    this.show();
  }

  ngOnInit() {
    this.toastElement = document.getElementById('toh-toast');
  }

  // métodos privados
  private hide() {
    this.toastElement.style.opacity = 0;
    window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);
  }

  private show() {
    console.log(this.message);
    this.toastElement.style.opacity = 1;
    this.toastElement.style.zIndex = 9999;
    window.setTimeout(() => this.hide(), 2500);
  }
}

```

[Volver al principio](#)

Delegar la lógica de los componentes complejos a los servicios

Hacer lógica límite en un componente para que sólo se requiere para la vista. Toda otra lógica debe delegarse a los servicios.

No mover la lógica reutilizable a los servicios y mantener los componentes sencillo y centrado en su propósito previsto.

¿Por qué? Varios componentes pueden reutilizar la lógica cuando se colocan dentro de un servicio y se exponen como una función.

¿Por qué? La lógica en 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 dependencias y oculta detalles de implementación del componente.

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

```
/* evitar */

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>) {
    // registra y maneja la excepción
    return new Observable();
  }

  private hideSpinner() {
    // esconde la ruleta
  }
}

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();
  }
}

```

[Volver al principio](#)

No prefijas `output` propiedades de salida

Estilo 05-16

Hacer eventos de nombre sin el prefijo `on`.

Hacer nombre métodos de controlador de eventos con el prefijo `on` seguida del nombre del evento.

¿Por qué? Esto es coherente con los eventos integrados, como los clics en los botones.

¿Por qué? Angular permite una [sintaxis alternativa](#) `on-*`. Si el evento en sí tuviera el prefijo `on`, esto daría como resultado una expresión de enlace `on-onEvent`.

```

/* evitar */

@Component({
  selector: 'toh-hero',
  template: `...
})

export class HeroComponent {
  @Output() onSaveTheDay = new EventEmitter<boolean>();
}

<! - evitar ->

<toh-hero (onSavedTheDay)="onSavedTheDay($event)"></toh-hero>

```

```

export class HeroComponent {
  @Output() savedTheDay = new EventEmitter<boolean>();
}

<toh-hero (savedTheDay)="onSavedTheDay($event)"></toh-hero>

```

[Volver al principio](#)

Poner la lógica de presentación en la clase de componentes

Estilo 05-17

Do puso 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 extenderse 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, la capacidad de mantenimiento y la reutilización.

```

/* evitar */

@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;
}

@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: {{avgPower}}
    </section>
  `
})

```

```

        </section>
      }

export class HeroListComponent {
  heroes: Hero[];
  totalPowers = 0;

  get avgPower() {
    return this.totalPowers / this.heroes.length;
  }
}

```

[Volver al principio](#)

Initialize inputs

Style 05-18

La opción del compilador `--strictPropertyInitialization` de TypeScript asegura que una clase inicialice sus propiedades durante la construcción. Cuando está habilitada, esta opción hace que el compilador de TypeScript informe un error si la clase no establece un valor para ninguna propiedad que no esté explícitamente marcada como opcional.

Por diseño, Angular trata todas las propiedades de `@Input` como opcionales. Cuando sea posible, debe satisfacer `--strictPropertyInitialization` proporcionando un valor predeterminado.

```

@Component({
  selector: 'toh-hero',
  template: `...
})
export class HeroComponent {
  @Input() id = 'default_id';
}

```

Si es difícil construir un valor predeterminado para la propiedad, utilice `?` para marcar explícitamente la propiedad como opcional.

```

@Component({
  selector: 'toh-hero',
  template: `...
})
export class HeroComponent {
  @Input() id?: string;

  process() {
    if (this.id) {
      // ...
    }
  }
}

```

Es posible que desee tener un campo `@Input` obligatorio , lo que significa que todos los usuarios de sus componentes deben pasar ese atributo. En tales casos, utilice un valor predeterminado. ¡Simplemente suprimiendo el error de TypeScript con `!` es insuficiente y debe evitarse porque evitará que el verificador de tipo se asegure de proporcionar el valor de entrada.

```
@Component({
  selector: 'toh-hero',
  template: `...
})
export class HeroComponent {
  // El signo de exclamación suprime los errores de que una propiedad
  // no inicializado.
  // Ignorar esta aplicación puede evitar que el verificador de tipos
  // de encontrar problemas potenciales.
  @Input() id!: string;
}
```

Directives

Usar directivas para mejorar un elemento

Estilo 06-01

Hacer uso de directivas de atributos cuando se tiene la lógica de presentación sin una plantilla.

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

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

```
@Directive({
  selector: '[tohHighlight]'
})
export class HighlightDirective {
  @HostListener('mouseover') onMouseEnter() {
    // resaltar el trabajo
  }
}
```

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

[Volver al principio](#)

HostListener / HostBinding versus metadatos del host

Estilo 06-03

Consideré la posibilidad de preferir `@HostListener` y `@HostBinding` a la propiedad de `host` de los decoradores `@Directive` y `@Component` .

Sea consistente en su elección.

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

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

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

Compare con la alternativa de metadatos de `host` menos preferida .

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

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

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

[Volver al principio](#)

Services

Los servicios son monolíticos

Estilo 07-01

Hacer servicios de uso como simple dentro del mismo inyector. Úselos para compartir datos y funcionalidad.

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

¿Por qué? Los servicios son ideales para compartir datos en memoria con estado.

```
export class HeroService {  
  constructor(private http: HttpClient) { }  
  
  getHeroes() {  
    return this.http.get<Hero[]>('api/heroes');  
  }  
}
```

[Volver al principio](#)

English

Consider using [snippets](#) for [Sublime Text](#) that follow these styles and guidelines.

[Feedback](#)

[Hide](#)

ada por

Cree un nuevo servicio una vez que el servicio comience a exceder ese propósito singular.

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

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

[Volver al principio](#)

Proporcionar un servicio

Estilo 07-03

Do proporcionar un servicio con el inyector raíz de la aplicación en el `@Injectable` decorador del servicio.

¿Por qué? El inyector angular es jerárquico.

¿Por qué? Cuando proporciona el servicio a un inyector raíz, esa instancia del servicio se comparte y está disponible en todas las clases que lo necesitan. Esto es ideal cuando un servicio comparte métodos o estados.

¿Por qué? Cuando registra un servicio en el decorador `@Injectable` del servicio, las herramientas de optimización como las utilizadas por las compilaciones de producción de [Angular CLI](#) pueden realizar sacudidas de árboles y eliminar servicios que su aplicación no utiliza.

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

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

[Volver al principio](#)

Usa el decorador de la clase `@Injectable()`

Estilo 07-04

No utilizar el `@Injectable()` decorador de clase en lugar de la `@Inject` parámetro decorador utilizando tipos como fichas para las dependencias de un servicio.

¿Por qué? El mecanismo de inyección de dependencia angular (DI) resuelve las dependencias propias de un servicio en función de los tipos declarados de los parámetros del constructor de ese servicio.

¿Por qué? Cuando un servicio acepta solo dependencias asociadas con tokens de tipo, la sintaxis `@Injectable()` es mucho menos detallada en comparación con el uso de `@Inject()` en cada parámetro de constructor individual.

```
/* evitar */

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

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

[Volver al principio](#)

Servicios de datos

Hablar con el servidor a través de un servicio

Estilo 08-01

Hacer lógica refactor para realizar operaciones de datos e interactuar con los datos a un servicio.

Haga que los servicios de datos sean responsables de las llamadas XHR, el almacenamiento local, el almacenamiento en la 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, solo que sabe a quién pedirlos. La separación de los servicios de datos mueve la lógica sobre cómo llevarlos al servicio de datos y permite que el componente sea más simple y más centrado en la vista.

¿Por qué? Esto hace que sea más fácil probar (simulacro o real) las llamadas de datos al probar un componente que utiliza un servicio de datos.

¿Por qué? Los detalles de la gestión de datos, como encabezados, métodos HTTP, almacenamiento en caché, manejo de errores y lógica de reintento, 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.

[Volver al principio](#)

Los ganchos del ciclo de vida

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

[Volver al principio](#)

Implementar interfaces de gancho de ciclo de vida

Estilo 09-01

No aplicar las interfaces de gancho ciclo de vida.

¿Por qué? Las interfaces del ciclo de vida prescriben firmas de métodos mecanografiados. Use esas firmas para marcar errores de ortografía y sintaxis.

```
/* evitar */

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

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

[Volver al principio](#)

Appendix

Herramientas y consejos útiles para Angular.

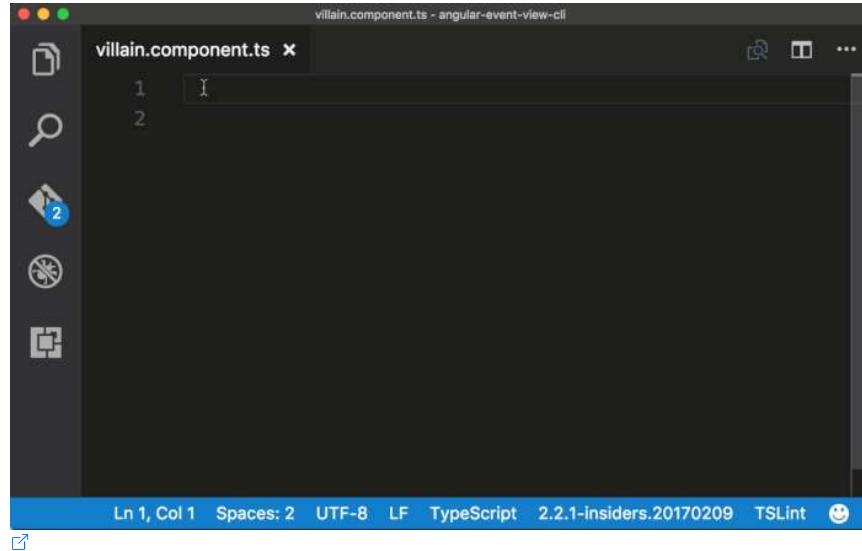
[Volver al principio](#)

Plantillas de archivos y recortes

Estilo A-02

Hacer plantillas de archivo de uso o fragmentos de ayuda seguimiento estilos y patrones consistentes. Aquí hay plantillas y / o fragmentos para algunos de los editores de desarrollo web e IDE.

Consideré usar [fragmentos](#) para [Visual Studio Code](#) que sigan estos estilos y pautas.



Consideré usar [fragmentos](#) para [Atom](#) que sigan estos estilos y pautas.

Consideré usar [fragmentos](#) de [text sublime](#) que sigan estos estilos y pautas.

Consideré usar [fragmentos](#) de [Vim](#) que sigan estos estilos y pautas.

[Volver al principio](#)

© 2010–2022 Google, Inc.

Licenciado bajo la Creative Commons Attribution License 4.0.

<https://angular.io/guide/styleguide>

<input type="text" value="Search..."/>	<input type="button" value=""/>
\$ 33.600	\$ 7.800
Cursos Capacitarte	
Capacitarte	



Seguridad e Higiene

Anuncio Capacitarte

Escritorio de