Ingeniería en Sistemas de Información

Cátedra: Ingeniería de Software

Trabajo Práctico Nº6

Curso: 4K4.

Grupo 9:

- Amante, Osmar Roberto   70224
- Miszczuk, Solange Daiana   58535
- Romero, Daniela Mayra   64457
- Sarmiento, Victoria María   70542
- Wannaz Gomez, Santiago   82594

Docentes:

- Ing. Boiero, Gerardo
- Ing. Covaro, Laura
- Ing. Crespo, Mickaela

2022

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Índice

Introducción

En este trabajo práctico se busca implementar la user story solicitada por la cátedra, para así comprender que esta corresponde a una porción transversal de funcionalidad que requiere la colaboración y el trabajo en equipo.

Se utilizaran distintas tecnologías para cumplir con dicho objetivo, cumpliendo con todos los criterios de aceptación de la user story, como sus diferentes pruebas.

Así, previo a la conclusión, se encuentra un link con el acceso al repositorio que deja plasmada esta user story.

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Unidad Nro. 3: Gestión del Software como producto

Consigna: Implementar una User Story determinada usando un lenguaje de programación elegido por el grupo respetando un documento de reglas de estilo.

Objetivo: Que el estudiante comprenda la implementación de una User Story como una porción transversal de funcionalidad que requiere la colaboración de un equipo multidisciplinario.

Propósito:

Familiarizarse con los conceptos de requerimientos ágiles y en particular con User Stories en conjunto con la aplicación de las actividades de SCM correspondientes.

Entradas: Conceptos teóricos sobre el tema desarrollados en clase. Definición completa de las User Stories correspondientes al Trabajo Práctico 2 "Requerimientos Ágiles – User Stories y Estimaciones"

Salida:

Implementación de la User Story correspondiente en un programa ejecutable

Documento de estilo de código

Instrucciones:

- Seleccionar una User Story a implementar de entre las siguientes opciones:
  - Realizar Pedido a Comercio adherido (grupos pares)
  - Realizar un Pedido de "lo que sea" (grupos impares)
- Seleccionar el conjunto de tecnologías para implementar la funcionalidad elegida.
- Buscar y seleccionar un documento de buenas prácticas y/o reglas de estilo de código para el lenguaje de programación a utilizar.
- Implementar la US siguiendo las reglas de estilo determinadas.

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Desarrollo

User Story: Realizar un pedido de "lo que sea"

Realizar Pedido de "lo que sea"   8

Como Solicitante quiero realizar un Pedido de "lo que sea" para recibir algo en mi domicilio que no está disponible en los comercios adheridos

Criterios de Aceptación

- Se debe indicar qué es lo que debe buscar el Cadete con un campo de texto
- Se puede adjuntar opcionalmente una foto en formato JPG con un tamaño máximo de 5 MB.
- Se debe indicar la dirección del comercio en forma textual (calle, número, ciudad y referencia opcional en formato de texto) o seleccionando un punto en un mapa interactivo de Google Maps.
- Se debe indicar la dirección de entrega (calle, número, ciudad y referencia opcional en formato de texto).
- Se debe poder seleccionar la ciudad de un listado de Ciudades disponibles.
- Se debe poder visualizar el total a pagar, en función de la distancia y/o del costo del producto, antes de elegir la forma de pago.
- Se debe seleccionar la forma de pago: Efectivo o Tarjeta VISA,
- Si paga en Efectivo debe indicar el monto con el que va a pagar.
- Si paga con Tarjeta VISA debe ingresar el número de la tarjeta, nombre y apellido del Titular, fecha de vencimiento (MM/AAAA) y CVC.
- Debe ingresar cuando quiere recibirlo: "Lo antes posible" o una fecha/hora de recepción.

- Probar realizar un Pedido de "lo que sea" en efectivo "lo antes posible" (pasa)
- Probar realizar un Pedido de "lo que sea" con tarjeta "lo antes posible" (pasa)
- Probar realizar un Pedido de "lo que sea" programando la fecha/hora de entrega (pasa)
- Probar realizar un Pedido de "lo que sea" con una tarjeta inválida (falla)
- Probar realizar un Pedido de "lo que sea" con una tarjeta MasterCard (falla)
- Probar realizar un Pedido de "lo que sea" en efectivo sin indicar el monto a pagar (falla)
- Probar realizar un Pedido de "lo que sea" programando una fecha/hora de entrega no válida (falla)
- Probar realizar un Pedido de "lo que sea" sin especificar qué buscar (falla)
- Probar realizar un Pedido de "lo que sea" adjuntando una foto (pasa)
- Probar realizar un Pedido de "lo que sea" sin indicar la dirección del comercio (falla)
- Probar realizar un Pedido de "lo que sea" seleccionando la dirección del comercio en el mapa interactivo (pasa)

El conjunto de tecnologías utilizados para implementar la user story fueron:

- HTML
- CSS
- Javascript
- Angular

Link del trabajó: https://gomezwannaz-santiago.github.io/

* Se adjuntan a continuación las guías de buenas prácticas.

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Angular coding style guide

Looking for an opinionated guide to Angular syntax, conventions, and application structure? Step right in. This style guide presents preferred conventions and, as importantly, explains why.

Style vocabulary

Each guideline describes either a good or bad practice, and all have a consistent presentation.

The wording of each guideline indicates how strong the recommendation is.

Do is one that should always be followed. *Always* might be a bit too strong of a word. Guidelines that literally should always be followed are extremely rare. On the other hand, you need a really unusual case for breaking a *Do* guideline.

Consider guidelines should generally be followed. If you fully understand the meaning behind the guideline and have a good reason to deviate, then do so. Aim to be consistent.

Avoid indicates something you should almost never do. Code examples to *avoid* have an unmistakable red header.

Why?

Gives reasons for following the previous recommendations.

File structure conventions

Some code examples display a file that has one or more similarly named companion files. For example, `hero.component.ts` and `hero.component.html`.

The guideline uses the shortcut `hero.component.ts|html|css|spec` to represent those various files. Using this shortcut makes this guide's file structures easier to read and more terse.

Single responsibility

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Apply the *single responsibility principle (SRP)* to all components, services, and other symbols. This helps make the application cleaner, easier to read and maintain, and more testable.

Rule of One

Style 01-01

Do define one thing, such as a service or component, per file.

Consider limiting files to 400 lines of code.

Why?

One component per file makes it far easier to read, maintain, and avoid collisions with teams in source control.

Why?

One component per file avoids hidden bugs that often arise when combining components in a file where they may share variables, create unwanted closures, or unwanted coupling with dependencies.

Why?

A single component can be the default export for its file which facilitates lazy loading with the router.

The key is to make the code more reusable, easier to read, and less mistake prone.

The following *negative* example defines the `AppComponent`, bootstraps the app, defines the `Hero` model object, and loads heroes from the server all in the same file. *Don't do this*.

app/heroes/hero.component.ts

```
content_copy/* avoid */

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

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

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```typescript
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));
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   Fecha:26/04/22

```typescript
    }

  }


@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;

}
```

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

It is a better practice to redistribute the component and its supporting classes into their own, dedicated files.

main.ts

app/app.module.ts

app/app.component.ts

app/heroes/heroes.component.ts

app/heroes/shared/hero.service.ts

app/heroes/shared/hero.model.ts

app/heroes/shared/mock-heroes.ts

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


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


platformBrowserDynamic().bootstrapModule(AppModule);
```

As the application grows, this rule becomes even more important.

[Back to top](#)

Small functions

Style 01-02

Do define small functions

Consider limiting to no more than 75 lines.

Why?

Small functions are easier to test, especially when they do one thing and serve one purpose.

Why?

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Small functions promote reuse.

Why?

Small functions are easier to read.

Why?

Small functions are easier to maintain.

Why?

Small functions help avoid hidden bugs that come with large functions that share variables with external scope, create unwanted closures, or unwanted coupling with dependencies.

[Back to top](#)

Naming

Naming conventions are hugely important to maintainability and readability. This guide recommends naming conventions for the file name and the symbol name.

General Naming Guidelines

Style 02-01

Do use consistent names for all symbols.

Do follow a pattern that describes the symbol's feature then its type. The recommended pattern is `feature.type.ts`.

Why?

Naming conventions help provide a consistent way to find content at a glance. Consistency within the project is vital. Consistency with a team is important. Consistency across a company provides tremendous efficiency.

Why?

The naming conventions should help find desired code faster and make it easier to understand.

Why?

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Names of folders and files should clearly convey their intent. For example, `app/heroes/hero-list.component.ts` may contain a component that manages a list of heroes.

[Back to top](#)

Separate file names with dots and dashes

Style 02-02

Do use dashes to separate words in the descriptive name.

Do use dots to separate the descriptive name from the type.

Do use consistent type names for all components following a pattern that describes the component's feature then its type. A recommended pattern is `feature.type.ts`.

Do use conventional type names including `.service`, `.component`, `.pipe`, `.module`, and `.directive`. Invent additional type names if you must but take care not to create too many.

Why?

Type names provide a consistent way to quickly identify what is in the file.

Why?

Type names make it easy to find a specific file type using an editor or IDE's fuzzy search techniques.

Why?

Unabbreviated type names such as `.service` are descriptive and unambiguous. Abbreviations such as `.srv`, `.svc`, and `.serv` can be confusing.

Why?

Type names provide pattern matching for any automated tasks.

[Back to top](#)

Symbols and file names

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Style 02-03

Do use consistent names for all assets named after what they represent.

Do use upper camel case for class names.

Do match the name of the symbol to the name of the file.

Do append the symbol name with the conventional suffix (such as Component, Directive, Module, Pipe, or Service) for a thing of that type.

Do give the filename the conventional suffix (such as .component.ts, .directive.ts, .module.ts, .pipe.ts, or .service.ts) for a file of that type.

Why?

Consistent conventions make it easy to quickly identify and reference assets of different types.

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

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
@Component({ ... })                              hero-detail.compone
                                                 nt.ts
export class HeroDetailComponent {
}
```

```
@Directive({ ... })                              validation.directive.t
                                                 s
export class ValidationDirective {
}
```

```
@NgModule({ ... })                               app.module.ts

export class AppModule
```

```
@Pipe({ name: 'initCaps' })                      init-caps.pipe.ts

export     class       InitCapsPipe
implements PipeTransform { }
```

```
@Injectable()                                    user-profile.service.t
                                                 s
export class UserProfileService {
}
```

[Back to top](#)

Service names

Style 02-04

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Do use consistent names for all services named after their feature.

Do suffix a service class name with `Service`. For example, something that gets data or heroes should be called a `DataService` or a `HeroService`.

A few terms are unambiguously services. They typically indicate agency by ending in "-er". You may prefer to name a service that logs messages `Logger` rather than `LoggerService`. Decide if this exception is agreeable in your project. As always, strive for consistency.

Why?

Provides a consistent way to quickly identify and reference services.

Why?

Clear service names such as `Logger` do not require a suffix.

Why?

Service names such as `Credit` are nouns and require a suffix and should be named with a suffix when it is not obvious if it is a service or something else.

SYMBOL NAME	FILE NAME
`@Injectable()` `export class HeroDataService {` `}`	hero-data.service.ts
`@Injectable()` `export class CreditService { }`	credit.service.ts

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
@Injectable()                                    logger.service.ts

export class Logger { }
```

Bootstrapping

Style 02-05

Do put bootstrapping and platform logic for the application in a file named `main.ts`.

Do include error handling in the bootstrapping logic.

Avoid putting application logic in `main.ts`. Instead, consider placing it in a component or service.

Why?

Follows a consistent convention for the startup logic of an app.

Why?

Follows a familiar convention from other technology platforms.

main.ts

```typescript
content_copyimport { 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));
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Back to top

Component selectors

Style 05-02

Do use *dashed-case* or *kebab-case* for naming the element selectors of components.

Why?

Keeps the element names consistent with the specification for Custom Elements.

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

```
content_copy/* avoid */



@Component({

  selector: 'tohHeroButton',

  templateUrl: './hero-button.component.html'

})

export class HeroButtonComponent {}
```

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

app/app.component.html

```
content_copy@Component({

  selector: 'toh-hero-button',

  templateUrl: './hero-button.component.html'

})

export class HeroButtonComponent {}
```

Back to top

Component custom prefix

Asignatura: Ingeniería de Software **Fecha:26/04/22**

Style 02-07

Do use a hyphenated, lowercase element selector value; for example, `admin-users`.

Do use a custom prefix for a component selector. For example, the prefix `toh` represents Tour of Heroes and the prefix `admin` represents an admin feature area.

Do use a prefix that identifies the feature area or the application itself.

Why?

Prevents element name collisions with components in other applications and with native HTML elements.

Why?

Makes it easier to promote and share the component in other applications.

Why?

Components are easy to identify in the DOM.

app/heroes/hero.component.ts

```
content_copy/* avoid */



// HeroComponent is in the Tour of Heroes feature

@Component({

  selector: 'hero'

})

export class HeroComponent {}
```

app/users/users.component.ts

```
content_copy/* avoid */



// UsersComponent is in an Admin feature
```

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

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

app/heroes/hero.component.ts

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

app/users/users.component.ts

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

Back to top

Directive selectors

Style 02-06

Do Use lower camel case for naming the selectors of directives.

Why?

Keeps the names of the properties defined in the directives that are bound to the view consistent with the attribute names.

Why?

The Angular HTML parser is case sensitive and recognizes lower camel case.

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Directive custom prefix

Style 02-08

Do use a custom prefix for the selector of directives (for example, the prefix `toh` from Tour of Heroes).

Do spell non-element selectors in lower camel case unless the selector is meant to match a native HTML attribute.

Don't prefix a directive name with `ng` because that prefix is reserved for Angular and using it could cause bugs that are difficult to diagnose.

Why?

Prevents name collisions.

Why?

Directives are easily identified.

app/shared/validate.directive.ts

```
content_copy/* avoid */


@Directive({

  selector: '[validate]'

})

export class ValidateDirective {}
```

app/shared/validate.directive.ts

```
content_copy@Directive({

  selector: '[tohValidate]'

})
```

19 Amante-Miszczuk-Romero-Sarmiento-Wannaz

```
export class ValidateDirective {}
```

[Back to top](#)

Pipe names

Style 02-09

Do use consistent names for all pipes, named after their feature. The pipe class name should use [UpperCamelCase](#) (the general convention for class names), and the corresponding `name` string should use *lowerCamelCase*. The `name` string cannot use hyphens ("dash-case" or "kebab-case").

Why?

Provides a consistent way to quickly identify and reference pipes.

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

[Back to top](#)

Unit test file names

Style 02-10

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Do name test specification files the same as the component they test.

Do name test specification files with a suffix of `.spec`.

Why?

Provides a consistent way to quickly identify tests.

Why?

Provides pattern matching for karma or other test runners.

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

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

| Pipes | ellipsis.pipe.spec.ts |
| | init-caps.pipe.spec.ts |

End-to-End (E2E) test file names

Style 02-11

Do name end-to-end test specification files after the feature they test with a suffix of `.e2e-spec`.

Why?

Provides a consistent way to quickly identify end-to-end tests.

Why?

Provides pattern matching for test runners and build automation.

TEST TYPE	FILE NAMES
End-to-End Tests	app.e2e-spec.ts
	heroes.e2e-spec.ts

Angular NgModule names

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Style 02-12

Do append the symbol name with the suffix `Module`.

Do give the file name the `.module.ts` extension.

Do name the module after the feature and folder it resides in.

Why?

Provides a consistent way to quickly identify and reference modules.

Why?

Upper camel case is conventional for identifying objects that can be instantiated using a constructor.

Why?

Easily identifies the module as the root of the same named feature.

Do suffix a `RoutingModule` class name with `RoutingModule`.

Do end the filename of a `RoutingModule` with `-routing.module.ts`.

Why?

A `RoutingModule` is a module dedicated exclusively to configuring the Angular router. A consistent class and file name convention make these modules easy to spot and verify.

SYMBOL NAME	FILE NAME
@NgModule({ ... }) export class AppModule { }	app.module.ts

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
@NgModule({ ... })                              heroes.module.ts

export class HeroesModule { }
```

```
@NgModule({ ... })                              villains.module.ts

export class VillainsModule { }
```

```
@NgModule({ ... })                              app-routing.module.t
                                                s
export class AppRoutingModule { }
```

```
@NgModule({ ... })                              heroes-routing.modul
                                                e.ts
export  class  HeroesRoutingModule
{ }
```

[Back to top](#)

Application structure and NgModules

Have a near-term view of implementation and a long-term vision. Start small but keep in mind where the application is heading.

All of the application's code goes in a folder named `src`. All feature areas are in their own folder, with their own NgModule.

All content is one asset per file. Each component, service, and pipe is in its own file. All third party vendor scripts are stored in another folder and not in the `src` folder. You didn't write them and you don't want them cluttering `src`. Use the naming conventions for files in this guide.

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

LIFT

Style 04-01

Do structure the application such that you can Locate code quickly, Identify the code at a glance, keep the Flattest structure you can, and Try to be DRY.

Do define the structure to follow these four basic guidelines, listed in order of importance.

Why?

LIFT provides a consistent structure that scales well, is modular, and makes it easier to increase developer efficiency by finding code quickly. To confirm your intuition about a particular structure, ask: *Can I quickly open and start work in all of the related files for this feature*?

Locate

Style 04-02

Do make locating code intuitive and fast.

Why?

To work efficiently you must be able to find files quickly, especially when you do not know (or do not remember) the file *names*. Keeping related files near each other in an intuitive location saves time. A descriptive folder structure makes a world of difference to you and the people who come after you.

Identify

Style 04-03

Do name the file such that you instantly know what it contains and represents.

Do be descriptive with file names and keep the contents of the file to exactly one component.

Avoid files with multiple components, multiple services, or a mixture.

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software **Fecha:26/04/22**

Why?

Spend less time hunting and pecking for code, and become more efficient. Longer file names are far better than *short-but-obscure* abbreviated names.

It may be advantageous to deviate from the *one-thing-per-file* rule when you have a set of small, closely-related features that are better discovered and understood in a single file than as multiple files. Be wary of this loophole.

Back to top

Flat

Style 04-04

Do keep a flat folder structure as long as possible.

Consider creating sub-folders when a folder reaches seven or more files.

Consider configuring the IDE to hide distracting, irrelevant files such as generated `.js` and `.js.map` files.

Why?

No one wants to search for a file through seven levels of folders. A flat structure is easy to scan.

On the other hand, psychologists believe that humans start to struggle when the number of adjacent interesting things exceeds nine. So when a folder has ten or more files, it may be time to create subfolders.

Base your decision on your comfort level. Use a flatter structure until there is an obvious value to creating a new folder.

Back to top

T-DRY (Try to be *DRY*)

Style 04-05

Do be DRY (Don't Repeat Yourself).

Avoid being so DRY that you sacrifice readability.

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Why?

Being DRY is important, but not crucial if it sacrifices the other elements of LIFT. That's why it's called *T-DRY*. For example, it's redundant to name a template `hero-view.component.html` because with the `.html` extension, it is obviously a view. But if something is not obvious or departs from a convention, then spell it out.

[Back to top](#)

Overall structural guidelines

Style 04-06

Do start small but keep in mind where the application is heading down the road.

Do have a near term view of implementation and a long term vision.

Do put all of the application's code in a folder named `src`.

Consider creating a folder for a component when it has multiple accompanying files (`.ts`, `.html`, `.css`, and `.spec`).

Why?

Helps keep the application structure small and easy to maintain in the early stages, while being easy to evolve as the application grows.

Why?

Components often have four files (for example, `*.html`, `*.css`, `*.ts`, and `*.spec.ts`) and can clutter a folder quickly.

Here is a compliant folder and file structure:

<project root>

src

app

core

exception.service.ts|spec.ts

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

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

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

...

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/...

...

While components in dedicated folders are widely preferred, another option for small applications is to keep components flat (not in a dedicated folder). This adds up to four files to the existing folder, but also reduces the folder nesting. Whatever you choose, be consistent.

[Back to top](#)

Folders-by-feature structure

Style 04-07

Do create folders named for the feature area they represent.

Why?

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software **Fecha:26/04/22**

A developer can locate the code and identify what each file represents at a glance. The structure is as flat as it can be and there are no repetitive or redundant names.

Why?

The LIFT guidelines are all covered.

Why?

Helps reduce the application from becoming cluttered through organizing the content and keeping them aligned with the LIFT guidelines.

Why?

When there are a lot of files, for example 10+, locating them is easier with a consistent folder structure and more difficult in a flat structure.

Do create an NgModule for each feature area.

Why?

NgModules make it easy to lazy load routable features.

Why?

NgModules make it easier to isolate, test, and reuse features.

For more information, refer to [this folder and file structure example](#).

[Back to top](#)

App *root module*

Style 04-08

Do create an NgModule in the application's root folder, for example, in `/src/app`.

Why?

Every application requires at least one root NgModule.

Consider naming the root module `app.module.ts`.

Why?

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Makes it easier to locate and identify the root module.

app/app.module.ts

```typescript
content_copyimport { 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 {}
```

Back to top

Feature modules

Style 04-09

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Do create an NgModule for all distinct features in an application; for example, a `Heroes` feature.

Do place the feature module in the same named folder as the feature area; for example, in `app/heroes`.

Do name the feature module file reflecting the name of the feature area and folder; for example, `app/heroes/heroes.module.ts`.

Do name the feature module symbol reflecting the name of the feature area, folder, and file; for example, `app/heroes/heroes.module.ts` defines `HeroesModule`.

Why?

A feature module can expose or hide its implementation from other modules.

Why?

A feature module identifies distinct sets of related components that comprise the feature area.

Why?

A feature module can easily be routed to both eagerly and lazily.

Why?

A feature module defines clear boundaries between specific functionality and other application features.

Why?

A feature module helps clarify and make it easier to assign development responsibilities to different teams.

Why?

A feature module can easily be isolated for testing.

[Back to top](#)

Shared feature module

Style 04-10

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Do create a feature module named `SharedModule` in a `shared` folder; for example, `app/shared/shared.module.ts` defines `SharedModule`.

Do declare components, directives, and pipes in a shared module when those items will be re-used and referenced by the components declared in other feature modules.

Consider using the name SharedModule when the contents of a shared module are referenced across the entire application.

Consider *not* providing services in shared modules. Services are usually singletons that are provided once for the entire application or in a particular feature module. There are exceptions, however. For example, in the sample code that follows, notice that the `SharedModule` provides `FilterTextService`. This is acceptable here because the service is stateless;that is, the consumers of the service aren't impacted by new instances.

Do import all modules required by the assets in the `SharedModule`; for example, CommonModule and FormsModule.

Why?

`SharedModule` will contain components, directives and pipes that may need features from another common module; for example, ngFor in CommonModule.

Do declare all components, directives, and pipes in the `SharedModule`.

Do export all symbols from the `SharedModule` that other feature modules need to use.

Why?

`SharedModule` exists to make commonly used components, directives and pipes available for use in the templates of components in many other modules.

Avoid specifying app-wide singleton providers in a `SharedModule`. Intentional singletons are OK. Take care.

Why?

A lazy loaded feature module that imports that shared module will make its own copy of the service and likely have undesirable results.

Why?

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

You don't want each module to have its own separate instance of singleton services. Yet there is a real danger of that happening if the `SharedModule` provides a service.

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

...

app/shared/shared.module.ts

app/shared/init-caps.pipe.ts

app/shared/filter-text/filter-text.component.ts

app/shared/filter-text/filter-text.service.ts

app/heroes/heroes.component.ts

app/heroes/heroes.component.html

```
content_copyimport { NgModule } from '@angular/core';

import { CommonModule } from '@angular/common';
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   Fecha:26/04/22

```typescript
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 { }
```

Back to top

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Lazy Loaded folders

Style 04-11

A distinct application feature or workflow may be *lazy loaded* or *loaded on demand* rather than when the application starts.

Do put the contents of lazy loaded features in a *lazy loaded folder*. A typical *lazy loaded folder* contains a *routing component*, its child components, and their related assets and modules.

Why?

The folder makes it easy to identify and isolate the feature content.

[Back to top](#)

Never directly import lazy loaded folders

Style 04-12

Avoid allowing modules in sibling and parent folders to directly import a module in a *lazy loaded feature*.

Why?

Directly importing and using a module will load it immediately when the intention is to load it on demand.

[Back to top](#)

Do not add filtering and sorting logic to pipes

Style 04-13

Avoid adding filtering or sorting logic into custom pipes.

Do pre-compute the filtering and sorting logic in components or services before binding the model in templates.

Why?

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Filtering and especially sorting are expensive operations. As Angular can call pipe methods many times per second, sorting and filtering operations can degrade the user experience severely for even moderately-sized lists.

[Back to top](#)

Components

Components as elements

Style 05-03

Consider giving components an *element* selector, as opposed to *attribute* or *class* selectors.

Why?

Components have templates containing HTML and optional Angular template syntax. They display content. Developers place components on the page as they would native HTML elements and web components.

Why?

It is easier to recognize that a symbol is a component by looking at the template's html.

> There are a few cases where you give a component an attribute, such as when you want to augment a built-in element. For example, [Material Design](#) uses this technique with `<button mat-button>`. However, you wouldn't use this technique on a custom element.

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

```
content_copy/* avoid */



@Component({

  selector: '[tohHeroButton]',

  templateUrl: './hero-button.component.html'

})

export class HeroButtonComponent {}
```

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

app/app.component.html

```
content_copy<!-- avoid -->



<div tohHeroButton></div>
```

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

app/app.component.html

```
content_copy@Component({

  selector: 'toh-hero-button',

  templateUrl: './hero-button.component.html'

})

export class HeroButtonComponent {}
```

Back to top

Extract templates and styles to their own files

Style 05-04

Do extract templates and styles into a separate file, when more than 3 lines.

Do name the template file `[component-name].component.html`, where [component-name] is the component name.

Do name the style file `[component-name].component.css`, where [component-name] is the component name.

Do specify *component-relative* URLs, prefixed with `./`.

Why?

Large, inline templates and styles obscure the component's purpose and implementation, reducing readability and maintainability.

Why?

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

In most editors, syntax hints and code snippets aren't available when developing inline templates and styles. The Angular TypeScript Language Service (forthcoming) promises to overcome this deficiency for HTML templates in those editors that support it; it won't help with CSS styles.

Why?

A *component relative* URL requires no change when you move the component files, as long as the files stay together.

Why?

The `./` prefix is standard syntax for relative URLs; don't depend on Angular's current ability to do without that prefix.

app/heroes/heroes.component.ts

```
content_copy/* 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">
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   Fecha:26/04/22

```
    <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;
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
    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();

  }

}
```

app/heroes/heroes.component.ts

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

app/heroes/heroes.component.html

app/heroes/heroes.component.css

```
content_copy@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();

  }

}
```

Back to top

Decorate `input` and `output` properties

Style 05-12

Do use the @Input() and @Output() class decorators instead of the inputs and outputs properties of the @Directive and @Component metadata:

Consider placing @Input() or @Output() on the same line as the property it decorates.

Why?

It is easier and more readable to identify which properties in a class are inputs or outputs.

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Why?

If you ever need to rename the property or event name associated with @Input() or @Output(), you can modify it in a single place.

Why?

The metadata declaration attached to the directive is shorter and thus more readable.

Why?

Placing the decorator on the same line *usually* makes for shorter code and still easily identifies the property as an input or output. Put it on the line above when doing so is clearly more readable.

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

```
content_copy/* avoid */


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

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
    label: string;

}
```

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

```
content_copy@Component({

  selector: 'toh-hero-button',

  template: `<button type="button">{{label}}</button>`

})

export class HeroButtonComponent {

  @Output() heroChange = new EventEmitter<any>();

  @Input() label = '';

}
```

Back to top

Avoid aliasing `inputs` and `outputs`

Style 05-13

Avoid `input` and `output` aliases except when it serves an important purpose.

Why?

Two names for the same property (one private, one public) is inherently confusing.

Why?

You should use an alias when the directive name is also an `input` property, and the directive name doesn't describe the property.

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

```
content_copy/* avoid pointless aliasing */
```

```typescript
@Component({

  selector: 'toh-hero-button',

  template: `<button type="button">{{label}}</button>`

})

export class HeroButtonComponent {

  // Pointless aliases

    @Output('heroChangeEvent') heroChange = new
EventEmitter<any>();

  @Input('labelAttribute') label: string;

}
```

app/app.component.html

```html
content_copy<!-- avoid -->


<toh-hero-button                          labelAttribute="OK"
(changeEvent)="doSomething()">

</toh-hero-button>
```

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

app/heroes/shared/hero-button/hero-highlight.directive.ts

app/app.component.html

```typescript
content_copy@Component({

  selector: 'toh-hero-button',

  template: `<button type="button" >{{label}}</button>`

})

export class HeroButtonComponent {
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
// No aliases

@Output() heroChange = new EventEmitter<any>();

@Input() label = '';

}
```

[Back to top](#)

Member sequence

Style 05-14

Do place properties up top followed by methods.

Do place private members after public members, alphabetized.

Why?

Placing members in a consistent sequence makes it easy to read and helps instantly identify which members of the component serve which purpose.

app/shared/toast/toast.component.ts

```
content_copy/* avoid */


export class ToastComponent implements OnInit {


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

  message: string;

  title: string;
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```typescript
  private toastElement: any;



  ngOnInit() {

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

   }



   // private methods

   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;
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
    this.toastElement.style.zIndex = 9999;



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

  }

}
```

app/shared/toast/toast.component.ts

```
content_copyexport class ToastComponent implements OnInit {

  // public properties

  message = '';

  title = '';



  // private fields

  private defaults = {

   title: '',

   message: 'May the Force be with you'

  };

  private toastElement: any;



  // public methods

    activate(message = this.defaults.message, title =
  this.defaults.title) {

   this.title = title;

   this.message = message;

   this.show();
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   Fecha:26/04/22

```typescript
  }


  ngOnInit() {

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

  }



  // private methods

  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);

  }

 }
```

Back to top

Delegate complex component logic to services

Style 05-15

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Do limit logic in a component to only that required for the view. All other logic should be delegated to services.

Do move reusable logic to services and keep components simple and focused on their intended purpose.

Why?

Logic may be reused by multiple components when placed within a service and exposed as a function.

Why?

Logic in a service can more easily be isolated in a unit test, while the calling logic in the component can be easily mocked.

Why?

Removes dependencies and hides implementation details from the component.

Why?

Keeps the component slim, trim, and focused.

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

```
content_copy/* 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';
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```typescript
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
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   Fecha:26/04/22

```
    }

  }

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

content_copyimport { 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();
  }
}
```

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

[Back to top](#)

Don't prefix `output` properties

Style 05-16

Do name events without the prefix `on`.

Do name event handler methods with the prefix `on` followed by the event name.

Why?

This is consistent with built-in events such as button clicks.

Why?

Angular allows for an [alternative syntax](#) `on-*`. If the event itself was prefixed with `on` this would result in an `on-onEvent` binding expression.

app/heroes/hero.component.ts

```
content_copy/* avoid */



@Component({

  selector: 'toh-hero',

  template: `...`

})

export class HeroComponent {

  @Output() onSavedTheDay = new EventEmitter<boolean>();

}
```

app/app.component.html

```
content_copy<!-- avoid -->
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

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

app/heroes/hero.component.ts

app/app.component.html

```typescript
content_copyexport class HeroComponent {

  @Output() savedTheDay = new EventEmitter<boolean>();

}
```

[Back to top](#)

Put presentation logic in the component class

Style 05-17

Do put presentation logic in the component class, and not in the template.

Why?

Logic will be contained in one place (the component class) instead of being spread in two places.

Why?

Keeping the component's presentation logic in the class instead of the template improves testability, maintainability, and reusability.

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

```typescript
content_copy/* avoid */


@Component({

  selector: 'toh-hero-list',

  template: `

  <section>
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
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;

}
```

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

```
content_copy@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>
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   Fecha:26/04/22

```
  `

})

export class HeroListComponent {

  heroes: Hero[];

  totalPowers = 0;


  get avgPower() {

   return this.totalPowers / this.heroes.length;

  }

}
```

[Back to top](#)

Initialize inputs

Style 05-18

TypeScript's `--strictPropertyInitialization` compiler option ensures that a class initializes its properties during construction. When enabled, this option causes the TypeScript compiler to report an error if the class does not set a value to any property that is not explicitly marked as optional.

By design, Angular treats all @[Input](#) properties as optional. When possible, you should satisfy `--strictPropertyInitialization` by providing a default value.

app/heroes/hero/hero.component.ts

```
content_copy@Component({

  selector: 'toh-hero',

  template: `...`

})
```

```typescript
export class HeroComponent {

  @Input() id = 'default_id';

}
```

If the property is hard to construct a default value for, use ? to explicitly mark the property as optional.

app/heroes/hero/hero.component.ts

```typescript
content_copy@Component({

  selector: 'toh-hero',

  template: `...`

})

export class HeroComponent {

  @Input() id?: string;


  process() {

   if (this.id) {

   // ...

    }

  }

}
```

You may want to have a required @Input field, meaning all your component users are required to pass that attribute. In such cases, use a default value. Just suppressing the TypeScript error with ! is insufficient and should be avoided because it will prevent the type checker ensure the input value is provided.

app/heroes/hero/hero.component.ts

```
content_copy@Component({

  selector: 'toh-hero',

  template: `...`

})

export class HeroComponent {

    // The exclamation mark suppresses errors that a property is

  // not initialized.

    // Ignoring this enforcement can prevent the type checker

  // from finding potential issues.

  @Input() id!: string;

}
```

Directives

Use directives to enhance an element

Style 06-01

Do use attribute directives when you have presentation logic without a template.

Why?

Attribute directives don't have an associated template.

Why?

An element may have more than one attribute directive applied.

app/shared/highlight.directive.ts

```
content_copy@Directive({
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
  selector: '[tohHighlight]'

})

export class HighlightDirective {

  @HostListener('mouseover') onMouseEnter() {

   // do highlight work

  }

}
```

app/app.component.html

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

[Back to top]()

[HostListener]()/[HostBinding]() decorators versus `host` metadata

Style 06-03

Consider preferring the @[HostListener]() and @[HostBinding]() to the `host` property of the @[Directive]() and @[Component]() decorators.

Do be consistent in your choice.

Why?

The property associated with @[HostBinding]() or the method associated with @[HostListener]() can be modified only in a single place —in the directive's class. If you use the `host` metadata property, you must modify both the property/method declaration in the directive's class and the metadata in the decorator associated with the directive.

app/shared/validator.directive.ts

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


@Directive({
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```typescript
  selector: '[tohValidator]'

})

export class ValidatorDirective {

  @HostBinding('attr.role') role = 'button';

  @HostListener('mouseenter') onMouseEnter() {

  // do work

  }

}
```

Compare with the less preferred `host` metadata alternative.

Why?

The `host` metadata is only one term to remember and doesn't require extra ES imports.

app/shared/validator2.directive.ts

```typescript
content_copyimport { Directive } from '@angular/core';


@Directive({

  selector: '[tohValidator2]',

  host: {

    '[attr.role]': 'role',

   '(mouseenter)': 'onMouseEnter()'

  }

})

export class Validator2Directive {

  role = 'button';
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```
onMouseEnter() {

  // do work

}

}
```

Back to top

Services

Services are singletons

Style 07-01

Do use services as singletons within the same injector. Use them for sharing data and functionality.

Why?

Services are ideal for sharing methods across a feature area or an app.

Why?

Services are ideal for sharing stateful in-memory data.

app/heroes/shared/hero.service.ts

```
content_copyexport class HeroService {

  constructor(private http: HttpClient) { }


  getHeroes() {

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

  }

}
```

Back to top

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Single responsibility

Style 07-02

Do create services with a single responsibility that is encapsulated by its context.

Do create a new service once the service begins to exceed that singular purpose.

Why?

When a service has multiple responsibilities, it becomes difficult to test.

Why?

When a service has multiple responsibilities, every component or service that injects it now carries the weight of them all.

[Back to top](#)

Providing a service

Style 07-03

Do provide a service with the application root injector in the @Injectable decorator of the service.

Why?

The Angular injector is hierarchical.

Why?

When you provide the service to a root injector, that instance of the service is shared and available in every class that needs the service. This is ideal when a service is sharing methods or state.

Why?

When you register a service in the @Injectable decorator of the service, optimization tools such as those used by the Angular CLI's production builds can perform tree shaking and remove services that aren't used by your app.

Why?

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

This is not ideal when two different components need different instances of a service. In this scenario it would be better to provide the service at the component level that needs the new and separate instance.

src/app/treeshaking/service.ts

```
content_copy@Injectable({

  providedIn: 'root',

})

export class Service {

}
```

Back to top

Use the @Injectable() class decorator

Style 07-04

Do use the @Injectable() class decorator instead of the @Inject parameter decorator when using types as tokens for the dependencies of a service.

Why?

The Angular Dependency Injection (DI) mechanism resolves a service's own dependencies based on the declared types of that service's constructor parameters.

Why?

When a service accepts only dependencies associated with type tokens, the @Injectable() syntax is much less verbose compared to using @Inject() on each individual constructor parameter.

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

```
content_copy/* avoid */


export class HeroArena {
```

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Dpto. Ingeniería en Sistemas de Información

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```typescript
  constructor(

   @Inject(HeroService) private heroService: HeroService,

   @Inject(HttpClient) private http: HttpClient) {}

 }
```

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

```typescript
content_copy@Injectable()

export class HeroArena {

  constructor(

   private heroService: HeroService,

   private http: HttpClient) {}

 }
```

[Back to top](#)

Data Services

Talk to the server through a service

Style 08-01

Do refactor logic for making data operations and interacting with data to a service.

Do make data services responsible for XHR calls, local storage, stashing in memory, or any other data operations.

Why?

The component's responsibility is for the presentation and gathering of information for the view. It should not care how it gets the data, just that it knows who to ask for it. Separating the data services moves the logic on how to get it to the data service, and lets the component be simpler and more focused on the view.

Why?

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

This makes it easier to test (mock or real) the data calls when testing a component that uses a data service.

Why?

The details of data management, such as headers, HTTP methods, caching, error handling, and retry logic, are irrelevant to components and other data consumers.

A data service encapsulates these details. It's easier to evolve these details inside the service without affecting its consumers. And it's easier to test the consumers with mock service implementations.

Back to top

Lifecycle hooks

Use Lifecycle hooks to tap into important events exposed by Angular.

Back to top

Implement lifecycle hook interfaces

Style 09-01

Do implement the lifecycle hook interfaces.

Why?

Lifecycle interfaces prescribe typed method signatures. Use those signatures to flag spelling and syntax mistakes.

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

```
content_copy/* avoid */


@Component({

  selector: 'toh-hero-button',

  template: `<button type="button">OK</button>`

})
```

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

```typescript
export class HeroButtonComponent {

  onInit() { // misspelled

   console.log('The component is initialized');

  }

}
```

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

```typescript
content_copy@Component({

  selector: 'toh-hero-button',

  template: `<button type="button">OK</button>`

})

export class HeroButtonComponent implements OnInit {

  ngOnInit() {

   console.log('The component is initialized');

  }

}
```

Back to top

Appendix

Useful tools and tips for Angular.

Back to top

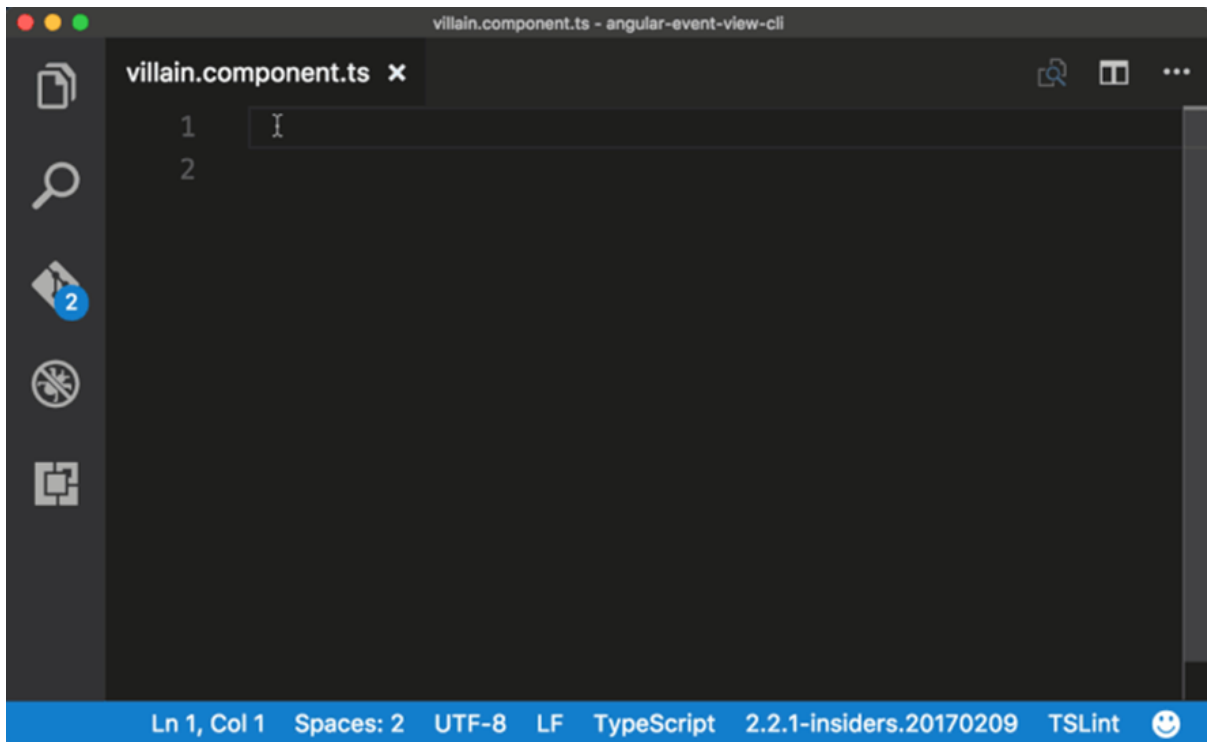## File templates and snippets

Style A-02

Do use file templates or snippets to help follow consistent styles and patterns. Here are templates and/or snippets for some of the web development editors and IDEs.

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Consider using snippets for Visual Studio Code that follow these styles and guidelines.

Consider using snippets for Atom that follow these styles and guidelines.

Consider using snippets for Sublime Text that follow these styles and guidelines.

Consider using snippets for Vim that follow these styles and guidelines.

Back to top

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Conclusión

En resumen, los integrantes del grupo tuvimos que trabajar en conjunto para lograr la solución a este trabajo práctico, el cual representó bastantes dificultades ya que no teníamos los conocimientos suficientes para la correcta implementación de la user story, lo que implicó un aprendizaje que va más allá de los contenidos de la materia pero que nos servirán para el desarrollo de nuestra carrera profesional.

Así mismo, consideramos que el uso del repositorio fue fundamental para lograr un trabajo ordenado, en conjunto y sin errores.

En definitiva, si bien resultó una experiencia compleja, el resultado de ésta fue muy fructífero.

Asignatura: Ingeniería de Software   **Fecha:26/04/22**

Bibliografía

- Material de la cátedra de Ingeniería de Software.
- https://angular.io/guide/styleguide