**Universidad Tecnológica Nacional**

**Facultad Regional Córdoba**

**Ingeniería en Sistemas de Información**

**Cátedra: Ingeniería de Software**

**Trabajo Práctico N.º6:**

**“Implementación de users stories”**

**Curso: 4k1**

**GRUPO 2:**

Legajo Apellido y Nombre

83116 Bianchini, Alex Maximiliano

82257 Cagol, Fabrizio

83316 Cugura, Ignacio Cruz

82539 Passero, Sabrina Denise

84839 Mastrapasqua, Gastón

82517 Sanchez Cassia, María Salomé

**Docentes:**

* Ing. Judith Meles
* Ing. Gerardo Boiero

**Fecha de Presentación:** 13/09/2022

Índice

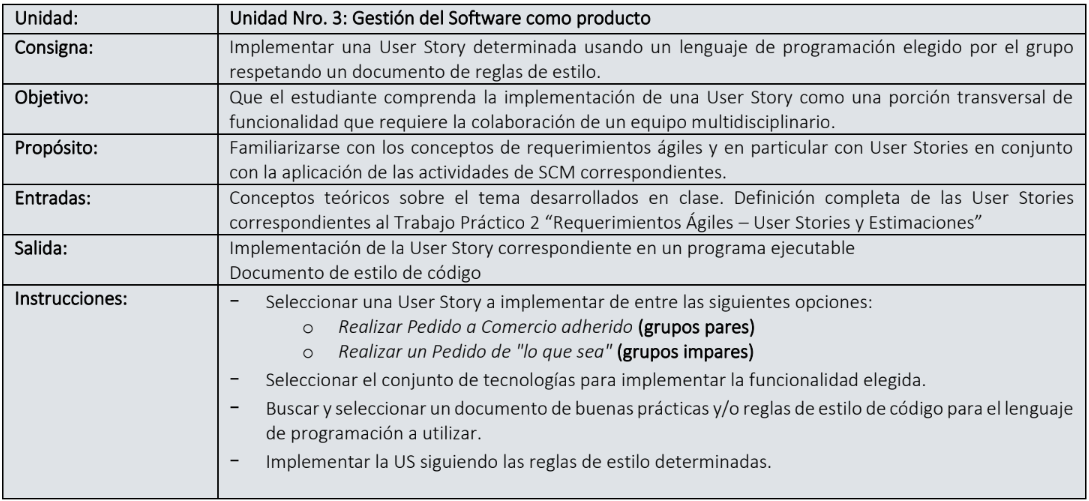
[**Enunciado**](#_f9f29ur92xpp) **2**

[**Resolución**](#_5ne21f5ikibm) **3**

[Tecnología](#_c3msafcx61m3) 3

[Guía de Estilo](#_f162o3921pjj) 3

# Enunciado





# Resolución

## Tecnología

La user se desarrolló con tecnología web, utilizando el lenguaje de programación Typescript, junto al framework Angular y la librería de estilos Bootstrap.

Se adjunta link del repositorio del código: https://github.com/Luuzaall/ISW-Grupo2-2022

## Guía de Estilo

Guía de estilo de la documentación de Angular: <https://angular.io/guide/styleguide>

**Single responsibility**

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.

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

**Small functions**

Style 01-02

Do define small functions

Consider limiting to no more than 75 lines.

**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.

**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.

**Symbols and file names**

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.

**Service names**

Style 02-04

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.

**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.

**Component selectors**

Style 05-02

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

**Component custom prefix**

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.

**Directive selectors**

Style 02-06

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

**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.

**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").

**Unit test file names**

Style 02-10

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

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

**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.

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.

**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.

**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.

**Locate**

Style 04-02

Do make locating code intuitive and fast.

**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.

**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.

**T-DRY (Try to be DRY)**

Style 04-05

Do be DRY (Don't Repeat Yourself).

Avoid being so DRY that you sacrifice readability.

**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).

**Folders-by-feature structure**

Style 04-07

Do create folders named for the feature area they represent.

**App root module**

Style 04-08

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

Feature modules

Style 04-09

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.

**Shared feature module**

Style 04-10

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.

**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.

**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.

**Do not add filtering and sorting logic to pipes**

Style 04-13

Avoid adding filtering or sorting logic into custom pipes.

**Components as elements**

Style 05-03

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

**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 ./.

**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.

**Avoid aliasing inputs and outputs**

Style 05-13

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

**Member sequence**

Style 05-14

Do place properties up top followed by methods.

Do place private members after public members, alphabetized.

**Delegate complex component logic to services**

Style 05-15

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.

**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.

**Put presentation logic in the component class**

Style 05-17

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

**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.

**Use directives to enhance an element**

Style 06-01

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

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.

**Services are singletons**

Style 07-01

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

**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.

**Providing a service**

Style 07-03

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

**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.

**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.

**Implement lifecycle hook interfaces**

Style 09-01

Do implement the lifecycle hook interfaces.

**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.