FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Trabajo Práctico Nº 6:

REQUERIMIENTOS ÁGILES - Implementación de User Stories

Ingeniería de Software - 4K2

> Docentes:
- Judith Meles
- Cecilia Massano
- Gerardo Boiero

> Grupo N° 6:
- 79110, Marandino Giovanna
- 72380, Acevedo Hernán
- 67298, Carranza Agustín
- 72459, Romero Alejandro

Índice

Introducción

Angular es un framework de diseño de aplicaciones y una plataforma de desarrollo para crear aplicaciones de una sola página eficientes y sofisticadas. Como características a destacar es código abierto a la comunidad y está en continuo desarrollo por 1395 contribuidores y es usado actualmente por 1,8 millones de personas según los datos que se encuentran en su repositorio de Github.

El objetivo de este documento es presentar de una forma resumida los estándares o reglas que se utilizan a la hora de escribir código fuente en el framework Angular en su versión 11.2.11.

Aplicando las buenas prácticas aumenta el valor del software, facilita el desarrollo para los programadores e incrementa la legibilidad y mantenibilidad si su utilizará el código en futuros proyectos.

Las buenas prácticas no tienen una traducción en español por lo que se procederá a traducir solamente los títulos.

Además se incluye en el trabajo práctico un apartado donde se describen las tecnologías utilizadas para llevar a cabo la implementación de la user story.

Tecnologías utilizadas para la implementación

	Angular 11 como framework de desarrollo web que se adapta a navegadores que se ejecutan en PC y en móviles.
	Google Firebase como base de datos Realtime .
	Bootstrap para diseño del sitio.
	Gitlab como repositorio de código remoto.
	Netlify como plataforma para hacer el deploy de la aplicación.

Link de acceso a implementación de US Realizar Pedido a Comercio Adherido

https://delivery-eat-grupo-6.netlify.app/

Forma de redacción de las buenas prácticas

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

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

Do es una directriz que siempre debe seguirse.

Consider las directrices generalmente se siguen. Si comprende completamente el significado detrás de la directriz y tiene una buena razón para desviarse, hágalo. Por favor, esfuércese por ser coherente.

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

Why? da razones para seguir las recomendaciones anteriores.

Guía de estilo de codificación Angular 11

01 - Responsabilidad Única

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

01-01 Regla del uno

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.

01-02 Funciones pequeñas

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? 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.

02 - Nombrado

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

02-01 Directrices de nombrado generales

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 simply help find desired code faster and make it easier to understand.

Why? 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.

02-02 Separar los nombres de archivos con puntos y guiones medios.

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.

02-03 Símbolos y nombres de archivos

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.

02-04 Nombres de servicios

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.

02-05 Bootstrapping

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

Do include error handling in the bootstrapping logic.

Avoid putting app 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.

05-02 Selectores de componentes

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.

02-07 Prefijo personalizado de componente

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 app itself.

Why? Prevents element name collisions with components in other apps and with native HTML elements.

Why? Makes it easier to promote and share the component in other apps.

Why? Components are easy to identify in the DOM.

02-06 Selector de directivas

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.

02-08 Prefijo personalizado de directiva

Do use a custom prefix for the selector of directives (e.g, 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.

02-09 Nombres de tuberías

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.

02-10 Nombres de archivo de prueba unitaria

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.

02-11 Nombres de archivo de pruebas End-to-End (E2E)

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.

02-12 Nombres de Angular NgModule

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.

04- Estructura de la aplicación y NgModules

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

All of the app'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.

04-01 LIFT

Do structure the app 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?

04-02 Localiza

Do make locating code intuitive, simple, 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.

04-03 Identifica

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.

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.

04-04 Estructura plana

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.

04-05 T-DRY (Intenta ser DRY)

Do be DRY (Don't Repeat Yourself).

Avoid being so DRY that you sacrifice readability.

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.

04-06 Directrices estructurales generales

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

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

Do put all of the app'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 app structure small and easy to maintain in the early stages, while being easy to evolve as the app grows.

Why? Components often have four files (e.g. *.html, *.css, *.ts, and *.spec.ts) and can clutter a folder quickly.

04-07 Estructura de carpetas por función

Do create folders named for the feature area they represent.

Why? 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 app 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.

04-08 Módulo raiz App

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

Why? Every app requires at least one root NgModule.

Consider naming the root module app.module.ts.

Why? Makes it easier to locate and identify the root module.

04-09 Módulo de funciones

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.

04-10 Módulo de funciones compartidas

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? 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.

04-11 Carpetas de carga perezosa

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.

04-12 Nunca importes carpetas de carga perezosa directamente

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.

05 - Componentes

05-03 Componentes como elementos

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.

05-04 Extraiga plantillas y estilos a sus propios archivos

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? 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.

05-12 Decora las propiedades de entrada y salida

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.

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.

05-13 Evite ponerle alias a las entradas y salidas

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.

05-14 Secuencia de los elementos (Members)

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.

05-15 Delegar la lógica de componentes complejos a los servicios

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 via 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.

05-16 No prefije las propiedades de salida

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.

05-17 Ponga lógica de presentación en la clase de componentes

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.

05-18 Inicializar entradas

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.

06 - Directivas

06-01 Usar directivas para realzar un elemento

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.

06-03 Decoradores HostListener/HostBinding decorators versus host metadata

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.

07- Servicios

07-01 Servicios son singletons

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.

07-02 Responsabilidad única

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.

07-03 Proveer un servicio

Do provide a service with the app 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? 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.

07-04 Usar el decorador de clase @Injectable()

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.

08 - Servicios de datos

08-01 Hablar con el servidor a través de un servicio.

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? 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.

09 - Lifecycle hooks

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

09-01 Implementar interfaces lifecycle hook

Do implement the lifecycle hook interfaces.

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

Apéndice

Useful tools and tips for Angular.

A-02 File templates and snippets

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.

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.

Referencias - Páginas de consulta

Angular coding style guide - Angular.io:
https://angular.io/guide/styleguide#file-structure-conventions

Angular - One framework. Mobile & desktop:
https://github.com/angular/angular