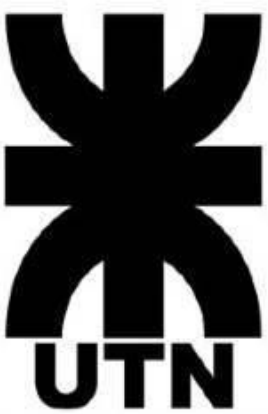


Universidad Tecnológica Nacional
Facultad Regional Córdoba

Ingeniería en Sistemas de Información
Ingeniería de Software

Trabajo Práctico N° 6
"REQUERIMIENTOS ÁGILES
Implementación de user stories"

Docentes:

- Adjunto: Ing. Covaro, Laura
- JTP: Ing. Massano, María Cecilia
- Ing. Robles, Joaquín Leonel

Integrantes:

- Acosta, Natalia 48605
- Irahol, Mauricio 38618
- Palacios, Aylén Macarena 69742
- Ponce, Santiago Javier 70083
- Quezada, José 55251
- Valles, Martín 71031

Curso: 4K3

Grupo n°: 1

Índice

Enunciado

Unidad:	**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:
	  o *Realizar Pedido a Comercio adherido* **(grupos pares)**
	  o *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.

A continuación se detalla la User Story correspondiente al Grupo N° 1.

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 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)

Desarrollo

A continuación detallaremos nuestro Manual de Buenas Prácticas al hacer aplicaciones en Angular.

Antes de comenzar, seguiremos la guía de estilos oficial de angular donde se describe de forma completa y actualizada como trabajar con Angular.
https://angular.io/guide/styleguide

1 Cómo declarar variables en JavaScript.

Existen 3 formas para declarar variables en JS (const - let - var). Cada una de estas tiene un significado y un uso específico.
El correcto manejo de las mismas permiten una lectura del código más clara, evitan errores y permiten generar un código más robusto.

Nota: Para comprender mejor el significado de los tipos de declaraciones de variables en JS hay que aclarar que existen varios scope.
Podemos definir como al scope como el alcance de un bloque de código, según donde se declare una variable tendrá un scope global siendo reconocida por todo el script o un scope de bloque siendo reconocida dentro del bloque de código declarado.

Significado:
`-const`: Se declara una variable como constante, es decir que una vez que se asigna un valor, este no puede ser modificado.

`-let`: Se declara una varia scope de bloque, por lo tanto solo existe dentro de su bloque de código, es decir todo lo que esté dentro de las mismas llaves ({})

`-var`: Se declara una variable dinámica dentro de un scope de función o global, es decir que va a existir dentro de toda la ejecución de la misma. Esta forma de declarar una variable es la que menos se debe utilizar.

2 Uso pipes de RxJs

El uso de RxJs es algo necesario al trabajar con observables, en vez de tener que importar cada operador de la librería que necesitamos para trabajar se puede importar pipe este operador permitiendo usar operadores de RxJs como map en secuencia, reduciendo los operadores a ejecutar solo los necesarios.

Además de permitir identificar casos de tener operadores “sueltos”. En caso que la ejecución corte a la mitad de la función solo se habrán importado y utilizado los operadores RxJs que se solicitaron entre medio, no todos los que estén en el código. El caso más claro se ve en el uso de un catch, el catch de rxjs solo se utilizará en caso de error sino, el compilador lo verá como si no estuviera ahí.

3 Utilizar servicios

En angular los injectables son elementos que se ejecutan de forma volátil, es decir existen durante la vida del componente que los llamo, por lo tanto las llamadas a APIs internas o externas a nuestro backend deben de realizarse por este medio, no debe de haber llamadas HTTP dentro de ningún componente (debido a la persistencia de los mismos).
De la misma manera para comunicarse entre componentes o incluso para reutilizar código los componentes deben poder comunicarse con servicios propios o con los servicios del padre.
En caso de tener un módulo con varios componentes hermanos, supongamos en el módulo 1 se encuentran los componentes Ca y Cb, cada uno con sus respectivos servicios Sa y Sb, en caso de que Ca necesite de utilizar un bloque de código de Sb, sería prudente que llame al servicio del padre S1 o en su defecto que realice su propia implementación en Sa.
Lo que nos lleva al número 5.

4 Parsear las respuestas de las API en los servicios

Debemos tener en cuenta que no siempre las respuestas de una API van a estar bien diseñadas, estas pueden estar mal formadas o incluso pueden no ser específicas para la funcionalidad que queremos usar.

Supongamos un API que devuelve todas las personas de una organización para cargar un combo, en el caso que este esté mal formado (en vez de ser un array de objetos es un objeto con un array de objetos dentro por ejemplo), o que no nos interesan todos los datos de las personas para el combo, suponiendo que solo necesitamos el id y el nombre, sería prudente solo pasarle dichos datos a el componente y no todos los datos para que cada componente deba pasarlos a su manera, así reutilizamos código y los servicios no proveen solo servicios sino que proveen buenos servicios.

5 Suscribirse a vistas, no a componentes

Los componentes pueden perdurar incluso una vez terminado su uso, las vistas no.
Un componente solo se destruye si se refresca el browser, o si se le indica la destrucción del mismo.
Por lo tanto resulta buena idea subscribirse en el HTML mediante el pipe async, evitando así manejar la desubscripción del mismo.

6) Cerrar las suscripciones

Así como se mencionó en el caso 5, al subscribirse a un observable es necesario de-suscribirse correctamente, para no perder memoria.
Mantener las suscripciones abiertas consume memoria y puede ralentizar la aplicación o la ocurrencia del cierre inesperado de la misma.
Para ello RxJs provee operadores específicos, como take o takeUntil. Siempre es una buena idea de-suscribirse una vez terminado el uso de un observable o incluso en la muerte del componente(ngOnDestroy).

7 Usar el operador apropiado para cada situación
Cada operador tiene una única función o al menos una en la que destaque, esto permite que a simple vista uno pueda interpretar que está realizando el código.
Supongamos el caso de usar un for o un map para iterar sobre un array, si bien los dos funcionan de manera similar, el map al ser un operador específico de colecciones, no hace falta saber que se está haciendo para saber que se está iterando una colección a diferencia del for. Al igual que este caso existen casos mucho más específicos, como switchMap o exhaustMap.
La especialización y el uso del operador correcto permite mayor lectura y menos comportamientos no esperados.

8 Uso de Lazy load y Eager load
A la hora de cargar una aplicación debemos ser cautelosos, cargar todos los módulos de una aplicación muy grande podría provocar un gasto innecesario de memoria y puede resultar en una aplicación lenta, sobre todo en el arranque, mientras que la implementación de lazy load o carga perezosa permite cargar los módulos a necesidad, “en caliente” si se puede decir.
¿Se recomienda usar siempre Lazy Load en vez de Eager load?
No, ya que el uso de lazy load puede resultar en una mala experiencia de usuario, ya que cada módulo debe buscarlo del servidor a disposición.
El mismo funciona bien para módulos pequeños o unos que no se utilizan en todos los casos, más que nada en usuarios con diferentes perfiles.

9 Utilizar estrategias de precarga
Sabemos que no es factible elegir una forma única para cargar los módulos de una aplicación en angular sino que debería analizarse según cada proyecto.

Angular nos provee de una prediseñada llamada PreloadAllModules, que permite realizar una carga perezosa de todos los módulos a medida que van terminando de cargar.
Una vez cargado el padre carga a los hijos. Pero a veces no nos alcanza con esta

estrategia, entonces debemos crear una estrategia propia de precarga, parte perezosa, parte ansiosa.

10 Evitar suscripciones anidadas
El uso de los operadores de cadena de RxJs como combineLatest o withLatestFrom, serán nuestros aliados a la hora de encadenar suscripciones, y así evitar anidarlas.
Esto permite una mejor lectura del código y un uso correcto de RxJs además de trabajar con el asincronismo con mayor facilidad evitando cadenas de “async y await”.

11 Tipificar los datos
Si bien TS es un lenguaje de tipado dinámico, que permite la creación de variables de tipo any que pueden ser cualquier tipo de dato, es una buena práctica tipificar correctamente cada dato.
En una variable que es un número declararla como un número, una que es una cadena de caracteres declararla como un string y una que es un array de estudiantes declararlos como un array de Estudiantes. Permitiendo una mejor visualización del código, siendo más robusto y legible, además también elimina problemas inesperados.

12 Usar reglas Lint
Lint son un conjunto de reglas que aplican a la aplicación en general, por ejemplo el uso de lint no-console, no permite que en la consola del browser se imprima nada, errores o incluso logs del programador a la mitad del debuggueo de un componente.
Para ver más sobre Lint: https://palantir.github.io/tslint/rules/

13 Los componentes deben ser atómicos
El uso de componentes grandes conlleva a errores, y mientras mayor tamaño tengan y serán mayores las funcionalidades que van a estar afectadas en la aplicación.
La práctica ideal consiste en reducir todos los componentes a su mínima expresión. Reduciendo código y reutilizando componentes, mientras más atómicos menos posibilidades habrá de que fallen, levantando menos errores en su camino.
Se busca llegar a “Un componente para una única funcionalidad”.

14 Los componentes deben lidiar solo con lógicas de visualización
Los componentes son los elementos con los que el usuario va a interactuar, estos no deben calcular precios ni promediar variables, para eso están las demás capas de la aplicación. El componente debe ser provisto de datos para visualizar, no entender lógicas de negocio, así cómo encargarse de mostrar esos datos.

15 Usar el caché a tu favor
El uso del mismo permite una mejora importante de performance de la aplicación. Hay que saber decidir que es necesario que maneje caché y que no. No se debe de dejar que el

browser decida por nosotros, esto termina implicando en cambios que no se visualizan a menos que se borre el mismo.
El buen uso del caché puede permitir ahorrar llamadas al backend, y reducir tiempos de espera por tener una mala conexión.

16 Versionar la aplicación
Cada release que se haga, debe de hacerse un versionado de la misma.
Esto incluye un número que indica en qué versión estamos, y qué cambios hubo en la última versión. Esta es una buena forma de rastrear de donde vino un bug que salió hoy de un cambio de hace semanas.
A su vez mediante el versionado le podemos indicar al cliente que debe actualizar su caché, si se encuentra en una versión anterior, permitiendo que este esté siempre actualizado con la última versión.

17 No utilizar lógica en la vista de un componente
Con angular es posible utilizar lógica directamente en el HTML, sin embargo el uso de la misma conlleva a una peor escalabilidad y no puede realizarse unitesting sobre ella.
En cada *ngIf debería de haber una función, no debería utilizarse la lógica de la misma en el mismo.
En el caso que no se desee usar una función puede usar una variable manejada en el componente, pero esta debe ser un booleano, y no se deben usar comparaciones en la misma.

18 Utilizar strings “seguras”
En caso de tener un string que solo puede tener valores fijos, supongamos tipos de perfiles, en vez de declararla como tipo string se la puede declarar como los valores posibles que puede llegar a utilizar, reduciendo la posibilidad de tener bugs en la aplicación.

19 Alias de importación
Algunos elementos de angular que van a ser utilizados por muchos lugares de la aplicación por ejemplo un servicio.
En caso que se quiera realizar esto, a veces resulta engorroso tipear toda la ruta hacia dicho elemento e incluso si realizamos un movimiento de un elemento de una carpeta a otra se puede desreferenciar de todos lados, para ello angular provee de la opción de guardar rutas absolutas en su configuración utilizando un alias, seguido de un @.
Suponiendo que queremos usar una variable de entorno podemos configurar un alias para eso escribiendo en el tsconfig.json bajo paths.
permitiendo así, importar los elementos mediante la línea:

```
import{ nombre1, nombre2 } from ‘@enviroment’
```

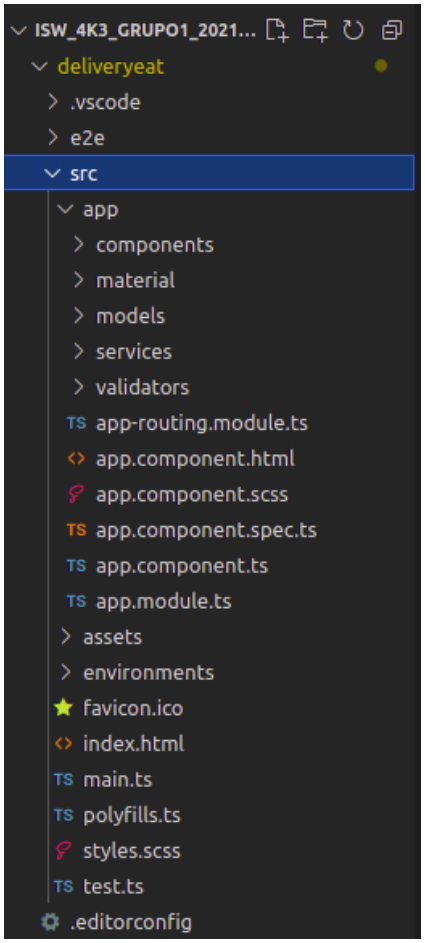
20) Comentarios y nombres en las variables

Al igual que en cualquier lenguaje de programación TS y JS proveen herramientas para hacer comentarios, si bien esto no es puntual de estos lenguajes o del framework siempre es bueno recordar que el que lee nuestro código puede o no saber que estamos intentando hacer, y una ayuda de nuestra parte, mediante comentarios o variables claras puede facilitarle la interpretación del mismo.
La nomenclación de una variable debería cumplir con el objetivo de mostrar de forma lo más clara posible su contenido y/o su tipo de datos sin necesidad de entender su contexto.
Ejemplo: indP => indicadorProductividad(esto puede ser un número o un booleano)
personaL => personasList(indica una lista de personas)

Mientras más claro sea uno a la hora de definir los nombres mientras escribe el código, mejor será su entendimiento a futuro o para alguien que intente leerlo y comprenderlo posteriormente facilitando el mantenimiento del mismo.

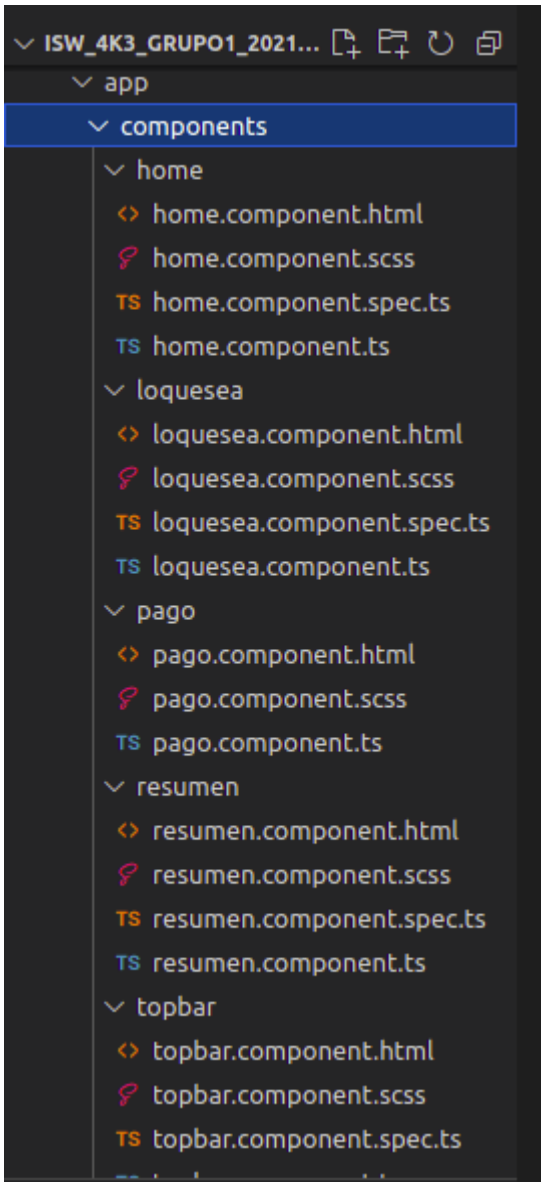
21 Estructura de archivos

Angular CLI nos provee una forma de crear un nuevo proyecto con una estructura bastante cómoda y práctica:



Generalmente se define la estructura de acuerdo a la complejidad del proyecto. Entre más grande sea, más orden y modularidad requiere.

Una de las mayores ventajas que proporciona angular frente a otros frameworks, es principalmente la capacidad de mantener ordenados proyectos complejos, haciendo uso de la excelente modularidad que presenta. A pesar de que el presente proyecto no es de gran envergadura, sigue siendo de gran utilidad la estructura proporcionada por el framework a la hora de organizar las carpetas del proyecto. Se logra en primer lugar una separación de intereses haciendo uso de componentes, los cuales a su vez, internamente proporcionan una clara separación entre la capa visual (la vista), la lógica de negocio (controlador), y los datos
(modelo). Tal modularidad permite una mayor facilidad en la gestión y mantenimiento del proyecto, haciendo posible la labor conjunta de un equipo de desarrollo de un modo más eficaz.



Tener una estructura bien definida nos ayudará a pensar en escalabilidad y permitirá ubicar fácilmente los distintos archivos a medida que el proyecto crezca.

22 Regla de los 5 segundos

Si al leer un bloque de código, nos toma más de 5 segundos entenderlo, es mejor considerar refactorizar. La idea es que el código se entienda en el menor tiempo posible. Se logra creando diferentes funciones pequeñas y haciendo composición en funciones más complejas.
En caso de que el código falle, es más fácil encontrar el error y corregirlo sin afectar otras funciones.

23 Organizar el código

Algunas formas de tener un archivo de código más organizado y legible son:

- Lo más importante debe ir arriba.
- Primero propiedades, después métodos.
- Un Ítem para un archivo: cada archivo debería contener solamente un componente, al igual que los servicios.
- Solo una responsabilidad: Cada clase o módulo debería tener solamente una responsabilidad.
- El nombre correcto: en angular se recomienda que las propiedades y métodos deberían usar el sistema de camel case (ej: getUserByName), al contrario, las clases (componentes, servicios, etc) deben usar upper camel case (ej: UserComponent).
- Los componentes y servicios deben tener su respectivo sufijo: UserComponent, UserService.
- Imports: los archivos externos van primero.

24 Ser claros

Al escribir código, es probable que alguien más lea el mismo en algún momento. Es por eso que lo ideal cuando escribimos es pensar en la persona que lo leerá, o incluso en ti mismo.

Código auto-descriptivo:

- Explicar en el mismo código, no en comentarios.
- Los comentarios deben ser claros y entendibles.

Evita comentar si:

- Se trata de explicar qué hace el código (dejar que este sea tan claro que se explique solo).
- Se tiene funciones y métodos bien nombrados. Evitar ser repetitivo.

Comentar cuando:

- Se trata de explicar por qué se hizo lo que se hizo.
- Se trata de explicar las consecuencias del código escrito.

25 Componentes

En Angular, lo más importante debe ir al inicio. Dentro de los componentes generalmente se escribe primero las propiedades y luego los métodos. Así mismo, a veces suelen agruparse propiedades o funciones alfabéticamente, mientras que en otros casos se ordenan por funcionalidad. Lo importante aquí es mantener una consistencia durante todo el proyecto.

Además:

- Es importante tratar de escribir código lo más compacto posible. Cada quien tiene una forma distinta de escribir y estructurar sus funciones.
- Es deseable que los componentes se mantengan lo más simple posible. En este contexto, se delega la mayor parte de la lógica a los servicios.
- Mantener la consistencia a la hora de declarar funciones, evitar las faltas ortográficas y mantener la seriedad de los nombres utilizado