Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Diseño de Aplicaciones II

Descripción del diseño

https://github.com/ORT-DA2/164660-185720-234059

Santiago Alvarez – N° Estudiante: 185720

Juan Castro – N° Estudiante: 164660

Marcelo Tilve – N° Estudiante: 234059

Índice

Descripción general

La solución de este obligatorio brinda la posibilidad de gestionar farmacias y sus medicamentos a partir de distintos tipos de usuarios teniendo presente los permisos de cada uno.

Se creó una Web Api llamada PharmaGo la cual expone recursos para que sean consumidos por los distintos tipos de usuarios que interactúen con ella. Algunos de sus servicios son manejo de farmacias, usuarios, medicamentos, gestión de stock de medicamentos, compras de medicamentos por usuarios anónimos, generación de reportes, entre otros.

Dentro del sistema existen 3 tipos de usuarios, Administradores, Empleados y Dueños. Como se mencionó anteriormente, cada uno tiene sus permisos, restringiendo a cada uno de la información a la que no debe tener acceso.

El alcance de esta entrega se basó en realizar toda la lógica del problema sin una interfaz gráfica de usuario, por lo que se utilizó Postman como herramienta para interactuar con los endpoints del sistema.

Para persistir la información se utilizó la herramienta Entity Framework Core aplicando la metodología Code First. Primero se definió el dominio de la solución para poder a través del uso de migraciones modelar la base de datos en SQL Server.

En cuanto al control de versionado, el repositorio de GitHub fue desarrollado teniendo en cuenta el modelo GitFlow. Se crea la rama develop de la cual va a salir una rama por funcionalidad creada o tarea a modificar o resolver. Una vez que la funcionalidad fue creada o corregida, es mergeada a develop. Por último, se realiza un último merge a la rama main donde se encuentra la aplicación compilada en release.

Para evidenciar TDD, se realizaron commits de tests reds, luego tests green y por último commits de refactor de ser necesarios. A su vez, se utilizó la librería Mock para poder mockear la interfaz que utiliza cada capa de la aplicación y de esta forma lograr tests unitarios de cada clase que se testea.

El flujo de trabajo para las funcionalidades solicitadas por la letra consistió en desarrollar requerimientos en profundidad desde afuera hacia adentro o desde adentro hacia afuera. Para cada requerimiento, se comenzó por el controlador que se encargaba del mismo, para luego hacer la lógica y por último la gestión de los datos en la base. Idem para el desarrollo desde adentro hacia afuera, pero comenzando desde la base hasta llegar al controlador.

La utilización de TDD con Mocks y la inyección de dependencias con interfaces nos permitió poder basarnos en este modelo de trabajo, ya que nuestros proyectos no dependen directamente uno del otro y por lo tanto no era necesario tener toda una funcionalidad completa para probarla.

Diagrama de paquetes

A continuación, se detalla el diagrama de paquetes resultante de la solución.

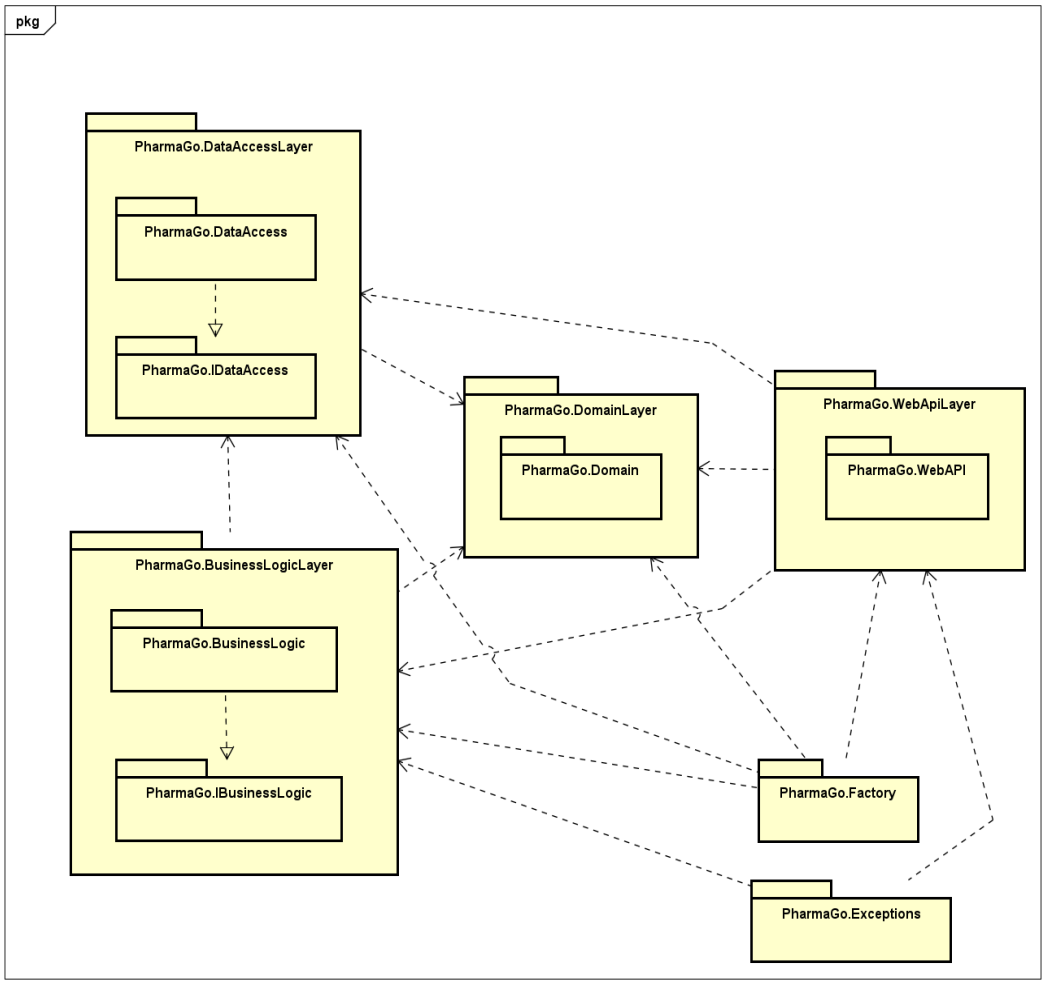


Ilustración 1: Diagrama de paquetes.

El diseño se basó en lograr obtener una solución que contemple la mantenibilidad y extensión del sistema, con el objetivo de que los cambios y nuevos desarrollos que serán solicitados en una segunda instancia,sean de bajo impacto al agregar nuevas características.

Para iniciar, en el paquete Domain, se encuentran las entidades necesarias para llevar a cabo el proyecto.

En el paquete ***IBusinessLogic***, se ubican las interfaces, que encapsulan operaciones que son específicas de las funcionalidades de cada controlador, que serán utilizadas por las clases del paquete ***BusinessLogic***, estas clases son las que proveen los servicios a la WebApi.

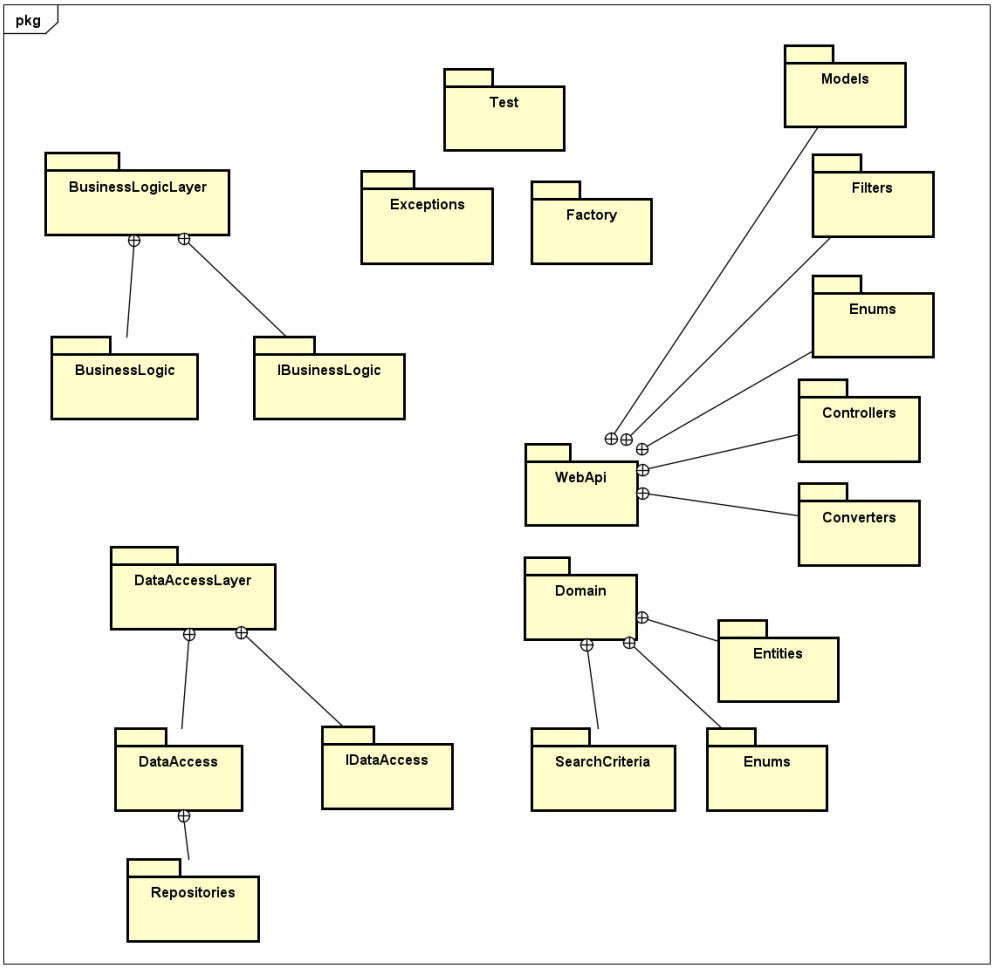
Lo mismo ocurre en el paquete ***IDataAccess*** y sus interfaces, que provee las operaciones que el paquete **DataAccess** implementará.

El paquete WebApi, mediante sus controladores, definen las rutas y se obtienen los datos de las request enviadas, con esa información, pide al servicio que resuelva la request, y el servicio mediante ***BusinessLogic*** accede a la base de datos a través del ***DataAccess***.

En este paquete también se encuentra el filtro de autorización, con el fin de autenticar y crear permisos de acceso para los diferentes roles de la aplicación.

Está división de paquetes y las interfaces especifica de manera clara qué responsabilidad tiene cada paquete, volviéndonos independientes unos de otros, por lo que si en el futuro, se decide por ejemplo, dejar de utilizar la WebApi y pasar a un Windows Form, se debería solamente desechar la WebApi y comenzar a implementar el Windows Form.

Diagrama de Paquetes Anidados



Modelo de tablas de la base de datos

Para llevar a cabo este primer obligatorio, se decidió crear el siguiente modelo de tablas de la base de datos PharmaGoDB.

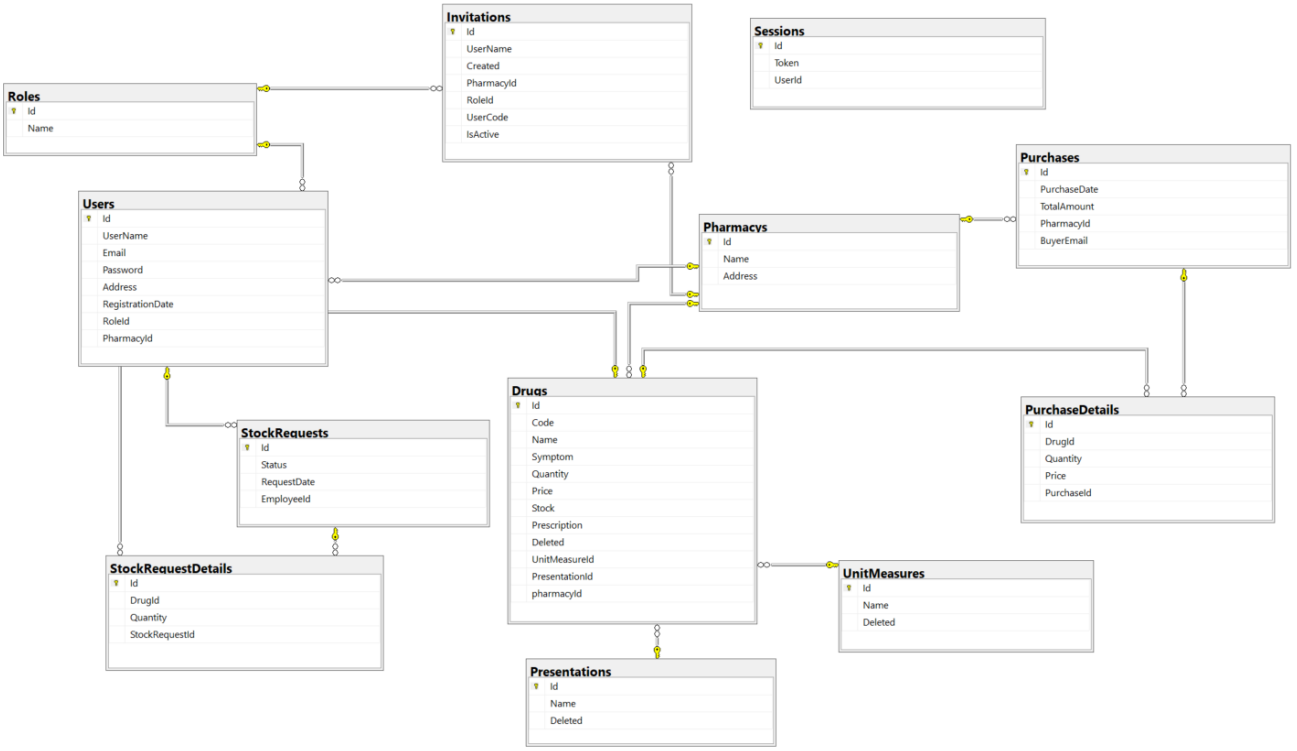


Ilustración 2: Modelo de tablas de la base de datos.

Diagramas de secuencia

- Crear Invitación

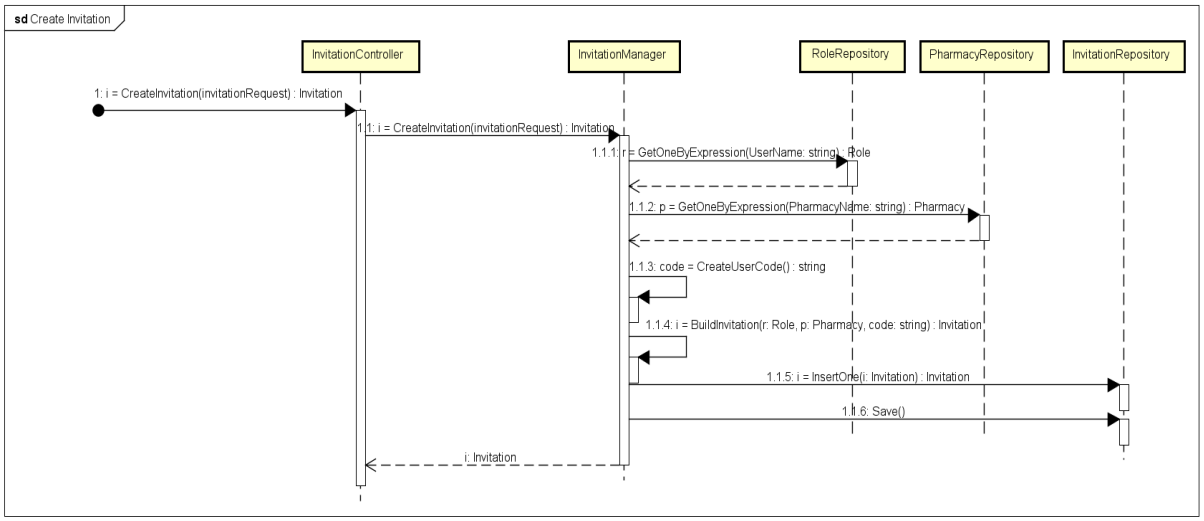


Ilustración 3: Diagrama de secuencia crear invitación.

- Crear Farmacia

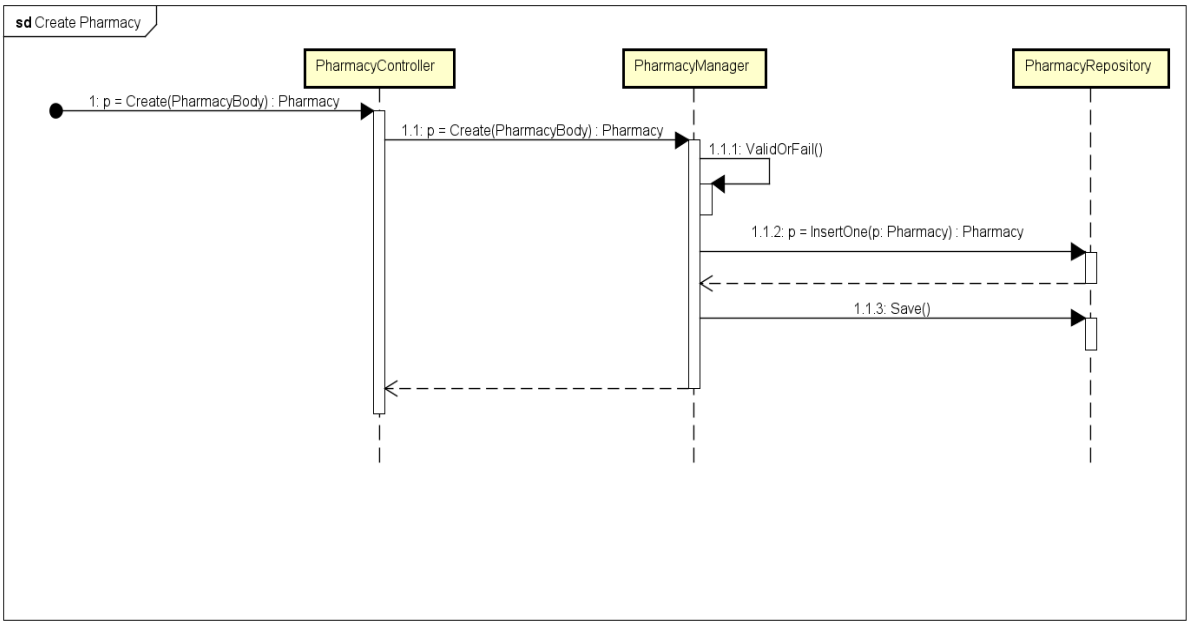


Ilustración 4: Diagrama de secuencia crear farmacia.

- Crear Compra

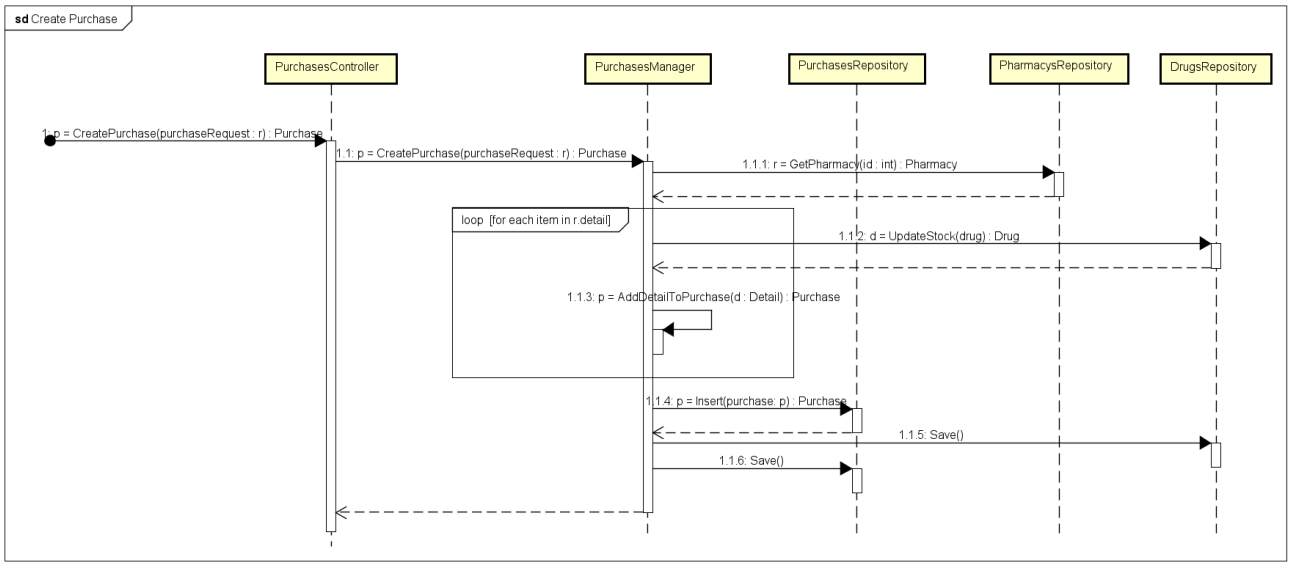


Ilustración 5: Diagrama de secuencia realizar compra.

- Crear solicitud de Stock

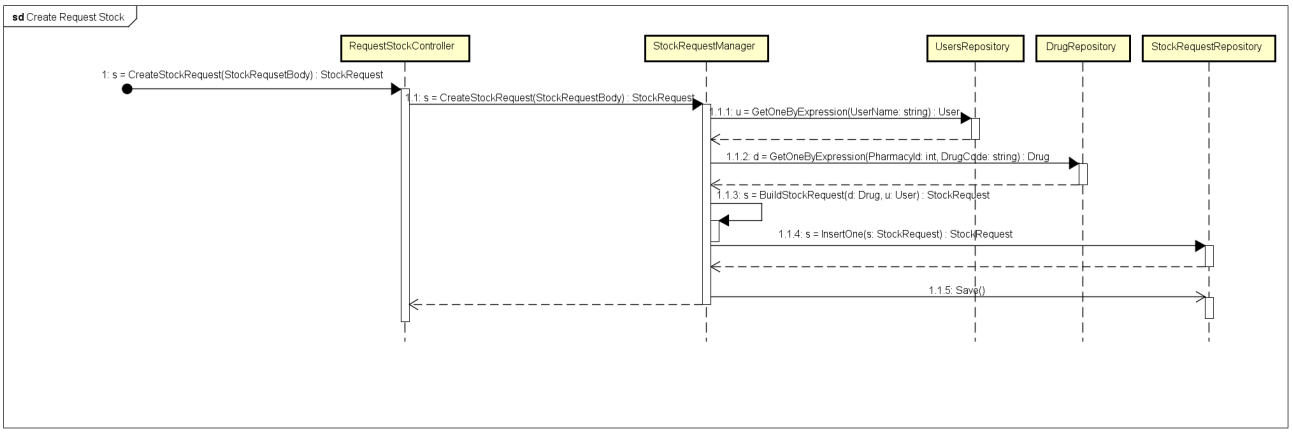


Ilustración 6: Diagrama de secuencia solicitud de reposición de stock de un medicamento.

- Crear Usuario

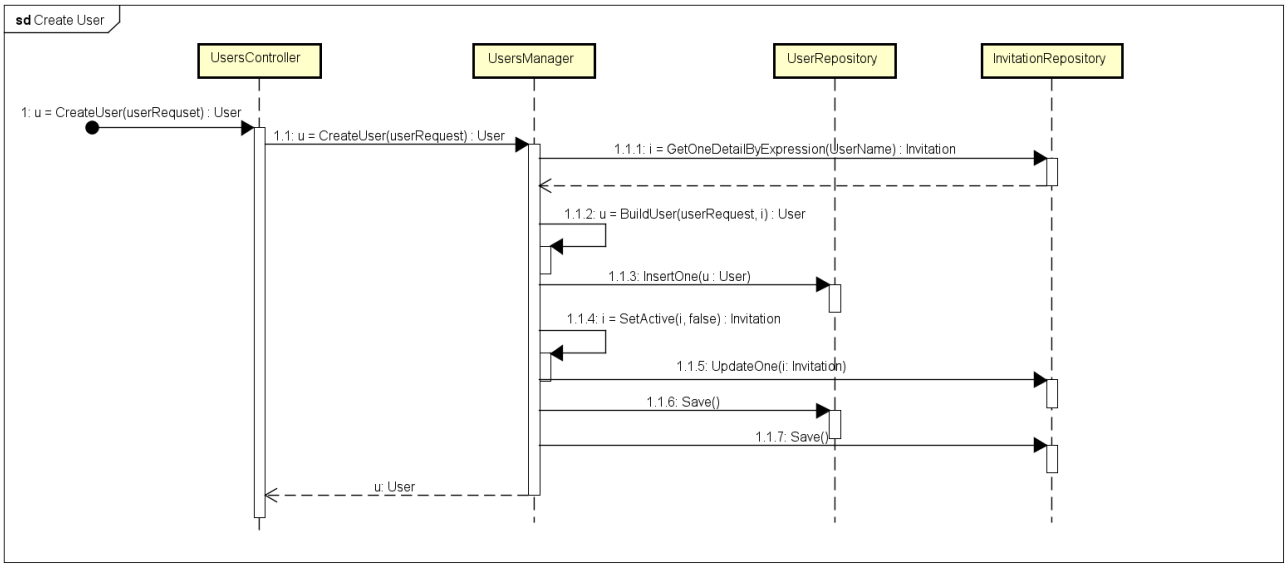


Ilustración 7: Diagrama de secuencia crear usuario.

- Login

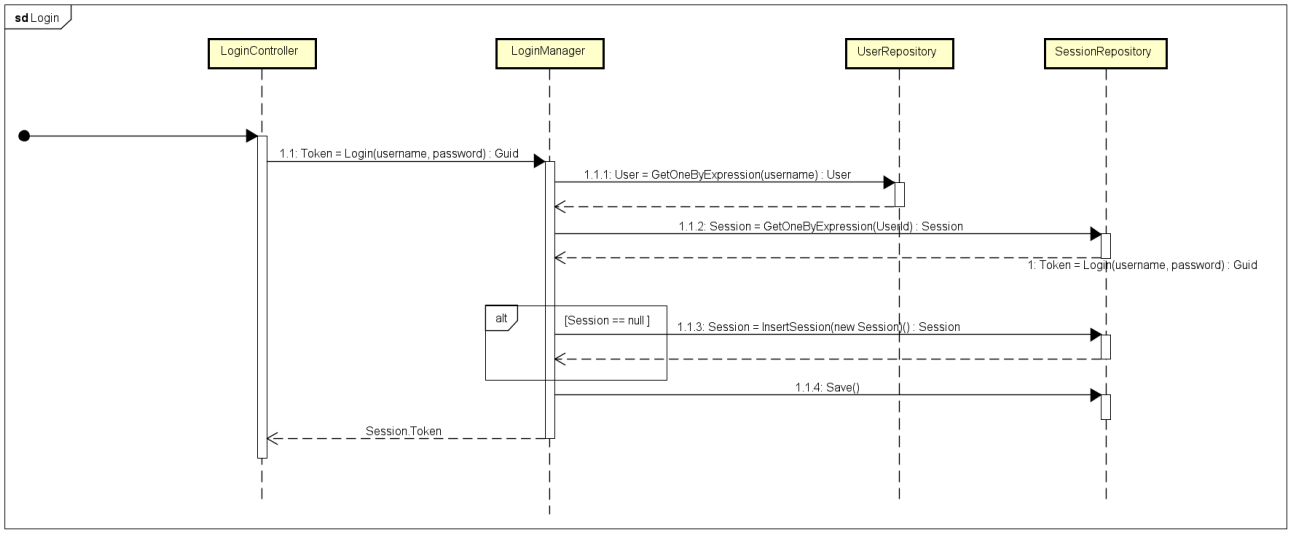


Ilustración 8: Diagrama de secuencia login.

- Detalle de compra por mes

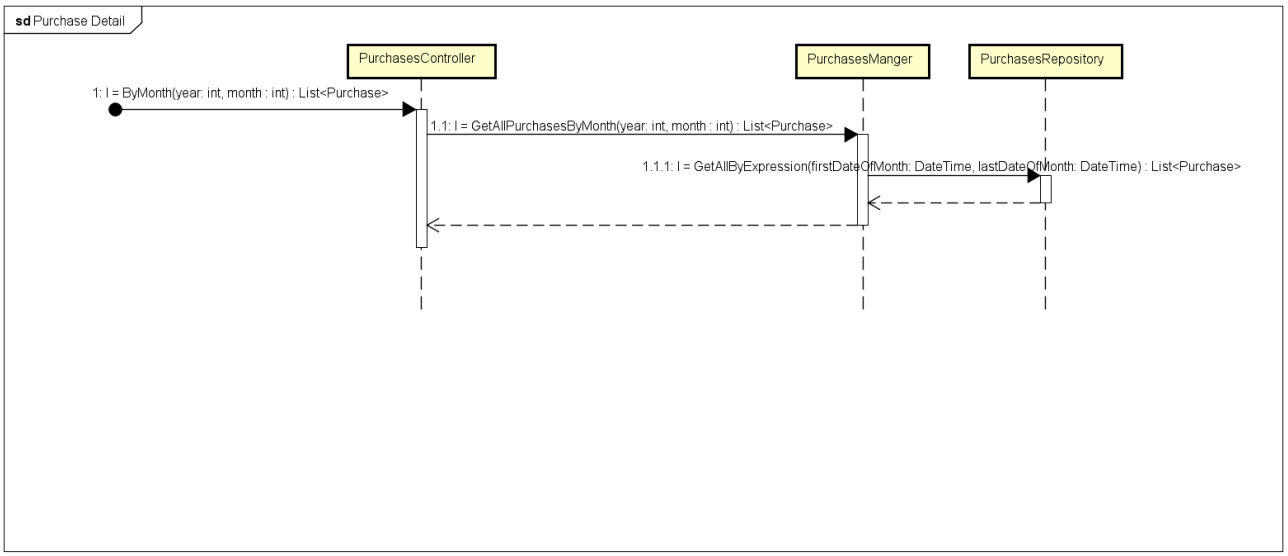


Ilustración 10: Diagrama de secuencia consulta compras realizadas por mes.

Justificación del diseño

Luego de analizar los requerimientos, se definieron las siguientes clases del dominio que brindarán una solución al problema.

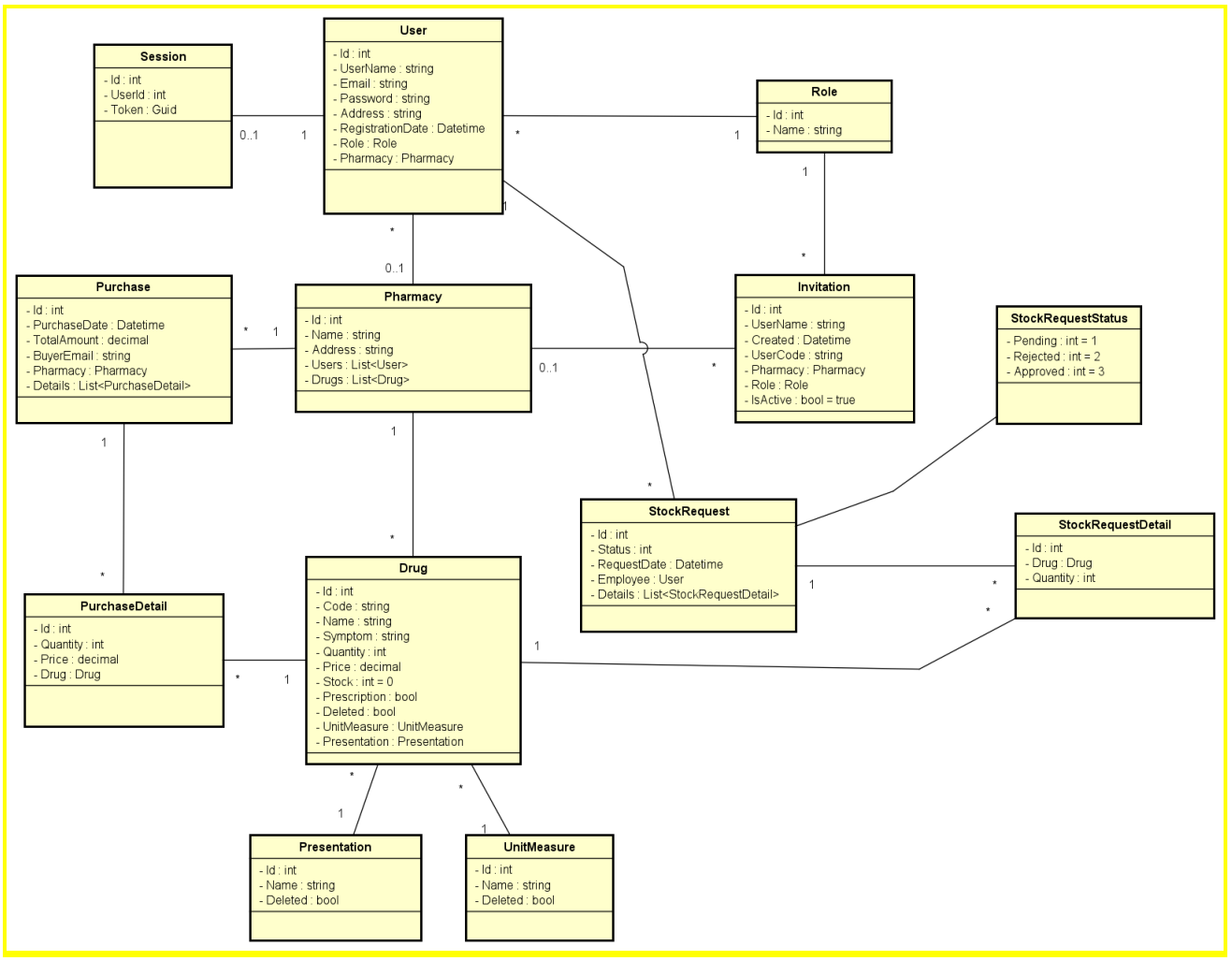


Ilustración 10: Diagrama de clases.

Patrones de Diseño

- Se puede entender a la api como la implementación del patrón Fachada.
- La realización de modelos (PharmaGo.WebApi.Models) es la implementación del patrón Adapter. De esta forma se está envolviendo las entidades para que los diferentes clientes no se vean afectados si las entidades cambian su estado.
- Dado que se puede cambiar la implementación de la interfaz en la Factory sin que esta se entere, y que también se podría hacer en tiempo de ejecución, se podría llegar a implementar a el patrón Strategy con Inyección de dependencia y la WebApi.

Decisiones de Diseño

- Se decidió que el borrado de las Drugs sea lógico, de esta forma se podrá acceder a dicha información en caso de ser eliminado del sistema.
- Se decidió mantener el Precio de una Compra en PurchaseDetail, pensando en el caso de que a futuro se desee actualizar los precios de las Drugs, de esta forma el histórico de precios de la compra será inalterable.
- Los diagramas indicados en el documento no cuentan con un detalle del 100% de las propiedades que pueden tener para mejorar la legibilidad en el documento.

Supuestos

StockRequest

Se supone que cuando se crea una solicitud de stock, la misma es creada con medicamentos existentes.

Aprobar una solicitud, puede demorar cierto tiempo en ser aprobada, en esa demora, es posible que un medicamento se elimine. Para tener presente este comportamiento, al momento de aprobar una solicitud, si hay algún medicamento eliminado en la misma, se muestra un mensaje de error.

PharmacyController y DrugController

Como extensión de la letra del obligatorio se decidió definir endpoints extra dentro de este controlador para completar las operaciones CRUD sobre la tabla de la base de datos. Ya que la letra no especifica nada acerca del rol que tienen que cumplir los usuarios que soliciten estos recursos se definió el rol sea Administrador.

Por otro lado, el delete de Farmacia no es posible hacerlo en el caso de que la misma tenga medicamentos asociados. Se tomó esta decisión ya que el delete de una drug realiza un borrado lógico de la tupla, para poder seguir mostrándola en reportes futuros. Por lo tanto, el CascadeDelete no era una opción a la hora de borrar una Farmacia ya que borraría sus drugs.

Diagrama de componentes

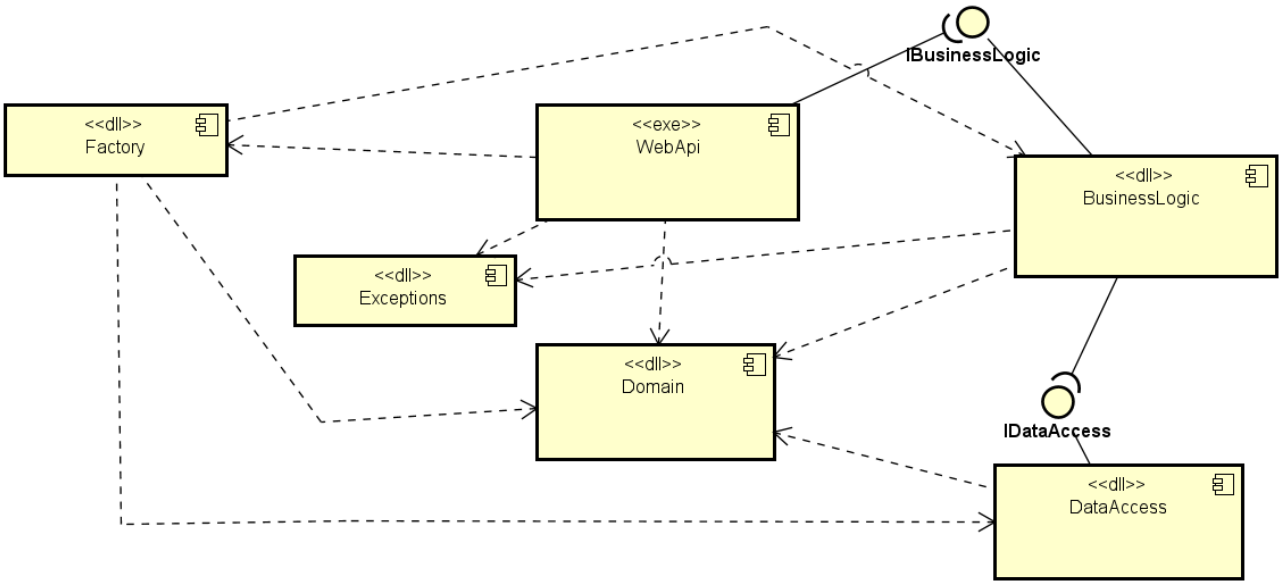


Ilustración 11: Diagrama de componentes.

Como se puede observar, en este caso a través de Postman (en el futuro el frontend Angular) utiliza la WebApi para traer las respuestas a las solicitudes del usuario, la misma a través de la interface IBusinessLogic provista por BusinessLogic accede a los servicios del mismo, para que estos a través de la interface IDataAccess, provista por DataAccess, extraiga los datos necesarios de la base de datos PharmaGoDb y los maneje de acuerdo a las necesidades del usuario y sus funcionalidades.

Esta división nos pareció la opción más correcta a desarrollar según los requerimientos, ya que tenemos una división de componentes clara, siendo lo más independiente posible uno de otro, dándole un enfoque importante a la mantenibilidad y teniendo fácilmente identificado las responsabilidades de cada uno, en el caso del Domain, se encargará de almacenar las entidades básicas requeridas para el funcionamiento de la aplicación.

DataAccess se encarga de almacenar, mantener y obtener los datos que se soliciten, en la BusinessLogic se encarga con dichos datos de realizar la lógica necesaria para que los requerimientos se cumplan, y a través de la capa de la WebApi proveemos a los usuarios acceso a las funciones del sistema, en esta versión a través de Postman.

Mecanismo de inyección de dependencias

La inyección de dependencias es un patrón de diseño que permite a PharmaGo romper el acoplamiento entre diferentes componentes. Para resolver la inyección de dependencias se creó un Proyecto PharmaGo.Factory que se encarga de resolver esto.

PharmaGo.Factory cuenta con una clase ServiceFactory que se encarga de registrar tanto las dependencias de la capa de negocios como la de acceso a datos:

- RegisterBusinessLogicServices: se encarga de registrar y asociar las interfaces IBussinesLogic con los Managers de BussinesLogic.
- RegisterDataAccessServices: se encarga de registrar y asociar las interfaces de cada repositorio IRepository de IDataAccess con los respositorios de DataAccess. Por otro lado se encarga de registrar el contexto de la base de datos PharmaGoDb obteniendo el connection string del archivo **appsettings.json.**

PharmaGo.WebApi se decidió que sea el proyecto ejecutable de inicio dentro de la solución, dentro del mismo se encuentra Program.cs en donde se llama a los métodos antes mencionados de la Factory para registrar las dependencias.

La inyección de dependencias nos permite un sistema extensible y que usa acoplamiento de interfaz y no de implementación, generando que este sea flexible a los cambios, que permita cambiar implementaciones en tiempo de ejecución y respetar los SOLID de diferentes formas.

Descripción del mecanismo de acceso a datos utilizado

Mediante ‘CodeFirst’ se definieron entidades en el proyecto **PharamaGo.Domain**, luego se definió el **PharmacyGoDbContext** en DataAccess, permitiendo así mediante el Entity Framework poder generar migrations y posteriormente generar la base de datos.

Las Migrations se encuentran en **PharmaGo.DataAccess.Migrations** y muestran la evolución del diseño a lo largo del tiempo y las decisiones que debieron tomarse de forma iterativa e incremental durante el desarrollo.

Se decidió utilizar un repositorio genérico **IRepository** con un tipo **T**, permitiendo utilizar las operaciones básicas como insertar, borrar, obtener y actualizar de forma sencilla y para todas las entidades. Luego se definieron repositorios específicos en caso de ser necesario para consultas más complejas de las que el repositorio base podría ofrecer.

El desarrollo se realizó mediante el Entity Framework Core 6.0.8 con SQL Server Express Edition 14.0.

Descripción del manejo de excepciones

Para el manejo de Excepciones se decidió crear un filtro ***ExceptionFilter*** encargado de capturar todas las excepciones que suceden en el sistema, estas se dividen en:

- **ResourceNotFoundException**: para capturar excepciones donde no se encuentra los recursos requeridos por las request del cliente, devolviendo un StatusCode de 404.
- **InvalidResourceException**: para capturar excepciones donde no se cumplen validaciones de los diferentes datos recibidos por el request del cliente, devolviendo un StatusCode de 400.
- **FormatException**: para el caso específico de que el token recibido por el header Authorization no cumple con el formato Guid, devolviendo un StatusCode de 400.
- **Exception**: para capturar el resto de las posibles excepciones, devolviendo un StatusCode de 500.

El formato de todas las respuestas de error devuelve un objeto JSON que tiene una key “message” en donde se muestra el mensaje de error de la excepción.

Las excepciones **ResourceNotFoundException** y **InvalidResourceException** son dos excepciones custom que heredan de **Exception** y cuentan con su propio proyecto PharmaGo.Exceptions dentro de la solución.

Dentro de Program.cs en el Proyecto PharmaGo.WebApi se agrega este filtro para que sea tenido en cuenta dentro del sistema.

Mejoras a tener en cuenta a futuro

- Normalizar las fechas recibidas, hoy en día persisten las fechas tal cual son recibidas desde el cliente.
- Paginar los reportes del lado del servidor, hoy en día no se paginan los reportes, devolviendo todos los datos, a futuro sería mejor implementar un *skip* y *top* para traer los datos de a partes desde la base de datos.
- Filtros aplicados en el reporte de solicitudes de stock por empleado, se pueden aplicar por separado y no en conjunto. Tienen bugs identificados por el equipo que se espera solucionar para la segunda entrega.
- Nos percatamos de que los nombres de los endpoint no quedaron correctamente creados, en algunos casos nos dimos cuenta que no se siguió las buenas prácticas de una API REST para el nombre de los endpoint. Se espera poder corregir esto para la segunda entrega.
- Las validaciones contra la unidad de medida y presentación del medicamento deberían realizarse validando el Id de cada tupla predefinida en la base de datos en vez del nombre. Esto permite agregar nuevas tuplas en las tablas sin tener que compilar o modificar el código de la aplicación.
- La creación de una solicitud de stock no está devolviendo el id cuando se crea, por lo tanto hay que ir a buscar el id a la base de datos para realizar alguna prueba, se pretende agregar dicho dato para la segunda parte.

Flujo de trabajo

Se utilizó durante todo el desarrollo el flujo de trabajo de GitFlow para gestionar las ramas de Github. De esta forma se creó un rama ‘main’, una rama ‘develop’ y múltiples ramas ‘feature/nombre-del-feature’ siguiendo rigurosamente los siguiente:

- Nunca se hace commit directamente a ‘develop’.
- Por cada nueva característica se crea una nueva rama ‘feature’.
- Las ramas ‘feature’ se mergean a ‘develop’ solo por PR.
- Al finalizar el desarrollo se mergea ‘develop’ en ‘main’ y se crea un tag de la versión.