Universidad ORT
Facultad de Ingeniería

Descripción del Diseño

Diseño de Aplicaciones 2 - 6/10/2022

Repositorio Git
https://github.com/ORT-DA2/DA2-163471-223139-201250.git

Federico Czarnievicz - 201250 - M6C-1
Ignacio Olivera - 223139 - M6C
Cristhian Maciel - 163471 - N6A

Índice

1. Descripción general del trabajo

1.1 Introducción

Se nos solicitó realizar un sistema que cumpla con la capacidad de administrar diferentes tipos de usuarios para el manejo del negocio de una red de farmacias. La aplicación permite a los usuarios realizar peticiones bajo un perfil anónimo, como también registrarse siguiendo los requisitos correspondientes. En esta primera versión existen cuatro roles de usuario diferentes que modelan las reglas del negocio y forman en conjunto los variados clientes que interactúan con la plataforma de farmacias, la cual denominamos: **PharmacyManager.**
Para esta primera entrega, solamente debemos desarrollar el backend de nuestra aplicación sin la necesidad de realizar el frontend, es por esto que el cliente interactúa con el sistema mediante los endpoints expuestos por una Web Api Rest, accediendo desde Postman.

Los principales servicios ofrecidos son:

- Registrar un usuario
- Dar de alta/baja a un medicamento
- Solicitar Stock de un medicamento
- Aprobar/Rechazar la solicitud de stock
- Realizar la compra de un medicamento
- Ver solicitudes de reposición de stock, pudiendo filtrar la búsqueda.

1.2 Bugs y mejoras a futuro

Creemos que es importante documentar distintos inconvenientes que surgieron en cuanto a diseño e implementación para poder tenerlos en cuenta y refactorizar en la siguiente entrega:

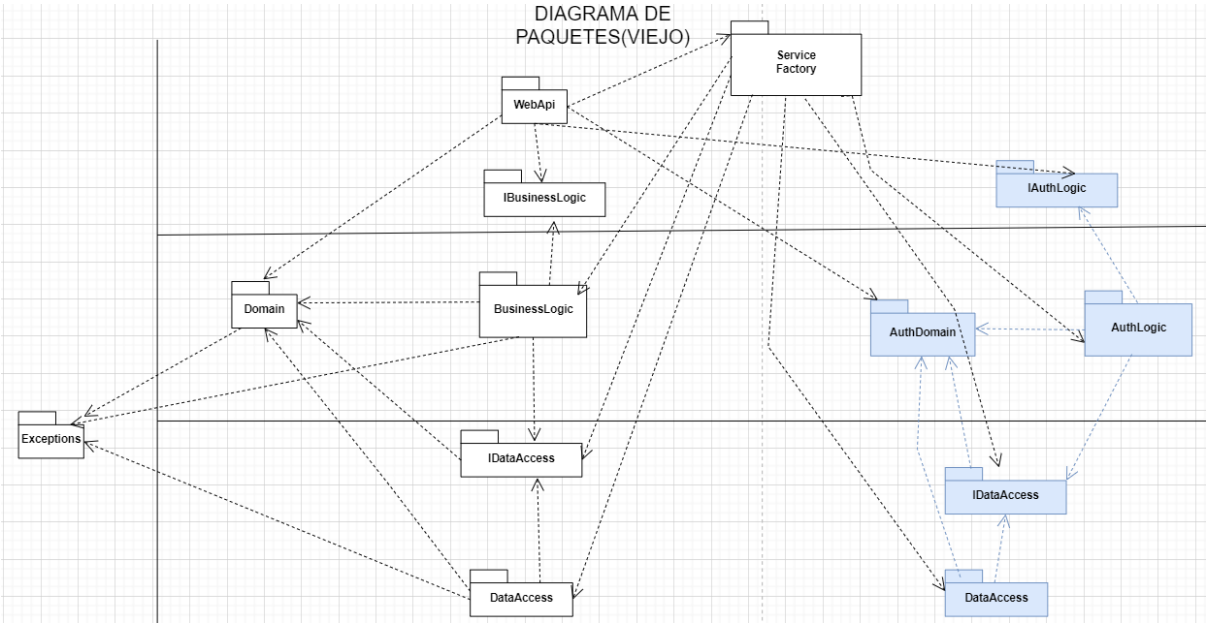
Case insensitive: Refactorizar en un futuro para que el nombre de los roles sea Case insensitive. Es decir, al asignarle un nombre a cada uno de los roles existentes, el mismo pueda empezar tanto con minúscula como con mayúscula.

Delete de Drug: En una próxima instancia, se debe integrar el delete de la droga con las solicitudes pendientes. Es decir, cuando la droga existe y se hace una solicitud de reposición, se realiza la misma con normalidad. En cambio, si se borra dicha droga, la solicitud continúa en estado pendiente sin eliminarse la misma, lo que puede generar un error en caso de intentar aceptar la solicitud pendiente de la cual estamos ejemplificando. En este caso, se estaría aceptando una solicitud de una droga que ya no existe - por haber sido eliminada - y por ende, se obtendrá una excepción.

Actualizar solicitud: Al actualizar una solicitud a un estado inexistente no se cachea la excepción.

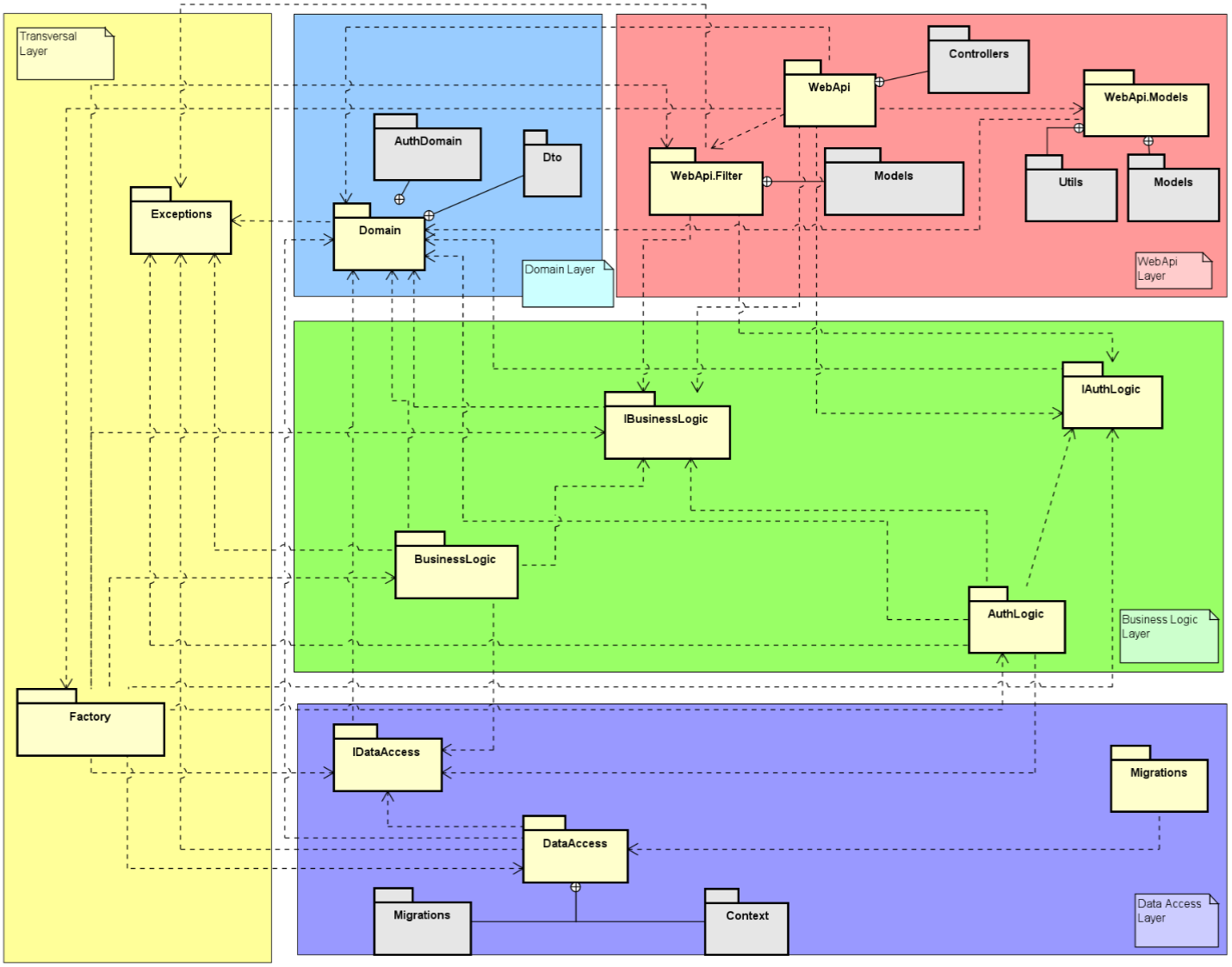
1.3 Diagrama general de Paquetes

En un principio habíamos planteado el siguiente diagrama de paquetes:



El razonamiento que consideramos en su momento fue tener completamente separados y aislados los paquetes relacionados a la lógica del negocio propia de la aplicación (BusinessLogic), de los componentes encargados de validar los permisos del usuario, autorizaciones y sesiones (Authlogic). La idea original surgió con el objetivo de desacoplar totalmente las reglas del negocio de la lógica de permisos, loggeo, etc.
La polémica estaba en la sincronización e integridad entre las dos bases de datos de cada subsistema.

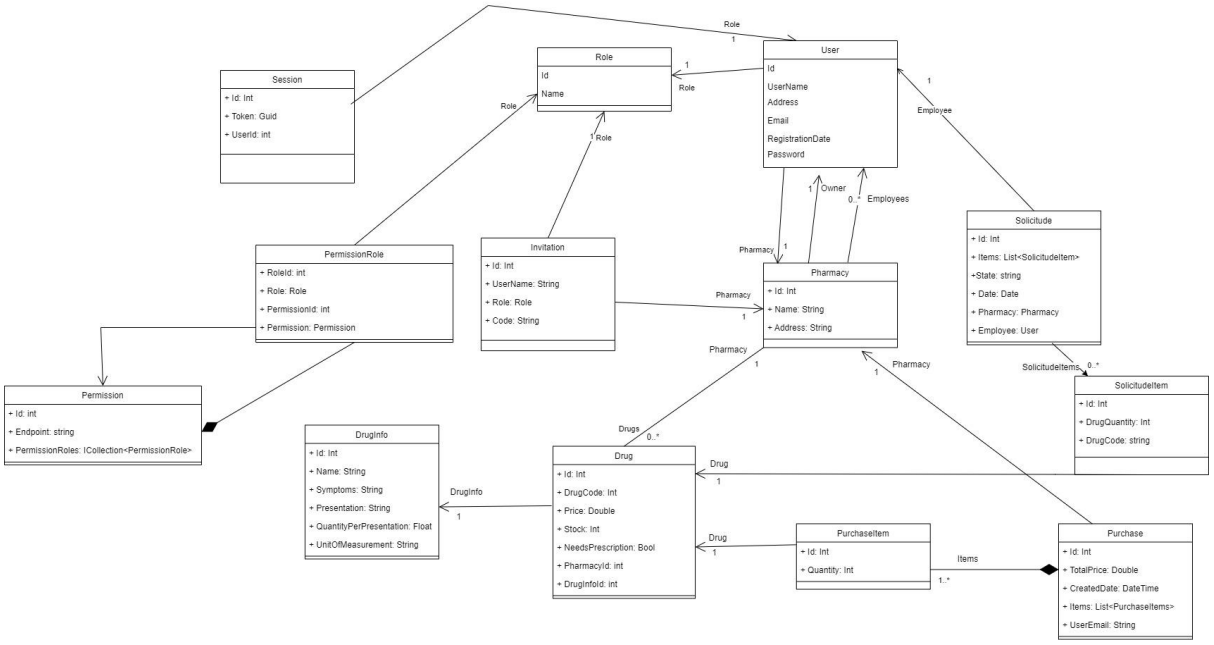
Si bien eso nos convencía, debido al alcance y tiempo de la consigna, decidimos simplificar todo en un mismo paquete y subdividir al paquete por responsabilidades para no tener que mantener a la vez algunas entidades como User en ambas bases de datos. El diagrama final resultante se puede observar a continuación:


Transversal Layer
Exceptions
Factory
AuthDomain
Dto
Domain
Domain Layer
Controllers
WebApi
WebApi.Models
WebApi.Filter
Models
Utils
Models
WebApi Layer
IBusinessLogic
IAuthLogic
BusinessLogic
AuthLogic
Business Logic Layer
IDataAccess
Migrations
DataAccess
Migrations
Context
Data Access Layer

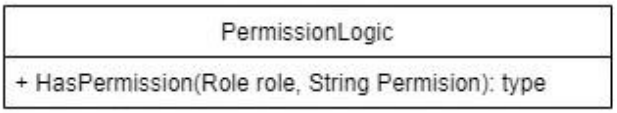
1.3.1 Responsabilidades de paquetes

A continuación se detallan las responsabilidades de los paquetes más trascendentes de nuestro sistema:

Domain

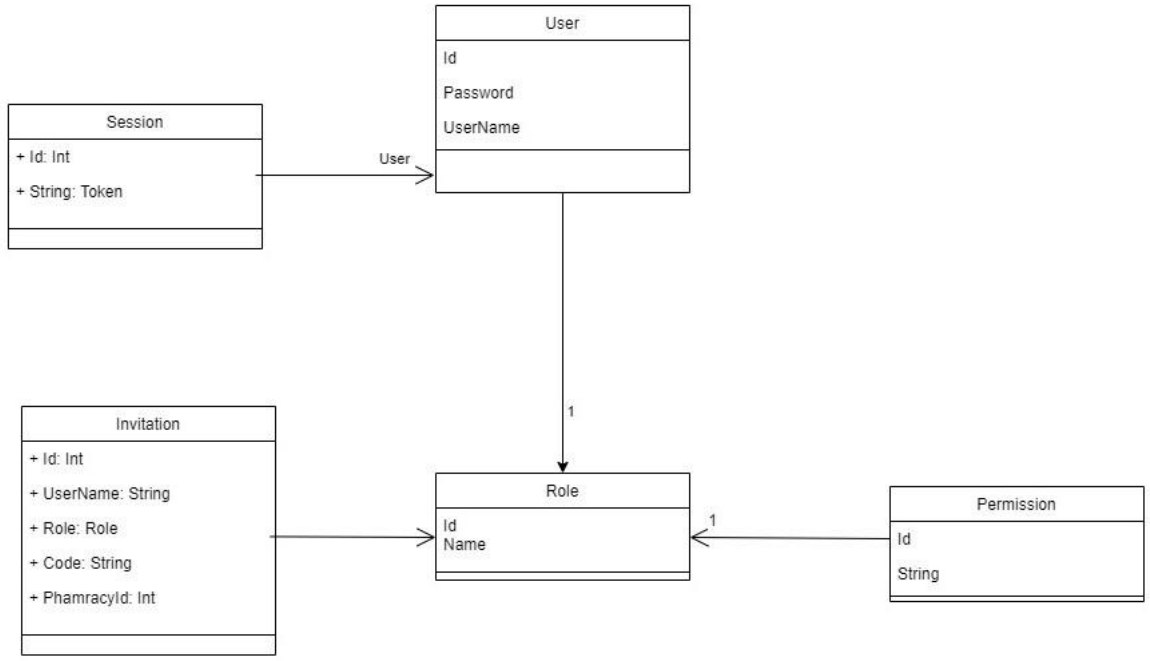


El paquete Domain es el encargado de modelar las entidades necesarias para llevar a cabo el sistema. Es donde se definen las características y comportamiento de las clases más simples. Respetando el principio de responsabilidad única, tanto el paquete en sí como las clases.

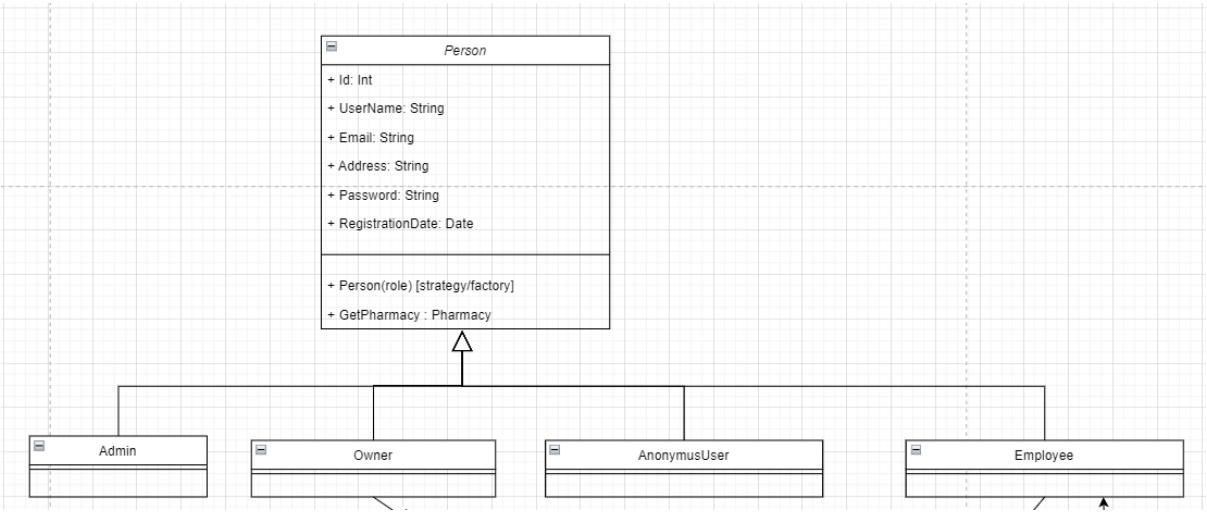


IAuthLogic: Contiene las interfaces de la AuthLogic

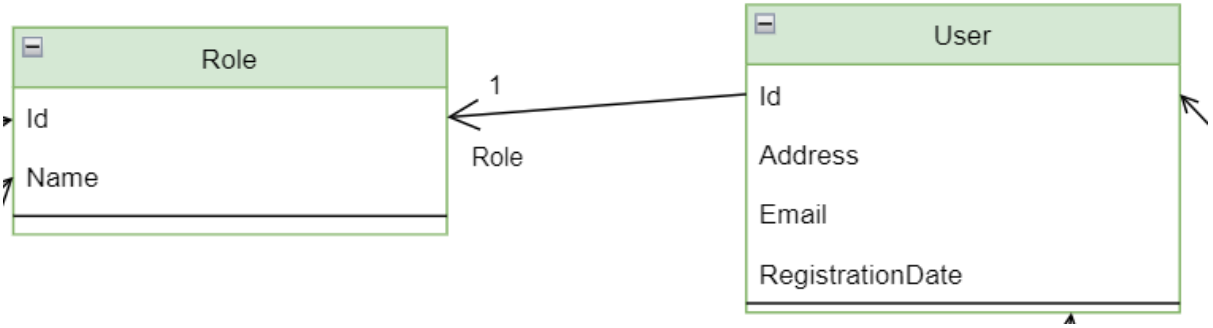
AuthLogic: Se encarga de manejar la lógica de la autenticación de los usuarios y datos de la aplicación.



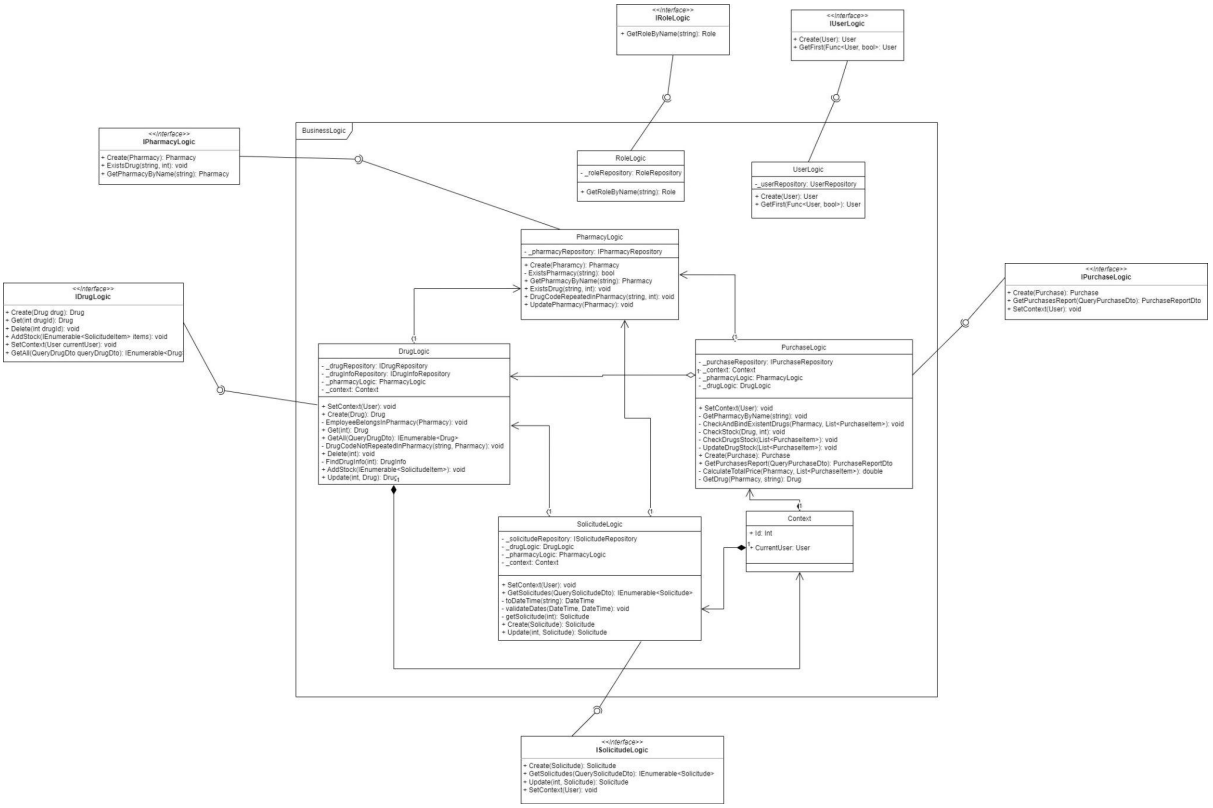
AuthDomain: Se encuentran todas las clases relacionadas con la autenticación de los usuarios. En este sentido, un aspecto que se tuvo en cuenta fue el manejo de los roles, en lo cual discutimos si modelarlo con una herencia donde las subclases compartían la mayoría de properties entre sí, pero no coincidían en comportamiento puesto que cada rol realiza una acción única en el sistema y ningún otro rol lo réplica. En el siguiente diagrama se pueden ver ilustrados los roles:



La idea mencionada anteriormente nos llevó a optar por manejar los roles en una clase Role separada. En la BusinessLogic se controla qué tipo de rol es el cliente y se hacen las validaciones correspondientes.



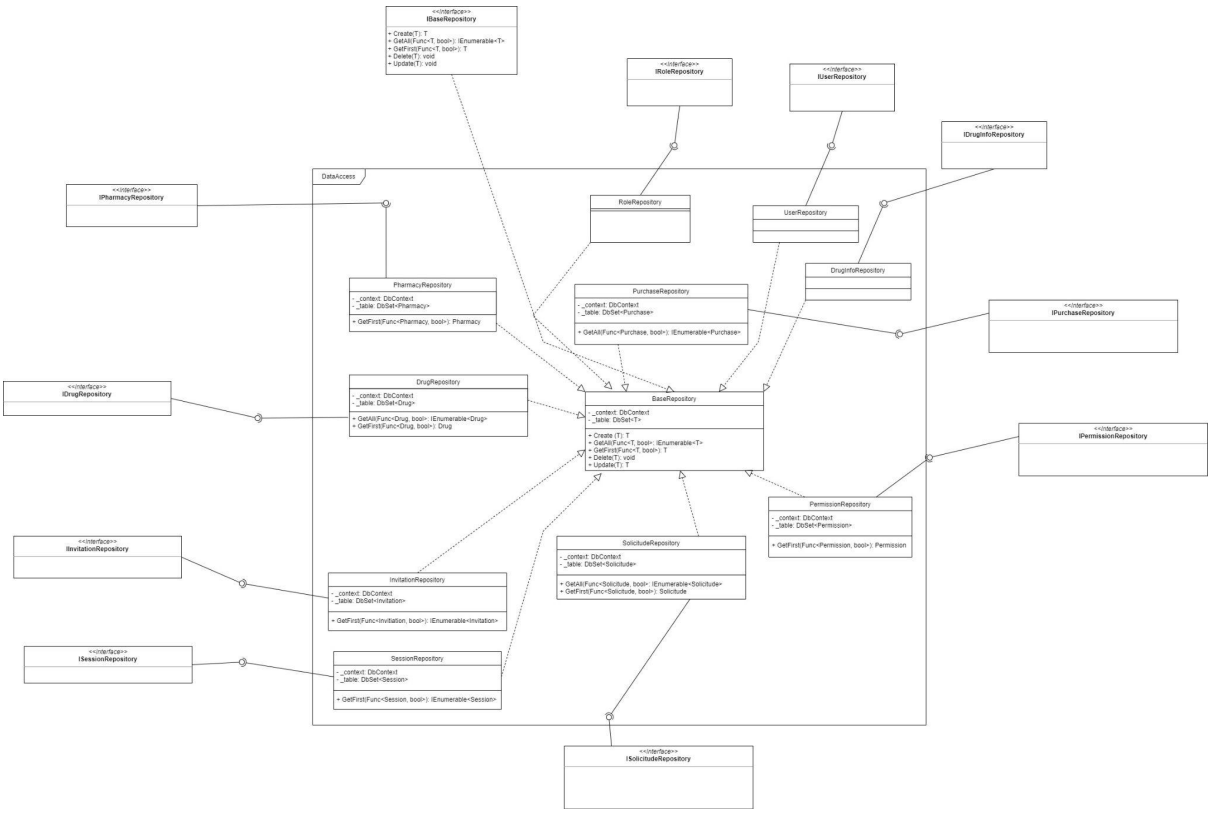
Dto: Aquí se guardan los datos creados del dominio para ciertas cosas puntuales.



BusinessLogic: El paquete BusinessLogic es el responsable de validar y controlar todas las reglas del negocio. Es decir, que todas las operaciones que se hagan en la aplicaciones sigan los criterios establecidos por el cliente. Las clases que se encuentran en el paquete están divididas por feature y son las encargadas de verificar las peticiones que llegan desde la WebApi, cumpliendo así con el principio de responsabilidad única(la validación).

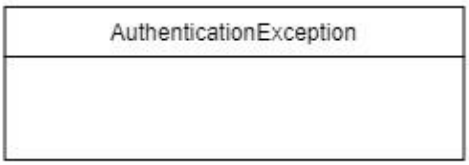
IBussinessLogic: Dicho paquete tiene las interfaces que provee la BusinessLogic.

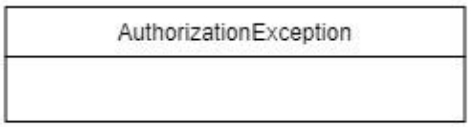
BusinessLogic.Tests: contiene las clases de pruebas de BusinessLogic.

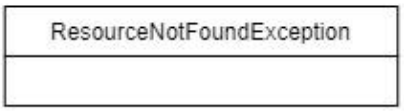


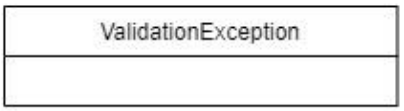
DataAccess: Contiene las clases que interactúan con el contexto de la base de datos. En la subcarpeta Migrations se almacenan las migraciones realizadas sobre la base de datos y en la carpeta Context se maneja el contexto de la base de datos. Todas las clases implementan BaseRepository

IDataAccess: al ser de bajo nivel el DataAccess, se creó una interfaz que los conecta con los Services de alto nivel.



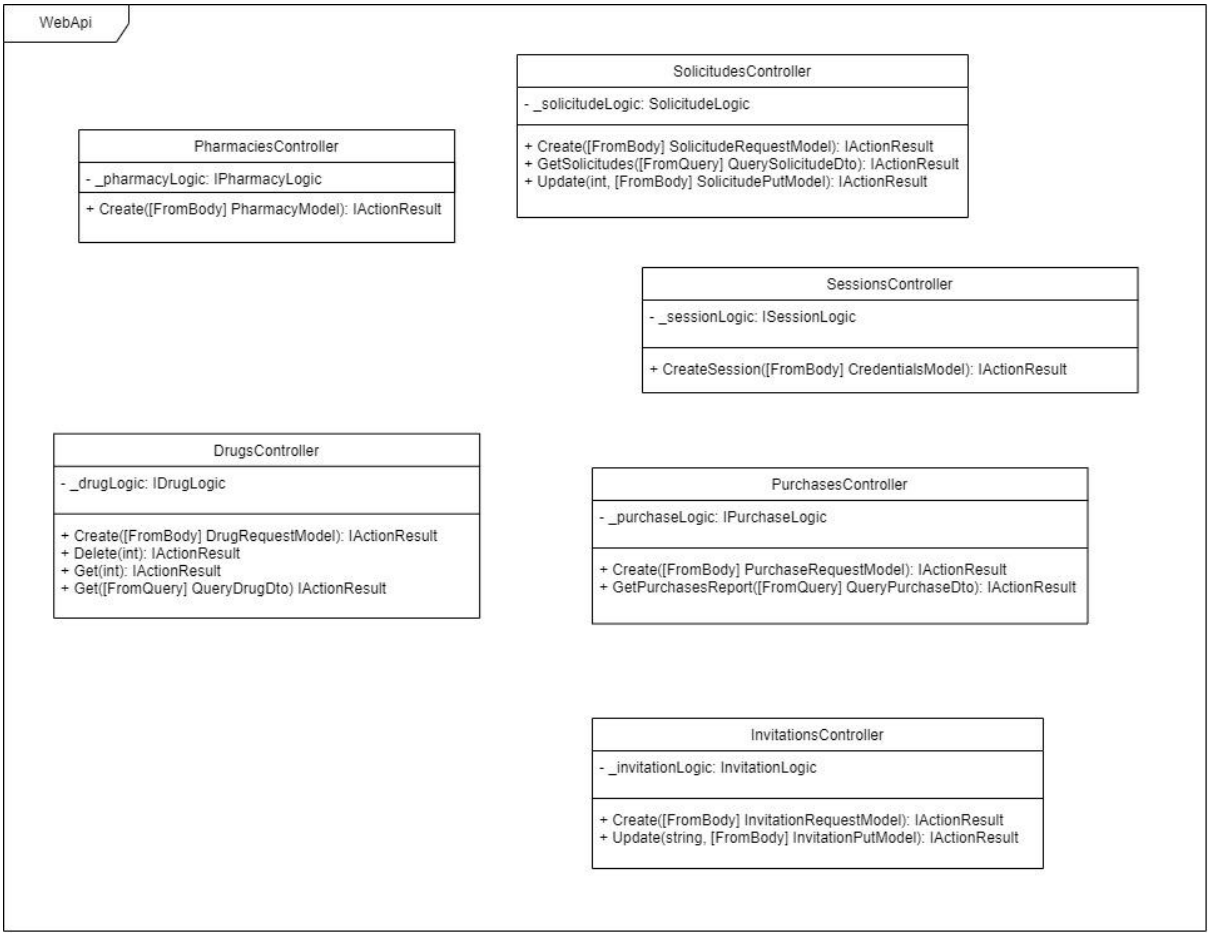






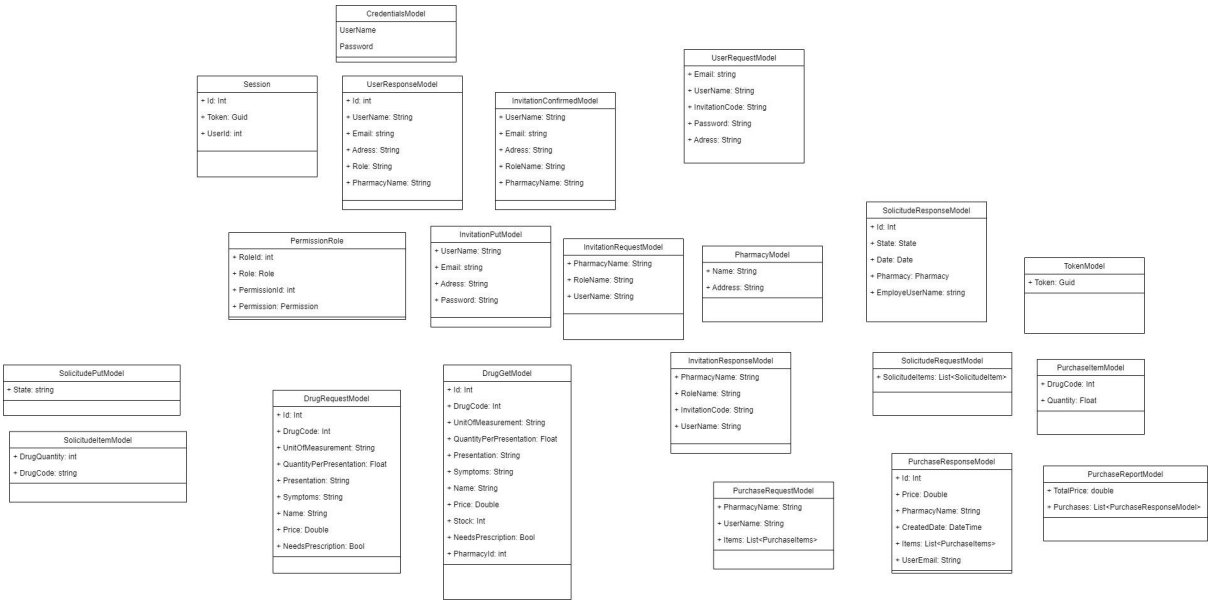
Exceptions: Creamos excepciones propias para manejar nuestras validaciones.

Permissions: Valida que solo los usuarios con los permisos adecuados accedan a los recursos.



WebApi: Es el paquete de más alto nivel, debido a que es quien recibe directamente las peticiones que vienen desde el cliente. Forma la capa de presentación y contiene varios paquetes que contribuyen a su buen funcionamiento.

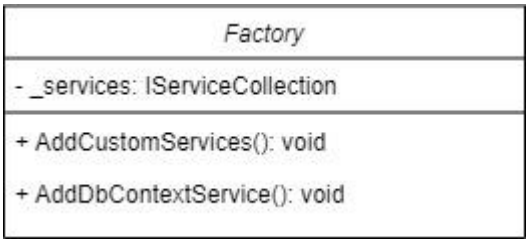
Controller: Aquí se encuentran los controladores de la api, con los endpoints implementados.



WebApi.Models: Aquí, dentro de una subcarpeta llamada models, están los modelos que usa la api para recibir los datos, los cuales después se mapean en nuestros objetos de la

lógica. En la subcarpeta Utils está la clase encargada de mapear los modelos a entidades y viceversa, así como listas y lo que haga falta.

WebApi.Filter: En esta carpeta es donde se mapean las excepciones y errores a códigos de error de HTML, en la subcarpeta models se modelan las excepciones.



Factory: Su única responsabilidad es centralizar la inyección de dependencias en el sistema. En otras palabras, este paquete está formado por una única clase Service Factory que se encarga de crear los servicios, respetando la inyección de las interfaces junto a su implementación.

2. Decisiones de diseño

2.1 Capa de presentación o WebApi

Utilizamos el framework ASP .NET Core para crear una **WebApi** donde se definieron los controllers que atienden las solicitudes REST y esta forma lograr una arquitectura cliente-servidor.

Aplicando el principio de responsabilidad única a nivel del paquete **WebApi** llevamos la responsabilidad de filtros, definición del contenedor de dependencias, definición y mapeo de modelos a otros paquetes y de esta forma **WebApi** queda más descongestionado en responsabilidades.

En el paquete **WebApi.Modules** observamos que definimos un subpaquete Utils el cual nos permite reutilizar funcionalidades de mapeo de entidades y seguir con el principio DRY. En este paquete también se definen los modelos pertenecientes a la capa de presentación desacoplándose de la definición de entidades de dominio para poder tener más control en la respuesta que damos al cliente.

En el paquete **WebApi.Filters** aplicamos el patrón Factory por eso las dependencias tanto a las interfaces de **BusinessLogic** y **AuthLogic** como a sus implementaciones. Esto para definir el contenedor de instancias utilizadas durante la inyección de dependencias.

2.2 Inyección de dependencias

En primer lugar, antes de detallar la implementación de la inyección de dependencias en nuestro proyecto, es inevitable dejar de mencionar el principio de **inversión de control**. El

mismo consiste en eliminar las dependencias de un módulo de un programa con el resto para poder reducir el acoplamiento de ciertos componentes de la solución y hacerla así, más mantenible. De manera tal de lograr cumplir con dicho principio, se utilizó la inyección de dependencias.

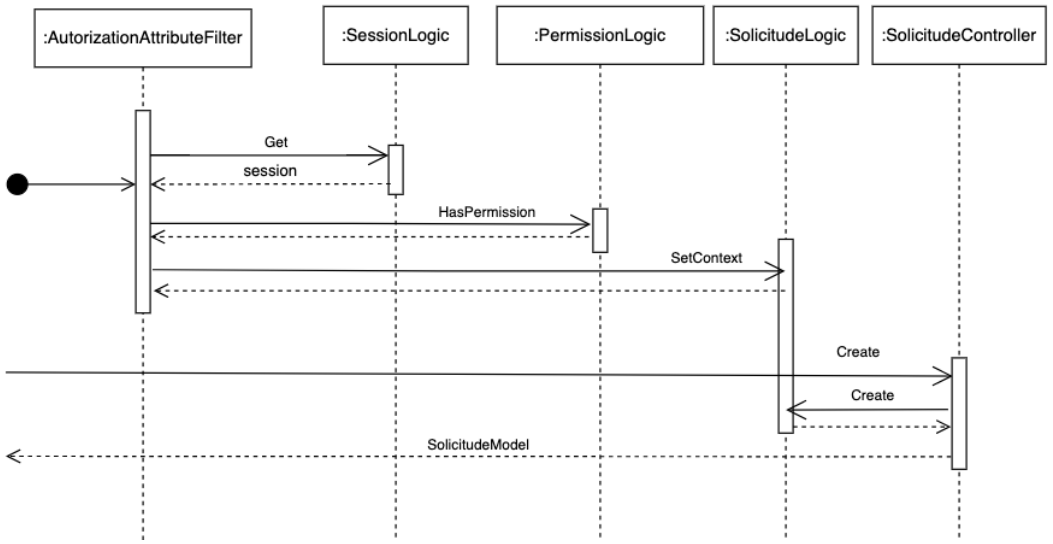
Para su implementación, se creó una clase **ServiceFactory** - dentro de **Factory** - con el objetivo de disminuir la dependencia entre la lógica de negocio y DataAccess. De esta manera, en dicha clase se crearon instancias de las interfaces mediante el método de AddScope tal como se muestra en la siguiente imagen:

```
public void AddCustomServices()
{
    _services.AddScoped<RoleLogic>();
    _services.AddScoped<UserLogic>();

    _services.AddScoped<ISessionLogic, SessionLogic>();
    _services.AddScoped<IPharmacyLogic, PharmacyLogic>();
    _services.AddScoped<IInvitationLogic, InvitationLogic>();
    _services.AddScoped<IDrugLogic, DrugLogic>();
    _services.AddScoped<ISolicitudeLogic, SolicitudeLogic>();
    _services.AddScoped<DrugLogic>();
    _services.AddScoped<PharmacyLogic>();
    _services.AddScoped<IPermissionLogic, PermissionLogic>();

    _services.AddScoped<AuthorizationAttributeFilter>();


    _services.AddScoped<IRoleRepository, RoleRepository>();
    _services.AddScoped<IUserRepository, UserRepository>();

    _services.AddScoped<ISessionRepository, SessionRepository>();
    _services.AddScoped<IPharmacyRepository, PharmacyRepository>();
    _services.AddScoped<IInvitationRepository, InvitationRepository>();
    _services.AddScoped<IDrugRepository, DrugRepository>();
    _services.AddScoped<IDrugInfoRepository, DrugInfoRepository>();
    _services.AddScoped<ISolicitudeRepository, SolicitudeRepository>();
    _services.AddScoped<IPermissionRepository, PermissionRepository>();
}
```

Cada vez que se quiere acceder a una clase de la BussinessLogic o DataAcces se utiliza la abstracción de las mismas en lugar de directamente depender de la implementación lo que hace un sistema abierto a la extensión y cerrado a la modificación, cumpliendo así, a su vez, con el principio de **Open-Close**.

En resumen, haber utilizado la inyección de dependencias, permitió tener un código más modular y menos acoplado logrando un sistema extensible y robusto, dependiendo de las interfaces y no directamente de las implementaciones.

En el siguiente diagrama de interacción se muestra como un mensaje entrante pasa por filtros de autorización antes de invocar el un método del controller y luego la lógica de negocio correspondiente. La orquestación de invocar al filtro y luego al controller la lleva adelante el framework de WebApi.



2.3 Lógica de negocio

El paquete **BusinessLogic** está compuesto de varias clases que resuelven las funcionalidades de nuestro sistema. Tuvimos en cuenta el principio de responsabilidad única para crear clases con funcionalidades sobre cada recurso. Cada clase es un proveedor de una interfaz definida en **IBusinessLogic** favoreciendo así el principio de segregación de interfaces y la alta cohesión.

La abstracción que nos brinda el paquete **IBusinessLogic** nos permite definir un contrato entre **WebApi** y **BussinessLogic**, logrando un código fácil de testear ya que podemos crear mocks de manera sencilla en nuestros tests unitarios de los controller. De todas formas podríamos no contar con esta abracción y aún así estaríamos cumpliendo con el

principio DIP ya que **WebApi** es un paquete de tecnología (bajo nivel) que puede depender del negocio (alto nivel) sin violar dicho principio.

2.4 Entidades de dominio

En el paquete **Domain** ubicamos las entidades de dominio fueron las primeras que diseñamos para poder modelar la realidad del sistema. También se encuentran definidas DTO's que fueron surgiendo a partir de la necesidad de definir atributos que necesitábamos recibir o retornar en la lógica de negocio diferentes a las entidades de dominio.

Como observamos en el diagrama de paquetes final, el paquete Domain quedó como dependiente de la mayoría de paquetes. Aquí identificamos una oportunidad de mejora para poder aumentar el bajo acoplamiento definiendo aún más DTO's, Object Values o POJOS para la comunicación entre WebApi y BusinessLogic
En cuanto a las validaciones de los atributos de las entidades de dominio, como formatos y otras restricciones vimos conveniente seguir el principio de experto de información y ubicarlas en estas clases. También para reuso de funciones de validación surgió la creación de una clase utilitaria **FormatValidator**.

Otras validaciones como nombres ya existentes en la base de datos decidimos ubicarlas en BusinessLogic ya que conlleva una interacción con la capa de acceso a datos.

2.5 Capa de acceso a datos

Para lograr una persistencia hacia una base de datos relacional decidimos utilizar la estrategia Code First mediante el uso del ORM Entity Framework. Esto nos permitió mapear entidades de dominio a tablas de base de datos de una forma automática sin la necesidad de definir un modelo de tablas. Aquí nos vimos atados al framework ya que en cada entidad de dominio teníamos que agregar un atributo Id correspondiente a la clave foránea de la entidad con relación de asociación. También al momento de crear una relación n:n entre roles y permisos nos vimos obligados a crear una tabla intermedia. Por ello identificamos una oportunidad de mejora ya que podríamos definir dentro del paquete **DataAccess** los modelos a persistir en la base de datos y las correspondientes funciones de mapeo al igual que definimos en **WebApi** y de esta forma desacoplaríamos impactos de cambio del framework sobre el paquete de Dominio respetando aún más DIP.

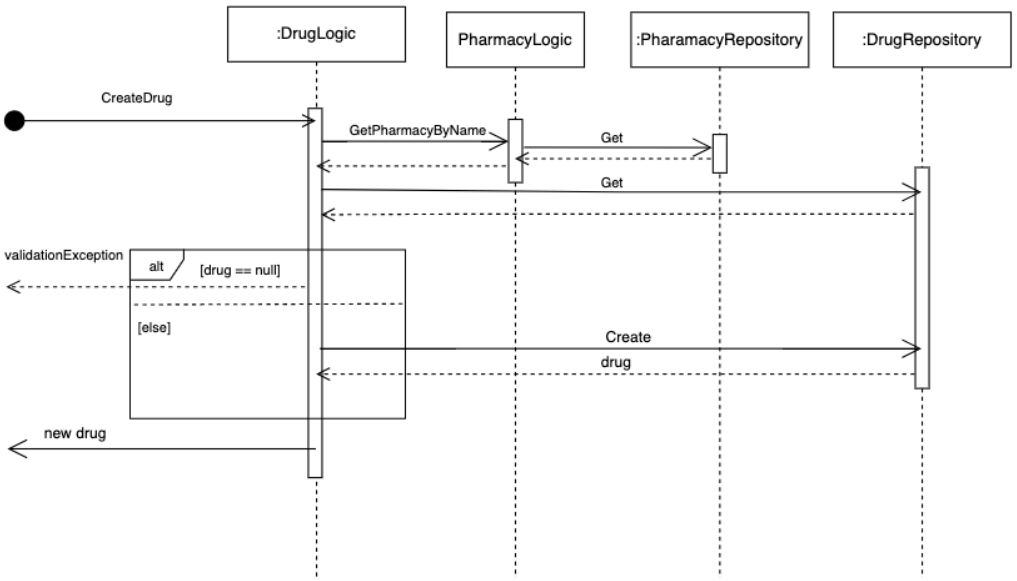
Siguiendo con el principio DIP se creó el paquete **IDataAccess** que se ubica entre **BusinessLogic** y las clases concretas de **DataAccess**. De esta forma cortamos el impacto de cambio de las clases de tecnología hacia las de lógica de negocio. Y damos la posibilidad a futuro de implementar otras formas de persistencia de datos sin impactar el las capas de alto nivel.

A medida que fuimos construyendo los repositorios de cada entidad nos percatamos que los métodos CRUD eran funcionalidades en común a reusar. De ahí creamos la clase

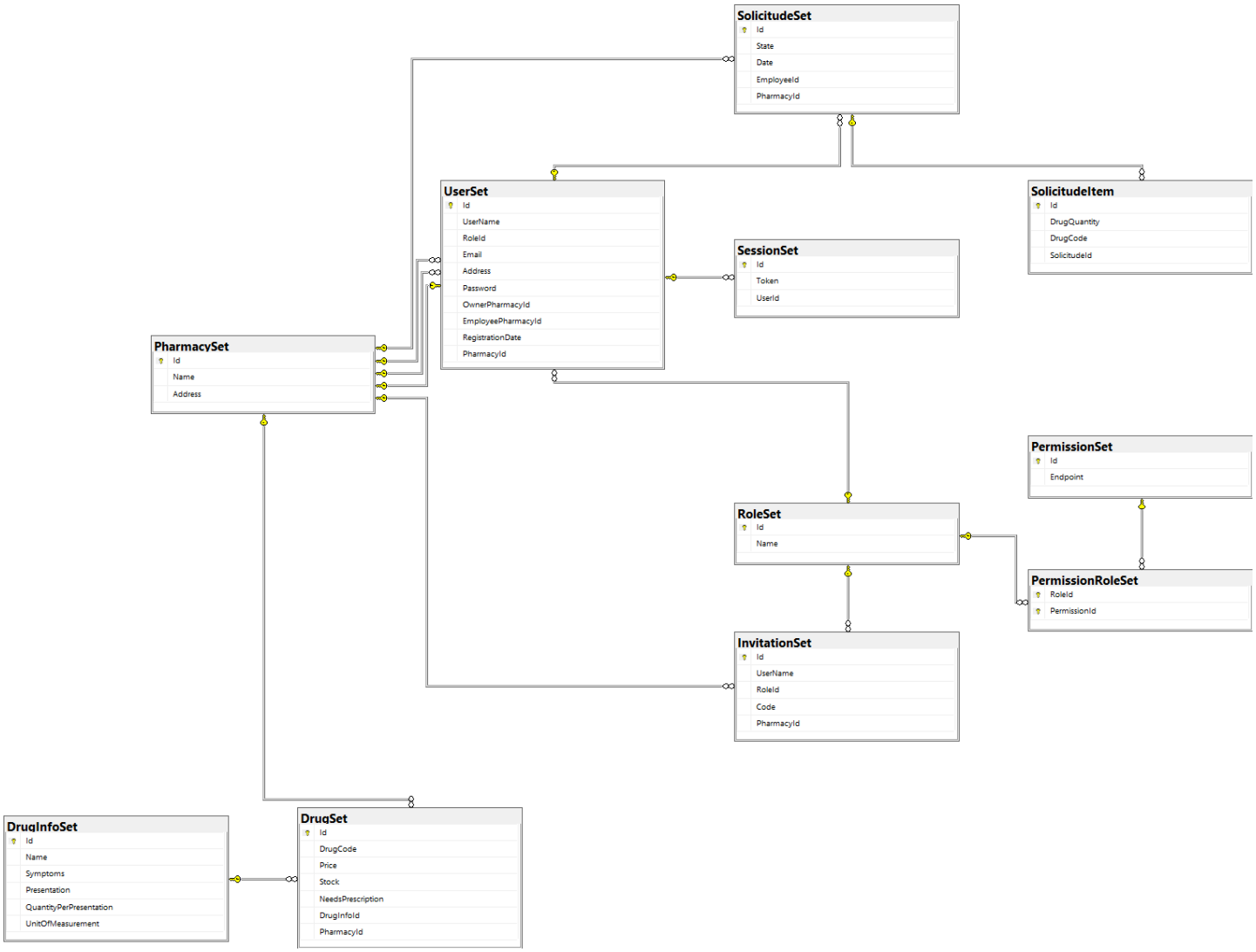
BaseRepository que implementa el patrón repositorio genérico. De esta forma logramos ahorrar código duplicado y tests unitarios.

Para la obtención de datos en base decidimos utilizar el mecanismo de Eager Loading lo que nos llevó a sobreescribir algunos métodos Get y GetAll del repositorio base para poder realizar Joins con otras tablas y recuperar entidades dependientes. Aquí identificamos una oportunidad de mejora ya que podríamos implementar el patrón template method para sobreescribir en cada repositorio específico las entidades dependientes que queremos traer.

El siguiente diagrama muestra la interacción de las clases de lógica negocio entre sí y los correspondientes repositorios de cada entidad:
Diagramas de interacción:



3. Modelo de tablas de la estructura de la base de datos



Este diagrama representa como modelamos nuestra base de datos, junto con sus relaciones y atributos. Decidimos tener una tabla por cada entidad grande de nuestro dominio, mapeando relaciones como Foreign Keys y dependencias entre ellos.

3.1 Descripción del mecanismo de acceso a datos utilizado

Tablas:

SolicitudeSet: Aquí se almacenan las solicitudes para los medicamentos, tiene Id, State, Data, EmployeeId y PharmacyId. Tanto EmployeeId como PharmacyId son Foreign Keys de las tablas UserSet y PharmacySet.

UserSet: Aqui se guardan todos los usuarios que se registran en el sistema, sus atributos son Id, UserName, RoleId, Email, Address, Password, OwnerPharmacyId,

EmployeePharmacyId, RegistrationDate y PharmacyId. RoleId es Foreign Key de RoleSet, PharmacyId dePharmacySet, y segun si es empleado o owner, EmployeePharmacyId o OwnerPharmacyId va a ser activo, mientras que el otro será nulo, ambas son Foreign Key de PharmacyId.

SolicitudeItem: Aquí se guardan las solicitudes. Tienen Id, DrugQuantity, DrugCode, SolicitudeId. SolicitudeId es Foreign Key de Solicitude.

RoleSet: Aquí se guardan los roles de los usuarios. Tiene Id y Name.

PermissionSet: Aquí se guardan los permisos asociados a los endpoints de la API. Tiene Id y Endpoint.

PermissionRoleSet: Aquí se guarda la relación entre los roles y los permisos, se compone de RoleId, Foreign Key de RoleSet, y PermissionId, Foreign Key de PermissionSet.

InvitationSet: Aquí se guardan las invitaciones para los usuarios de la aplicación, para que puedan loguearse al sistema. Tiene Id, UserName, RoleId, Code y PharmacyId. RoleId es Foreign Key de RoleSet y PharmacyId es Foreign Key de PharmacySet.

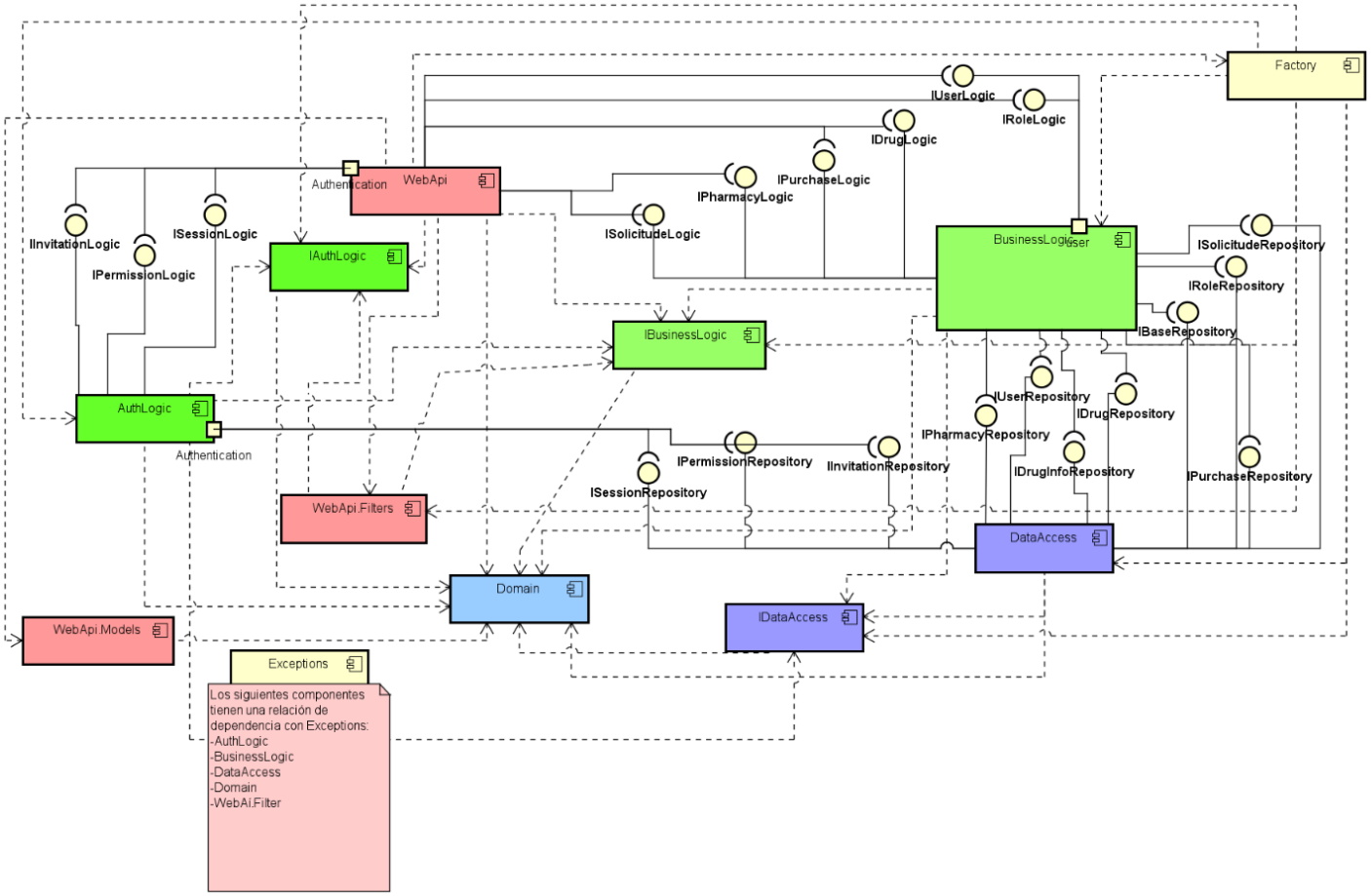
PharmacySet: Aquí se guardan las farmacias del sistema. Tiene Id, Name y Address.

DrugSet: Aquí se guardan los medicamentos de la farmacia. Tiene Id, DrugCode, Price, Stock, NeedsPrescription, DrugInfoId y PharmacyId. DrugInfoId es Foreign Key de DrugInfoSet y PharmacyId es Foreign Key de Pharmacy.

DrugInfoSet: Aquí se guarda la información del medicamento. Tiene Id, Name, Symptoms, Presentation, QuantityPerPresentation y UnitOfMeasurement.

4. Diagrama de componentes

Mediante el diagrama de componentes(externo), podemos representar la vista de implementación. Aca es donde se muestra el sistema desde la perspectiva de un desarrollador y está enfocada en los diferentes artefactos y/o componentes del software. Este diagrama deja en evidencia el cumplimiento del principio de inversión de dependencias. Se puede apreciar como los módulos de alto nivel no dependen de los de bajo nivel, sino que ambos dependen de abstracciones. Esto se ve reflejado a través de las Interfaces que se proveen que también sirven como fachada entre ambos módulos.

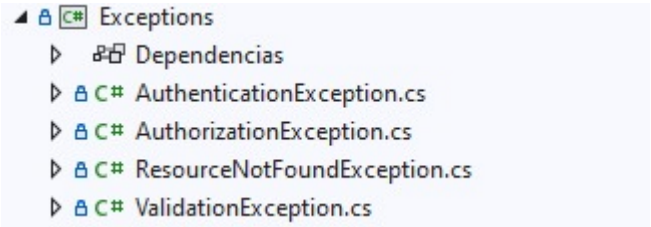


5. Manejo de excepciones

Se crearon las excepciones dentro del sistema de forma tal de identificar todos los casos de error y aportar un mensaje explicativo para el usuario sobre los mismos, según surjan. Las excepciones de cada una de las clases heredan de Exceptions, siendo este un proyecto separado para poder respetar el principio de **responsabilidad única,** teniendo en cuenta que son una cantidad importante de paquetes los que dependen de él.

Para controlar las excepciones desde la API, los filters hacen de intermediario. El modelo WebApi.Filters es el encargado de retornar al cliente un mensaje de error. El mismo contiene una clase ExceptionFilter, encargada de cachear las diferentes excepciones del sistema y mapearlas al modelo creado para retornar en la API.

Como podemos visualizar en la siguiente imagen, existen cuatro tipo de excepciones que se están manejando:



Authentication: Relacionado al loggeo del usuario con el header de auth

Authorization: Errores relacionados a los permisos y roles que tienen acceso a las diferentes operaciones.

ResourceNotFound: Cuando se intenta acceder a un elemento que no existe o dejó de existir en el sistema

Validation: Errores de validación, formato o reglas del negocio mismo.