Universidad ORT Uruguay

Facultad de Ingeniería

Descripción del diseño

Ignacio Loureiro - 191659
Malvina Jaume - 151281

Entregado como requisito de la materia
Diseño de Aplicaciones 2

25 de junio de 2020

Índice general

1. Introducción

El siguiente documento ha sido realizado para describir el diseño realizado para solución de la implementación del API REST descripta en la letra del obligatorio entregado.

1.1. Objetivos del documento

- Descripción general del trabajo
- Construcción de diagramas
- Descripción de jerarquías
- Modelado de entidades
- Modelado de estructura de datos
- Justificación del diseño
- Diagramas de implementación

2. Descripción general del trabajo

La Intendencia Municipal de Montevideo (IMM) desea desarrollar una aplicación que permita que los ciudadanos informen sobre diferentes situaciones que ocurren en la ciudad, permitiendo que la intendencia pueda reaccionar rápidamente. Dicha aplicación se llamara **IMMRequest**.

Las situaciones detectadas por los ciudadanos serán informadas a través de solicitudes las cuales obligatoriamente tienen un detalle, nombre del solicitante, correo electrónico y opcionalmente un teléfono, además de otros datos dependientes de su área, tema y tipo de solicitud. Esto significa que cada solicitud esta relacionada a un área de un tema en particular y dentro de ese tema un tipo. Lo importante a tener en cuenta aquí es que tipo de solicitud brinda la capacidad de que la solicitud pueda tener otros campos adicionales.

Una vez ingresada la solicitud el usuario obtiene un número de solicitud, el cual permitirá realizar el seguimiento de la misma. No puede existir áreas con el mismo nombre en el sistema, lo mismo para los temas de un área y tipos dentro de un tema.

Un tipo debe tener nombre, debe estar asociado a un tema (de un área) y puede contener una serie de campos adicionales que son los que agregará a las solicitudes que se asocien a ese tipo. Cada campo adicional tiene un nombre, el tipo y opcionalmente un rango.

En esta primera entrega el alcance establecido corresponde a la implementación de una API REST que ofrezca todas las operaciones necesarias para soportar la aplicación anteriormente descripta. Por ende, se trabajó en backend de la aplicación (lógica de negocio). Como la aplicación no tiene interfaz de usuario en esta instancia, se realizó la interacción con la misma mediante el cliente HTTP Postman.

2.1. Especificación de Requerimiento

Luego de realizarse un análisis inicial se detectaron los requerimientos y actores involucrados en el sistema, que se detallaron en el documento que adjunto *DA2_Requerimientos.pdf*

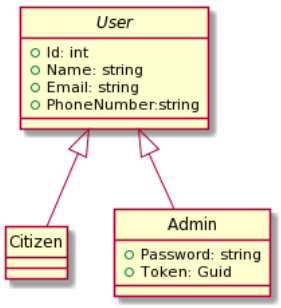
2.2. Arquitectura definida para la solución

Artefacto	Descripción
BD	Base de datos relacional (Sql Server) en donde se almacenan los datos de la aplicación.
Api REST	Toda interacción con el repositorio de datos se realiza mediante un Api REST, la que ofrece operaciones para resolver todo lo necesario y donde se implementa la lógica de negocio.
Web Front End	Aplicación que permite a los usuarios registrarse e interactuar con la aplicación.

- El Web Front End es la aplicación que permite a los ciudadanos y administradores registrarse e interactuar con el sistema de IMMRequest.
- La base de datos relacional SQL Server es en donde se almacenan los datos de la aplicación. Nuestra base de datos se llama *IMMRequestDB*.
- La API REST realiza toda la interacción con el repositorio de datos. Ofrece operaciones para resolver todo lo necesario. Allí se implementa la lógica de negocios.

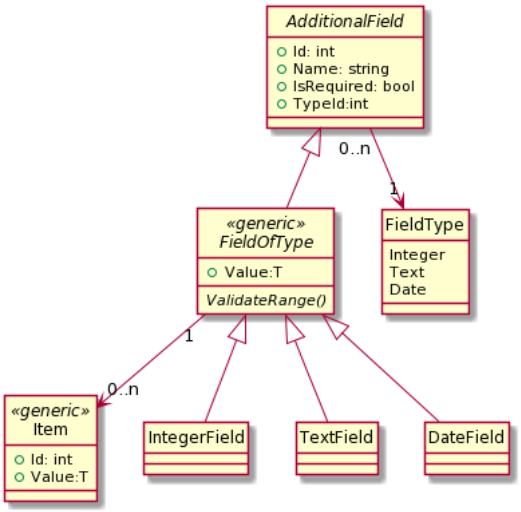
2.2.1. Justificación de la solución y diseño

El modelado de los usuarios del sistema se considera de la siguiente manera. Todos los Usuarios del sistema van a ser ó Administradores ó Ciudadanos que quieren hacer llegar sus pedidos la Intendencia.



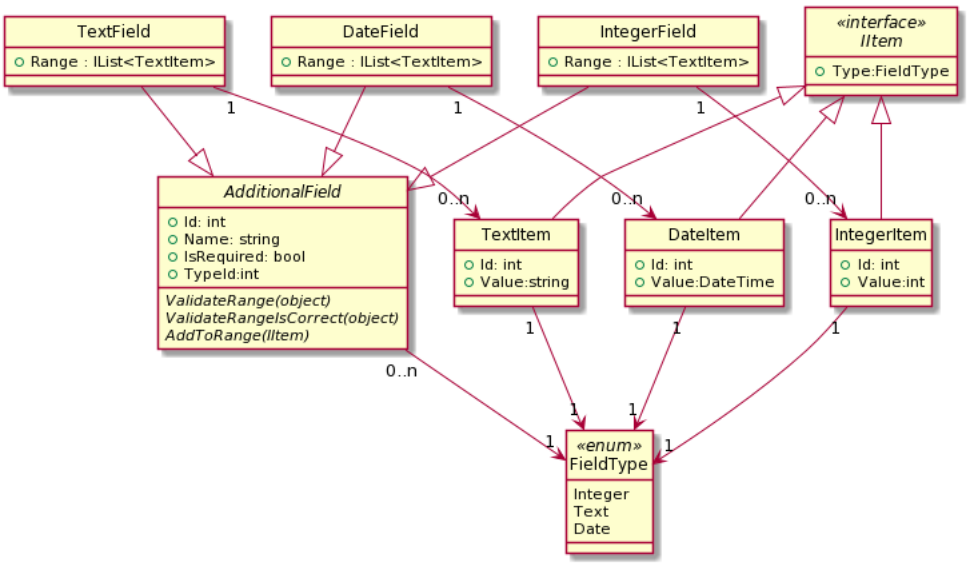
De esta manera, al tener una clase User abstracta, los usuarios, efectivamente serán insancias de Administradores ó Ciudadanos.

Por otra parte, para resolver el manejo de los campos adicionales en los tipos, teníamos, en un principio, la siguiente jerarquía.



Si bien éste diseño es elegante, se presentó un problema al intentar persistir esta jerarquía en la base de datos manejada por Entity Framework.

Para poder resolver el problema, la jerarquía quedó resuelta de ésta manera. Tiene un punto en contra muy grande y es que el Rango queda supeditados a las clases que extienden la base, y por lo tanto se hace más engorroso de manipular.



El token es la forma en la que se autenticará al usuario. Cada usuario podrá iniciar sesión a través del login con su nombre de usuario (mail) y contraseña. En caso de que el login sea correcto se devolverá un token, que en nuestro caso será un número aleatorio. Este token debe pasarse en los valores del header de todas las request.

2.2.2. Patrones utilizados

Para el diseño del sistema se utilizaron los siguientes patrones de diseño:

2.2.2.1. Repository

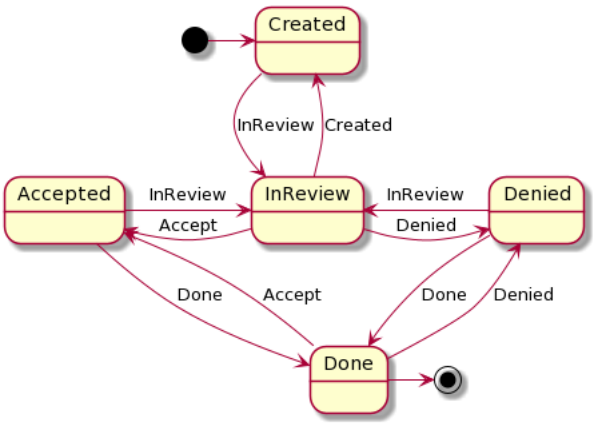
El patrón Repository nos permite tener una abstracción de la implementación de acceso de datos. De esta manera nuestra lógica de negocio no conoce ni está acoplada a la fuente de datos. Nuestro repository actúa como un intermediario entre nuestra lógica de negocio y la de acceso de datos.

2.2.2.2. Inyección de Dependencias

El objetivo de uso de este patrón es extraer la responsabilidad a un componente para delegarlas en otros, estableciendo un mecanismo a través del cual el nuevo componente pueda ser cambiado en tiempo de ejecución.

2.2.2.3. State

Este patrón es utilizado para que un objeto se comporte de forma diferente dependiendo el estado interno en el que se encuentre en cada momento.
Este patrón lo utilizamos para modelar los diferentes estados que van a tener las solicitudes en el sistema. La secuencia de los estados que se manejan serían los siguientes:



2.2.3. Ingreso de datos

Para poder tener una base de datos pronta para ser usada creamos una clase **IMMRequestDBSeeder**, que se encarga de "seedear"la base de datos, con datos hardcodeados en la clase. Esto sirve para tener un juego de datos con el que se pueda probar la aplicación sin tener que estar agregándolos a mano. Esta clase reside en IMMRequest.DataAccess.Core y se inyecta con un ServiceScopeFactory al contexto de la base de datos.

2.2.4. Repositorio de datos

Para realizar el un control de las diferentes versiones del código creamos un repositorio Git. Este nos permite tener una copia local del historial del desarrollo y los

cambios que se van propagando entre los repositorios locales. Además este repositorio nos permite realizar una gestión eficiente del proyecto que se está desarrollando dada la rapidez de gestión de diferencias entre archivos.

Creamos un repositorio llamado **IMMRequest_191659_151281**[1] en donde en la carpeta **Codigo** dejamos todo lo relacionado al desarrollo de la aplicación. Decidimos tener una rama **develop** en donde está la solución final, y con la cual se trabajará para la siguiente entrega. Las ramas de requerimientos y de fixes fueron creadas para que tanto los dos integrantes del grupo desarrollaron sobre ellas para después realizar un "pull request", en caso de ser necesario.

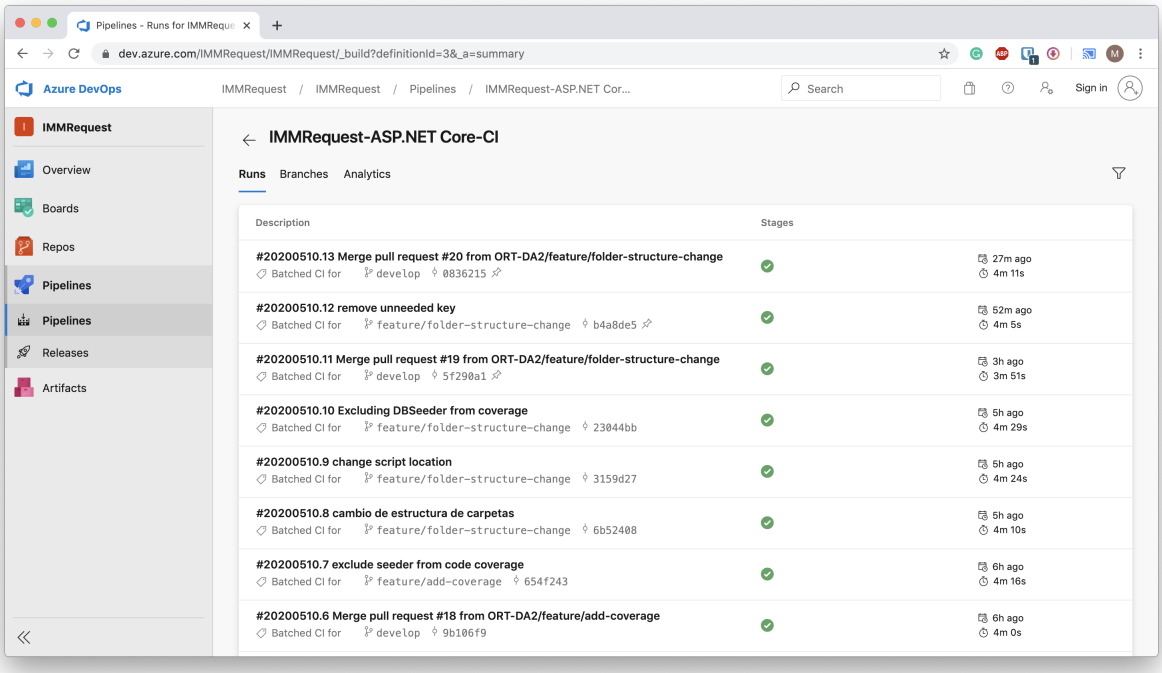
Tuvimos ciertos inconvenientes en *commits* por lo que el historial se puede ver confuso, se identificó sobre el final del desarrollo que algunos test que se habían realizado no estaban y se volvieron a agregar.

Algunos de los branch creados para requerimientos fueron borrados después de realizarse el merge con develop, luego nos enteremos que no era necesario realizar eso y por eso no aparecerán todos en la lista en git.

2.2.5. Desarrollo guiado por pruebas (TDD)

Al comienzo del desarrollo se realizó TDD para las clases del dominio que fuimos identificando, luego incorporamos el desarrollo con Moq para cuando comenzamos con la lógica. Posteriormente las pruebas se realizaron luego de tener las funcionalidades completas lo que nos resulto más ágil al momento del desarrollo. Para tener una constante ejecución de las pruebas creamos en **Azure DevOps**[2] un pipeline que se ejecuta automáticamente al engancharse a los hooks de GitHub para cada push o pull request. Además nos dio la posibilidad de tener esta validación al momento de crear el Pull Request, si las pruebas no pasaban este no se podía realizar.

[1] https://github.com/ORT-DA2/IMMRequest_191659_151281
[2] https://dev.azure.com/IMMRequest/IMMRequest/_build



2.2.6. Paquetes utilizados

Para realizar los diagramas de clases, interacción y paquetes utilizamos el Lenguaje Unificado de Modelado (UML), mediante el cual se realizó una descripción del proyecto a crear. En nuestra implementación del diseño, agrupamos las clases necesarias para el desarrollo del programa en diferentes paquetes: IMMRequest.DataAccess, IMMRequest.DataAccess.Core, IMMRequest.DataAccess.Test, IMMRequest.Domain, IMMRequest.Domain.Exceptions, IMMRequest.Domain.Tests, IMMRequest.Logic, IMMRequest.Logic.Core, IMMRequest.Logic.Tests y IMMRequest.WebApi.

Para la estructura del proyecto tomamos como ejemplo el proyecto Moodle visto en clase.

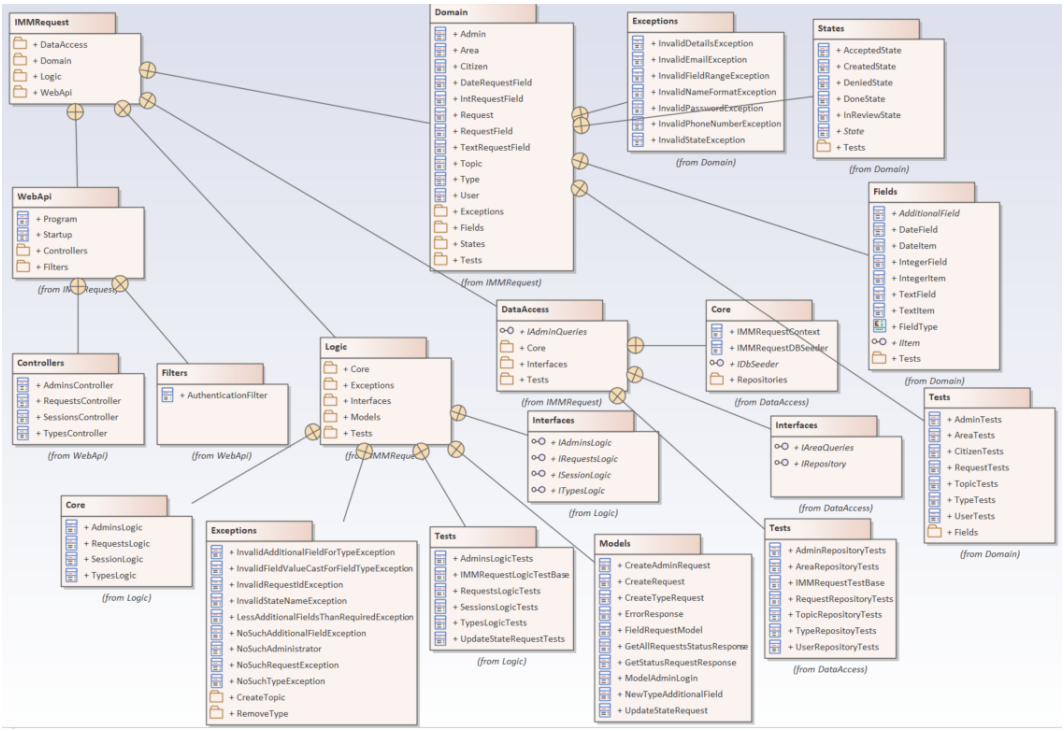
2.2.7. Errores conocidos

Cuando se finalizó la implementación se detectaron defectos que será tenidos en cuenta para corregir cuando se realice la próxima etapa:

- No se puede realizar una solicitud (request) con el mismo usuario (mail)
- Se pueden agregar más de un Tipo con el mismo nombre para una misma área. Esto no estaba especificada por requerimiento, pero debe manejarse como mejora.
- Al momento de crear un tipo nuevo, pueden agregarse varios campos adicionales con el mismo nombre, generando una redundancia de datos.

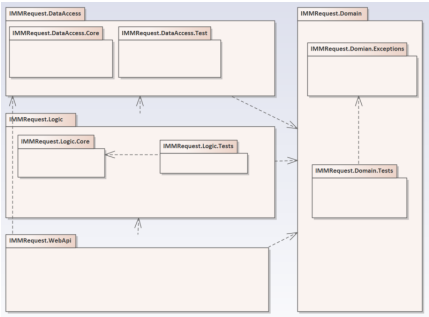
3. Diagrama de namespaces

Para la solución planteada quedaron definidos varios namespaces que están relacionados según el siguiente diagrama:

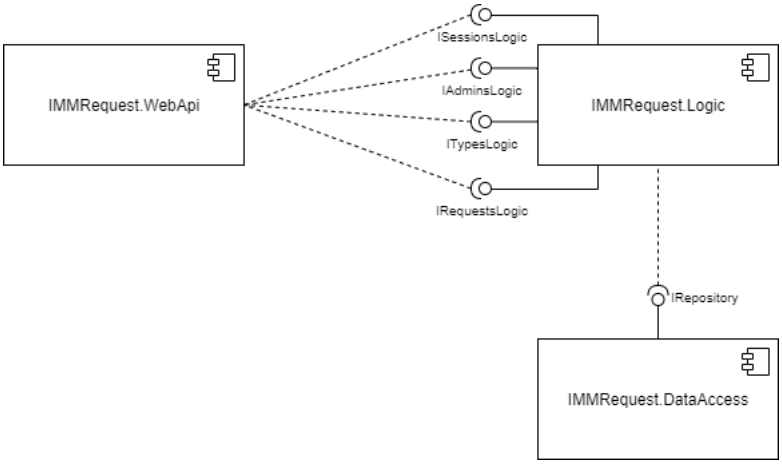


4. Diagrama general de paquetes

En la siguiente imagen detallamos la relación de los paquetes principales que se verán en la solución, y su dependencias.



Se puede notar que separamos los paquetes en función de su lógica, permitiéndonos realizar modificaciones de interfaz, código o pruebas de manera que las mismas funcionen de forma independiente, que que los otros paquetes se vean afectados.

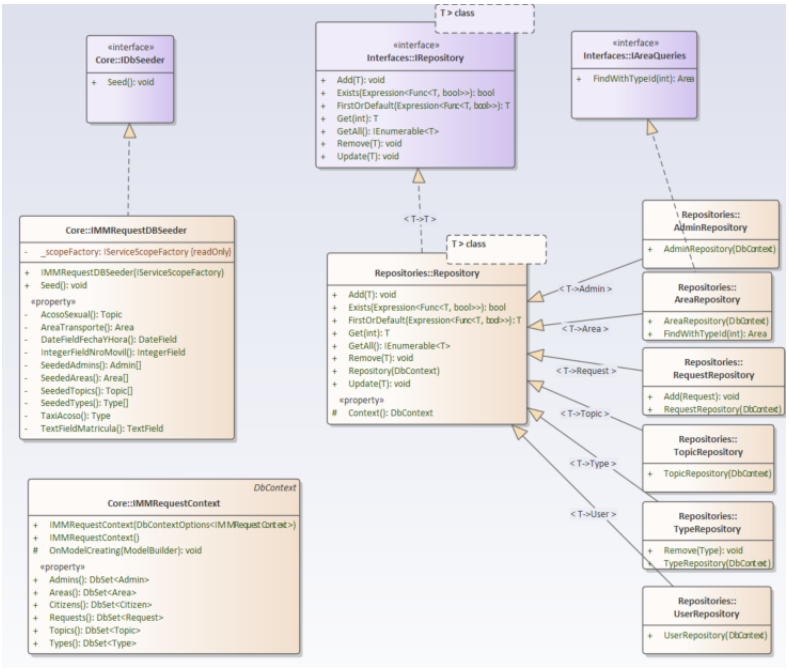


Los paquetes principales que se encuentran en nuestra solución serían:

4.1. Paquete *DataAccess*

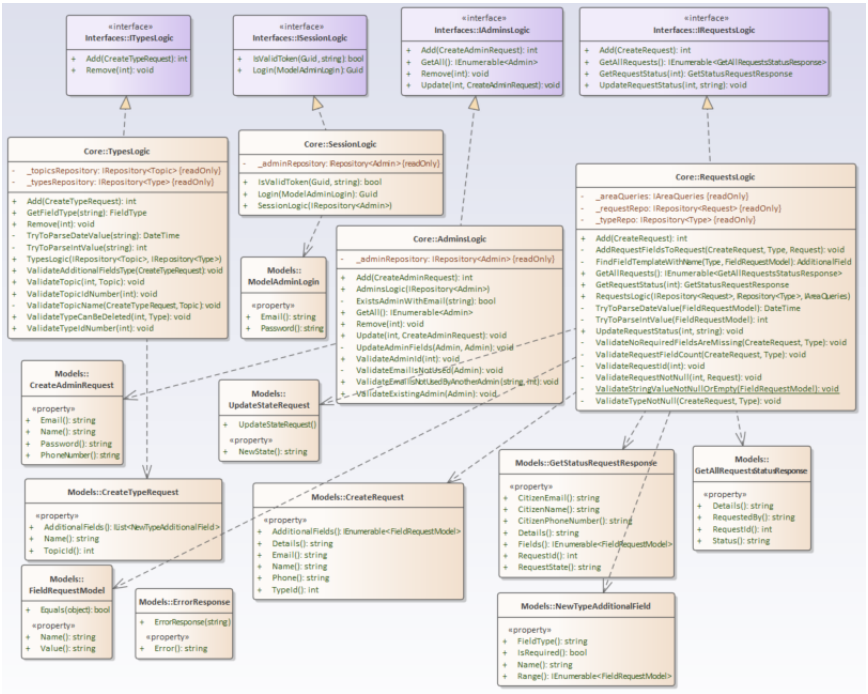
Este paquete tiene la responsabilidad del manejo de la conexión y manejo de datos en la base de datos. Brinda la operaciones necesarias para el manejo de la

persistencia necesaria para la solución. A travez de la interfaz IRepository quedan expuestas todas las funciones necesarias para el manejo de los datos



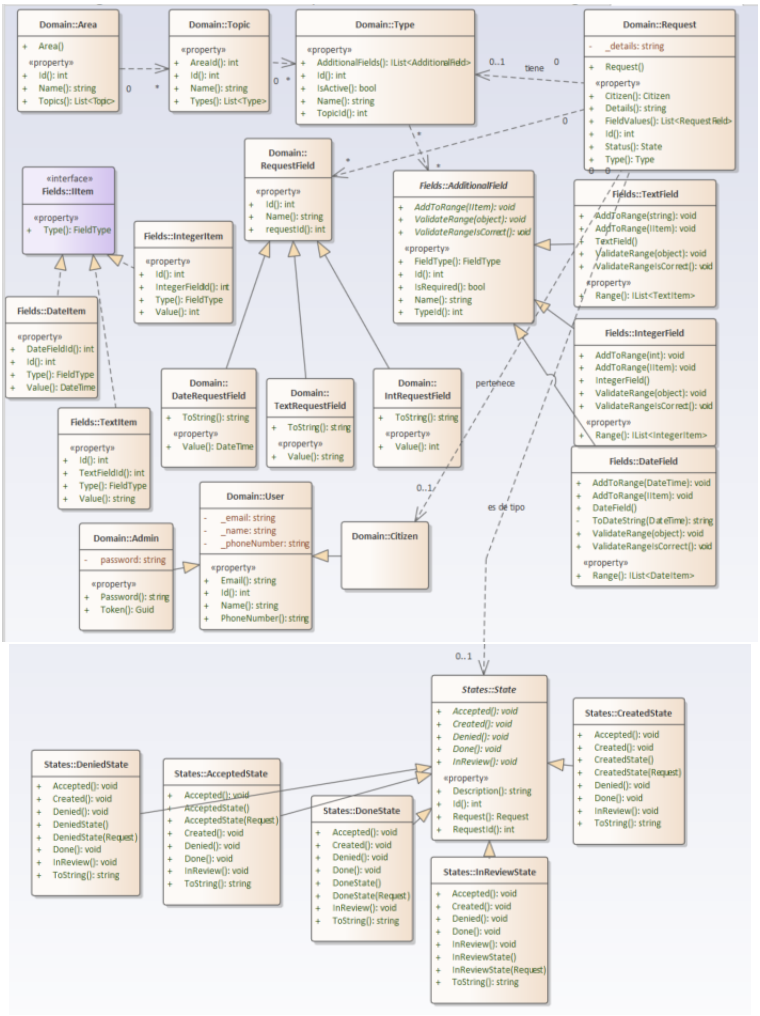
4.2. Paquete *Logic*

Este paquete tiene la responsabilidad del manejo de toda la lógica para el cumplimiento de los flujos que se deben cumplir según los requerimientos definidos. En este paquete también se manejan las Excepciones y clases de pruebas. En el diagrama de dominio se detallan las clases principales.



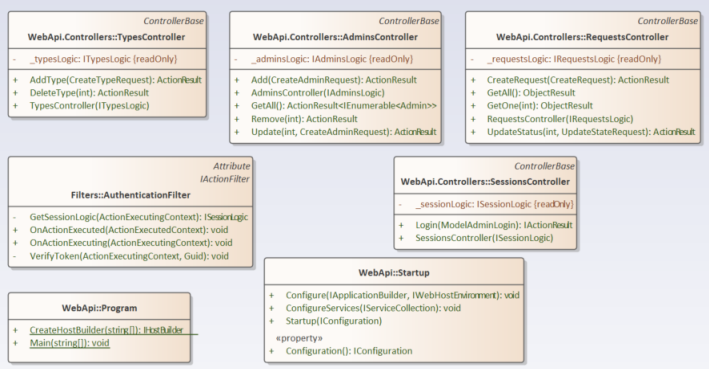
4.3. Paquete *Domain*

Este paquete tiene la responsabilidad del manejo de todas las entidades necesarias que modelan la solución. En el diagrama de dominio se pueden ver las entidades relevantes. En este paquete también se manejan las Excepciones y clases de pruebas.

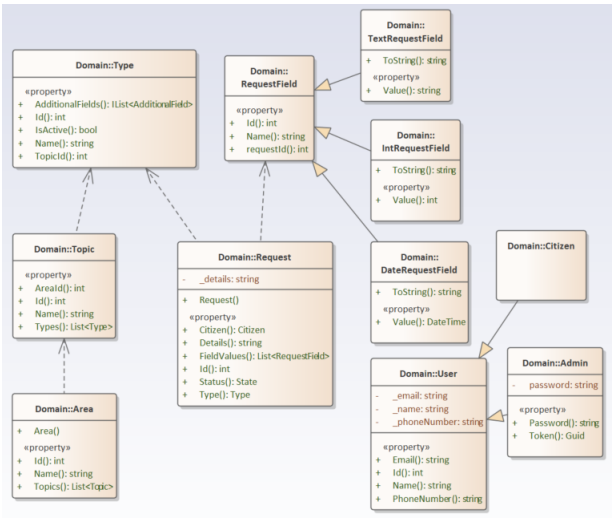


4.4. Paquete *WebApi*

Esta paquete tiene la responsabilidad del manejo de las solicitudes a los distintos endpoint de la aplicación. Consume la lógica a través de una atracción.

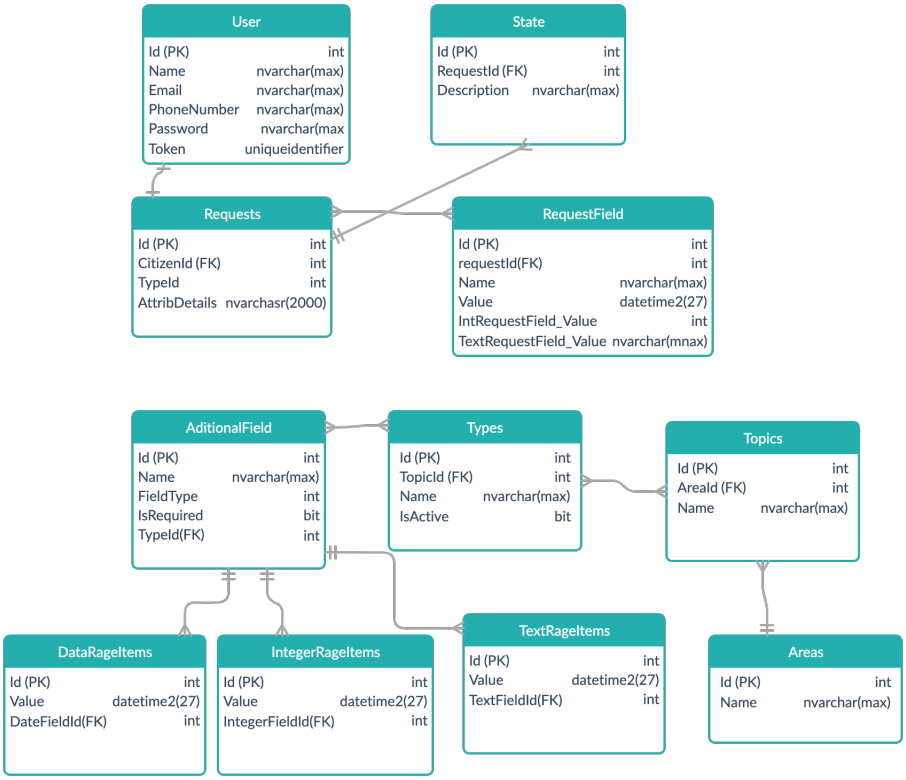


4.5. Modelo de entidades

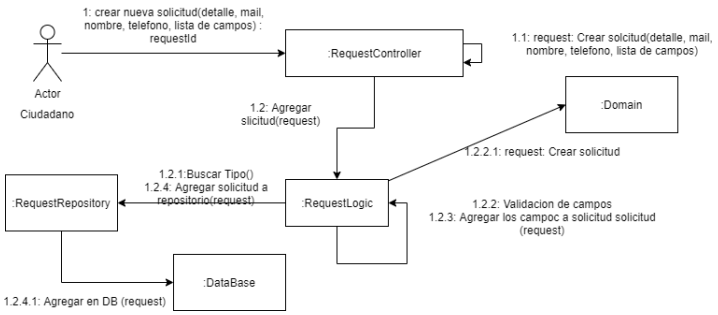


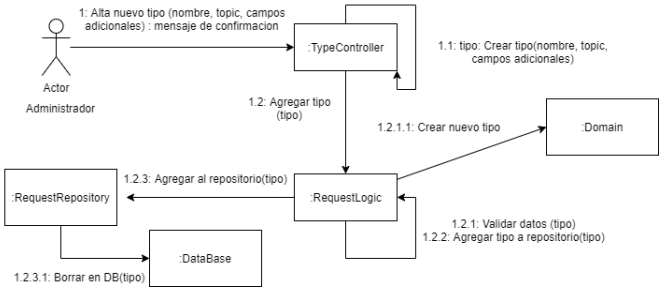
4.6. Modelo de tablas de la estructura de la base de datos

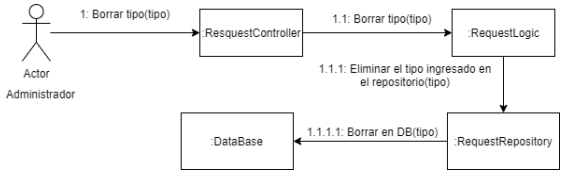
En esta versión de la aplicación realizamos la persistencia de la información en una base de datos. Para esta implementación se utilizó el motor de base de daos Microsoft SQL Server 2017. El siguiente modelo refleja la relación de las tablas de la misma.

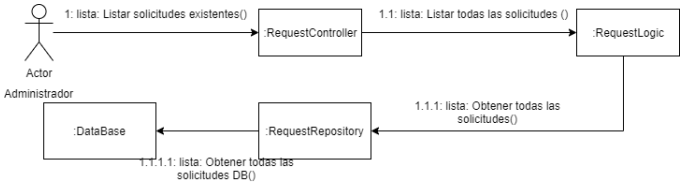


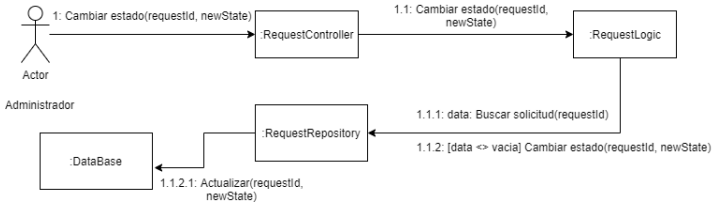
4.7. Diagramas de colaboración



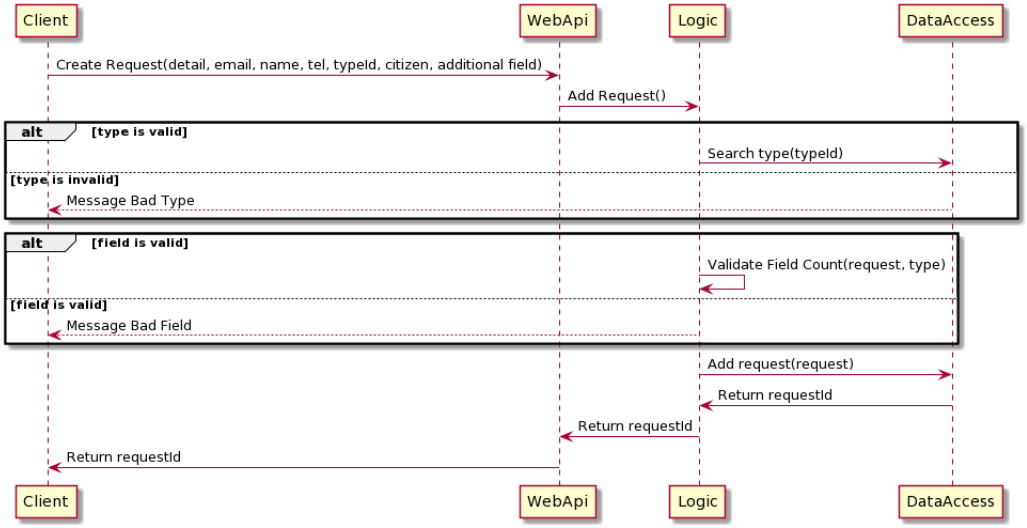


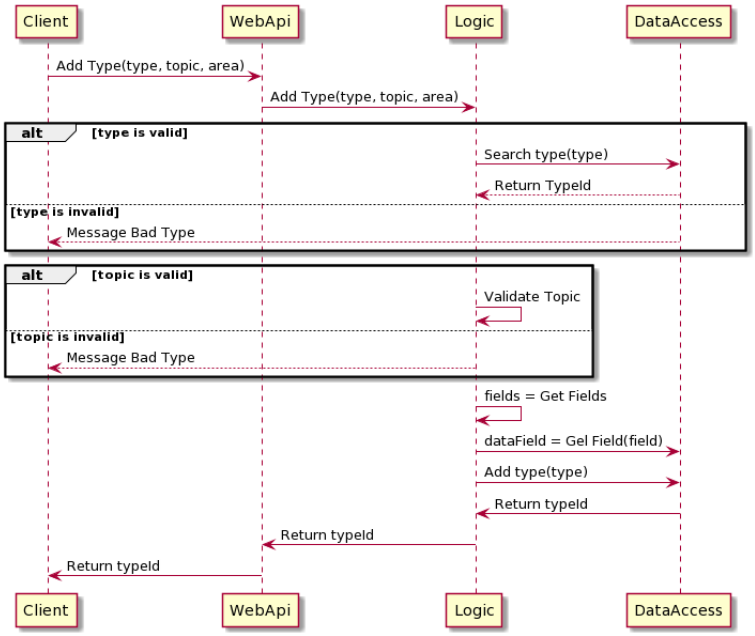

1: Borrar tipo(tipo)
:ResquestController
1.1: Borrar tipo(tipo)
:RequestLogic
Actor
Administrador
1.1.1: Eliminar el tipo ingresado en el repositorio(tipo)
:DataBase
1.1.1.1: Borrar en DB(tipo)
:RequestRepository

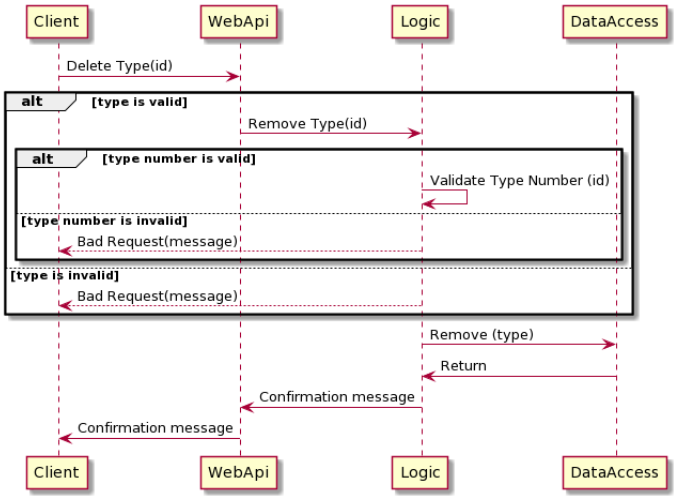

1: lista: Listar solicitudes existentes()
:RequestController
1.1: lista: Listar todas las solicitudes ()
:RequestLogic
Actor
Administrador
1.1.1: lista: Obtener todas las solicitudes()
:DataBase
:RequestRepository
1.1.1.1: lista: Obtener todas las solicitudes DB()

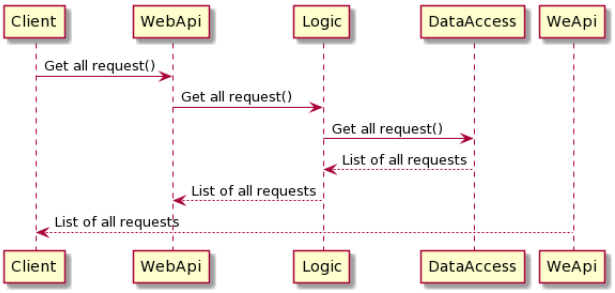

1: Cambiar estado(requestId, newState)
:RequestController
1.1: Cambiar estado(requestId, newState)
:RequestLogic
Actor
Administrador
1.1.1: data: Buscar solicitud(requestId)
:RequestRepository
1.1.2: [data <> vacia] Cambiar estado(requestId, newState)
:DataBase
1.1.2.1: Actualizar(requestId, newState)

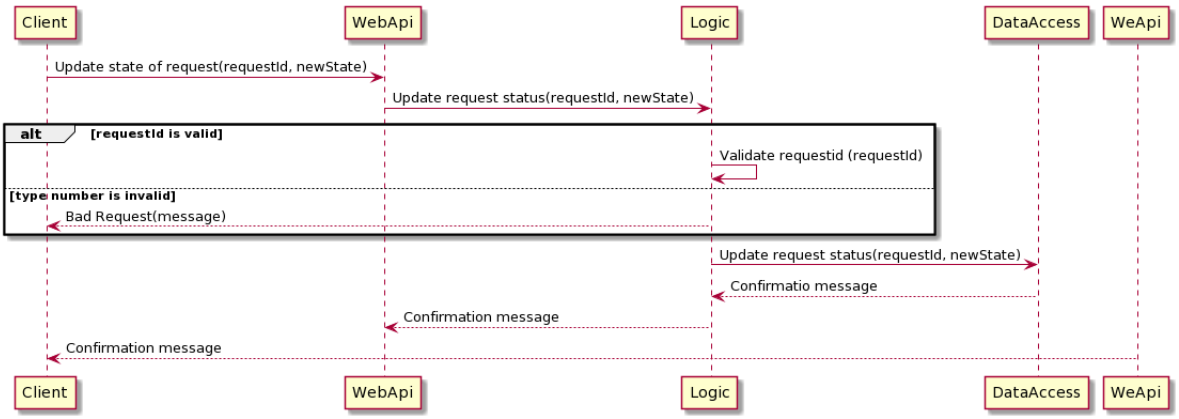
5. Diagramas de secuencia



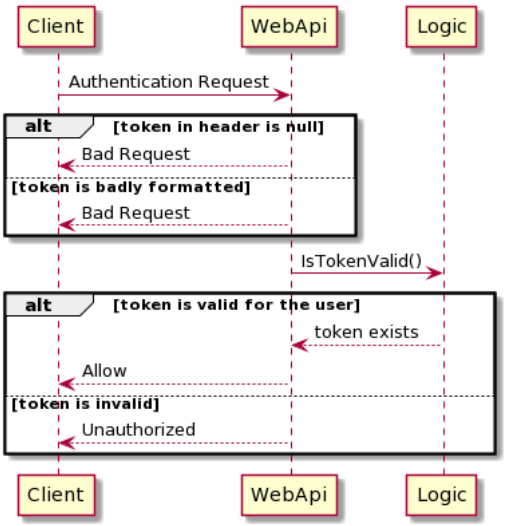








Otro diagrama de secuencia que consideramos importante es el correspondiente al filtro que realizamos para la autenticación.


Client
WebApi
Logic
Authentication Request
alt
[token in header is null]
Bad Request
[token is badly formatted]
Bad Request
IsTokenValid()
alt
[token is valid for the user]
token exists
Allow
[token is invalid]
Unauthorized
Client
WebApi
Logic

6. Manejo de Excepciones

Para el manejo de excepciones creamos dento de los paquetes Domain y Logic, un paquete de Exception. Realizamos esta implementación para tener el manejo de las excepciones centralizadas en un solo lugar e independiente de resto del desarrollo. Definimos un tipo de excepción dependiendo de error que queremos transmitir.