Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2
Obligatorio 1:
Justificación del Diseño

Juliette Ruchel - 203942
Francisco Martinez - 233126

Docente: `Ignacio Valle`

Entregado como requisito de la materia Diseño de Aplicaciones 2

https://github.com/ORT-DA2/203942-233126

14 de mayo de 2020

Declaraciones de autoría

Nosotros, Juliette Ruchel y Francisco Martinez, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos Diseño de aplicaciones 2;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Resumen

El presente documento tiene el propósito de exponer nuestra justificación del diseño de nuestro sistema para el problema presentado.

Índice general

1. Descripción del Sistema

1.1. Funcionalidades del Sistema

El sistema es una API que permite ofrece un servicio a dos tipos de usuarios diferentes. Por un lado, permite que ciudadanos realizar solicitudes a la intendencia sin necesidad de registrarse. A través de la API podrán saber el estado de su solicitud.
Por otro lado, permite que los trabajadores de la intendencia manejen estas solicitudes de manera eficiente, pudiendo cambiar el estado de las mismas y crear nuevos campos para solicitudes mas especificas. Finalmente, también tiene la capacidad de hacer un seguimiento de quienes tienen acceso al sistema, pudiendo crear, modificar e eliminar usuarios administradores.

1.2. Aclaraciones

- Para poder probar la API libremente con Postman, por favor desactivar en la configuración del mismo la verificación de certificados SSL.
- El único bug encontrado por el equipo actualmente, es que se permite crear una solicitud aunque no estén llenos todos los campos adicionales de un tipo dado. Se espera corregir en la siguiente entrega.

2. Diseño

2.1. Estructura de Paquetes

2.1.1. Dependencias de Paquetes

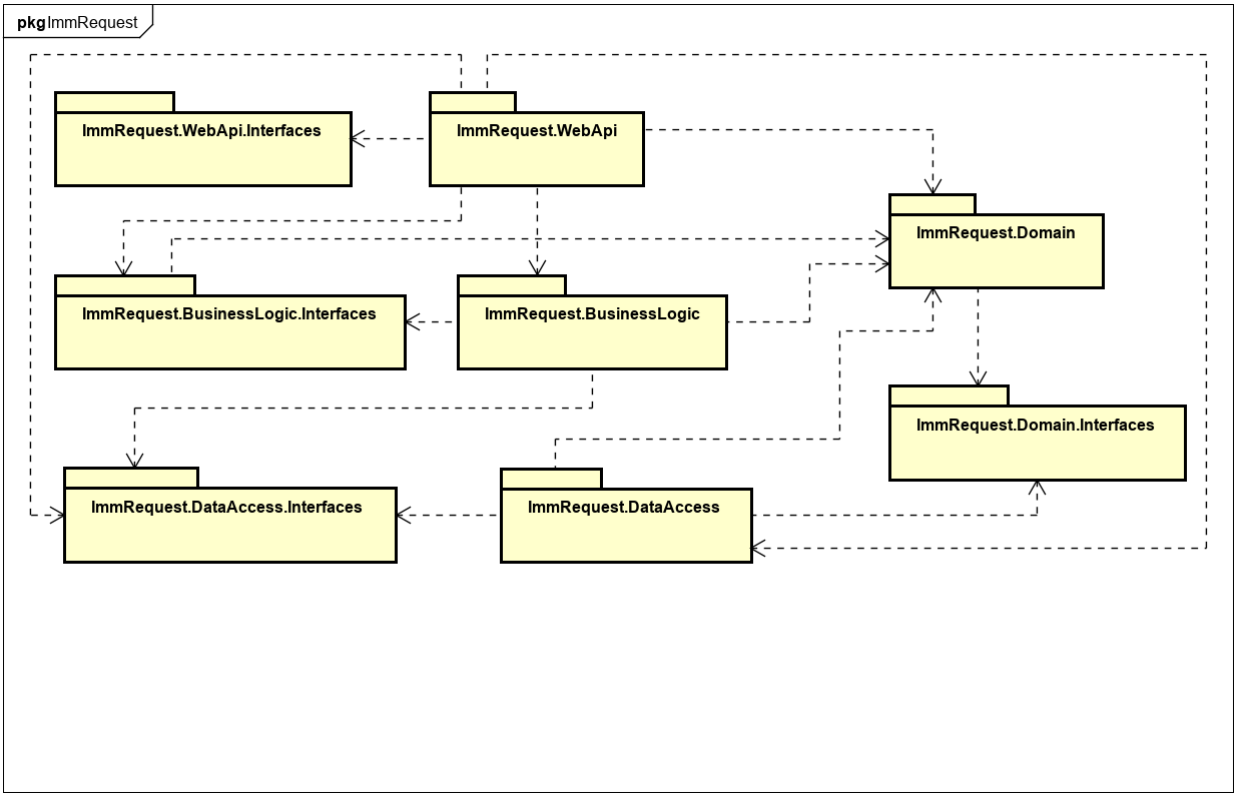

Figura 2.1: Diagrama de dependencias entre paquetes

Nuestra idea principal para el diseño de los paquetes fue generar niveles y conectar cada nivel con interfaces intermediaras con el objetivo de que nuestros paquetes más concretos dependan de paquetes más abstractos y por ende mas estables. Los paquetes de *DataAccess.Interfaces* y *BusinessLogic.Interfaces* buscan lograr justamente eso.
La otra razón detrás de los paquetes de *.Interfaces* es poder asignar la responsabilidad de establecer los contratos o pre-requisitos que las clases concretas deben

implementar en un lugar especifico y poder reusarlos para otros proyectos con requerimientos similares. *Domain.Interfaces* y *WebApi.Interfaces* son ejemplo de esto.

Descripción de cada paquete

- **ImmRequest.Domain:** Es el paquete que contiene las entidades de la solución que proporciona el sistema.
- **ImmRequest.Domain.Interfaces:** Define los contratos que deben proporcionar las entidades del sistema.
- **ImmRequest.DataAccess:** Es el paquete que tiene la responsabilidad de comunicarse con la base de datos. Se encarga de manejar el contexto y realizar las consultas necesarias para la lógica.
- **ImmRequest.DataAccess.Interfaces:** Contiene las interfaces que deben implementar las clases que se comuniquen con la base de datos. Conecta el paquete de la lógica con el de base de datos, para reducir el impacto de cambio.
- **ImmRequest.BusinessLogic:** Contiene la lógica de negocio para poder cumplir con los requerimientos funcionales de la aplicación.
- **ImmRequest.BusinessLogic.Interfaces:** Establece las interfaces necesarias para las clases de lógica. Conecta el paquete de WebApi y el paquete de Lógica.
- **ImmRequest.WebApi:** Es el paquete ejecutable del proyecto, en donde se encuentran los diferentes endpoints del sistema.
- **ImmRequest.WebApi.Interfaces:** Contiene las interfaces necesarias para la implementación de algunas clases del el paquete WebApi

2.1.2. Diagrama de paquetes con nesting

En el siguiente paquete se puede observar cada paquete principal con sus subpaquetes. El objetivo es motivar el reuso y asignarle una responsabilidad común a las clases a cada paquete.
Una decisión que se debió tomar, fue donde ubicar las excepciones del paquete de DataAccess. Originalmente, se hubiese ubicado en el paquete de *ImmRequest.DataAccess*, de forma que cada paquete concreto determinara sus propias excepciones. Sin embargo, realizarlo de tal forma implicaba que el paquete de *ImmRequest.BusinessLogic* dependiera directamente del paquete de *ImmRequest.DataAccess* y no solo del paquete de interfaces. Decidimos, entonces mover las excepciones al paquete de *ImmRequest.DataAccess.Interfaces* para que se pudiera hacer un manejo eficiente de ellas en la lógica del negocio y que el paquete de lógica del negocio dependa únicamente de un paquete de mayor estabilidad
En el caso de la relación entre la lógica del negocio y el paquete de web api no se presento la misma situación debido a que la dependencia era necesaria para poder

realizar la inyección de dependencias correctamente. De todas formas desacoplar, estos paquetes es un objetivo para futuras entregas.

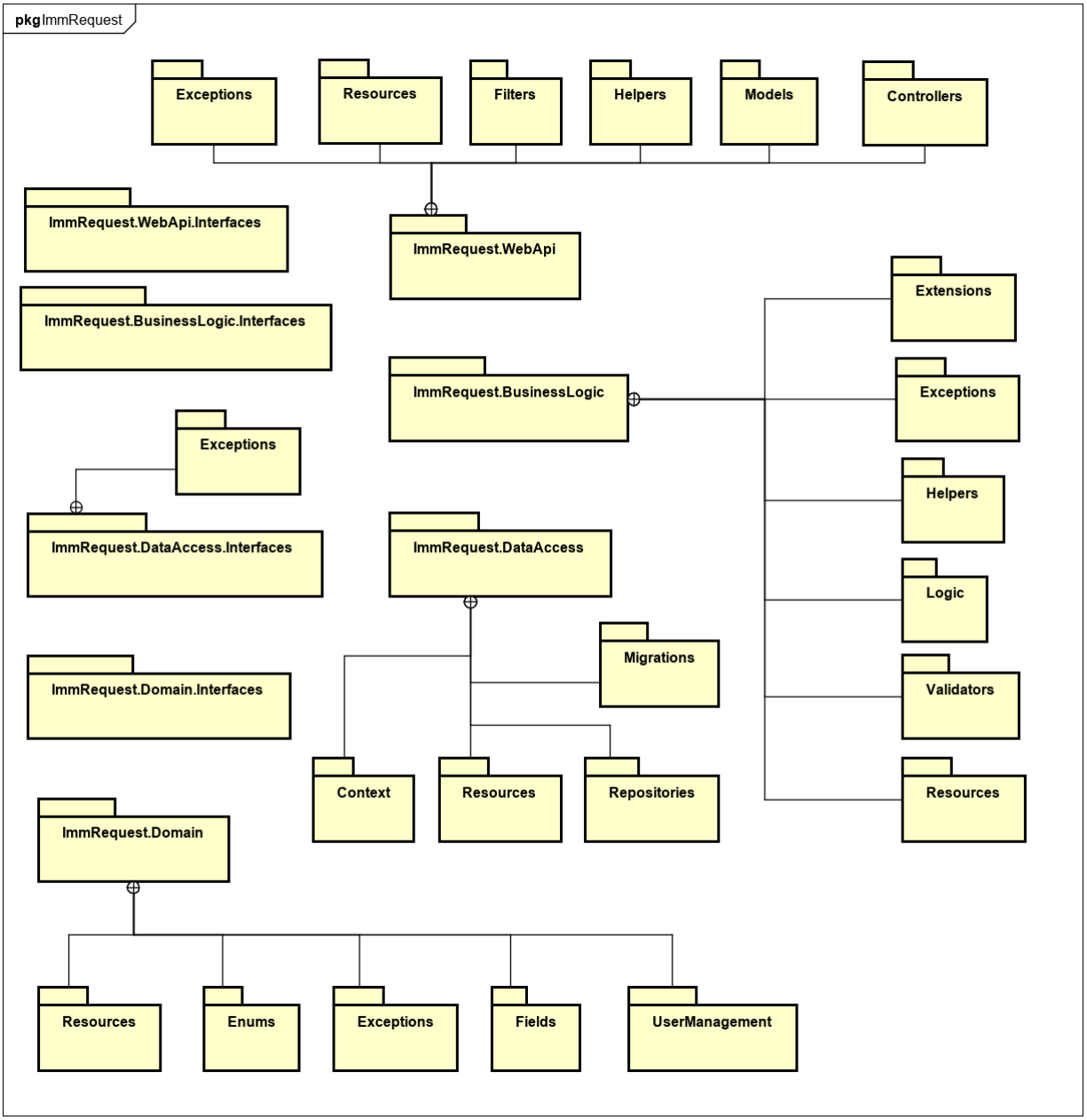

Figura 2.2: Diagrama de paquetes usando conector nesting

2.1.3. Diagrama de Componentes del sistema

En el siguiente diagrama de componentes podemos observar como los distintos componentes de cada paquete se comunican a través de las interfaces establecidas en cada uno de los paquetes de *.Interfaces*.

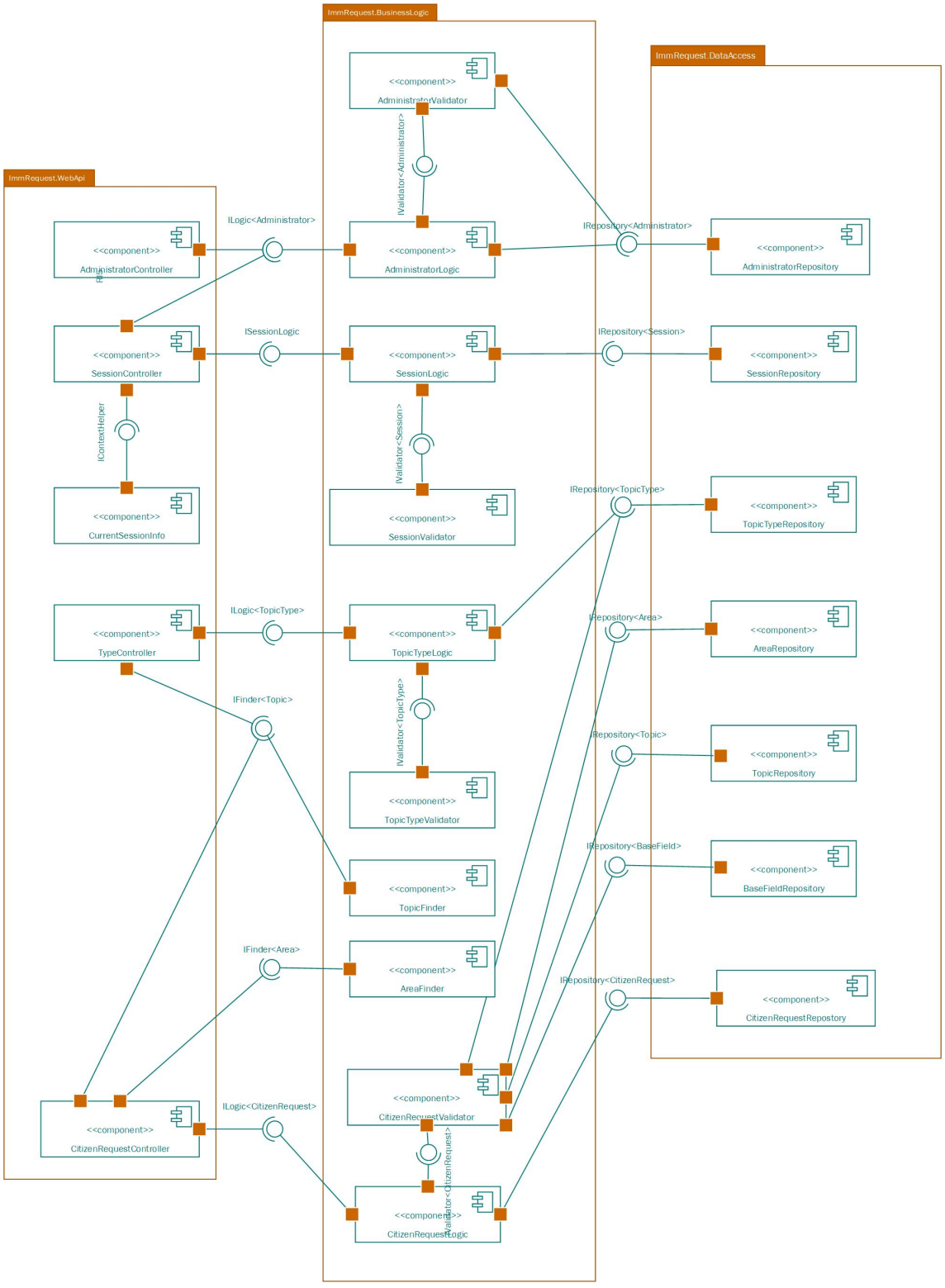

Figura 2.3: Diagrama de componentes del sistema

2.2. Paquete de Dominio

2.2.1. Diagrama de Clases

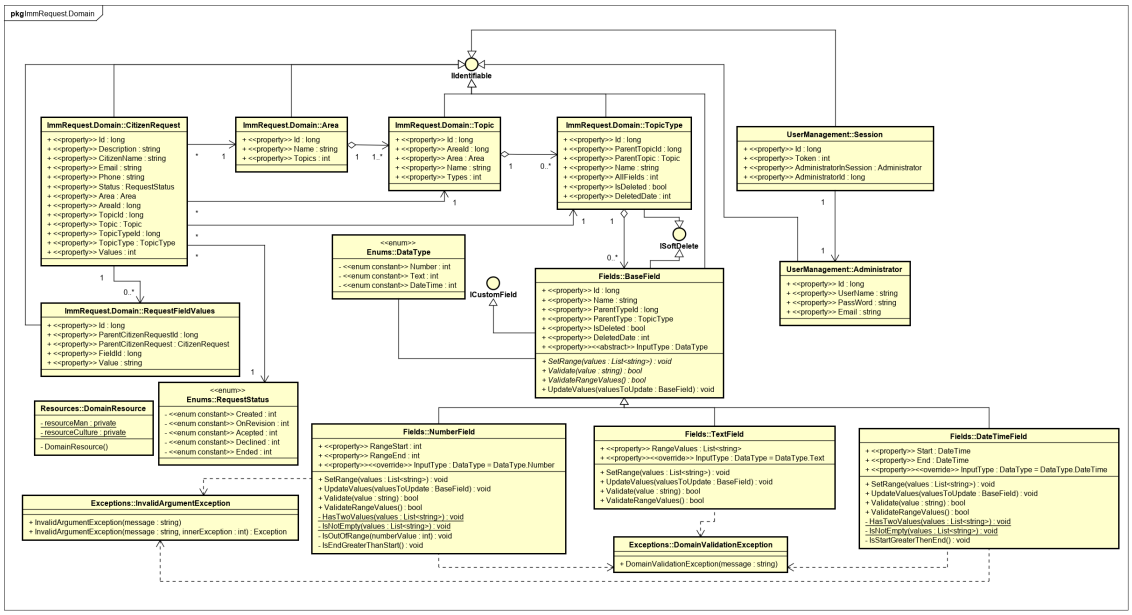

Figura 2.4: Diagrama de clases de domino

La decisiones mas importantes respecto al diseño de nuestra solución fueron tres. La primera, es que toda entidad que se fuera a guardar en la base de datos tenia que implementar la interfaz de *IIdentifiable*. Como forma de establecer un estándar para el ID de las entidades de la base de datos y asegurarse que toda entidad la cumpla.

La segunda, es la utilización de la interfaz *SoftDelete*. Aquellas entidades que no se deban eliminar permanentemente deberán implementar esta interfaz, la cual luego el contexto utilizara para determinar el modo de eliminación, pero ademas para llevar un control de la fecha de la misma. En caso de algún error, la entidad que se elimino se puede encontrar por su fecha siempre y cuando se sepa la misma.

Por ultimo, la utilización de polimorfismo para la implementación de los campos adicionales. Usando un patron strategy, en donde una interfaz *ICustomField* define el comportamiento que cualquier tipo de campo adicional debe implementar, nos permite dar la extensibilidad de agregar futuros campos adicionales cuya implementación del comportamiento difiera de la de *BaseField*. Al mismo tiempo se creó la clase *BaseField* para poder implementar el comportamiento común que tienen los campos adicionales actuales y utilizar el polimorfismo para asignarle la responsabilidad a cada campo de la asignación de su rango y validación del valor que se da en la solicitud. Esto permite que no se tengan que crear clases especiales que según el tipo campo validen o asignen el valor del mismo en la solicitud.

2.2.2. Diagramas de Interacción

A continuación se pueden encontrar diagramas de secuencia y colaboración que buscan ejemplificar el uso del polimorfismo para la creación y validación de los campos adicionales.
En ambos casos se esta creando un *Topic Type* que es quien contiene una lista de *BaseFields*. Por cada elemento en esa lista se va a llamar a los métodos de cada implementación de una BaseField para validar y definir su rango.

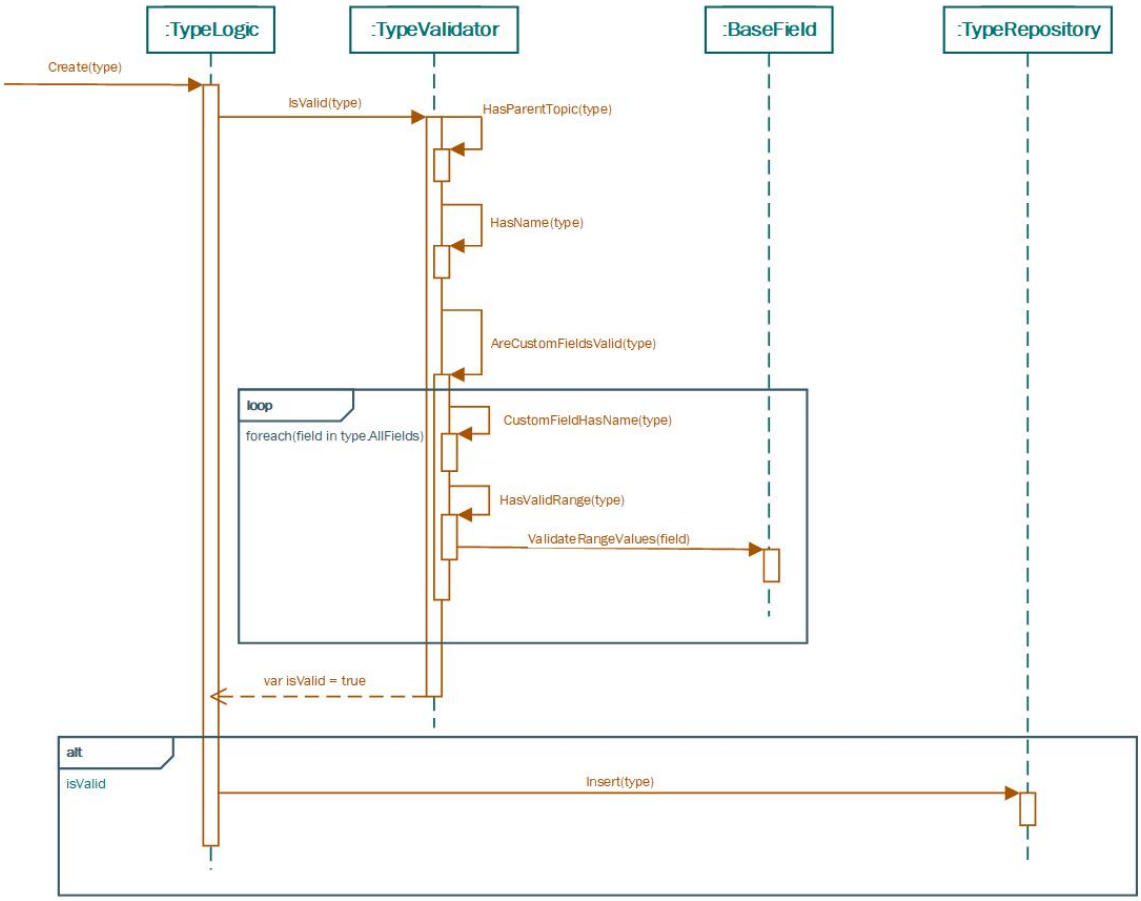

Figura 2.5: Crear Topic Type desde la lógica

En el siguiente diagrama se puede observar un caso concreto, asumiendo que el la lista de campos adicionales hay un campo de cada tipo y que la variable field cambia en cada iteración del loop.

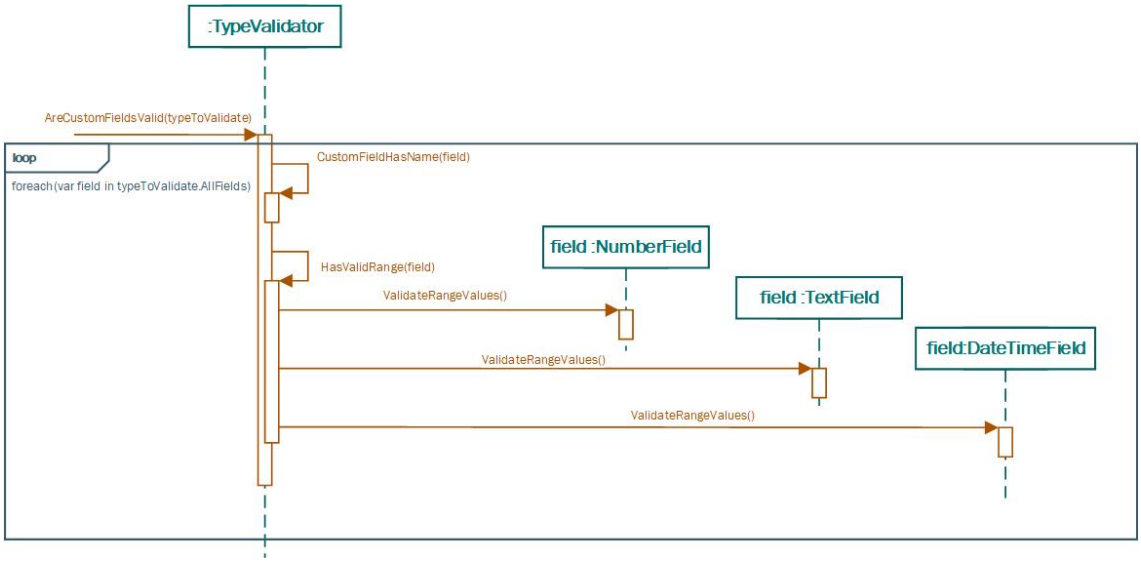

Figura 2.6: Validar rango de campos adicionales caso concreto

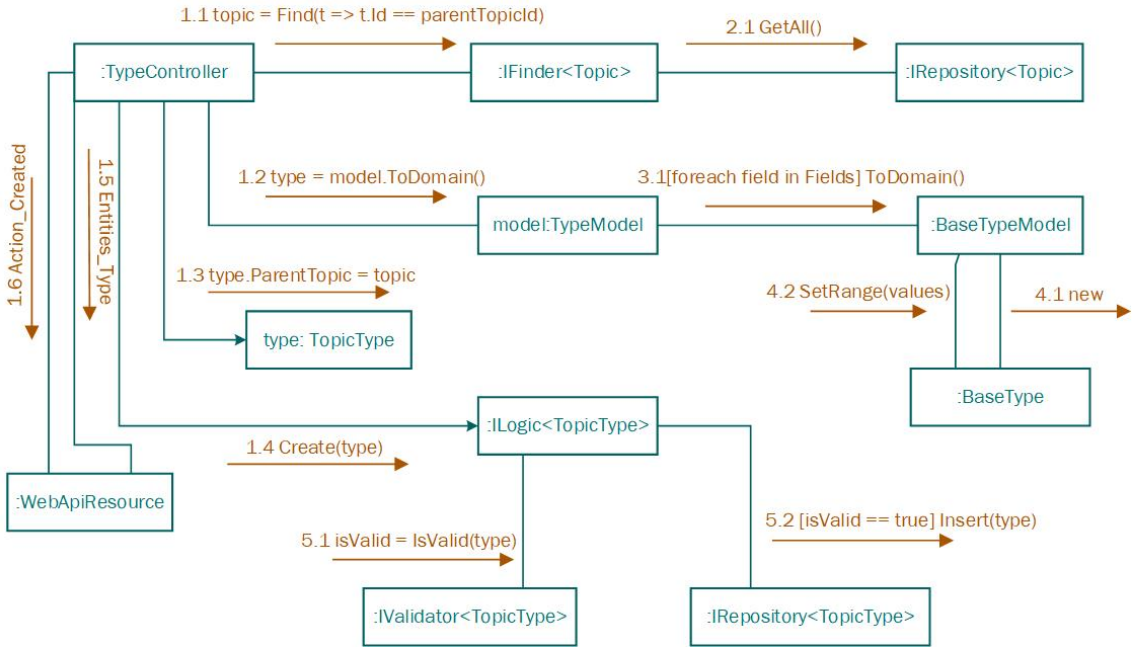

Figura 2.7: Crear un topic type

2.3. Paquete de Data Access

2.3.1. Diagrama de Clases

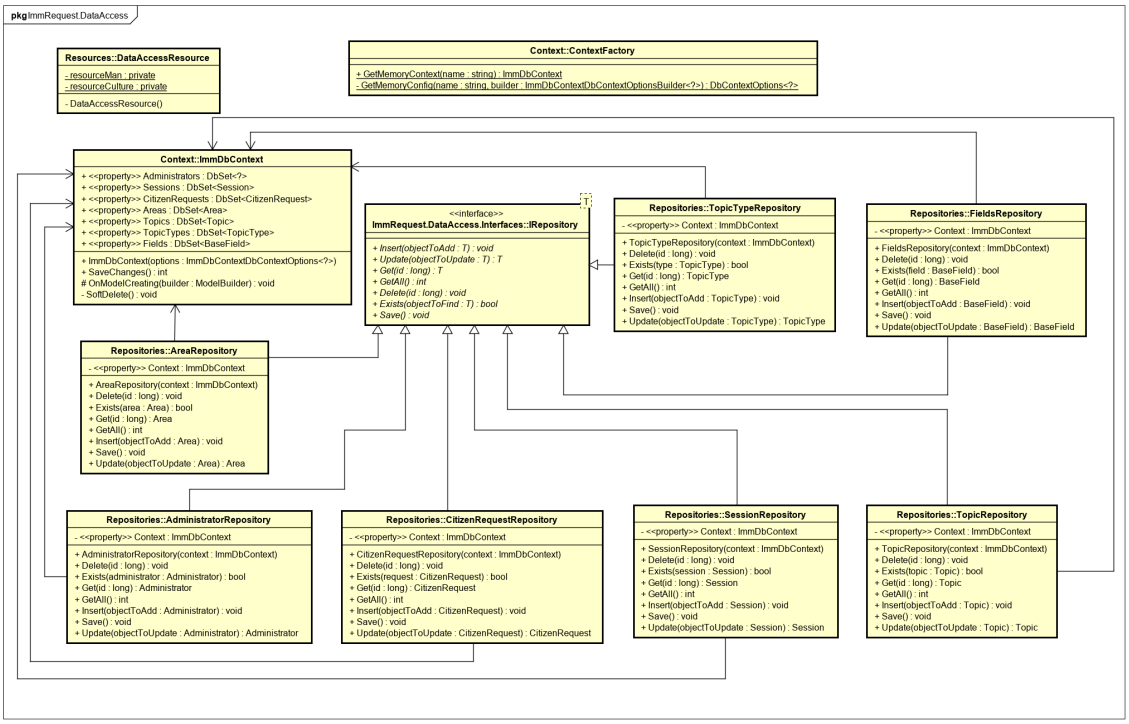

Figura 2.8: Diagrama de clases del paquete Data Access

Para el data access se determino una interfaz que establece las funciones básicas para cualquier repositorio. De esta forma cada repositorio implementa sus comportamiento especifico, lo que nos permite incluir las entidades de navegación requeridas para cada entidad.
Un detalle a destacar, se implemento un *Query Filter* para las entidades que implementen la interfaz *SoftDelete* que mencionamos en la sección anterior. La razón de esto, es para prevenir que cuando el repositorio retorne las entidades de la base, no retorne aquellos que fueron marcado como eliminados. El único caso en el que se ignora este filtro es en el *Get y GetAll* de la CitizenRequest por la forma en que Entity Framework resuelve los includes. De no ignorar este filtro, no traería aquellas solicitudes que cuyos tipos o campos adicionales fueron eliminados por algún administrador.

Otro dato interesante es que los datos de las Areas y Temas, al igual que un administrador, son creados con migrations a la base. Esas migrations no modifican en ninguna forma el diseño del sistema en ninguna forma, sino que aprovechan el Migration builder para ejecutar sentencias SQL e ingresar los datos necesarios para la ejecucion del sistema.
Nota importante:*Dichas migrations tienen como pre requisito que no hayan datos de prueba ingresados previamente a la base. Ya que la migration que ingresa los*

tipos depende de los IDs de las Areas creadas en la migration anterior. Si la base esta siendo creada por primera vez no va ver problema

2.3.2. Diagrama de Entidades

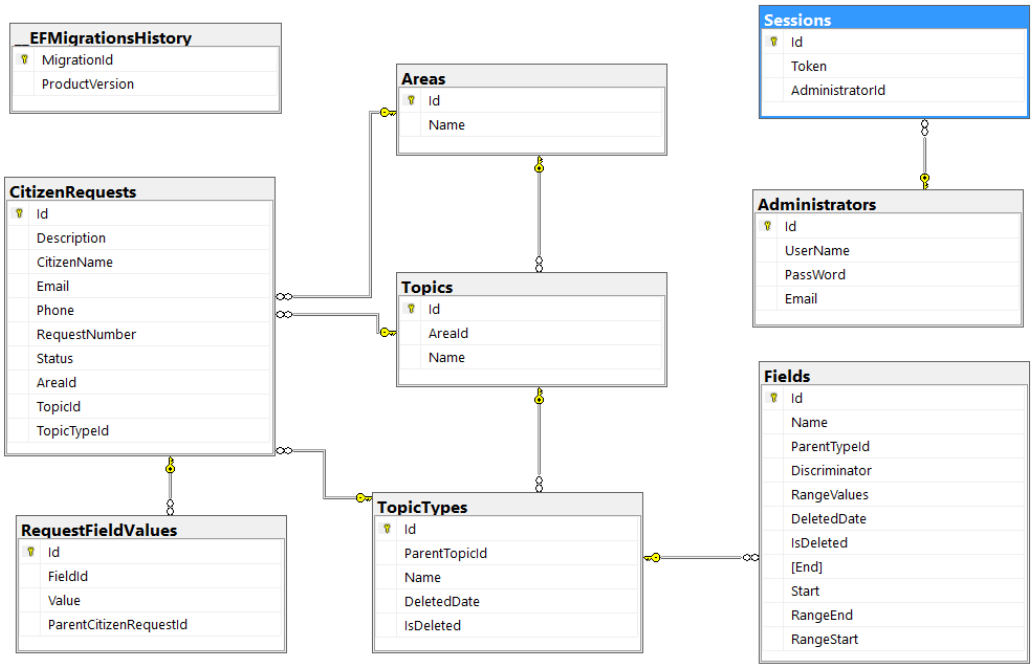

Figura 2.9: Tablas en la base de datos

Agregamos a este diagrama la tabla donde se guarda el registro de migrations aplicadas a la base.

2.4. Paquete de Lógica de Negocio

2.4.1. Diagrama de Clases

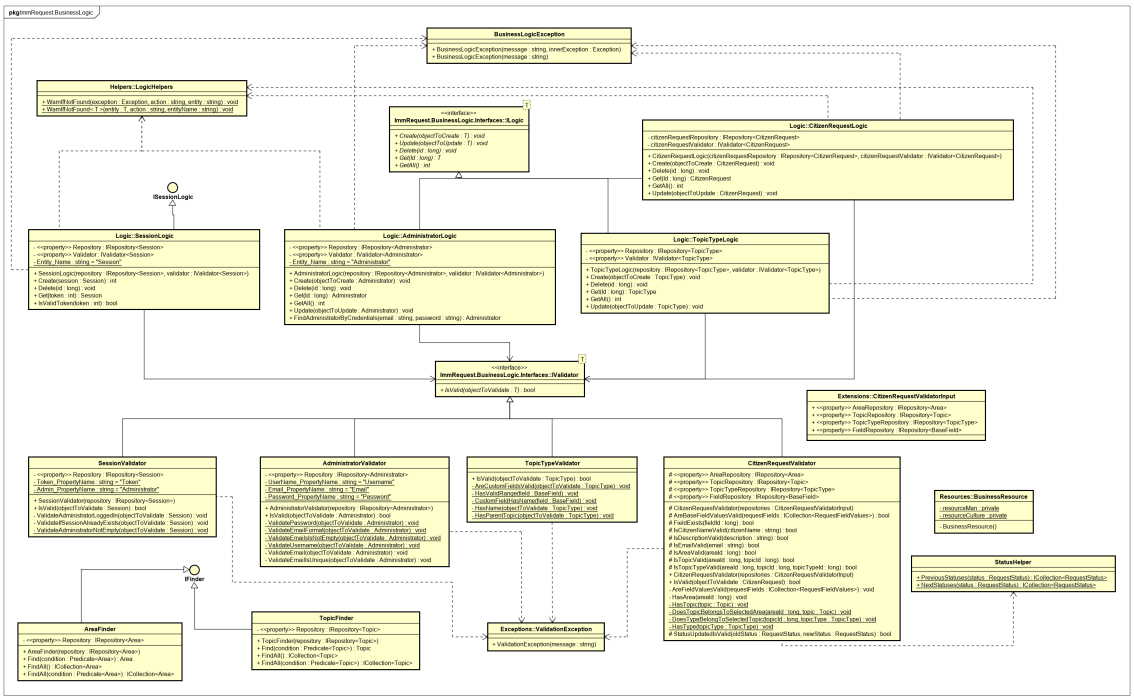

Figura 2.10: Diagrama de Clases de la lógica

Nuestra solución de paquete de lógica se divide en dos grandes pedazos. Las clases que implementan *ILogic*, cuya responsabilidad es manejar la lógica de negocio relacionada a alguna de las entidades, y las que implementan *IValidator*, cuya responsabilidad es validar las entidades que forman parte de las operaciones lógicas. Se tomo esta decisión para poder repartir la responsabilidad de realizar las operaciones del negocio de validar que cumplan las condiciones necesarias para guardar los datos a la base. Nos da la flexibilidad de poder crear nuevas clases de lógica, que no requieran de validadores y que estos puedan ser usados en otras partes del sistema si se requiere. Estas clases fueron separadas en paquetes distintos para que sean reusables, en caso necesario. Para tanto la Lógica como los Validadores se usaron patrones strategy para tener la flexbilidad de crear distintas implementaciones de cada uno que cumplan con el comportamiento y sean intercambiables si existiera la necesidad

La interfaz *IFinder* define un comportamiento de búsqueda para un tipo genérico. Su objetivo, es poder implementar una lógica reducida para entidades que no requieran operaciones crud de lógica, como la que propone *ILogic*. Su objetivo es evitar que se implementen funciones innecesarias que solo generarían ruido en el código.

2.4.2. Diagramas de Interacción

A continuación se pueden como funciona un validador, en el caso de una *CitizenRequest*, y como colaboran con las clases de lógica, en el caso de la creación de un *Administrador*.

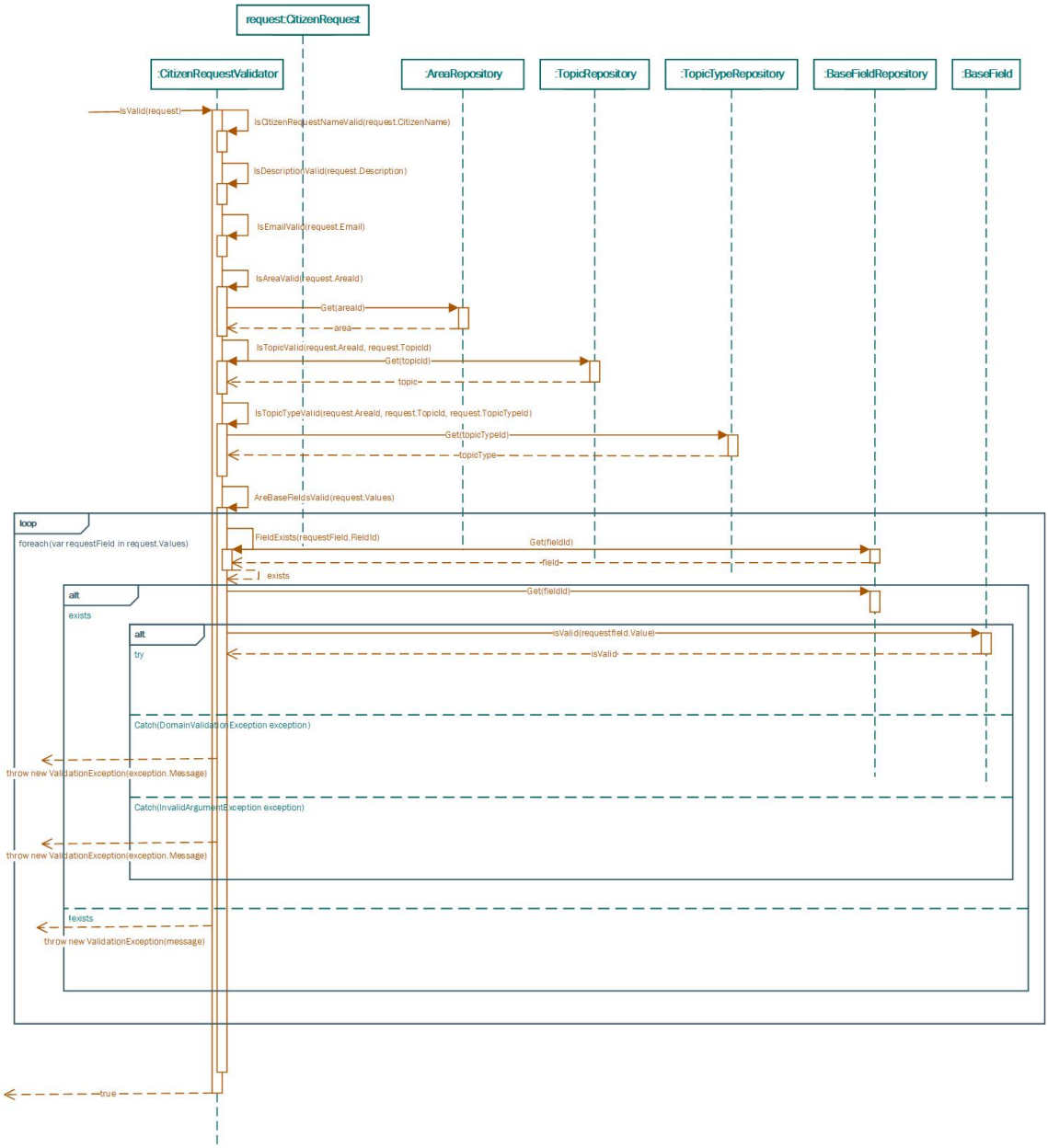

Figura 2.11: Validación de una citizen request

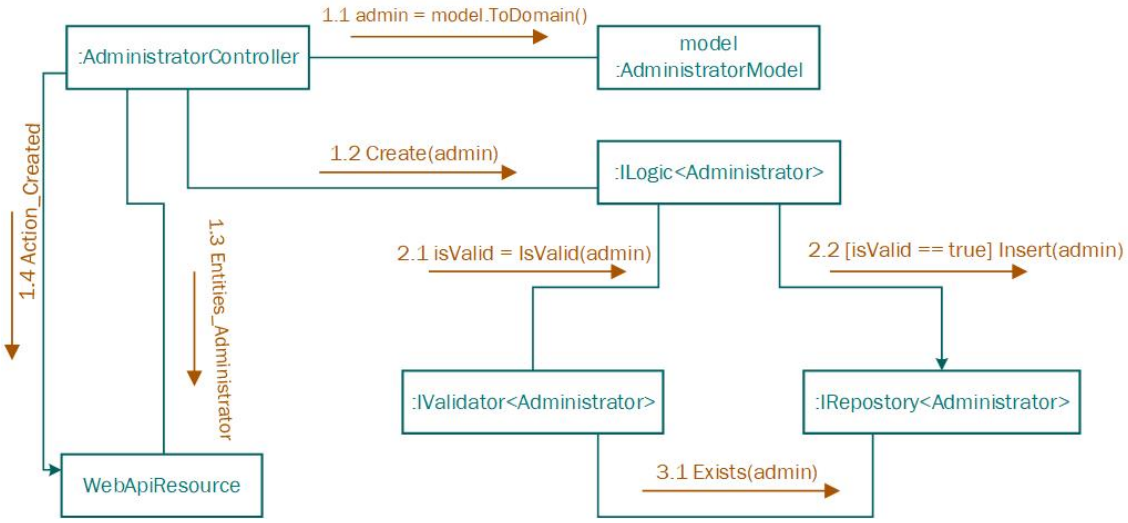

Figura 2.12: Creación de un administrador

2.5. Paquete Web Api

2.5.1. Diagrama de Clases

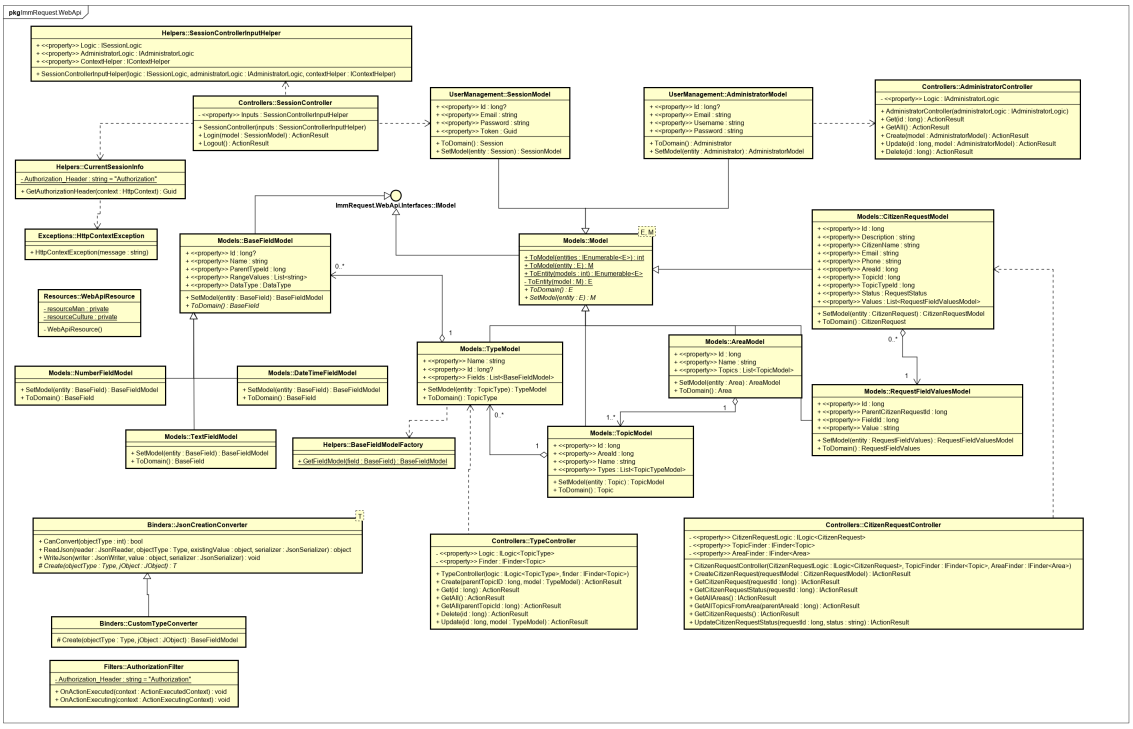

Figura 2.13: Diagrama de Clases de la Web Api

Para poder cumplir con los requerimientos necesarios se hicieron las cuatro controllers: *AdministratorController*, para el mantenimiento de Administradores, *CitizenController*, para la creación y mantenimiento de solicitudes, *TypeController*,

para el mantenimiento de tipos y sus campos adicionales y *SessionController* para el control de sesión de un administrador. El reparto se le dio a cada controller la responsabilidad sobre alguna entidad y sus operaciones crud, de forma que fuera facil ubicar que controller realiza que operación.

Los controllers reciben los datos en forma de json, los cuales un binder transforma en nuestros clases de modelos, que tiene la responsabilidad de convertir los datos recibidos a clases de dominio o viceversa.
Para el caso especial de los campos adicionales, aplicamos el patron *Factory Method*, la clase *BaseFieldModel* define el método abstracto para crear los distintos campos a su version de dominio. (Metodo *ToDomain()*). Luego cada modelo de cada campo crea su propio field y el *TypeModel* utiliza este método para crear en dominio todos sus campos sin conocer que tipo de field esta creado.
Sin embargo, en el caso de la creación de *Fields* a modelos tuvimos que utilizar una clase helper que nos permitiera crear la instancia del modelo segun el *DataType* de cada *field*, esto se debió a que la clase *BaseField* es abstracta. De igual formas el modelo de *TypeModel* desconoce como se crean tanto los modelos como los objetos del dominio de los campos adicionales y utiliza la fabrica.

2.5.2. Diagramas de Interacción

El siguiente diagrama ejemplifica la creación de los modelos *TypeModel* y *BaseFieldModel*. Aunque en este ejemplo no se usa clases concretas para simplificar el diagrama y reflejar la interacción entre los modelos.

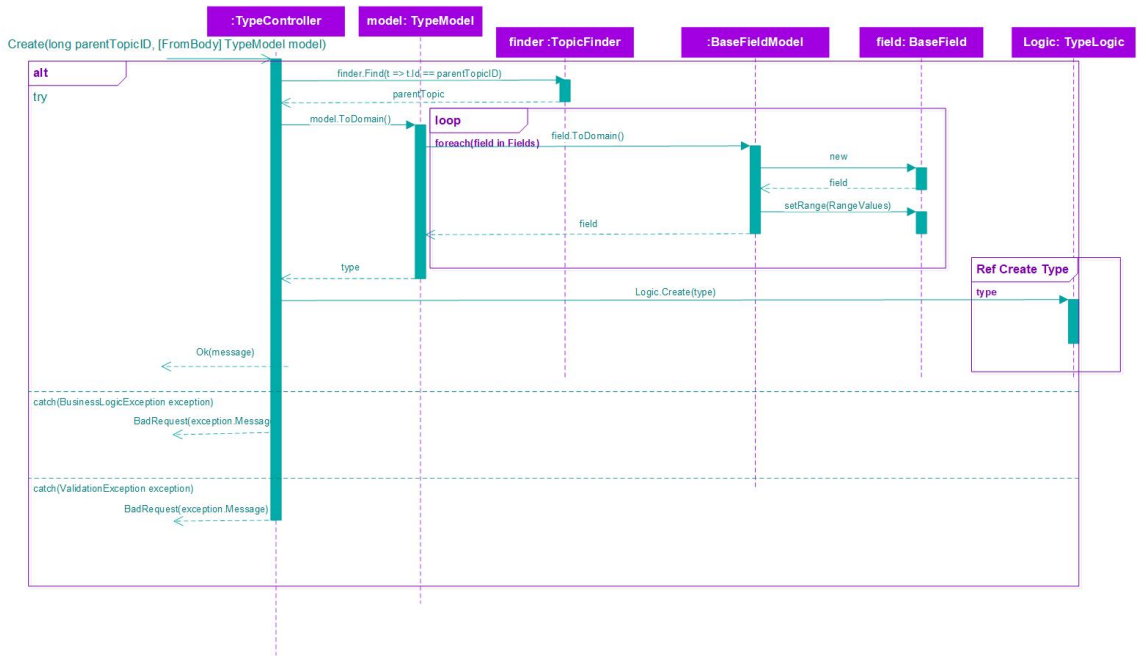

Figura 2.14: Crear un Type desde el Controller

2.6. Deploy de la Aplicacion

El deploy de la aplicación cuenta con tres nodos. El primer nodo contiene la base de datos de SQL Server, que se utilizara para persistir los datos. El segundo nodo contiene el servidor en que se encontrara el API desarrollada. El protocolo de comunicación para estos dos nodos es TCP/IP.
El tercer nodo es el servidor cliente, en esta primera etapa de desarrollo nuestro cliente sera postaman, que el front end no esta realizado. La comunicación con la API de este nodo se da a través de HTTP v 1.0.

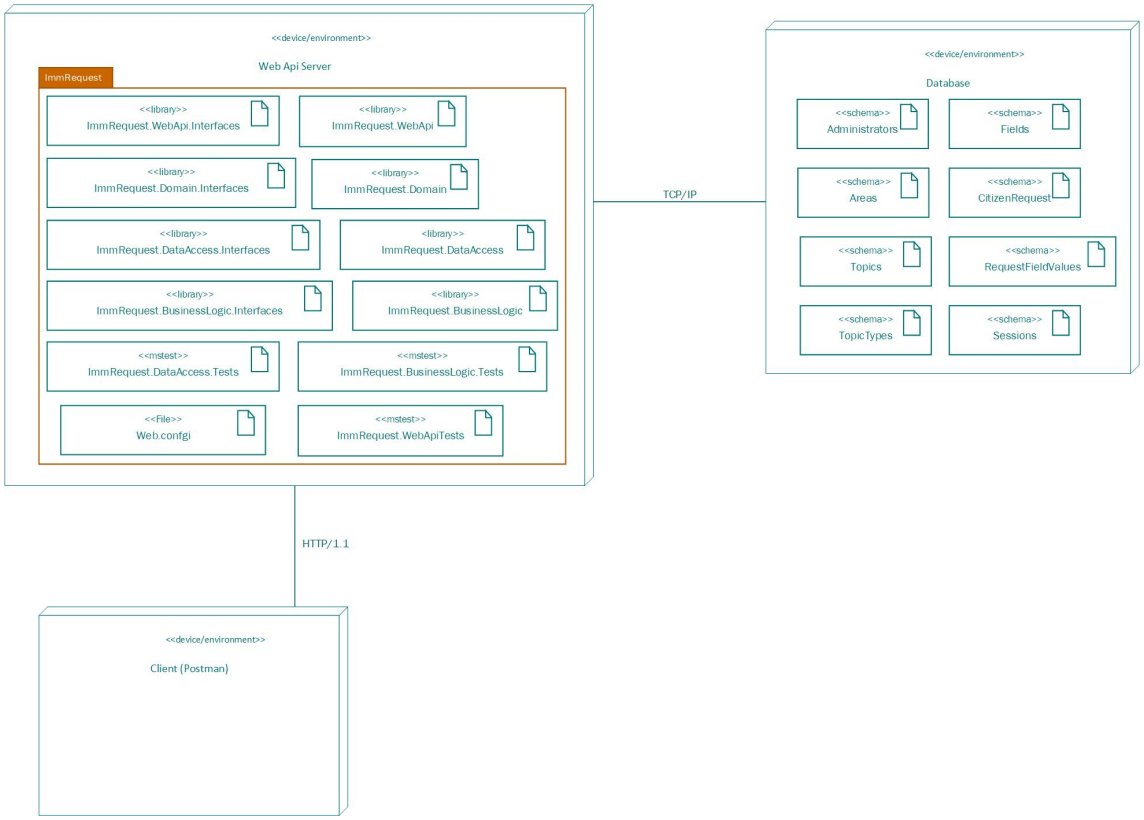

Figura 2.15: Diagrama de Deploy