Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Diseño de Aplicaciones 2

Obligatorio 1

1-Descripción de diseño

Santiago Castro – 168347

https://github.com/ORT-DA2/168347

Grupo M6B

Tabla de contenido

1. Descripción general del trabajo

En este trabajo se desarrolló una aplicación a ser usada por la Intendencia de Montevideo con la finalidad de ofrecer un servicio a los ciudadanos. Este servicio permite la recepción y procesamiento de solicitudes las cuales describen una situación la cual desean informar a la Intendencia.

En esta primera instancia del trabajo, se implementó una API REST como Backend, la cual utiliza una base de datos relacional SQL Server para la persistencia de la información. Esta API provee diferentes funcionalidades a ser utilizadas a modo futuro por la capa Frontend, las cuales son:

Ingreso de una nueva solicitud: El ciudadano ingresa los datos de la solicitud: Nombre, email y teléfono del solicitante, detalle y un tipo previamente seleccionado. Estos tipos tienen un tema los cuales pertenecen a un área. Algunos tipos contienen campos adicionales los cuales el ciudadano debe proveer cumpliendo con sus determinados criterios tales como tipo de campo (texto, número o fecha) y posibles valores los cuales debe cumplir.

Consulta del estado de una solicitud: El ciudadano puede consultar el estado de una solicitud ingresando el identificador de la misma. El sistema provee su estado y una descripción que indica información extra sobre el estado de la solicitud.

Inicio y cierre de sesión: Los usuarios administradores pueden iniciar sesión para poder realizar diferentes actividades de mantenimiento del sistema las cuales son las siguientes:

Crear o borrar un tipo: Los administradores pueden crear un tipo asociado a un tema, para esto se especifica un nombre, el tema asociado y los campos adicionales a ser requeridos. Adicionalmente el administrador puede eliminar un tipo existente del sistema.

Listar solicitudes existentes: Los administradores pueden mostrar todas las solicitudes que se encuentran registradas en el sistema.

Cambiar el estado de una solicitud: Los administradores pueden cambiar el estado de una solicitud, así como la descripción del mismo. Los posibles estados son: *Creada, En revisión, Aceptada o Denegada y Finalizada* los cuales existe un orden de precedencia el cual se debe cumplir, por ejemplo, se puede cambiar al estado *Finalizada* solo si la solicitud actualmente tiene alguno de estos estados: *Aceptada, Denegada o Finalizada.*

Cambiar datos de usuarios y borrar: Los administradores pueden cambiar el nombre, email y contraseña de usuarios existentes. También pueden borrar usuarios del sistema.

Se utilizó ASP.NET Core 3 para implementar el Backend el cual consta de una API REST que escucha peticiones HTTP por la red. Esta API es del estilo REST ya que cumple con su estilo arquitectónico detallado en el siguiente capítulo.

2. Estructura de la solución

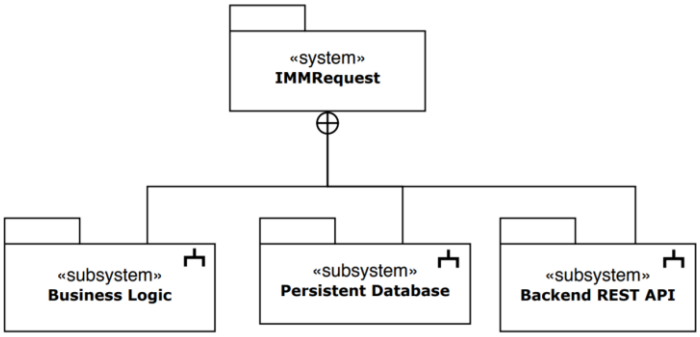
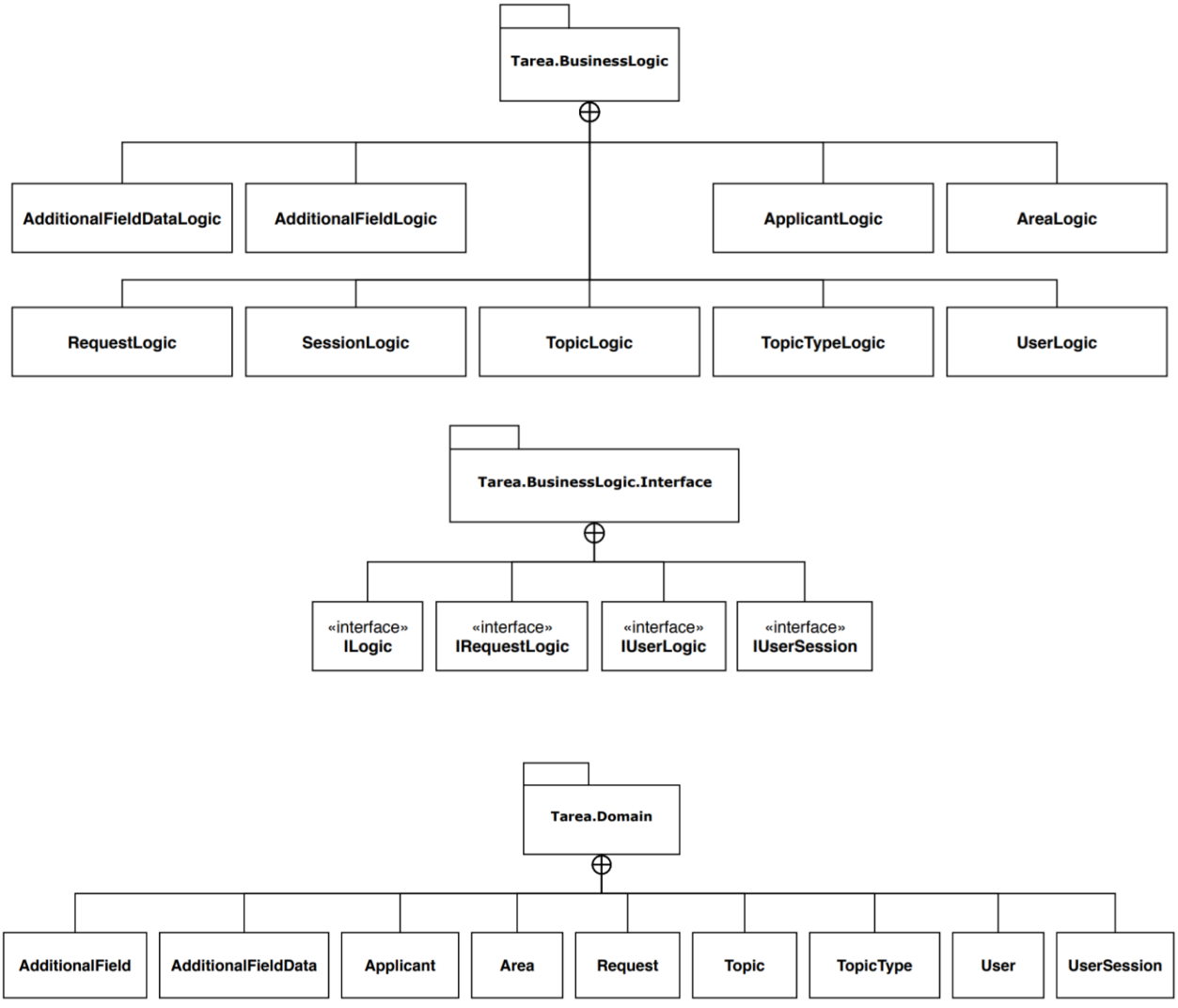


Diagrama de descomposición de namespaces:



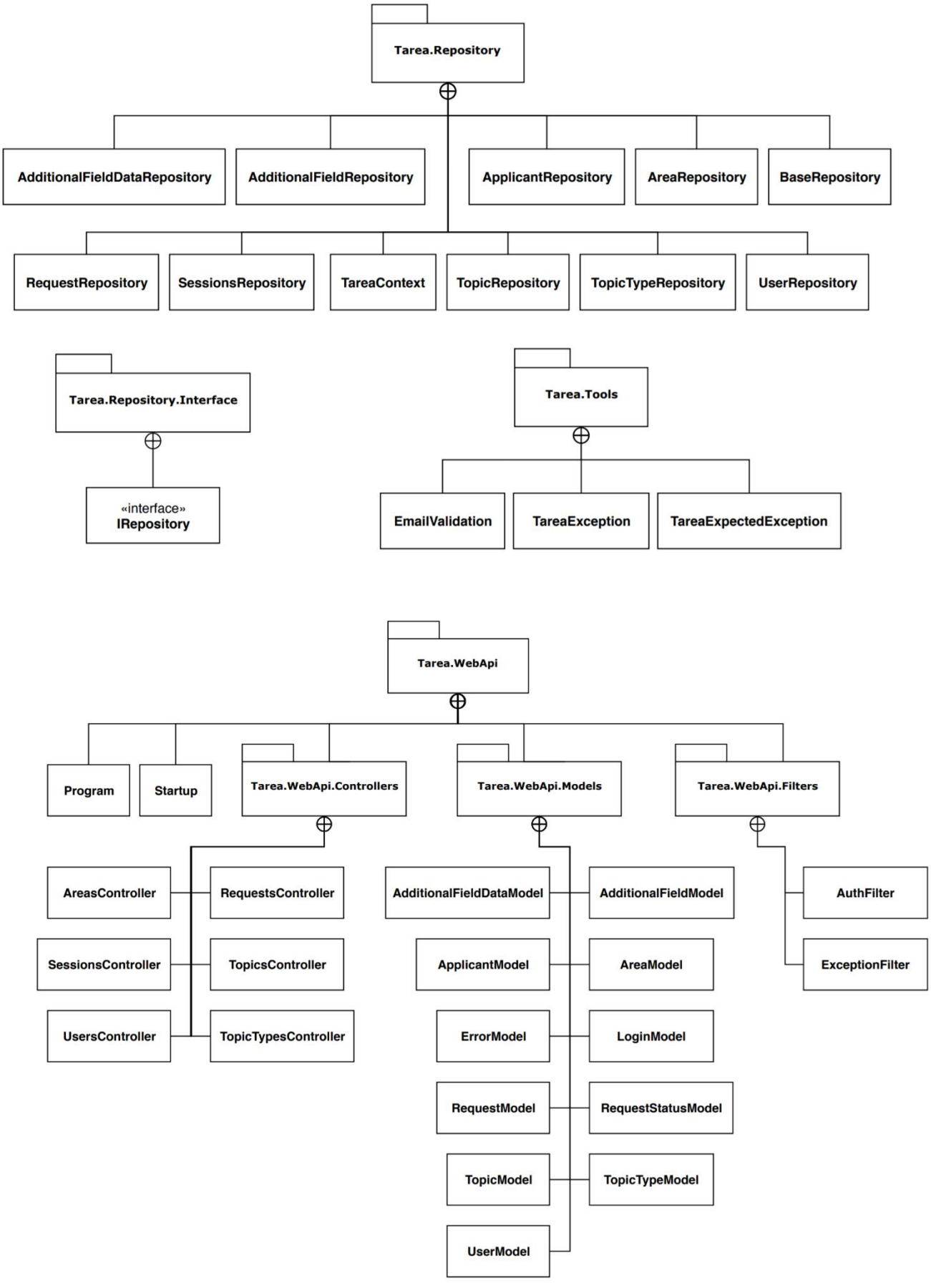
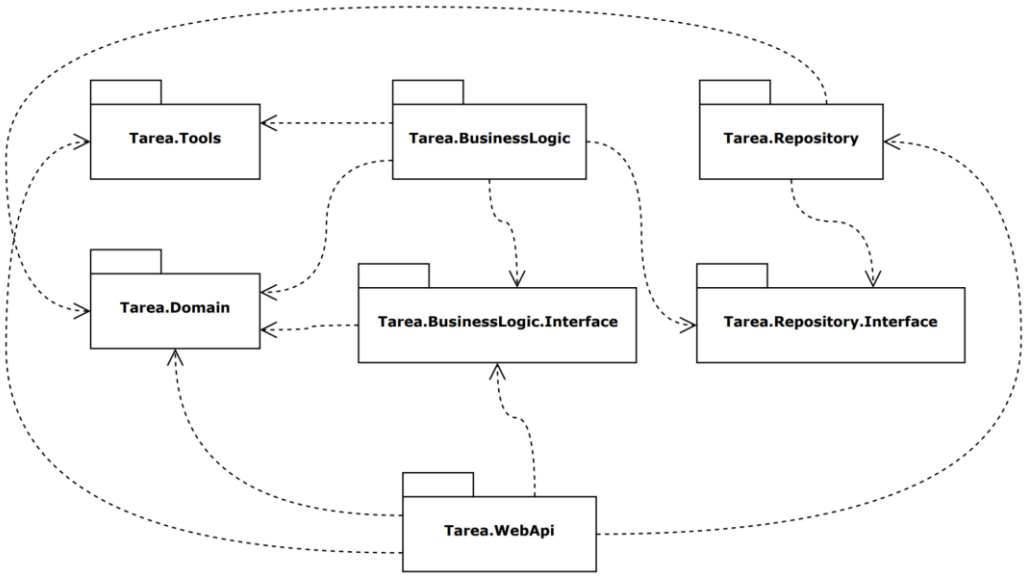
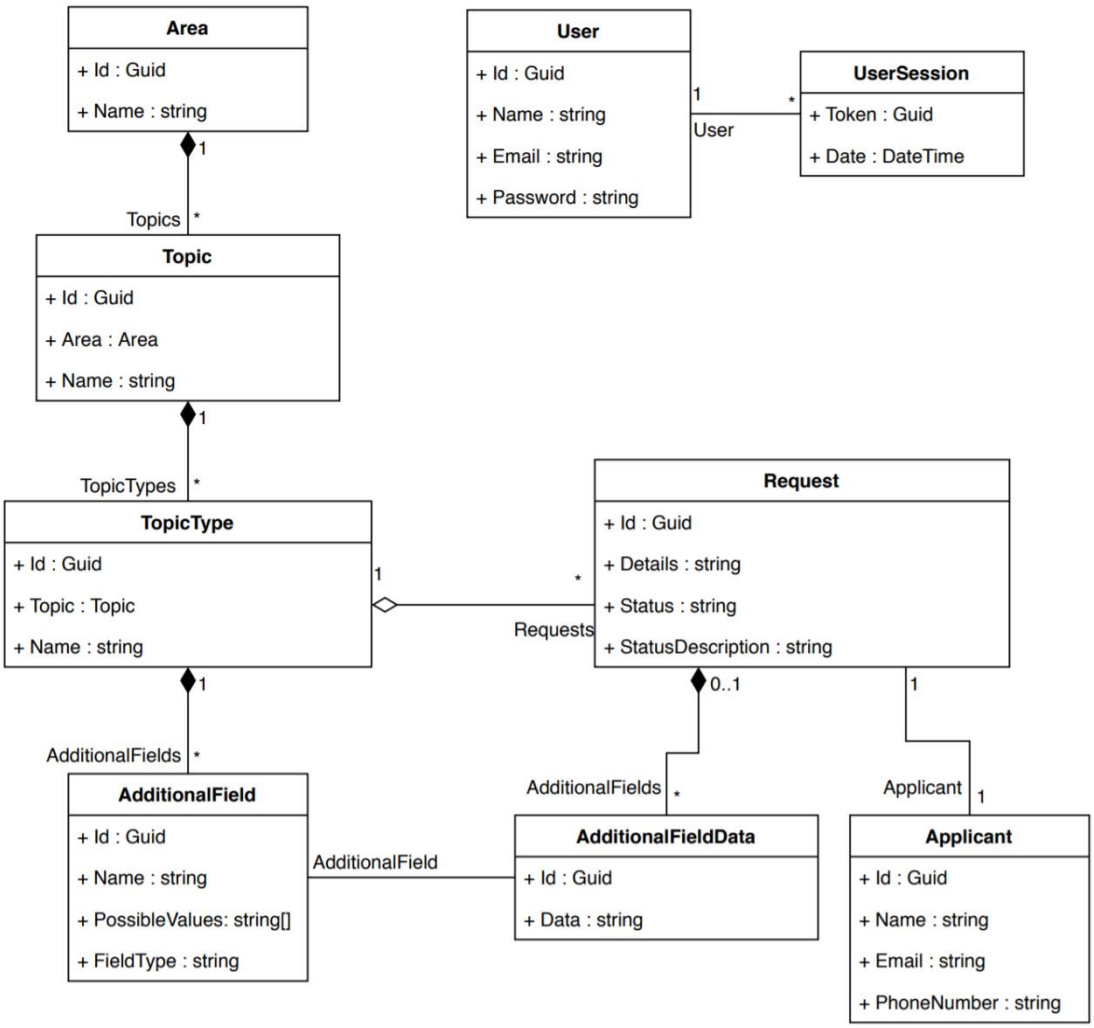

Tarea.Repository
AdditionalFieldDataRepository
AdditionalFieldRepository
ApplicantRepository
AreaRepository
BaseRepository
RequestRepository
SessionsRepository
TareaContext
TopicRepository
TopicTypeRepository
UserRepository
Tarea.Repository.Interface
«interface»
IRepository
Tarea.Tools
EmailValidation
TareaException
TareaExpectedException
Tarea.WebApi
Program
Startup
Tarea.WebApi.Controllers
Tarea.WebApi.Models
Tarea.WebApi.Filters
AreasController
RequestsController
SessionsController
TopicsController
UsersController
TopicTypesController
AdditionalFieldDataModel
AdditionalFieldModel
ApplicantModel
AreaModel
ErrorModel
LoginModel
RequestModel
RequestStatusModel
TopicModel
TopicTypeModel
UserModel
AuthFilter
ExceptionFilter

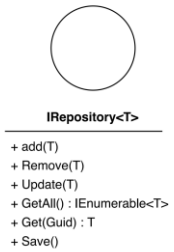
Diagrama de paquetes:



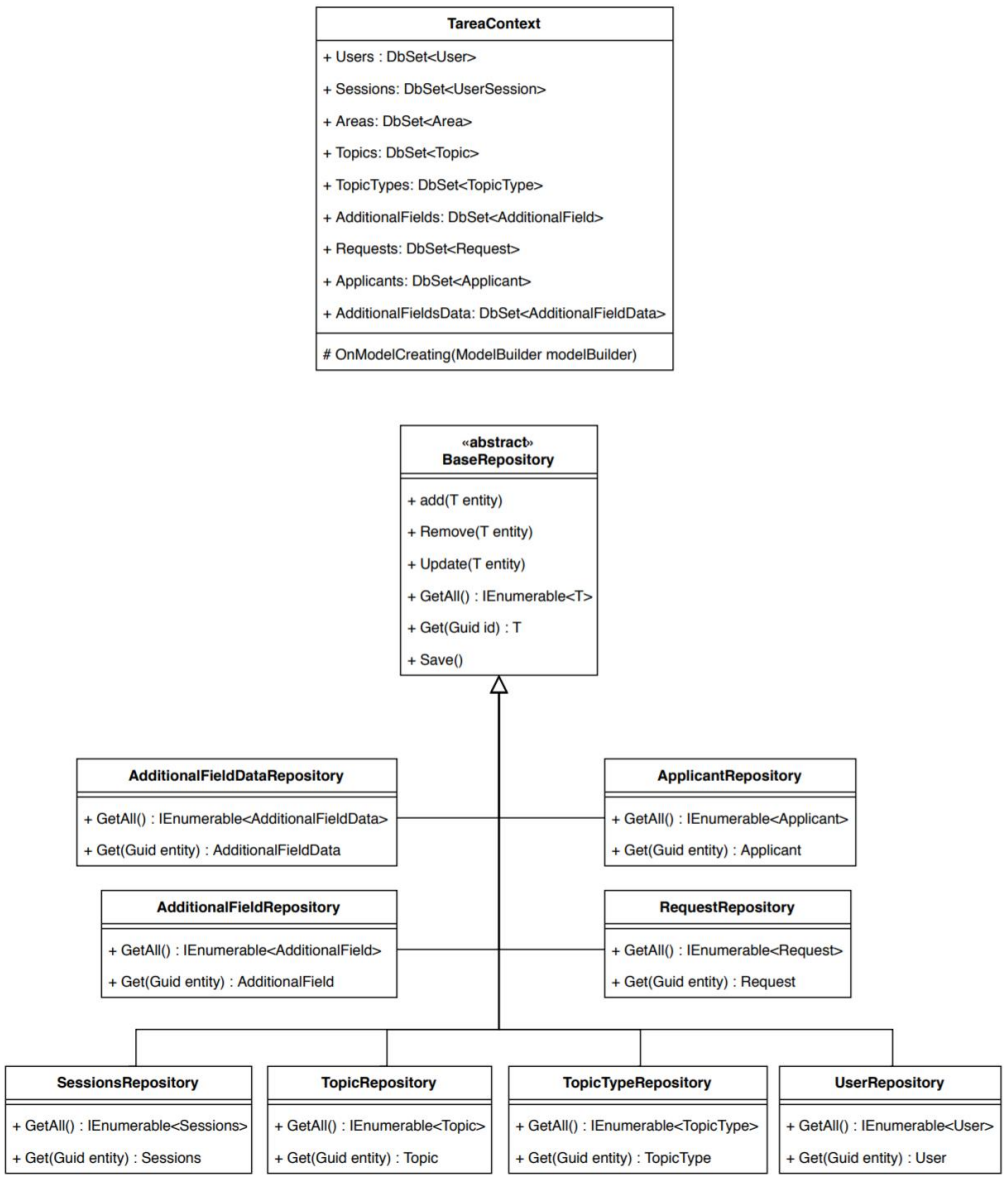
Tarea.Domain: Contiene el modelo del sistema, esto es, las clases tangibles como, por ejemplo: *Area, Topic, TopicType, User, Request,* etc. Es responsable de describir el modelo utilizado por los otros paquetes de la solución, las clases con sus atributos y relaciones.



Tarea.Repository.Interface: Contiene una única clase: *IRepository* la cual describe un contrato con los métodos comunes de uso para la base de datos: *Add, Remove, Update, Get, GetAll y Save*.

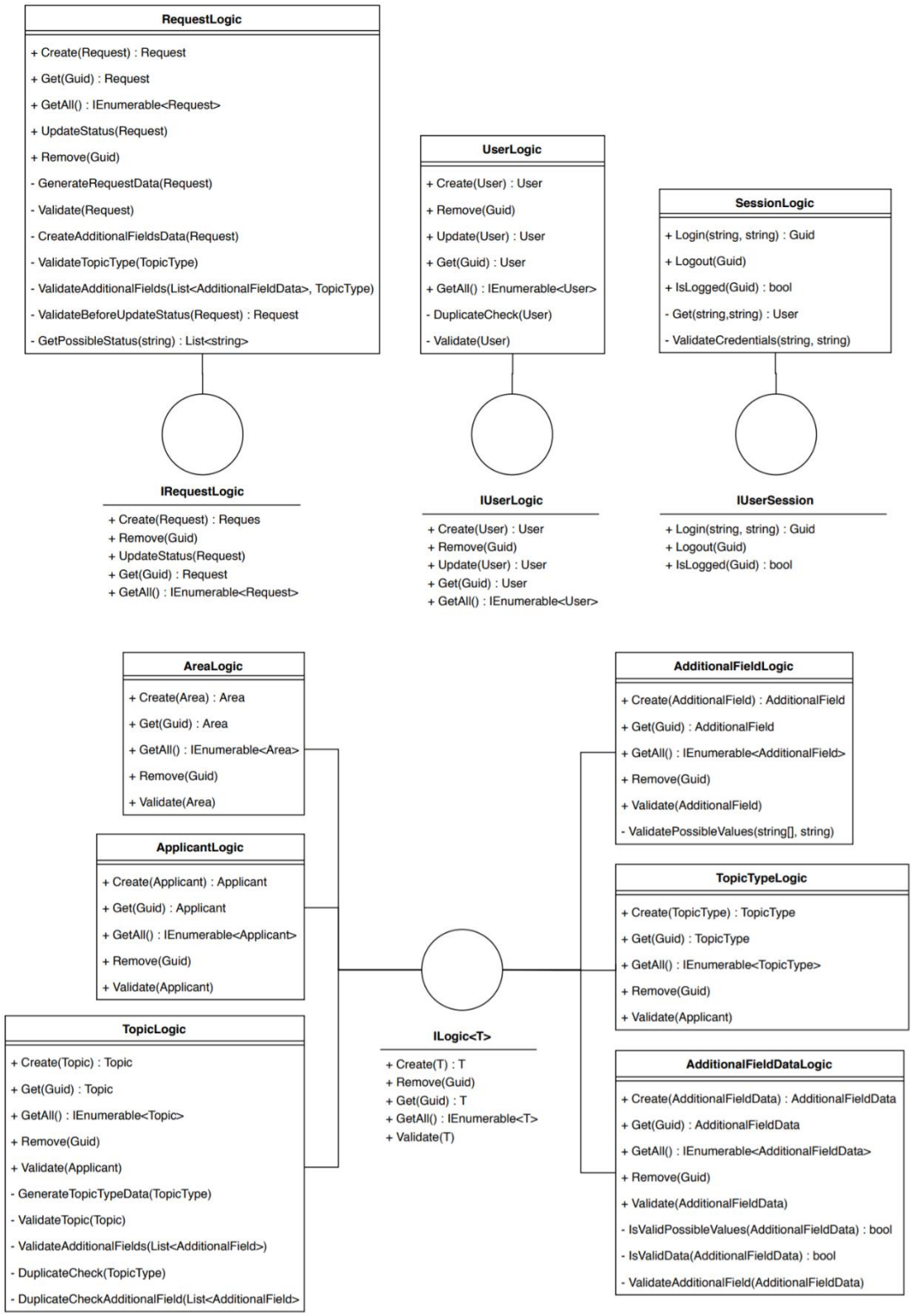


Tarea.Repository: Contiene clases que implementan los métodos de la clase abstracta *BaseRepository* la cual que provee la interfaz IRepository. Además, contiene el contexto de la base de datos en la clase *TareaContext.*

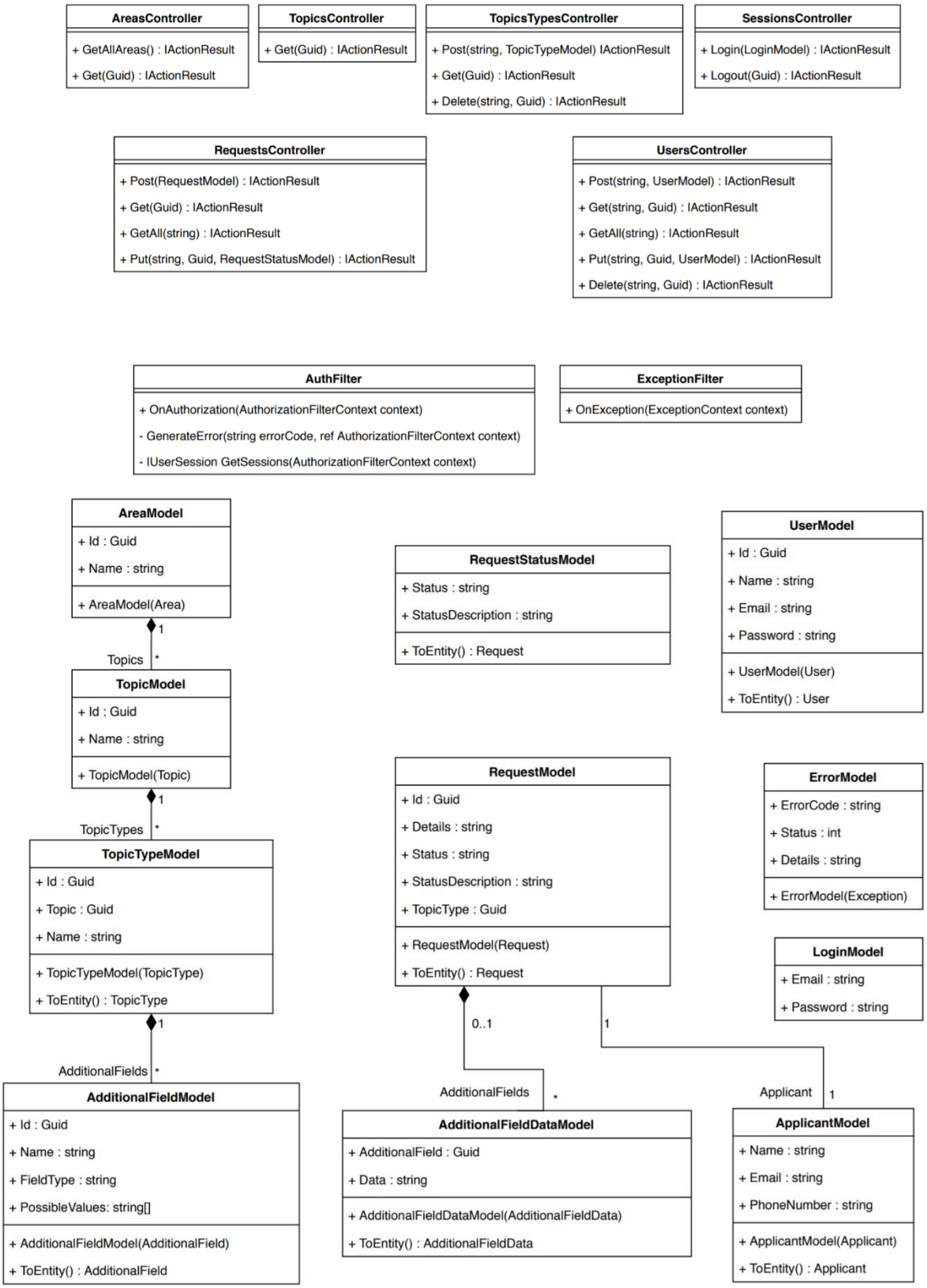


Tarea.BusinessLogic.Interface: Contiene las interfaces que describen métodos a ser usados por los controladores del paquete WebApi. La responsabilidad de este paquete es de exponer la funcionalidad de la lógica sin implementarlas.

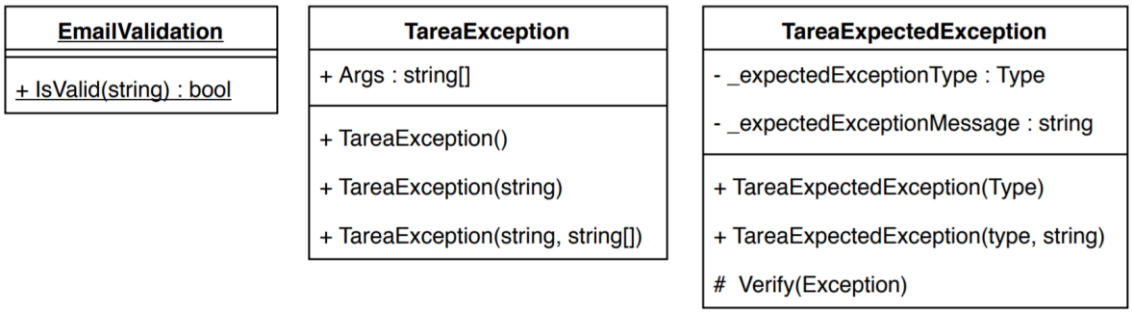
Tarea.BusinessLogic: Este paquete implementa la lógica de negocio descripta por las interfaces del paquete anterior. La responsabilidad de este paquete es de procesar la información proveniente del paquete WebApi y retornar un resultado como puede ser datos o excepciones.



Tarea.WebApi: Se encuentra el servidor Backend API REST el cual expone al mundo exterior los endpoints con las funcionalidades descritas en el capítulo anterior. La responsabilidad es de escuchar peticiones HTTP, llamar a la lógica de negocio a través de las interfaces del paquete *Tarea.BusinessLogic.Interfaces* y retornar una respuesta. Este proceso debe cumplir con la arquitectura REST.



Tarea.Tools: Incluye funcionalidad que es usada por múltiples paquetes. como puede ser la verificar si una string es una dirección email válida, la clase *TareaException* que hereda de *Exception* la cual permite empaquetar un mensaje y una lista de argumentos dentro de una excepción.



EmailValidation: Clase static la cual tiene un único método static que devuelve true cuando el string pasado por parámetro es de tipo email, retorna false en otro caso.

TareaException: Clase que hereda de System.Exception la cual define una excepción que se le puede pasar una lista de argumentos del tipo string. Esto es usado para comunicar la capa lógica con la de WebApi.

TareaExpectedException: Esta clase es utilizada por los paquetes de prueba unitaria cuando se debe comprobar que el resultado de la prueba es una excepción de determinado tipo y su mensaje corresponde al esperado.

Por ejemplo:

```
[TestMethod] //Empty request data
[TareaExpectedException(typeof(TareaException), "ERR_REQUEST_DETAILS_NULL_EMPTY")]
```

Quiere decir que este test debe retornar una excepción del tipo TareaException y su mensaje debe ser “ERR_REQUEST_DETAILS_NULL_EMPTY”.

Se decidió crear esta clase porque el tag ExpectedException convencional no compara el mensaje de la excepción lo que no es lo ideal.

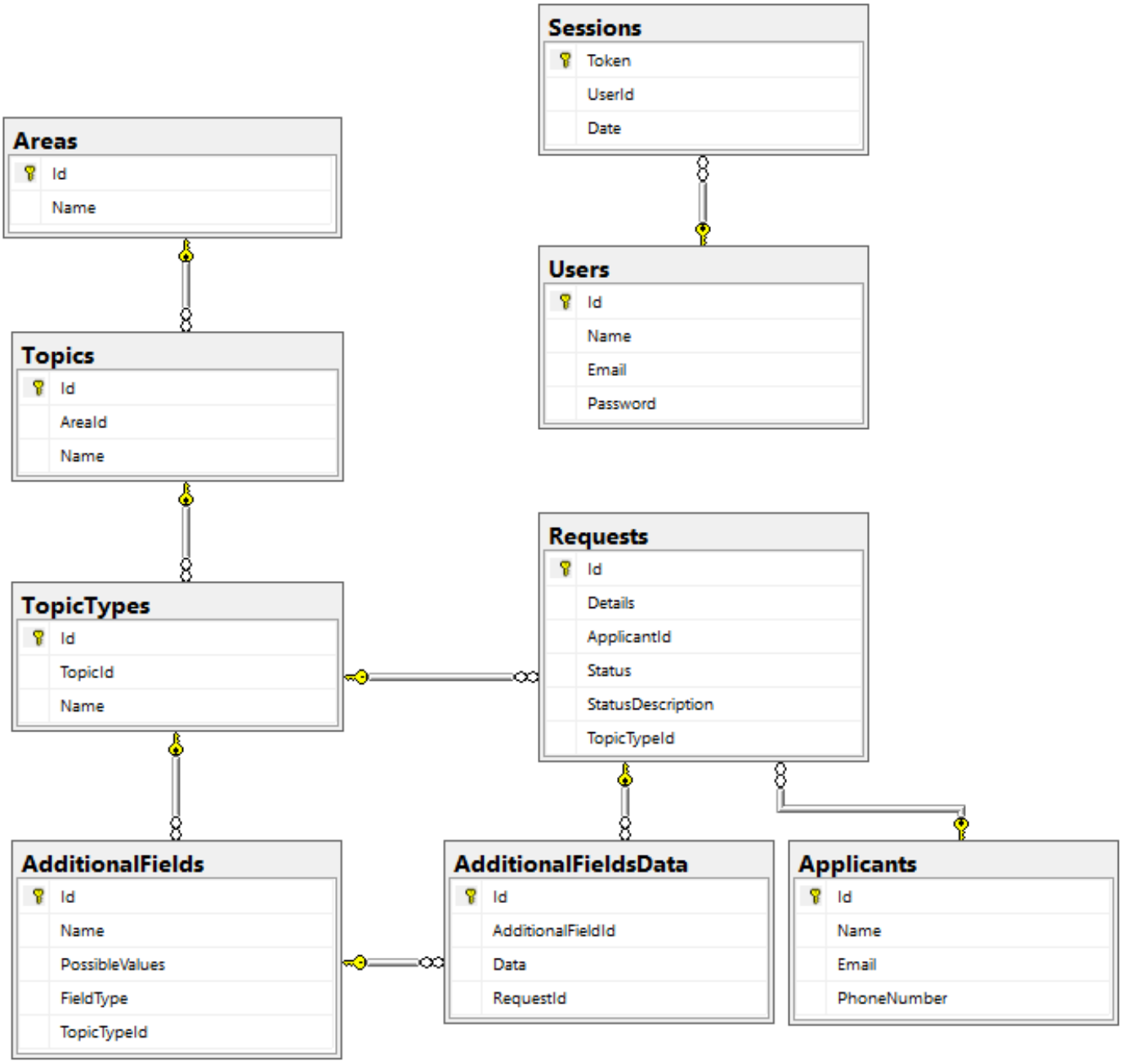
3. Jerarquías de herencia

Se utilizó polimorfismo para las clases del repositorio, para esto se define una interfaz *IRepository* con las operaciones comunes de la base de datos: *Add, Remove, Update, Get, GetAll y Save*. Esta interfaz es la utilizada por la capa BusinessLogic para acceder a los datos. La clase abstracta *BaseRepository* implementa la interfaz y define como abstractos los métodos *Get* y *GetAll* los cuales van a ser implementados por las clases del repositorio mostradas en el diagrama del paquete Tarea.Repository, estas clases heredan de BaseRepository y definen un comportamiento diferente al momento de obtener datos de la base.

Por ejemplo, la clase *RequestRepository* implementa métodos de la clase *BaseRepository:*

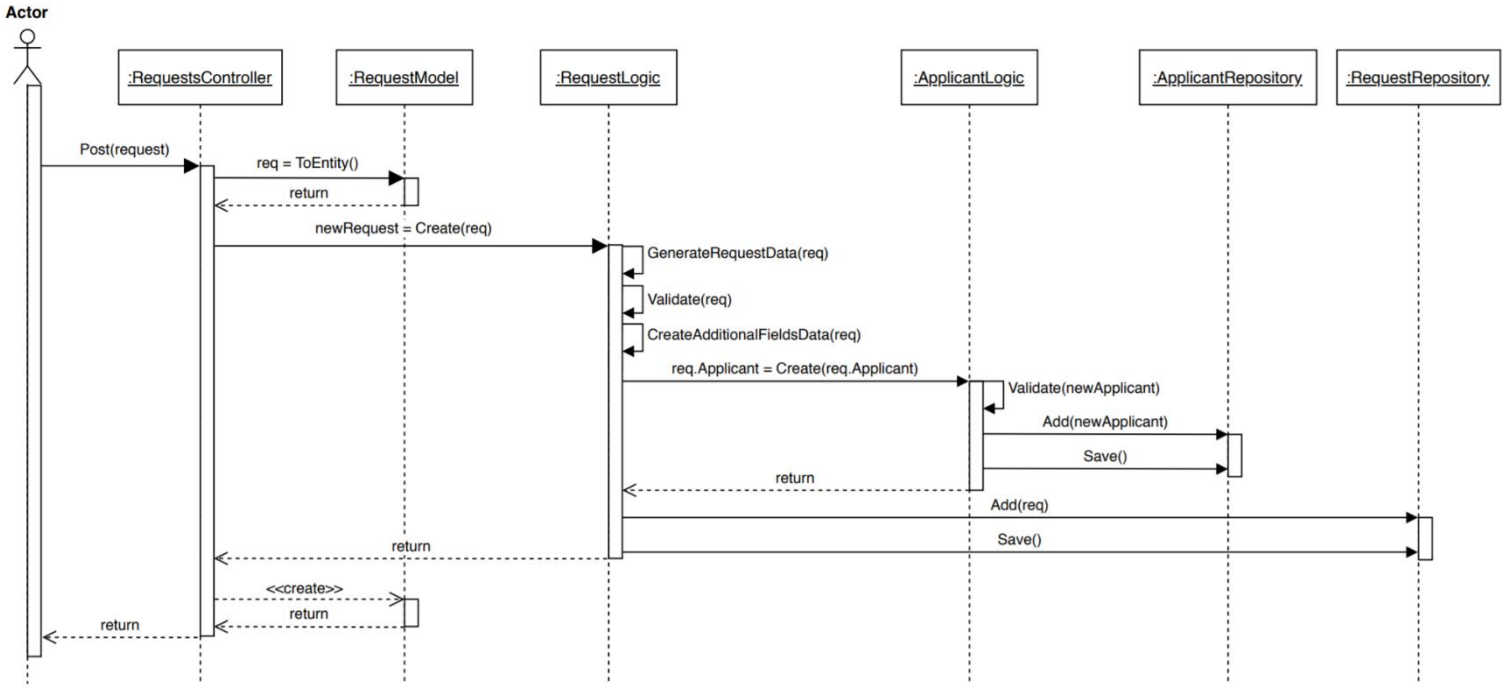
```
public class RequestRepository : BaseRepository<Request>
{
    0 references
    public RequestRepository(DbContext context)
    {
        Context = context;
    }

    67 references
    public override Request Get(Guid id)
    {
        return Context.Set<Request>().Include(r => r.Applicant)
            .Include(r => r.TopicType)
            .Include(r => r.AdditionalFields).ThenInclude(afd => afd.AdditionalField)
            .FirstOrDefault(x => x.Id == id);
    }

    32 references
    public override IEnumerable<Request> GetAll()
    {
        return Context.Set<Request>().Include(r => r.Applicant)
            .Include(r => r.TopicType)
            .Include(r => r.AdditionalFields).ThenInclude(afd => afd.AdditionalField)
            .ToList();
    }
}
```

4. Estructura de la base de datos



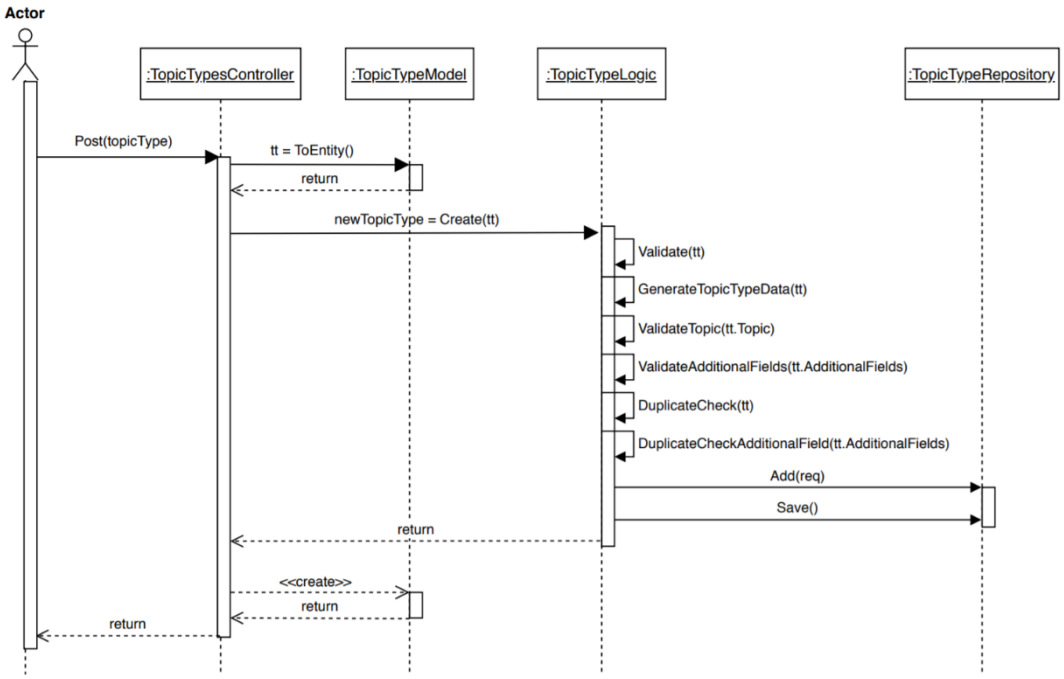
5. Implementación de las funcionalidades

6.1. Ingreso de una nueva solicitud:



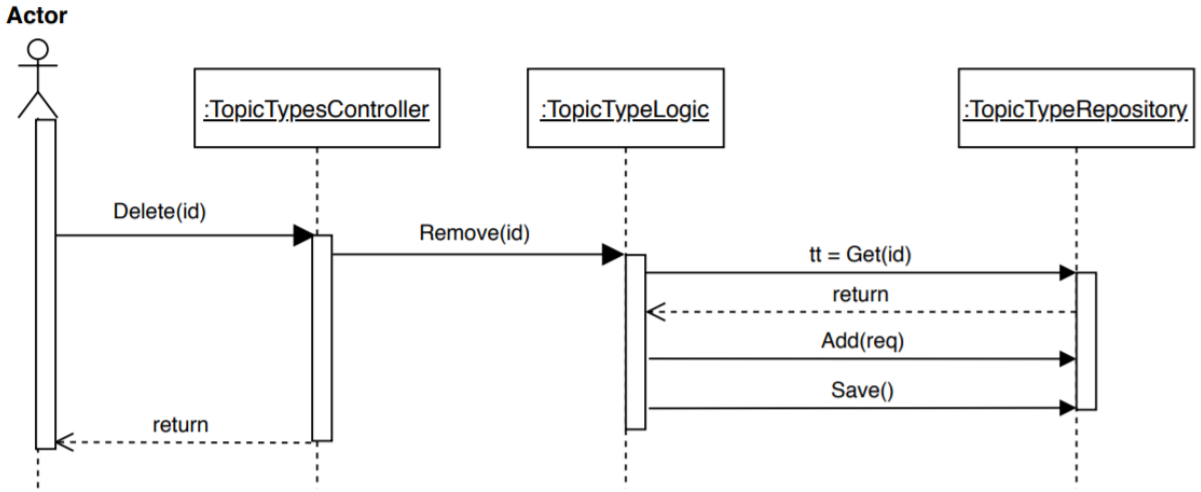
Clases involucradas: RequestController, RequestModel, RequestLogic, RequestRepository, ApplicantLogic, ApplicantRepository, TopicTypeLogic, TopicTypeRepository, AdditionalFieldDataLogic, AdditionalFieldDataRepository.

6.2. Crear un tipo:



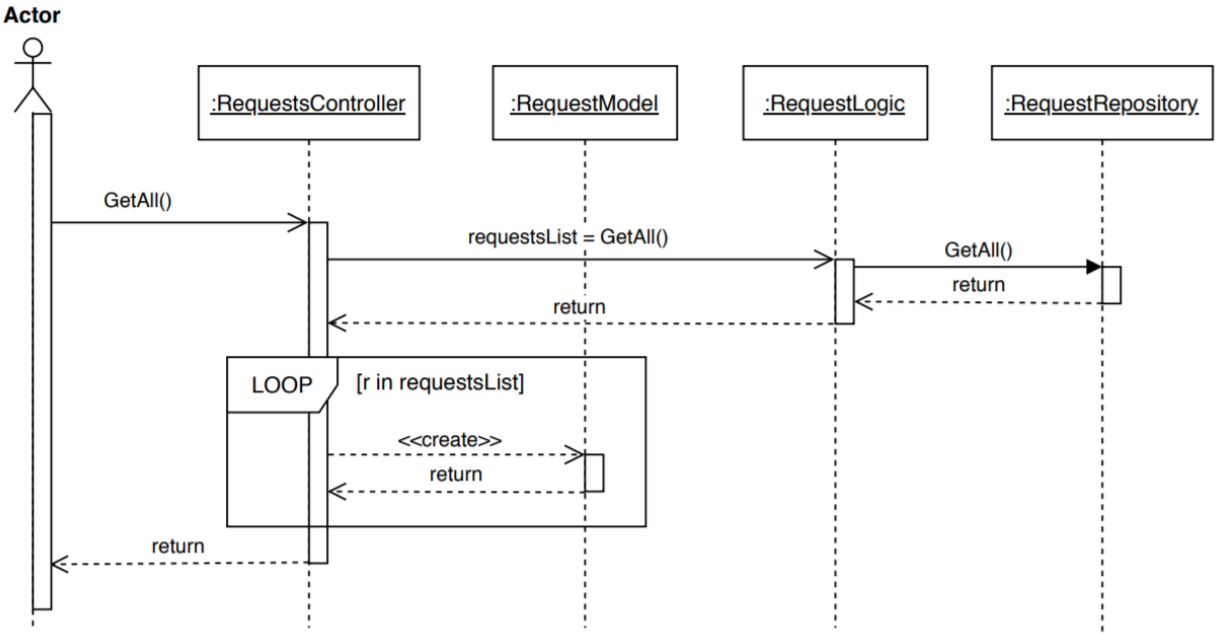
Clases involucradas: TopicTypeController, TopicTypeModel, TopicTypeLogic, TopicTypeRepository, TopicLogic, TopicRepository, AdditionalFieldLogic, AdditionalFieldRepository.

6.3. Borrar un tipo:



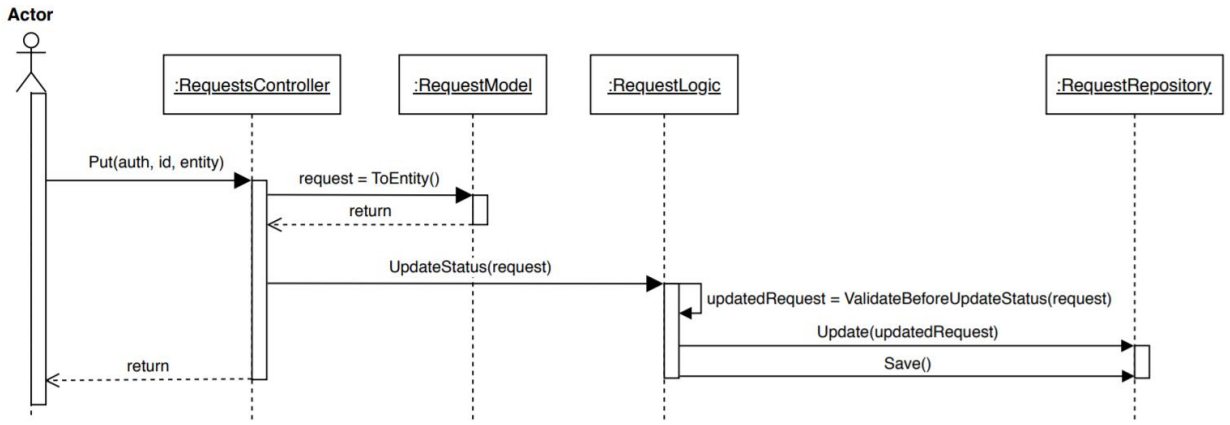
Colaboraciones: TopicTypesController, TopicTypeLogic, TopicTypeRepository.

6.4. Listar solicitudes existentes:



Colaboraciones: RequestController, RequestLogic, RequestRepository, RequestModel.

6.5. Cambiar el estado de una solicitud:



Colaboraciones: RequestsController, RequestModel, RequestLogic, RequestRepository.

7. Justificación de diseño

Inyección de dependencias: Se utilizó la técnica de Inyección de dependencias para instanciar las clases de la lógica y del repositorio desde la capa WebApi, de esta forma desacoplamos las capas de tal forma que, por ejemplo, la lógica no se encarga de crear instancias del repositorio. Esto trae muchos beneficios como facilitar el testeo, mejorar la calidad de código haciéndolo más fácil de leer y de modificar y cumplir con buenas prácticas de programación descritas en los principios SOLID.

GRASP - Algunos de los criterios utilizados para asignar responsabilidad a las clases fueron:

Bajo acoplamiento: Se intentó minimizar la cantidad de dependencias de las clases, para esto se utilizó la Inyección de dependencias

Alta cohesión: Los controladores contienen endpoints asociados a un solo tipo de dato, por ejemplo, *RequestsController*, procesa solo *Requests*. De forma similar para las clases de la lógica y del repositorio se encargan de manejar datos de un solo tipo.

Polimorfismo: Las clases *ApplicantLogic, AdditionalFieldDataLogic, AdditionalFieldLogic* implementan la interfáz *ILogic*, de esta forma depende del tipo de objeto el método que va a ser ejecutado. De forma similar para las clases del repositorio heredan de la clase abstracta *BaseRepository* y todas definen distinto comportamiento para sus métodos.

Controller: Se utilizaron controladores en la capa de WebApi de manera tal que recibe la información por medio de las peticiones y no saben bien qué hacer con esta información, sino que se llama a otras clases (BusinessLogic) para que hagan el trabajo pesado de procesar esta información.

SOLID - Se tomaron en cuenta los siguientes principios para mejorar el diseño:

SRP: Se intentó asignar la menor cantidad de responsabilidades a las clases, de esta forma decrementa la probabilidad de cambiar en el futuro potencialmente afectando a todos los que la usan. Ver Inyección de dependencias.

OCP: Se intentó que el código sea lo más abierto a la extensión y cerrado a la modificación para que si en el futuro se quiere agregar una nueva funcionalidad se modifique lo menos posible el código existente y que se implemente creando nuevas clases. La Inyección de dependencias ayuda a este principio.

LSP: En ningún momento se verifica si determinado objeto es de determinada clase, métodos que utilizan referencias a clases base pueden usar objetos de clases derivadas sin saberlo.

ISP: Si los controladores utilizan alguna interfaz, dan uso de todos los métodos definidos.

Mecanismo de acceso a datos: Se utilizó Entity Framework Code First para almacenar los datos del sistema en una base de datos SQL Server. Para esto se define la clase *TareaContext* del paquete *Tarea.Repository* la cual define el contexto a ser persistido, esto es, los DbSet de las clases del Dominio. Como se detalló en el punto Jerarquías de Herencia existe una interfaz llamada IRepository la cual define los métodos de acceso a estos DbSets los cuales son implementados por la clase *BaseRepository* y sus clases hijas, luego, la capa BusinessLogic es la que utiliza esta interfaz.

Manejo de Excepciones: BusinessLogic se encarga de procesar los datos provenientes de los controladores, si se encuentra algún tipo de error se retorna una excepción del tipo *TareaException* con un código de error y una lista de argumentos del tipo string. La clase *TareaException* hereda de *System.Exception* y se encuentra en el paquete *Tarea.Tools.*

```
throw new TareaException("ERR_REQUEST_MISSING_ADDITIONALFIELD", af.Id.ToString());
```

Esta excepción es procesada en el paquete de WebApi por un filtro de excepciones llamado *ExceptionFilter*. Este filtro es una clase que se encuentra en *Tarea.WebApi.Filters* la cual implementa las clases *Attribute* para que se pueda utilizar como data annotation y la clase *IExceptionfilter*. Esta clase se encarga de manejar cualquier error que ocurra al momento de procesar algún endpoint, para esto, crea un objeto del tipo *ErrorModel* el cual contiene un código de error, un número de status y un campo Details para mostrar información extra sobre el error.

```
2 references
public string ErrorCode { get; set; }
4 references
public int Status { get; set; }
2 references
public string Details { get; set; }
```

Adicionalmente consta con un constructor que toma como parámetro un objeto de tipo Exception, el cual asigna valores a los atributos de la clase dependiendo de la excepción.

Por ejemplo, si durante la creación de una nueva Request salta la excepción del tipo *TareaException* con código de error “ERR_REQUEST_DETAILS_NULL_EMPTY”, el constructor asigna los atributos:

ErrorCode = “ERR_REQUEST_DETAILS_NULL_EMPTY”

Status = 400

Details = "Request.Details required, max 2000 characters"

Finalmente, el filtro de excepciones modifica *Context.Response* de tal manera que en el body asigna el objeto creado del tipo *ErrorModel*, y asigna el StatusCode con el valor del atributo Status del objeto.

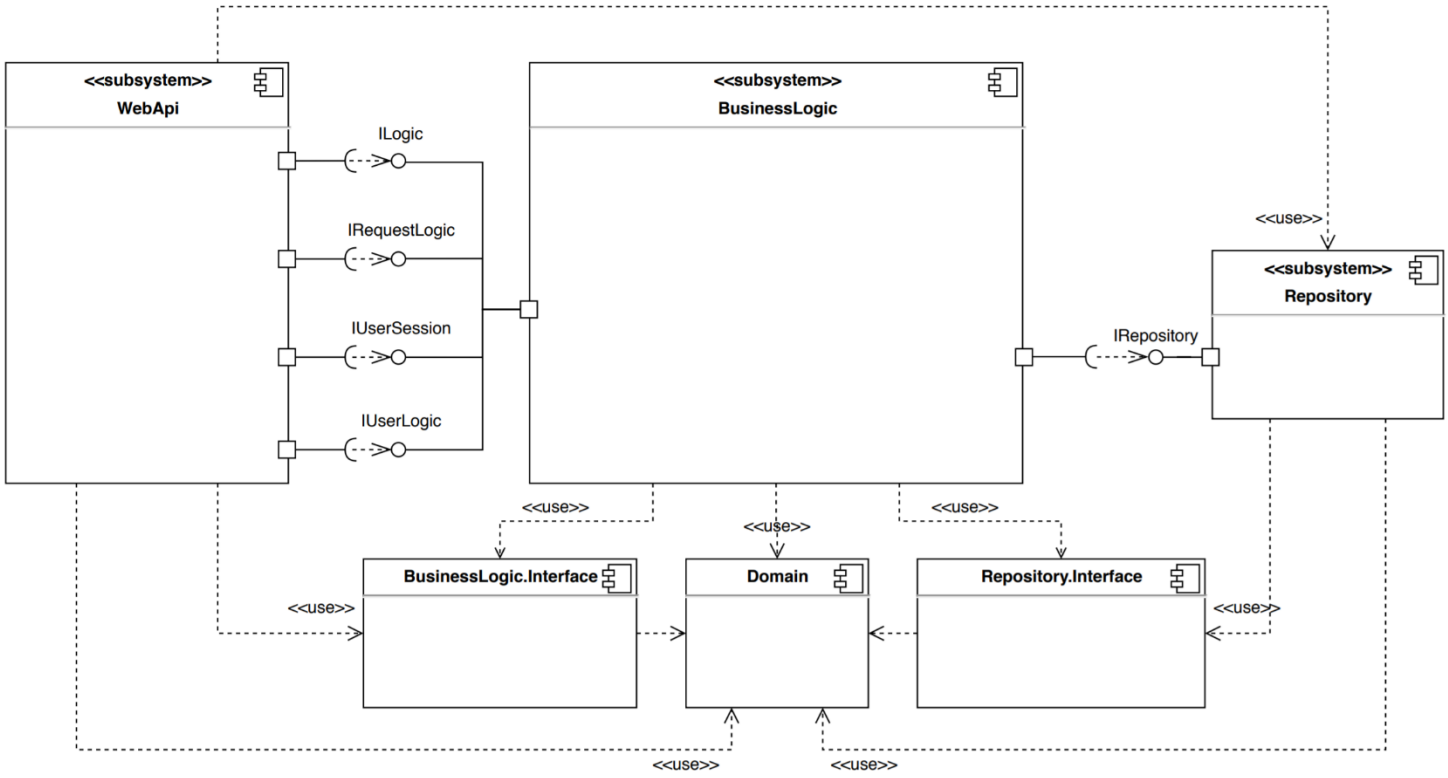
En caso de que ocurra una excepción de tipo diferente a *TareaException*, se retorna el Status 500 y el mensaje de la excepción. Esto no es lo ideal ya que se puede mostrar información importante sobre el servidor al usuario.

En caso de que ocurra un error en la base de datos, ya sea al momento de conectarse o usarla, se muestra un error con el siguiente formato:

```
{
    "errorCode": "A network-related or instance-specific error occurred while establishing a connection to SQL Server. The
        server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is
        configured to allow remote connections. (provider: Named Pipes Provider, error: 40 - Could not open a connection to
        SQL Server)",
    "status": 500,
    "details": "Error connecting to the database."
}
```

Este mecanismo de manejo de excepciones permite mostrar al usuario información extra sobre un error ocurrido al momento de realizar la petición, adicionalmente consolida el manejo de errores en un solo lugar el cual facilita al momento mantener el código. Adicionalmente impide que la capa de la lógica muestre información directamente al usuario, esta tarea la hace la WebApi.

8. Diagrama de componentes



El componente de la WebApi es el responsable de escuchar peticiones HTTP, procesarlas y enviar una respuesta, para esto se comunica con el componente de la lógica a través de interfaces. La lógica interactúa con la interfaz del repositorio que la provee el componente Repository, donde se encuentra el contexto de la base de datos y los manejadores para operar sobre la misma.