Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2

Obligatorio 1

BlogsApp

Fernando Spillere - 274924

Gimena Alamón - 243518

Carmela Sotuyo - 186554

Mayo 2023

Índice

1 Introducción **3**

1.1 Descripción General 3

1.2 Interfaces 3

1.3 Inyección de dependencias 4

1.4 Consideraciones de letra 4

2 Arquitectura de la solución **6**

2.1 Descripción General 6

2.2 Paquetes del dominio 7

2.3 Lógica de Negocio 8

2.4 API RESTFul 9

2.5 Base de datos 11

2.6 Manejo de exepciones 12

2.7 Tests 12

3 Diseño base de datos **15**

4 Diagramas de interacción **17**

Introducción

1.1 Descripción General

La solución que presentamos en este documento se basa en los requerimientos identificados de la letra así como en el foro de la materia. El código en general, así como nombres de entidades y componentes fue realizado en inglés. Se siguen los estándares, buenas prácticas, uso de patrones y recomendaciones que entendemos hacen a un código más legible, re utilizable y sencillo de entender. A continuación se detallan algunas consideraciones relevantes.

1.2 Interfaces

Se implementó la solución aplicando el principio de segregación de interfaces (Interface Segregation Principle, ISP) de los principios SOLID de diseño de software. Aplicando este patrón buscamos promover la construcción de software más modular y fácil de mantener a largo plazo.

Se definen interfaces tanto para la Lógica de Negocio (BlogsApp.IBusinessLogic) como para el acceso a datos (BlogsApp.IDataAccess) tratando de que sean específicas y que contengan solo lo necesario, de manera que los componentes que las usan solo necesiten conocer la firma de los métodos que utilizan. Destacamos que las clases que implementan las interfaces usan todos sus métodos, En el caso de Business Logic dado se crea una interfaz por cada entidad (Article, User, Comment, Reply, Session). Y en el caso de DataAccess además de poseer una interfaz por cada entidad, también estas implementan la interfaz IRepository, la cual dispone de métodos comunes al resto.

Partiendo de lo anterior logramos una mayor flexibilidad en la construcción de la aplicación, ya que habilita que los módulos y clases sean más independientes y abiertas al cambio, ya que se podrá en gran medida agregar nuevos comportamientos o modificar los existentes sin afectar el resto del sistema. A su vez se logró verificar cómo también facilita el testing del software.

1.3 Inyección de dependencias

Siguiendo los principios SOLID se implementa el principio de inyección de dependencias (Dependency Injection Principal) para el cual se utilizó el framework que nos ofrece .Net para C#.

Para lograr implementarlo, se crea el módulo ServiceFactory (BlogsApp.Factory/ServiceFactory.cs), el cual será esencial en el funcionamiento. Esta ”fábrica de servicios” representa una capa de abstracción entre la aplicación y los servicios que utiliza, brindando flexibilidad en la definición de la misma.

En términos generales, como se explicó anteriomente, se definen interfaces que describen las funcionalidades que se esperan de los diferentes servicios (y las clases implementan cada intefaz), y la aplicación utiliza la fábrica de servicios para crear instancias de los servicios que necesita en tiempo de ejecución.

La principal ventaja de este enfoque es que la aplicación no está vinculada a ninguna implementación específica de los servicios que necesita. En cambio, la fábrica de servicios determina dinámicamente qué instancia de servicio se debe crear en función de la configuración de la aplicación. Esto permite que la aplicación sea más flexible y fácil de mantener.

La inyección de dependencias es una técnica común que se utiliza en conjunción con el patrón de Service Factory para proporcionar una forma aún más flexible de configurar los servicios. En lugar de crear instancias de servicios directamente en la aplicación, se pasan instancias de servicio a través de constructores o propiedades. Esto permite que la aplicación sea aún más modular, ya que cada servicio se puede reemplazar fácilmente por otro sin tener que cambiar el código de la aplicación.

1.4 Consideraciones de letra

Se detalla a continuación información relevante para el uso de la aplicación, así como consideraciones de letra y posibles mejores para siguiente iteración:

- La búsqueda de texto en el endpoint de article es Case sensitive.

- Cuando un usuario no admin quiere actualizar su usuario para convertirse en admin, el sistema lo permite. Se considera una posible mejora.

- Cuando se realiza una request al endpoint que devuelve la cantidad de artículos por mes, los meses que no tienen artículos no se incluyen. Puede resultar confuso para el usuario dado que los meses no están identificados.

- Para que el usuario realice log out debe conocer el id de la sesión. Este dato no es accesible por el usuario. Para un próxima iteración se considera relevante que este dato no sea necesario para cerrar la sesión.

- El endpoint de ranking de usuarios no está devolviendo los valores correctos. Se tendrá en cuenta para próxima iteración.

- Las replies no se están vinculando correctamente al comentario, es por un bug de lógica identificado, que se espera corregir próximamente.

Arquitectura de la solución

2.1 Descripción General

La solución costa en los siguientes paquetes:

El paquete WebAPI contiene todos los controladores (Controllers), los filtros (Filters), Objetos de transferencia de datos (DTOs) y el modelo de estadísticas (StatsModel.cs). Este paquete se comunica con la lógica mediante el paquete de Interfaz de lógica (IBusinessLogic), el cual expone interfaces dependiendo de la entidad del dominio con la cual se trabaja. Luego la lógica contiene la implementación de los métodos expuestos por las interfaces, y se comunican con la base de datos mediante las interfaces expuestas por el paquete de IDataAccess/Interfaces. Luego en Domain se encuentran las entidades que componen la solución, y Service factory se encarga de la inyección de dependencias para la lógica y la API. A su vez, también existen 3 proyectos de tests que corresponden a la API, Lógica, Base de datos y Dominio.

Diagrama de Paquetes

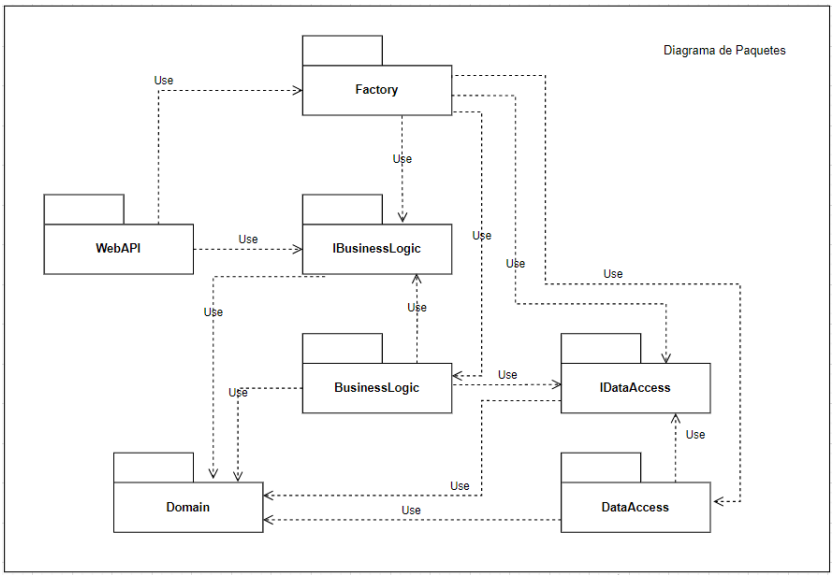

Figure 2.1: Diagrama de paquetes

2.2 Paquetes del dominio

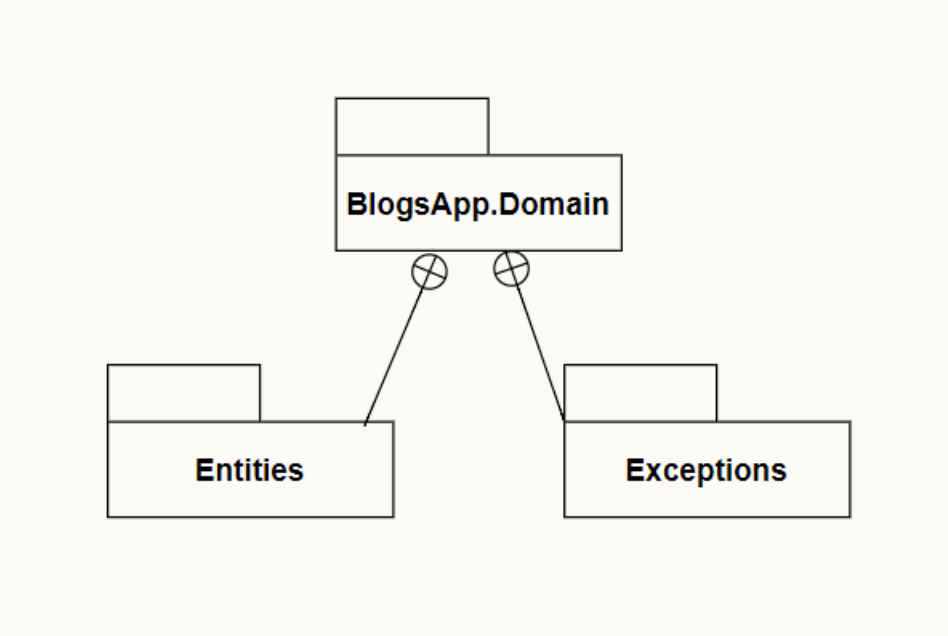

Figure 2.2: Diagrama de paquetes - Dominio

En el paquete Domain se definen:

- Las entidades clave de la solución, como ser las entidades de negocio (Dentro de ../Entities).

- Las excepciones relacionadas a errores de datos y lógica. Aquí se definieron BadInputException, InterruptedActionException y ExistenceException y NonExistantImplementationException (Dentro ../Exceptions)

Diagrama de objetos del dominio:

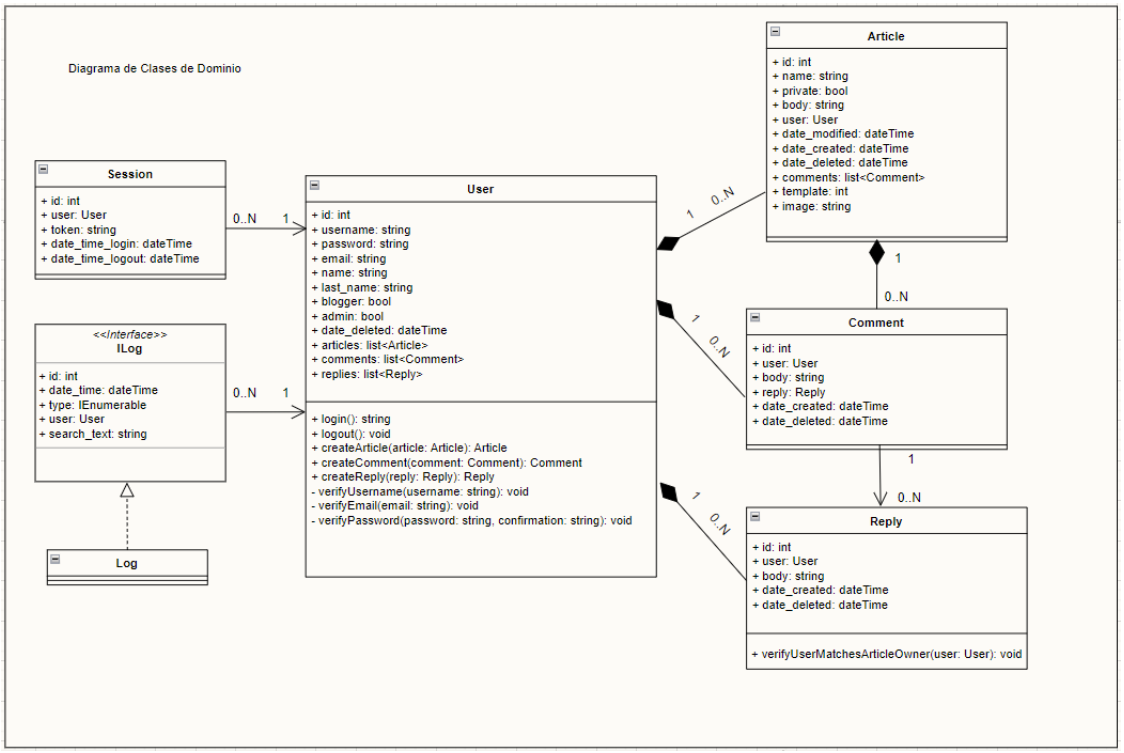

Figure 2.3: Diagrama de dominio

2.3 Lógica de Negocio

Paquetes de la lógica:

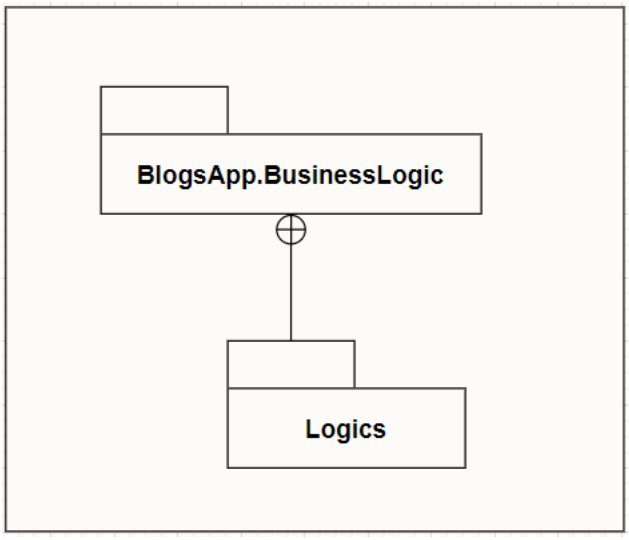

Figure 2.4: Diagrama de paquetes - Lógica e iterfaces

Clases de lógica y sus relaciones con las interfaces:

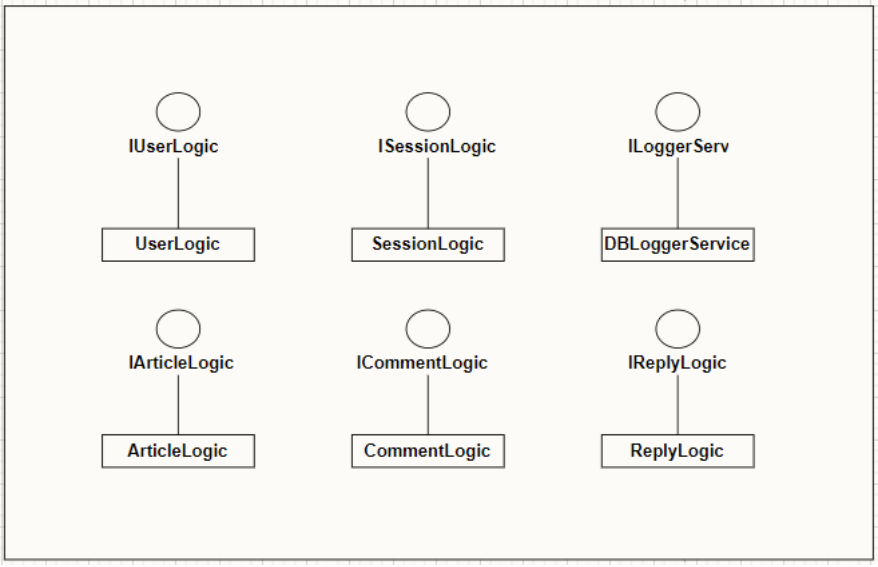

Figure 2.5: Diagrama de paquetes - API

2.4 API RESTFul

En cuanto a la arquitectura de la API se define un proyecto WebAPI, el cual contiene los siguientes paquetes:

- Controllers - DTOs - Filters

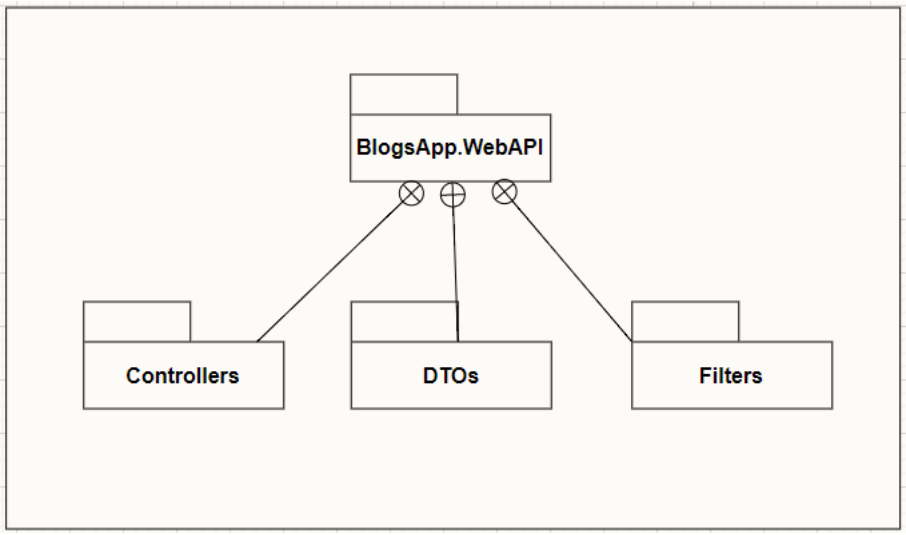

Figure 2.6: Diagrama de paquetes - Lógica

Existe un controller por cada recurso, el cual se encarga de gestionar las peticiones relacionadas al mismo, utilizando las interfaces de la lógica para cumplir con las funcionalidades.

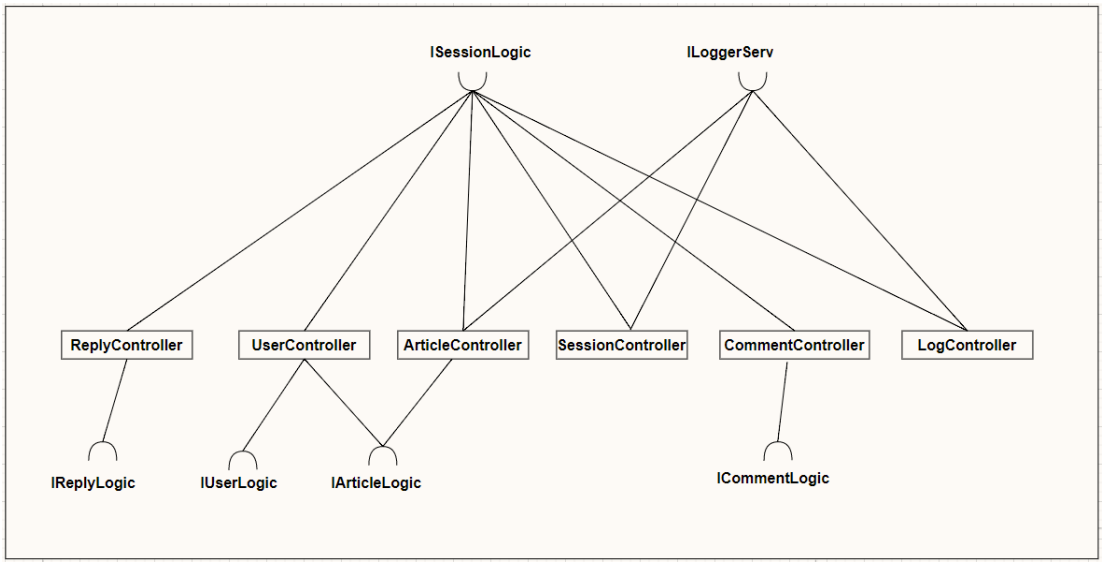

Figure 2.7: Diagrama de paquetes - Controles e interfaces

Con el objetivo de proteger el dominio pero también de no depender de información innecesaria al momento de crear o actualizar los recursos, se crean los siguientes DTOs:

- CreateUserRequestDTO

- UpdateUserRequestDTO

- ArticleConverter

- CreateArticleRequestDTO

- LoginRequestDTO

- LoginResponseDTO

- UpdateArticleRequestDTO

- UpdateUserRequestDTO

Sobre los filtros, se implementaron ”AuthorizationFilter”, el cual tiene la responsabilidad de

gestionar las sesiones de los usuarios, es decir, otorga Tokens de inicio de sesión a cada usuario , y evalúa en cada petición que el mismo se encuentre logueado en el sistema; y ”Exceptionfilter”, el cual se encarga del manejo de excepciones que surjan resultado de la ejecución de peticiones. Dependiendo del error que surja, retorna un mensaje y código de error.

2.5 Base de datos

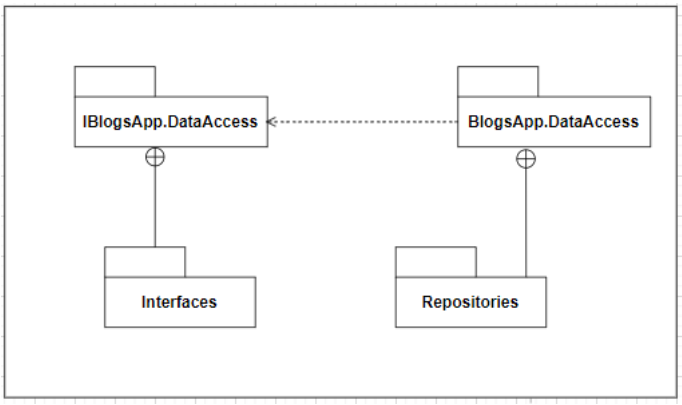

Figure 2.8: Diagrama de paquetes - DataAccess

Se define el paquete IDataAccess, el cual contiene las interfaces que representarán el nexo entre los repositorios y la lógica de negocio. Con el objetivo de minimizar el código repetido se crea una interfaz IRepository, de la cual implementan el resto de interfaces métodos que son comunes a todas.

También se encuentra el paquete Exceptions, donde se encuentran ”AlreadyExistsDBException” y ”NotFoundDBException”, representando las posibles excepciones que podrían surgir.

Por otro lado, la definición de los repositorios que implementan las interfaces comentadas anteriormente se encuentra en el proyecto DataAccess, el cual además posee Migrations y Context.

2.6 Manejo de exepciones

Con el objetivo de una comunicación clara y lo más efectiva posible hacia el cliente, se crean algunas excepciones que representan los errores más comunes o esperados por parte de la aplicaicón.

Se detallan a continuación:

Excepciones a nivel de lógica:

- BadInputException: Representa que el error es generado por el cliente, por ejemplo, si en el body se olvidó de agregar un dato obligatorio, o se equivoca en el tipo de datos de un atributo.

- ExistenceException: Representa que no se pudo encontrar la información que se solicitó. Por ejemplo si de adiciona un id en la URI pero es imposible para la lógica obtener el recurso.

- ActionInterruptedException: Representa un error que hace interrumpir el procesamiento de la solicitud.

Excepciones a nivel de Acceso a datos:

- NotFoundException: Representa que no se pudo encontrar el recurso o información solicitada. - AlreadyExistsDBException: Reprensenta que el objeto ya existe, por ejemplo si se desea dar de alta un usuario que ya fue registrado.

Excepciones a nivel de API:

Como se comentó en secciones anteriores, para el manejo de las excepciones relacionadas a la API, se encarga el ExceptionFilter, el cual, cuando se dispara una excepción, retorna una response con el código y mensaje que corresponda según la excepción, y corta la ejecución.

2.7 Tests

En la solución se incluyen tres paquetes con la responsabilidad de testear la API, BusinessLogic y DataAccess respectivamente. Los mismos contienen tests automáticos para cada una de sus

clases y su definición que hizo siguiendo TDD (Test Driven Development).

Se utilizó Mocks para simular objetos y comportamiento, y así poder generar tests específicos e independientes, evitando tener que manejar las dependencias que existen entre los paquetes. De esta forma los tests se encuentran aislados y no utilizan recursos innecesarios.

Al correr los tests de DataAccess todos juntos, sucede que algunos fallan no por un tema de definición, sino porque entendemos las ejecuciones se superponen, dejando datos inconsistentes o que se modifican entre si.

Ejemplo:

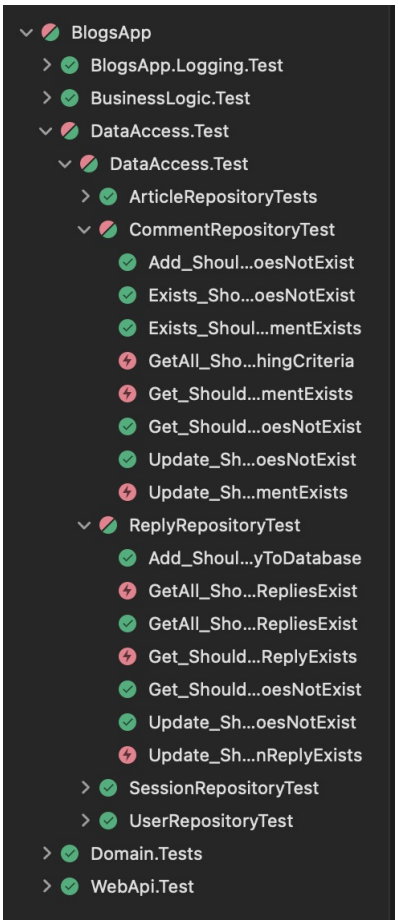

Figure 2.9: Ejecución de tests

Pero luego, al correr cada test por separado, se ejecuta sin problemas:

Figure 2.10: Ejecución de tests

Resultado final, corriendo los tests en conjunto:

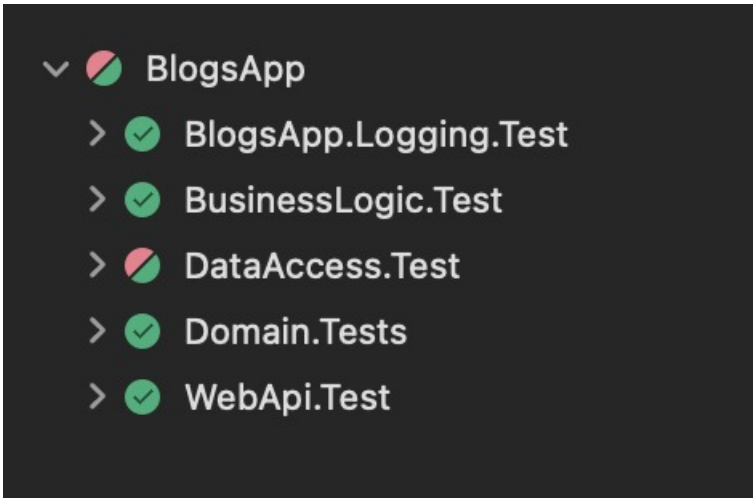

Figure 2.11: Ejecución de tests

Diseño base de datos

Se utilizó Entity Framework Core, modalidad Code-First para la definición de la Base de datos, así como para el manejo de los mismos.

Teniendo como base la defincón de las clases que representan el dominio, Entity Framework creó las tablas que contendrán la información de los objetos instanciados a partir de las clases del dominio.

Se detalla la estructura de la base de datos y la relaciones entre las tablas:

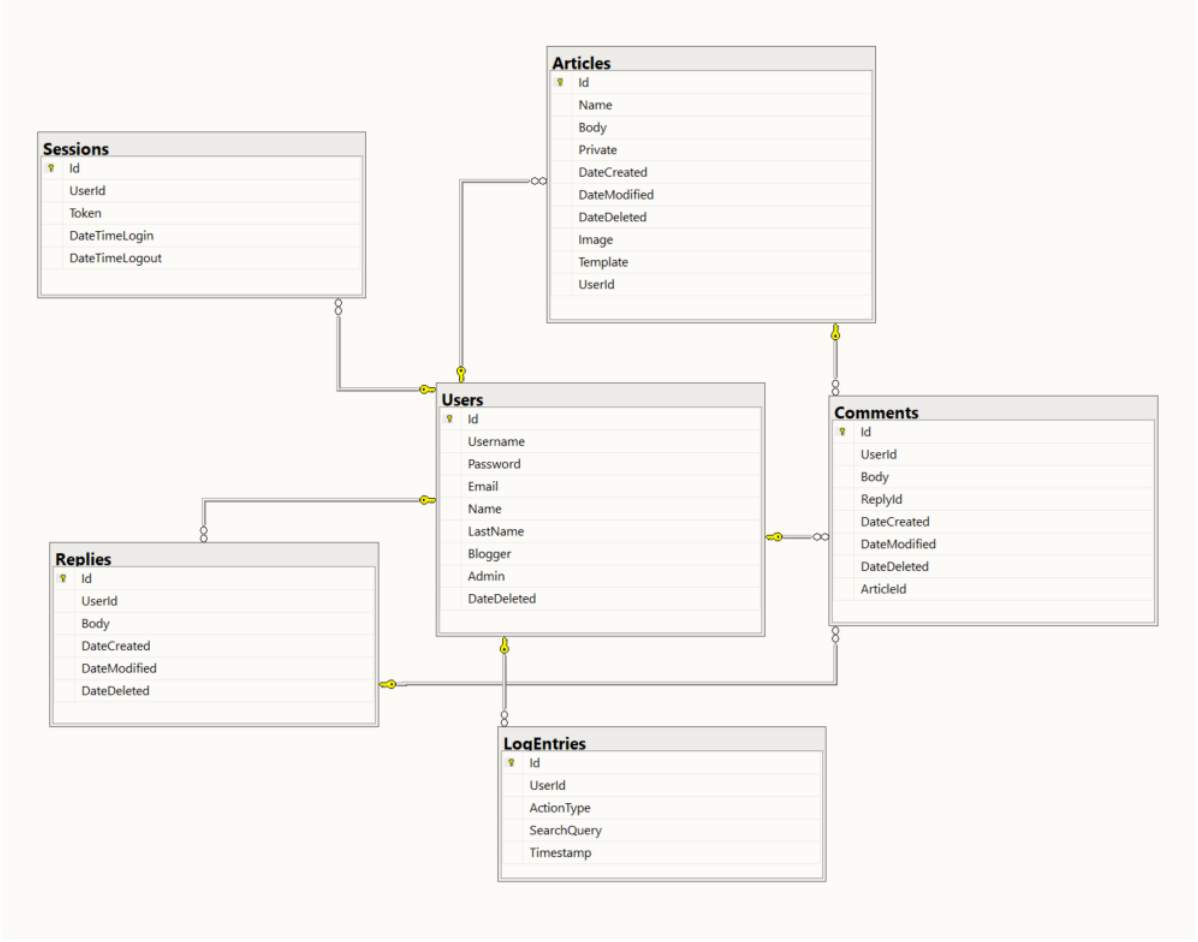

Figure 3.1: Diseño base de datos

Para el acceso a los datos desde la lógica, se definen cinco repositorios que implementan sus

respectivas interfaces. Para poder acceder a la base de datos desde los repositorios, se crea un contexto (el cual tiene el Connection String) y los DbSets, los cuales representan colecciones de las clases de nuestro dominio.

DBSets definidos:

```
public class Context : DbContext
{
    5 referencias | 0/3 pasando
    public DbSet<User> Users { get; set; }
    4 referencias | 0/3 pasando
    public DbSet<Article> Articles { get; set; }
    0 referencias
    public DbSet<Comment> Comments { get; set; }
    0 referencias
    public DbSet<Reply> Replies { get; set; }
    0 referencias
    public DbSet<Session> Sessions { get; set; }
```

Figure 3.2: DBSets en Context

Diagramas de interacción

Se presenta un diagrama de secuencia para representar la funcionalidad de eliminar un usuario.

Se considera que para este flujo puede ser importante representar cómo interactúan los objetos y los métodos que se invocan en la línea de tiempo. Destacamos que cuando un usuario es eliminado, deberán también eliminarse sus artículos creados, comentarios y respuestas realizadas.

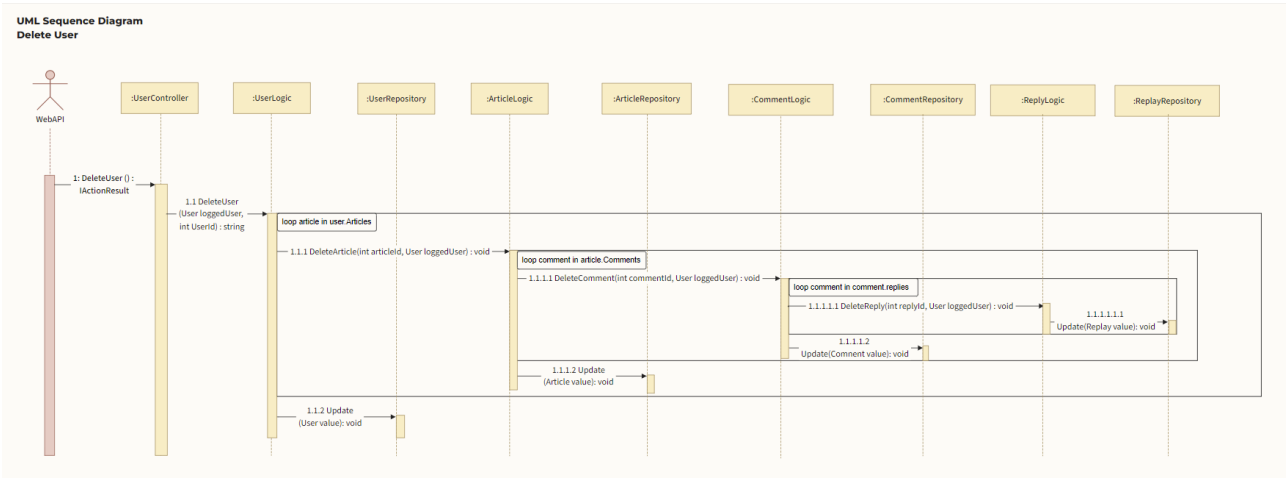

Figure 4.1: Diagrama de paquetes

Este diagrama muestra cómo la WebAPI invoca al controlador de usuario, que a su vez invoca a la lógica de negocio de usuario para eliminar al usuario. (Como se comentó en secciones anteriores la eliminación de los recursos es lógica, por tanto consta en asignar al recurso una fecha de eliminación).

La lógica de negocio de usuario actualiza la fecha de eliminación del usuario en el repositorio de mismo y, a su vez, invoca a la lógica de negocio de artículo para eliminar los artículos relacionados con el usuario.

La lógica de negocio de artículo actualiza la fecha de eliminación de los artículos en el repositorio de artículo y, a su vez, invoca a la lógica de negocio de comentario para eliminar los comentarios relacionados con los artículos.

La lógica de negocio de comentario actualiza la fecha de eliminación de los comentarios en el repositorio de comentario y, a su vez, invoca a la lógica de negocio de respuesta para eliminar las respuestas relacionadas con los comentarios.

Finalmente, la lógica de negocio de respuesta actualiza la fecha de eliminación de las respuestas en el repositorio de respuesta.