Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio 1 -Diseño de aplicaciones 2 - Descripción del diseño

Nicolas Hernandez – 229992
Francisco Aguilar – 230143

2023

https://github.com/ORT-DA2/DA2-Aguilar-Hernandez

Índice

Abstract

En este documento se puede encontrar la descripción general de la solución implementada para el obligatorio de diseño de aplicaciones 2, esa descripción es acompañada por diferentes diagramas utilizados para la fácil comprensión de la solución y algunos flujos de la misma. Aquí también se encuentra la justificación del diseño y algunos de los errores conocidos.

Descripción general

La solución consta de un servicio backend y un frontend con angular basado en un sistema de blog en el cual se permita el registro, logueo, deslogueo y modificación de usuarios. Un sistema de administración de usuarios para los usuarios con roles de Admin en donde el mismo pueda crear, modificar y eliminar usuarios del sistema y también ver los usuarios más activos entre 2 fechas definidas. También se implementó un sistema de creación , modificación y eliminación de artículos por parte de los usuarios logueados con el rol de blogger, también se permite visualizar

por otros usuarios los diferentes artículos públicos y comentar en los mismos, donde el creador del artículo también va a poder responder dichos comentarios. Los creadores de artículos también son capaces de colocar sus artículos en privados para que solo ellos sean capaces de visualizarlos. Otra característica del sistema es que se pueden buscar los artículos a partir de un texto.

Los administradores deben ser capaces también de llevar un registro de los artículos y comentarios que tienen un lenguaje ofensivo, pudiendo aprobarlos y también pudiendo editar los artículos en caso de que tengan alguna palabra ofensiva, el listado de palabras ofensivas del sistema puede ser modificado por los administradores.

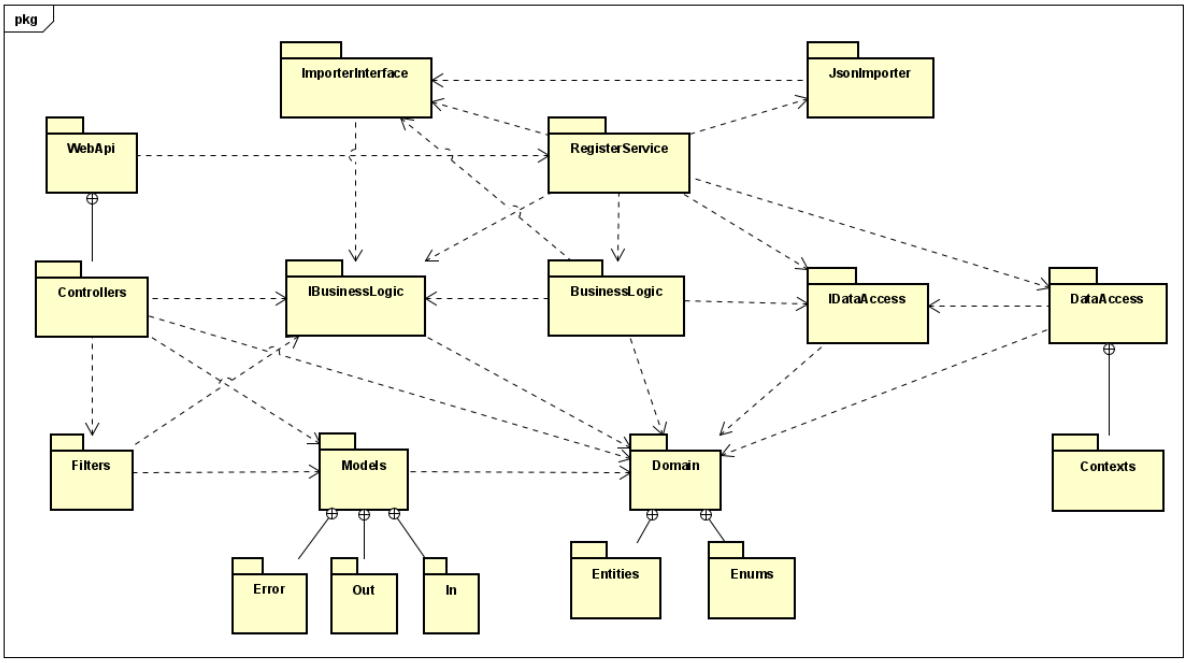
Los usuarios no logueados son solamente capaces de ver los 10 últimos artículos públicos creados en el sistema y de registrarse.

Errores conocidos

1. No se pueden responder rápidamente las notificaciones desde el frontend
2. A veces se muestra un error en el frontend que muestra [object] [object] al ver el perfil. El cual se debe a una modificación en si el perfil lo está viendo otra persona o el mismo usuario dueño del perfil.
3. Si el usuario tiene un artículo o comentario creado, falla el borrar user en el userManagement.
4. Luego de realizar algunas acciones se muestra el componente de unauthorized por error.
5. Al usar la barra de búsqueda mientra se esta en otro artículo hace que se no te lleve al artículo buscado
6. Si el archivo a importar no se le pasa un campo requerido por artículos, falla.

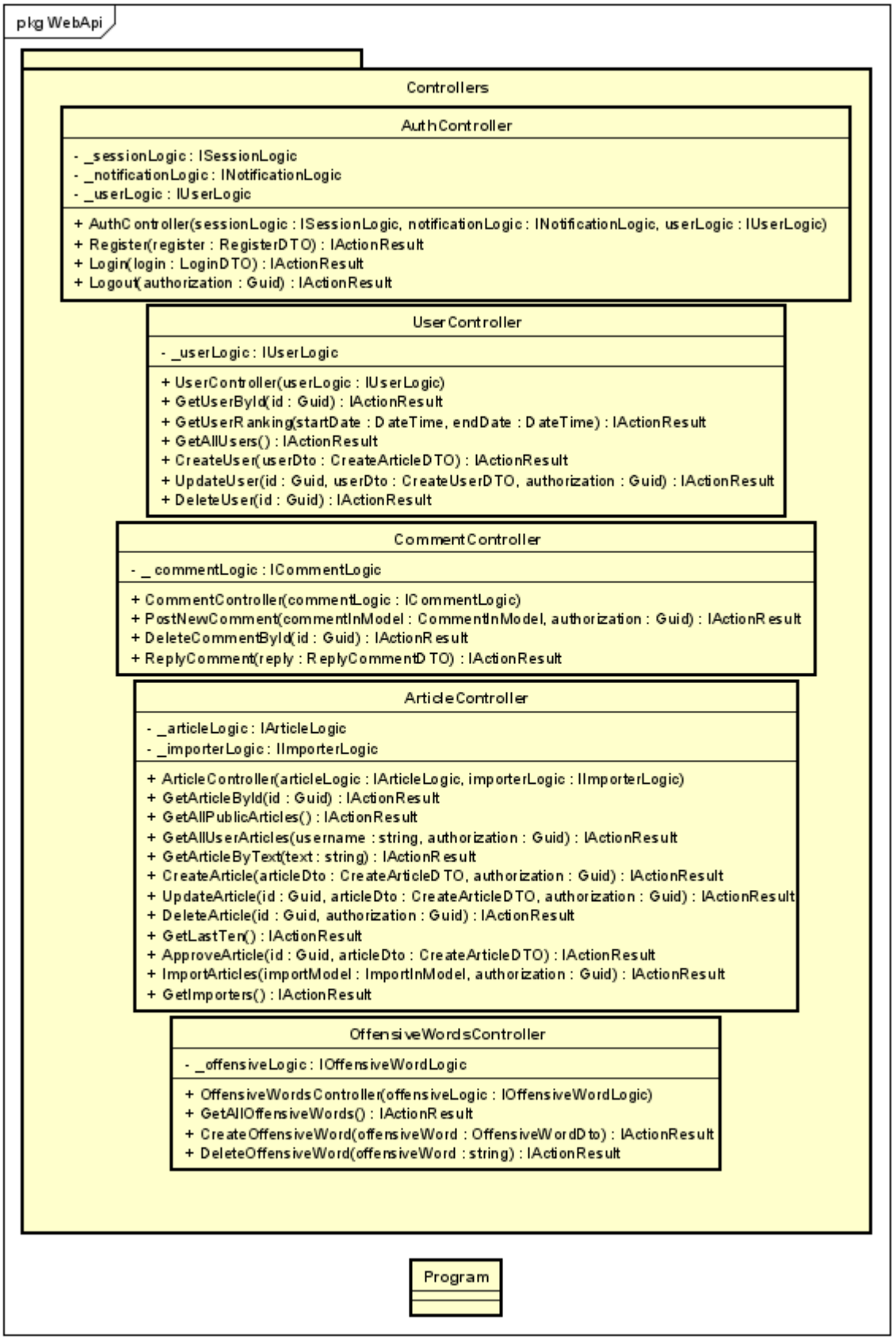
Diagrama general de paquetes

Nuestra solución se dividió en 18 paquetes los cuales cuentan con diferentes responsabilidades.



Blog.WebApi

Este paquete es el punto de ingreso a nuestro backend y es donde se encuentran todos los controladores con los diferentes endpoints.

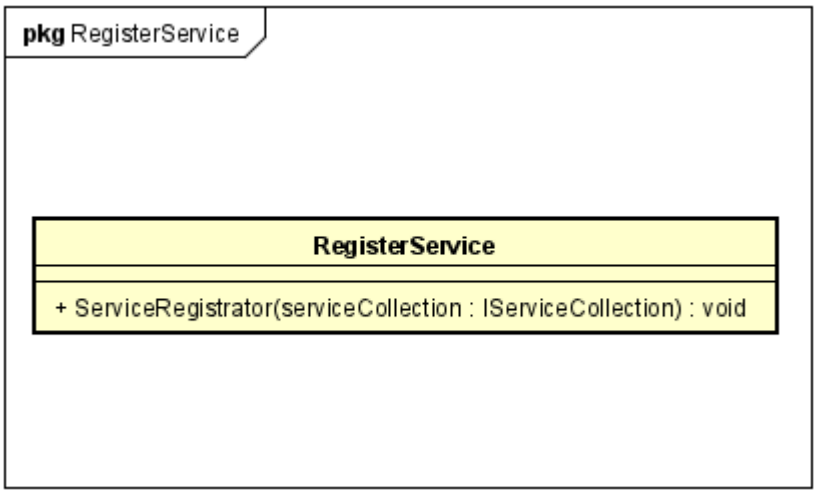

pkg WebApi
Controllers
AuthController
- _sessionLogic : ISessionLogic
- _notificationLogic : INotificationLogic
- _userLogic : IUserLogic
+ AuthController(sessionLogic : ISessionLogic, notificationLogic : INotificationLogic, userLogic : IUserLogic)
+ Register(register : RegisterDTO) : IActionResult
+ Login(login : LoginDTO) : IActionResult
+ Logout(authorization : Guid) : IActionResult
UserController
- _userLogic : IUserLogic
+ UserController(userLogic : IUserLogic)
+ GetUserById(id : Guid) : IActionResult
+ GetUserRanking(startDate : DateTime, endDate : DateTime) : IActionResult
+ GetAllUsers() : IActionResult
+ CreateUser(userDto : CreateArticleDTO) : IActionResult
+ UpdateUser(id : Guid, userDto : CreateUserDTO, authorization : Guid) : IActionResult
+ DeleteUser(id : Guid) : IActionResult
CommentController
- _commentLogic : ICommentLogic
+ CommentController(commentLogic : ICommentLogic)
+ PostNewComment(commentInModel : CommentInModel, authorization : Guid) : IActionResult
+ DeleteCommentById(id : Guid) : IActionResult
+ ReplyComment(reply : ReplyCommentDTO) : IActionResult
ArticleController
- _articleLogic : IArticleLogic
- _importerLogic : IImporterLogic
+ ArticleController(articleLogic : IArticleLogic, importerLogic : IImporterLogic)
+ GetArticleById(id : Guid) : IActionResult
+ GetAllPublicArticles() : IActionResult
+ GetAllUserArticles(username : string, authorization : Guid) : IActionResult
+ GetArticleByText(text : string) : IActionResult
+ CreateArticle(articleDto : CreateArticleDTO, authorization : Guid) : IActionResult
+ UpdateArticle(id : Guid, articleDto : CreateArticleDTO, authorization : Guid) : IActionResult
+ DeleteArticle(id : Guid, authorization : Guid) : IActionResult
+ GetLastTen() : IActionResult
+ ApproveArticle(id : Guid, articleDto : CreateArticleDTO) : IActionResult
+ ImportArticles(importModel : ImportInModel, authorization : Guid) : IActionResult
+ GetImporters() : IActionResult
OffensiveWordsController
- _offensiveLogic : IOffensiveWordLogic
+ OffensiveWordsController(offensiveLogic : IOffensiveWordLogic)
+ GetAllOffensiveWords() : IActionResult
+ CreateOffensiveWord(offensiveWord : OffensiveWordDto) : IActionResult
+ DeleteOffensiveWord(offensiveWord : string) : IActionResult
Program

Blog.Tests

Este paquete contiene los test unitarios de WebApi y de las entidades del dominio

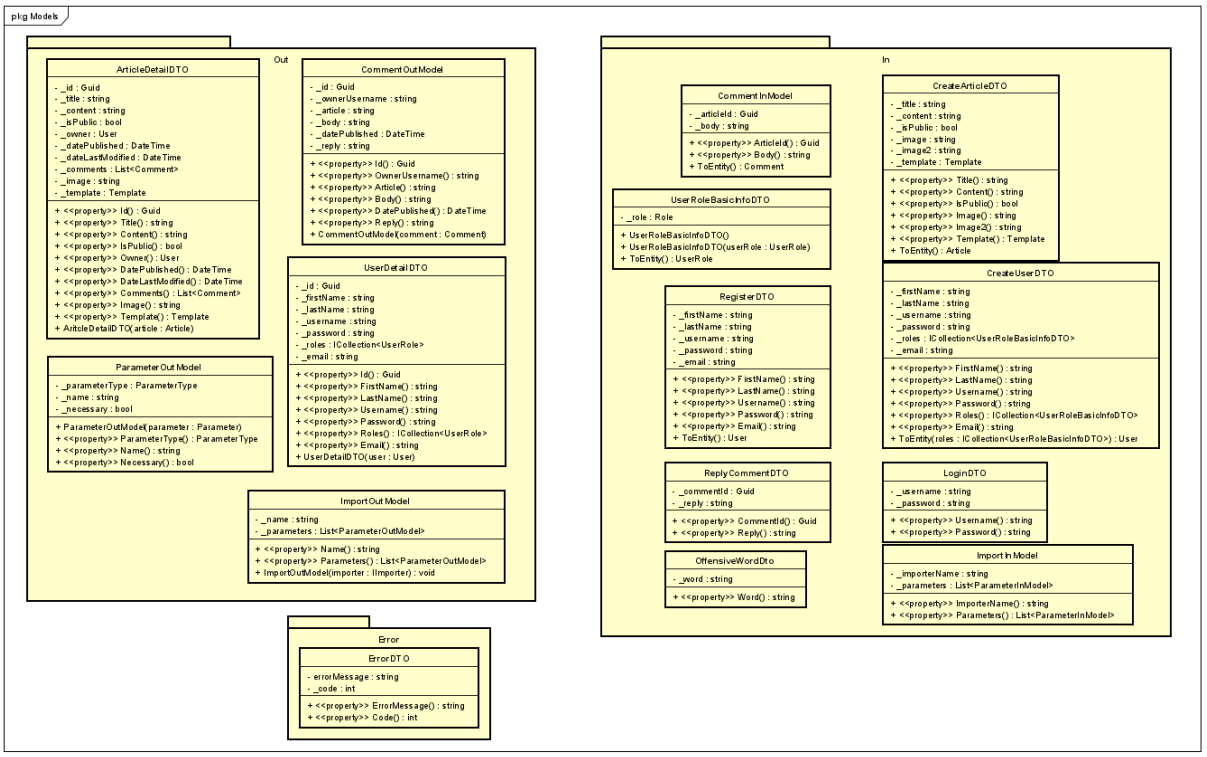
Blog.RegisterService

Este paquete es donde se realizan todas las inyecciones de dependencia, también está allí el registro del servicio del DbContext y de los filtros de autorización y de roles. Este paquete lo creamos para sacar esa responsabilidad del paquete de WebApi ya que anteriormente se realizaban en la clase program.



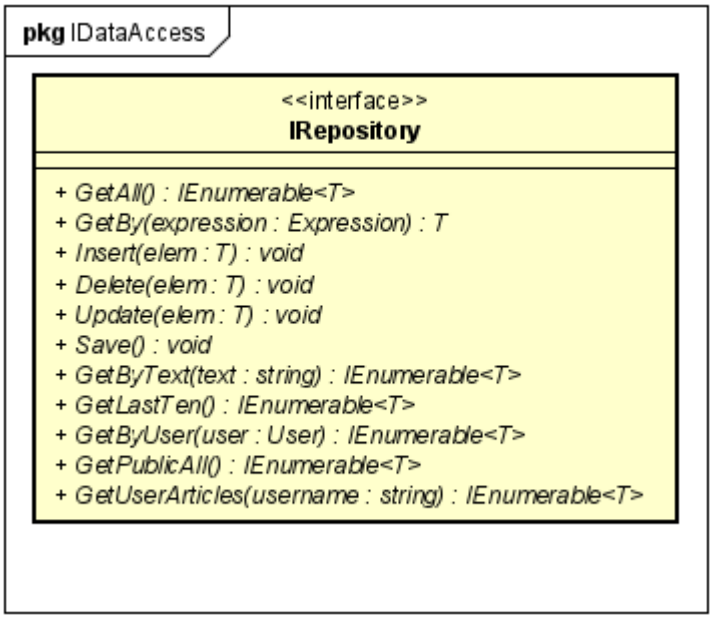
Blog.Models

Este paquete contiene todos los dtos que utilizamos en la aplicación, osea los que utilizamos como parámetro de ingreso en los controllers como los de salida de dichos controllers y también el dto de de errores.



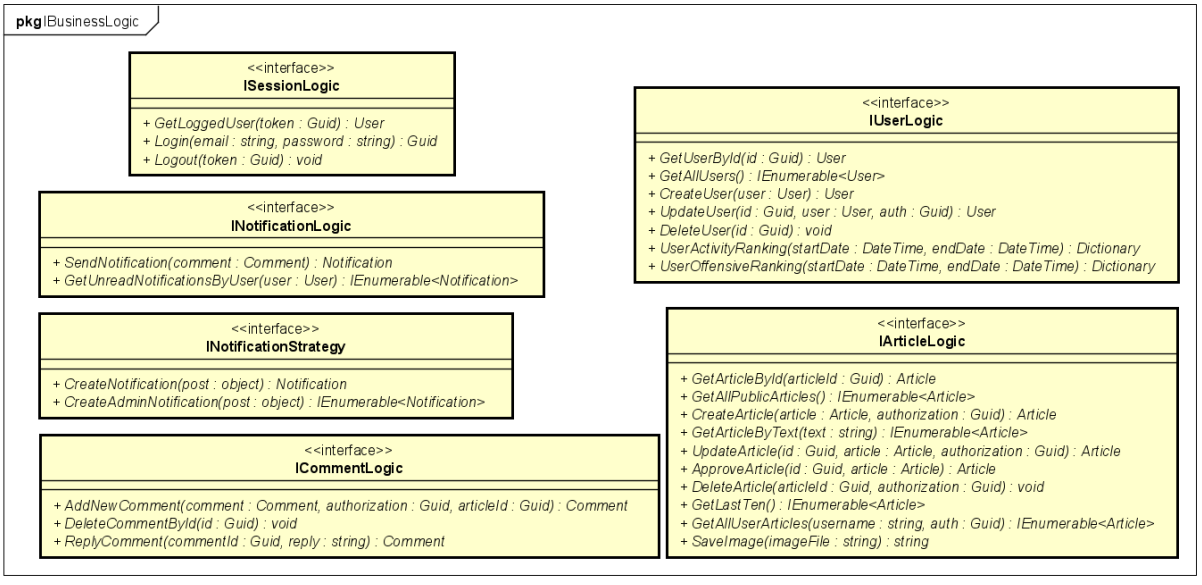
Blog.IDataAccess

Este paquete es el que contiene las interfaces que luego van a ser implementadas por el paquete Blog.DataAccess, esto lo realizamos para que se dependa de interfaces y no de implementaciones y así lograr más estabilidad en la solución y respetar el quinto principio solid, que las clases dependen de abstracciones y no de clases concretas. Las interfaces en este paquete luego van ser utilizadas en las clases del paquete de Blog.BusinessLogic. Aquí se encuentra solamente una clase, ya que decidimos crear un IRepository generico para evitar repetir código.



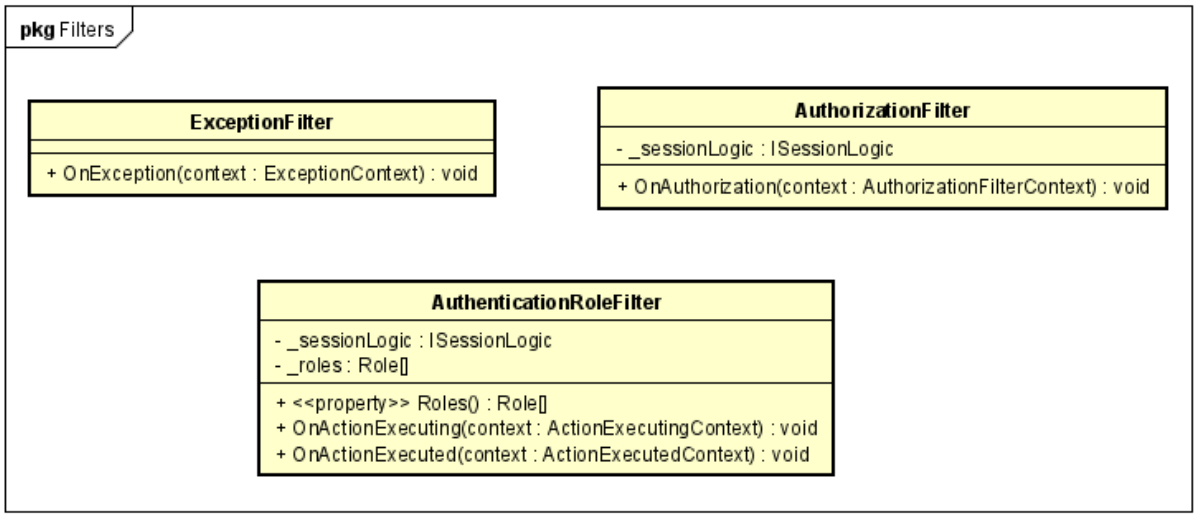
Blog.IBusinessLogic

Este paquete cumple la misma función que el de Blog.IDataAccess lo único que para las clases del dominio y que las interfaces creadas aquí son utilizadas posteriormente por los controladores del paquete de Blog.WebApi, en este paquete se encuentran las interfaces que luego van a ser implementadas en el paquete de Blog.BusinessLogic.



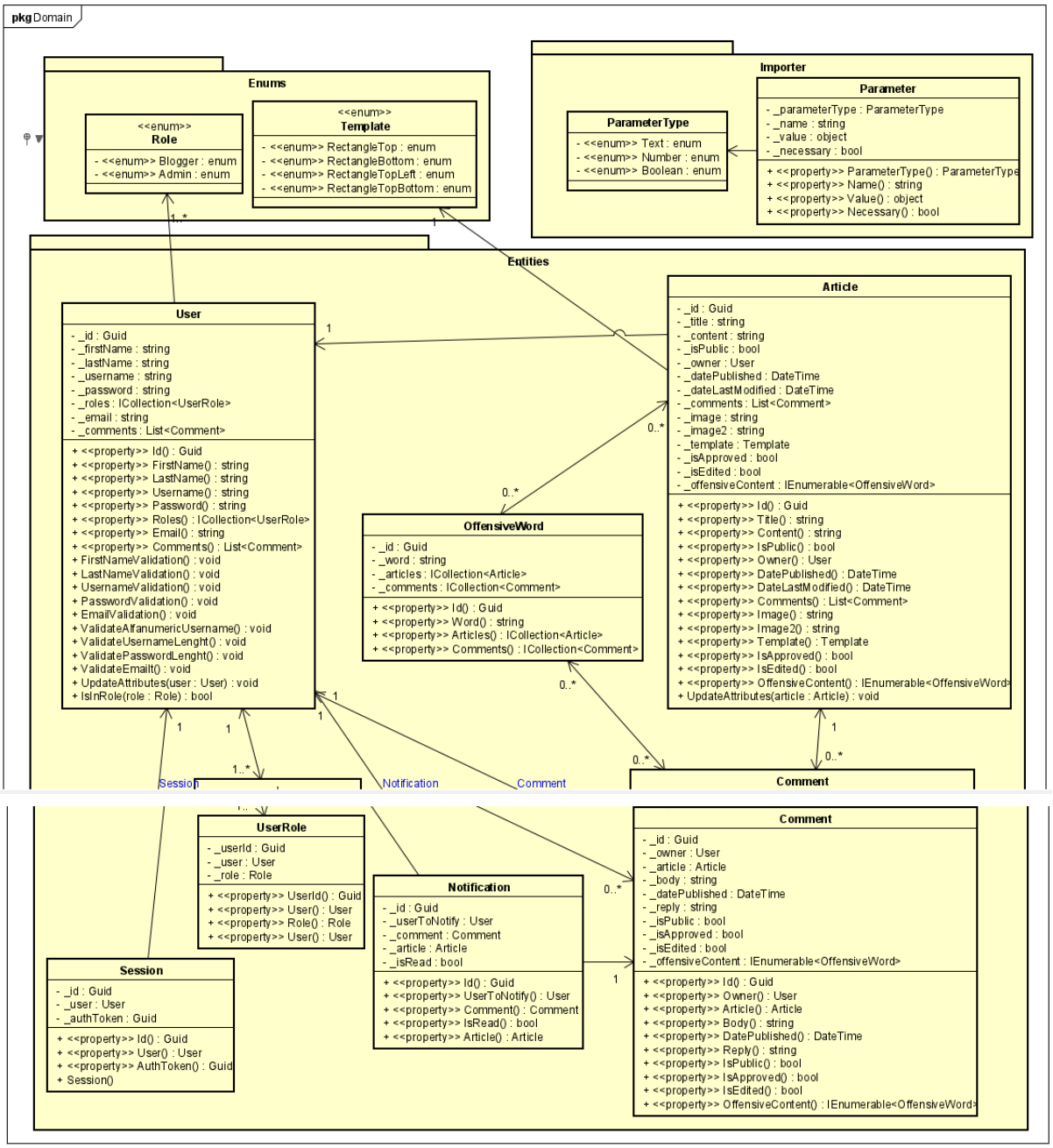
Blog.Filters

En este paquete se encuentran los diferentes filtros implementados para la solución, estos son: Filtro de autorización de usuarios, Filtro de autorización por roles de usuario y Filtro de excepciones. Decidimos poner estas clases en un paquete separado de Blog.WebApi para que allí se encuentren simplemente los controladores y así desacoplar esta lógica de este paquete.



Blog.Domain

Este paquete es donde se encuentran todas las clases del dominio, al igual que los diferentes enumerados y las excepciones que personalizadas que decidimos crear en la solución.

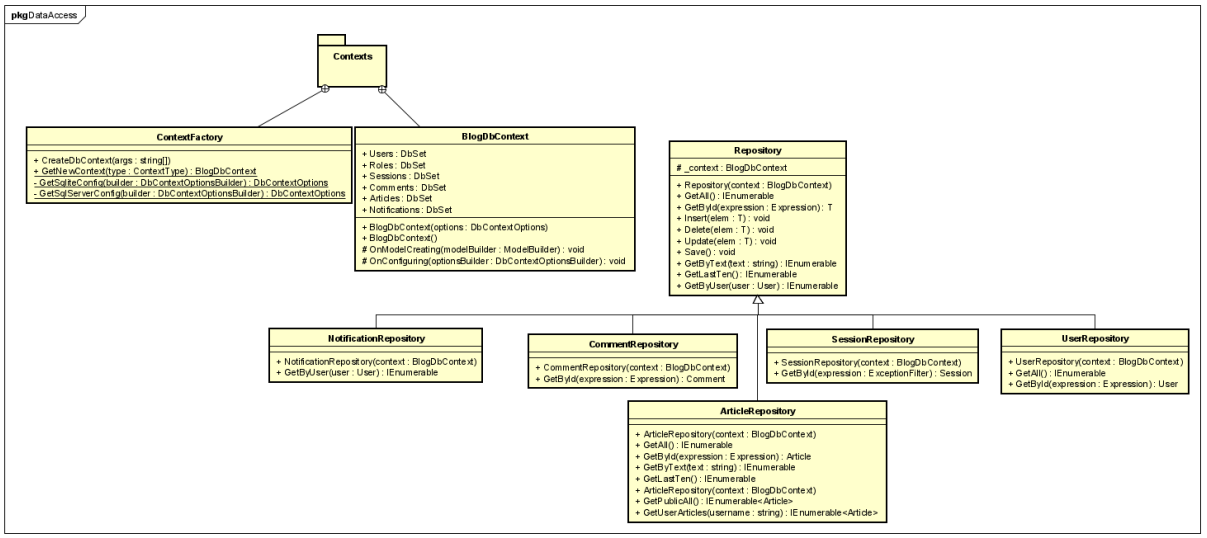


Blog.DataAccess.Test

Este paquete es el responsable de contener a todos los test unitarios que realizamos para las clases del paquete de Blog.DataAccess, en un principio teníamos todos los test unitarios dentro del paquete Blog.Tests, pero por el gran volumen de test utilizados con TDD y para que el paquete de Blog.Tests no quede muy sobrecargado, decidimos crear este paquete donde colocar solo los test unitarios creados para el acceso a datos.

Blog.DataAccess

En este paquete se encuentran todas las implementaciones que utilizamos para el acceso a datos, Aquí decidimos crear una implementación genérica con todos los métodos básicos de un acceso a datos, y luego crear las diferentes clases de repositorio para casos específicos, en estas implementaciones específicas se hace override de los métodos que requieren dicha lógica extra.

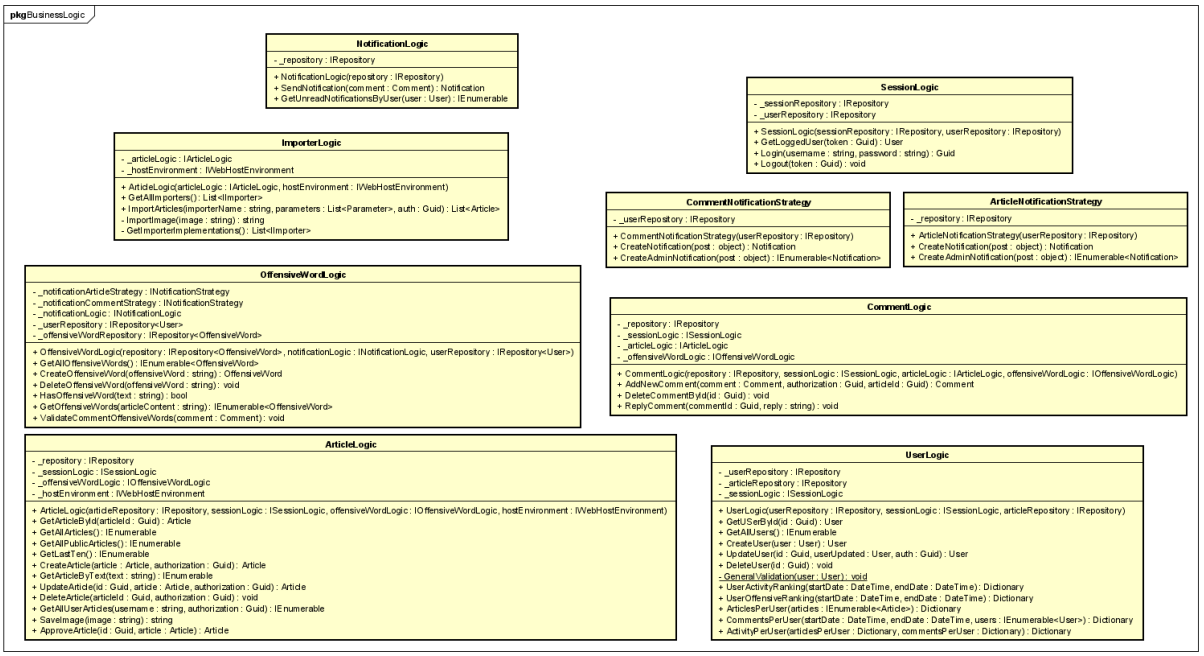


Blog.BusinessLogic.Test

En este paquete al igual que sucedió con el de Blog.DataAccess.Test, en un principio estaba dentro del paquete Blog.Test, pero decidimos crear un proyecto aparte para no sobrecargar al otro paquete con tantos test y que no cumpla con tantas responsabilidades. Aquí se encuentran todos los test unitarios creados para la lógica de negocio de la solución.

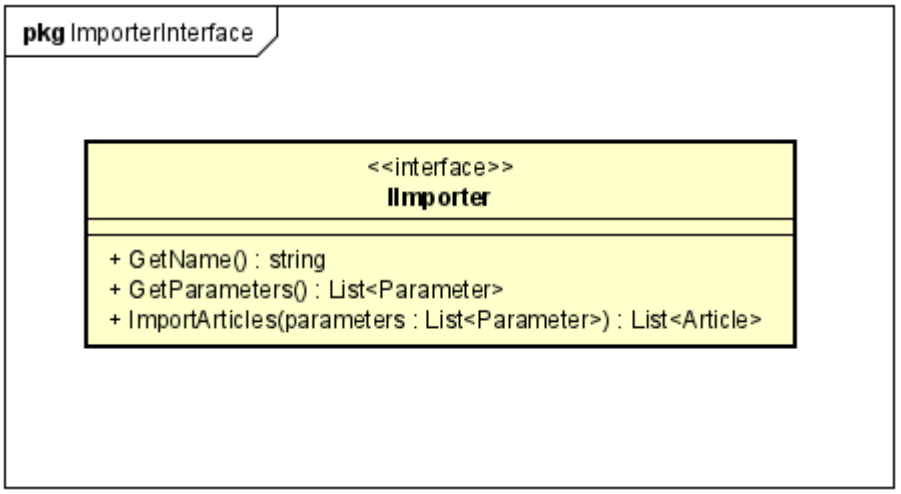
Blog.BusinessLogic

Este paquete es el encargado de contener a todas las clases responsables de la lógica de negocio de la solución, esta clase implementa todas las interfaces anteriormente mencionadas dentro del paquete de Blog.IBusinessLogic y así cumplir con el quinto principio solid como ya se mencionó.



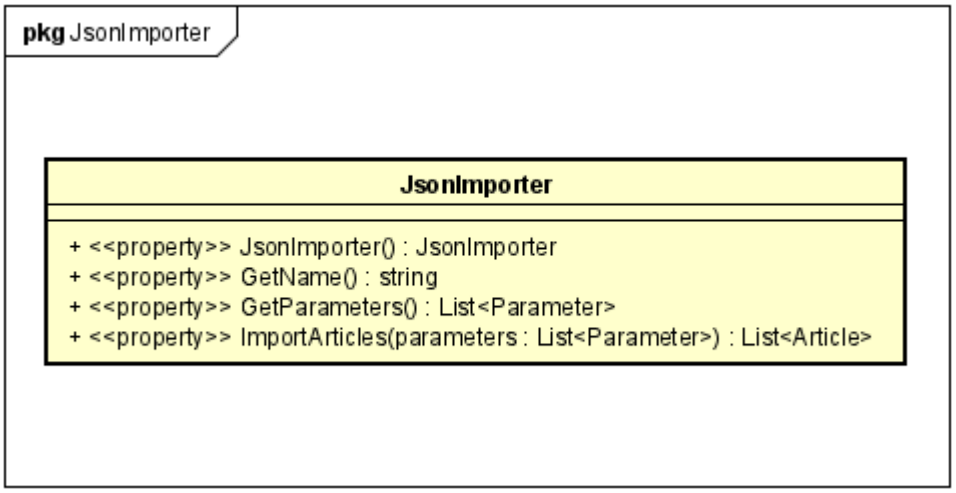
Blog.ImporterInterface

Este paquete se utiliza para poder ofrecer una interfaz mediante la cual un desarrollador pueda acoplar una implementación propia.

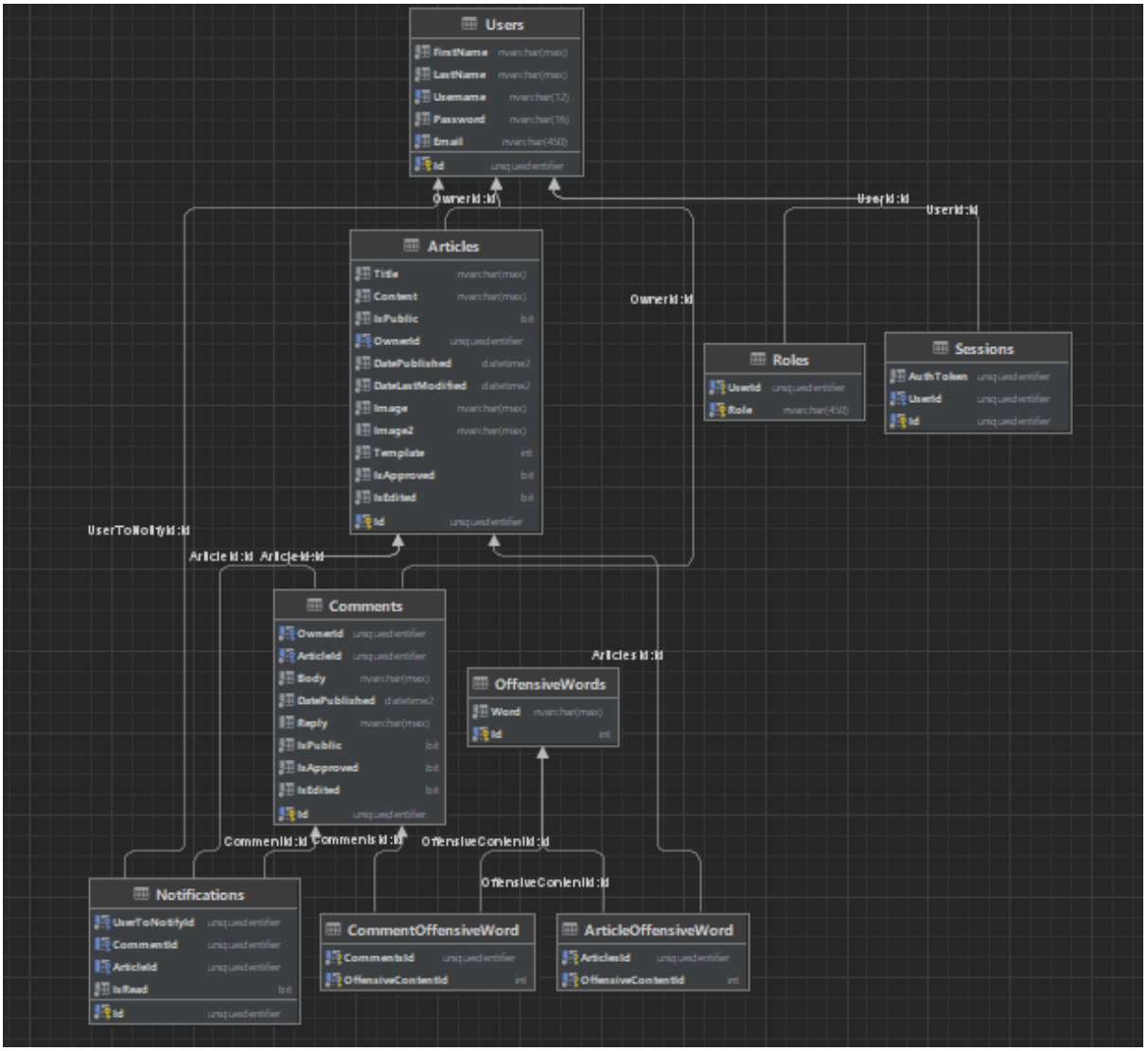


Blog.JsonImporter

Este paquete contiene la implementación del importador de artículos a través de Json. Al compilar este paquete el assembly resultante debe de ser colocado en la carpeta de “Importers” dentro del paquete WebApi para su ejecución.



Modelo de tablas



Diagramas de interacción

Diagrama de secuencia de Crear Artículo

Se puede encontrar una copia del diagrama en github y junto a la documentación por si no se puede apreciar al 100% todo

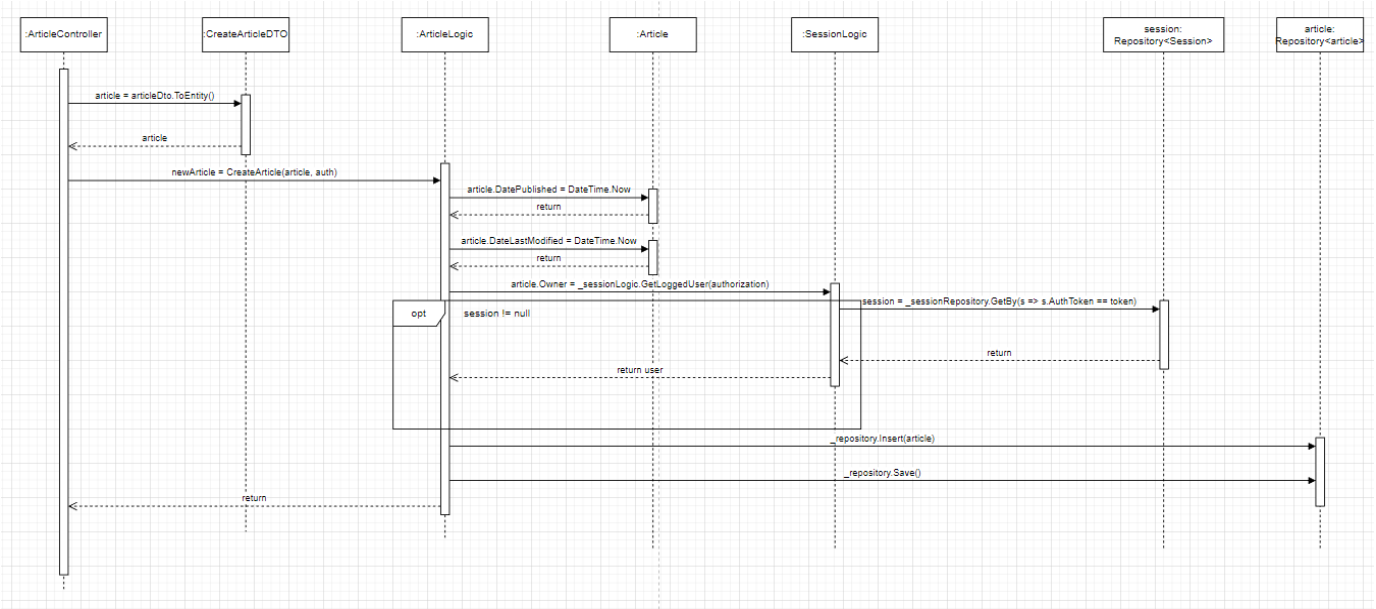
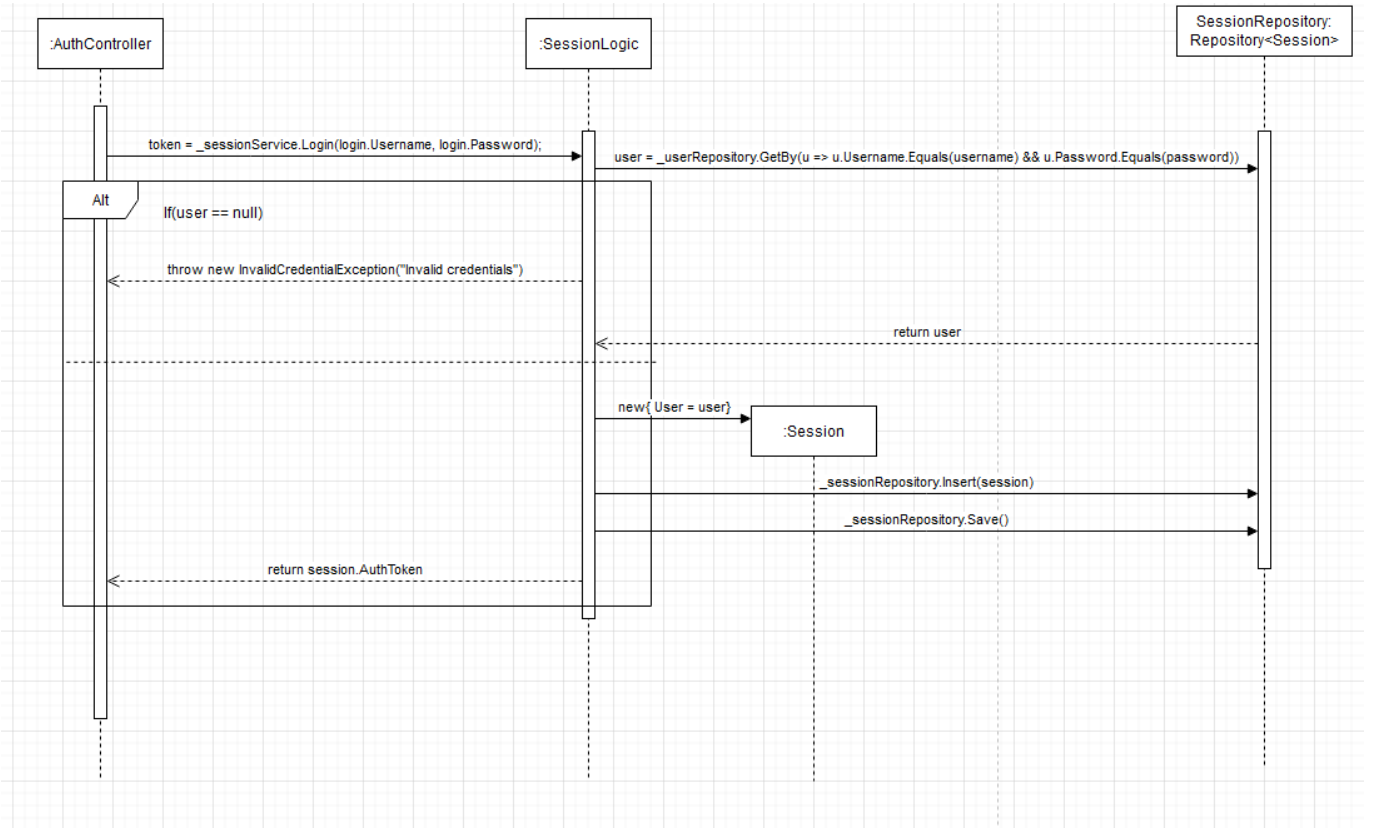


Diagrama de secuencia Login

Se puede encontrar una copia del diagrama en github y junto a la documentación por si no se puede apreciar al 100% todo



Justificación del diseño

Mecanismos de inyección de dependencias

Decidimos la utilización de un proyecto aparte para el manejo de la inyección de dependencias ya que nos parecía que ponerlo en la clase program del paquete de Blog.WebApi hubiese sobrecargado el paquete de dependencias que no eran necesarias para ese paquete como por ejemplo las interfaces de Blog.IDataAccess y las clases de DataAccess.

Filtros

Los filtros también decidimos ponerlos en un paquete aparte para que se cumpla con una sola responsabilidad en el y cumplir con las reglas vistas en clase sobre la agrupación de clases en paquetes a partir de responsabilidades, tratamos de que los diferentes paquetes están agrupados por responsabilidades y que las clases dentro de un paquete tengan responsabilidades similares.

Clase de contextFactory

Decidimos crear una clase dentro del paquete de Blog.DataAccess para que nos facilite al momento de los test unitarios, ya que para los mismos utilizamos una base de datos de SqlLite y para la base de datos del codigo fuente utilizamos una base de datos de SQLServer.

Repository

Utilizamos una clase llamada Repository ubicada en el paquete DataAccess, la cual implementa la interfaz IRepository, Los métodos se implementan en esta clase y a la vez cada repositorio en particular hereda estos métodos, donde algunos los sobreescriben para casos particulares, para así de esta manera poder reutilizar la mayor cantidad de código posible.

Manejo de excepciones

Para el manejo de excepciones utilizamos un filtro dentro del paquete Blog.Filters, este filtro fue utilizado en los diferentes controladores en caso de que una excepción salte, en dicho filtro utilizamos los códigos de error: 400, 401, 404 y 500. Siendo el

código de error 500 el código por defecto osea el que salta en caso de excepciones que no manejamos en nuestro sistema. También creamos una excepción custom para cuando no se encontraba un elemento la cual llamamos NotFoundException.

Principios aplicados

Como se podrá ver en nuestra solución buscamos acoplarnos lo máximo posible algunos de los principios SOLID, como por ejemplo el de responsabilidad única, ya que la gran mayoría de nuestras clases cumplen con una responsabilidad, por ejemplo la clase ArticleLogic cumple solamente con la responsabilidad de la lógica de los artículos.
También buscamos cumplir con el quinto principio SOLID y hacer que las clases concretas dependen de clases abstractas, como por ejemplo en los controladores los cuales usan las interfaces de IBusinessLogic y en la lógica de negocio que se utilizan las interfaces de IDataAccess.

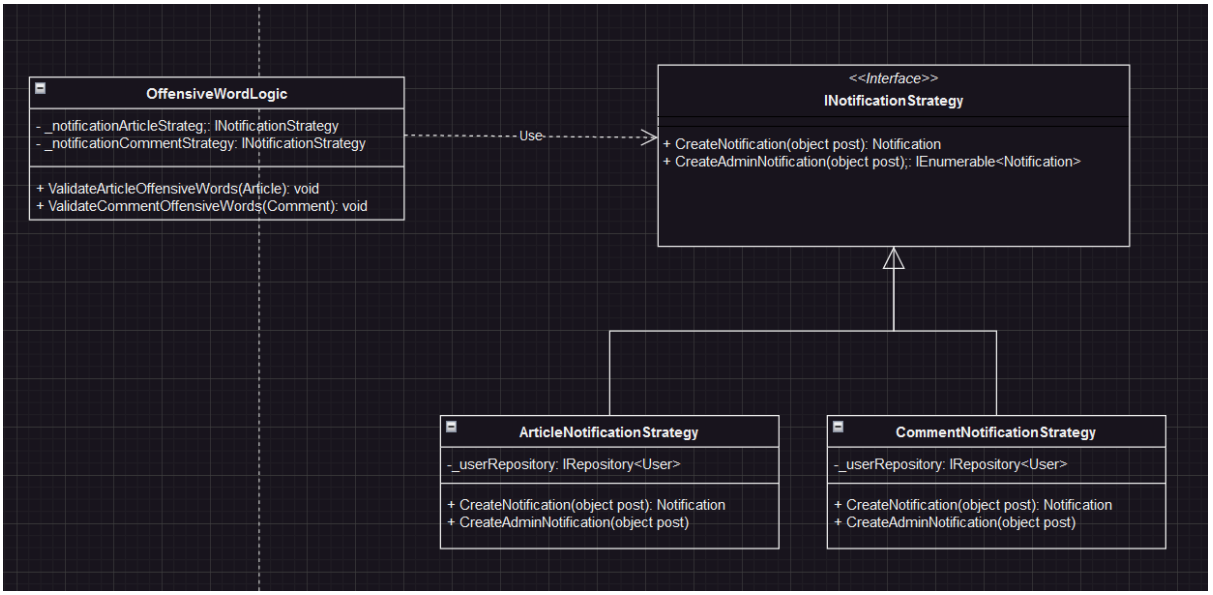
Patrones aplicados

Para el manejo de notificaciones utilizamos el patrón strategy, ya que para la segunda entrega se pidió que se agregue las notificaciones de palabras ofensivas por lo que creamos las clases ArticleNotificationStrategy y CommentNotificationStrategy que consumen la interfaz INotificationStrategy.

Decidimos utilizar este patrón ya que necesitábamos un mismo método con diferente funcionalidad dependiendo si es un comentario o un artículo, ya que los comentarios tienen otros tipos de notificaciones que son la que se agregó un nuevo comentario aparte de la de lenguaje ofensivo.

Luego en la lógica de palabras ofensivas, se llama a la clase en función a lo que se necesita, si se encuentra que un artículo tiene palabras ofensivas se llama a ArticleNotificationStrategy y si por otro lado se valida un comentario y se encuentra una palabra ofensiva se llama al método dentro de CommentNotificationStrategy.

En la imagen se muestran los métodos de OffensiveWordLogic que son relevantes para el patrón utilizado, la clase contiene otros métodos y atributos que no son relevantes para justificar al patrón.

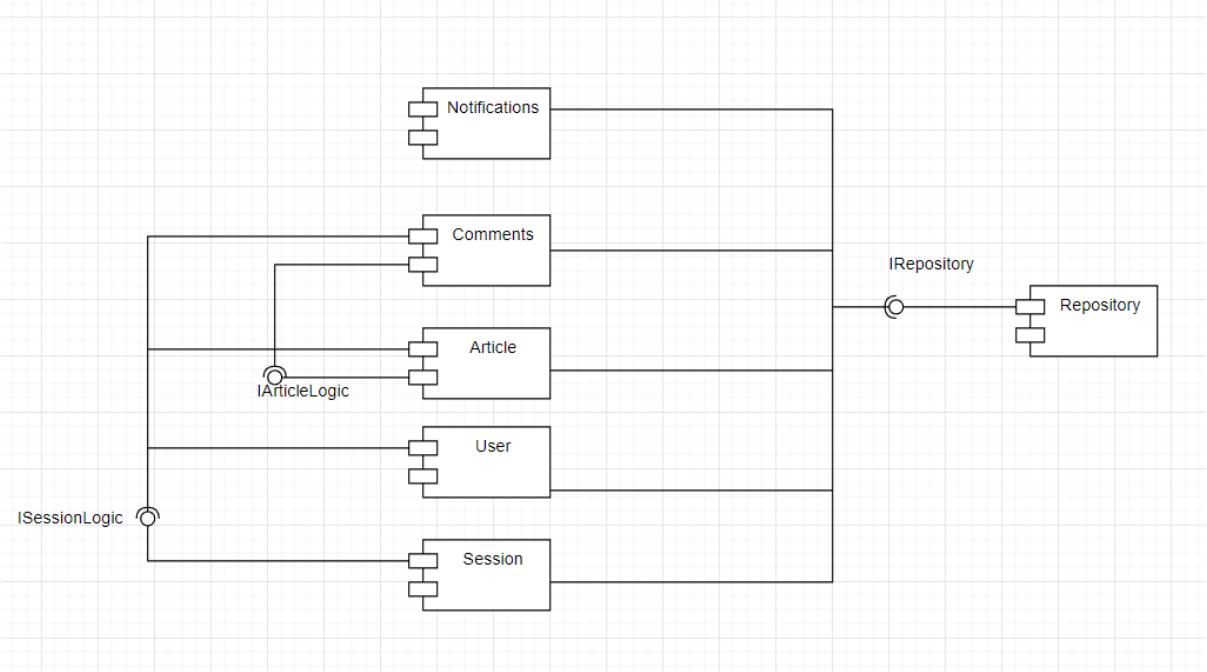


FrontEnd

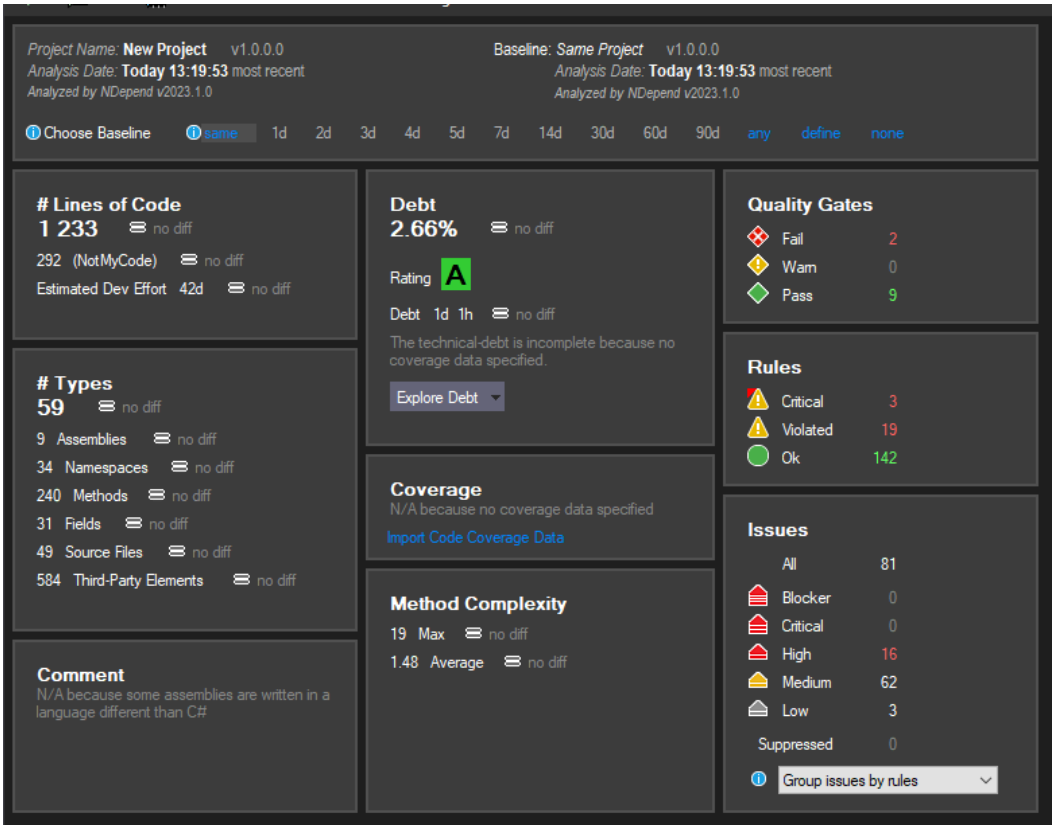
Para el frontend se utilizó angular y elegimos utilizar también bootstrap para el diseño y css de la aplicación ya que nos pareció el más fácil e intuitivo junto con angular. Tratamos de seguir las mejores prácticas de angular, typescript y los diferentes principios, tratando de hacer el código igual de limpio que en el backend siguiendo las prácticas de clean code. Gracias a que contábamos con un código bastante limpio y extensible en la primer entrega, la creación del frontend no nos hizo modificar casi nada de nuestro backend anterior, logrando una buena implementación en general.

Diagrama de componentes

La solución se dividió en los componentes de notification, comments, article, user, session y repository ya que son las partes fundamentales del blog, todos los componentes menos el article son componentes propios de un blog con autenticación y administración de usuarios, y luego se decidió agregar el componente repositorio ya que todos los componentes son guardados allí y los diferentes componentes necesitan de ella.



Métricas de diseño

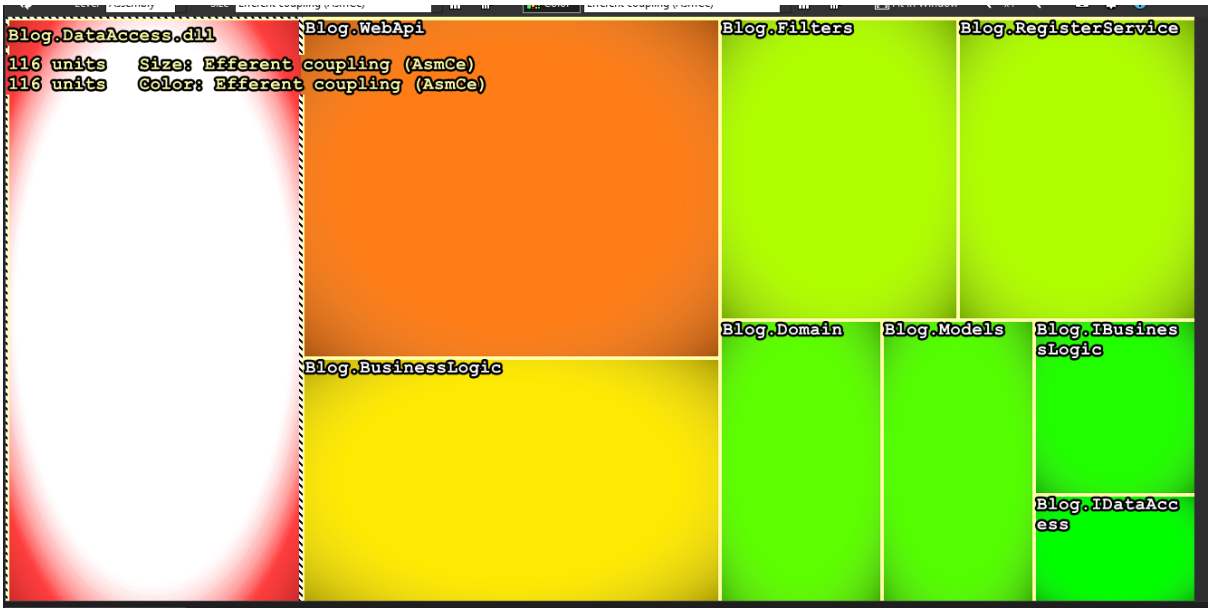


El programa nos calificó nuestra solución con una A, se observan algunos issues pero no son de mayor importancia y la solución parece ser buena a nivel de métricas.

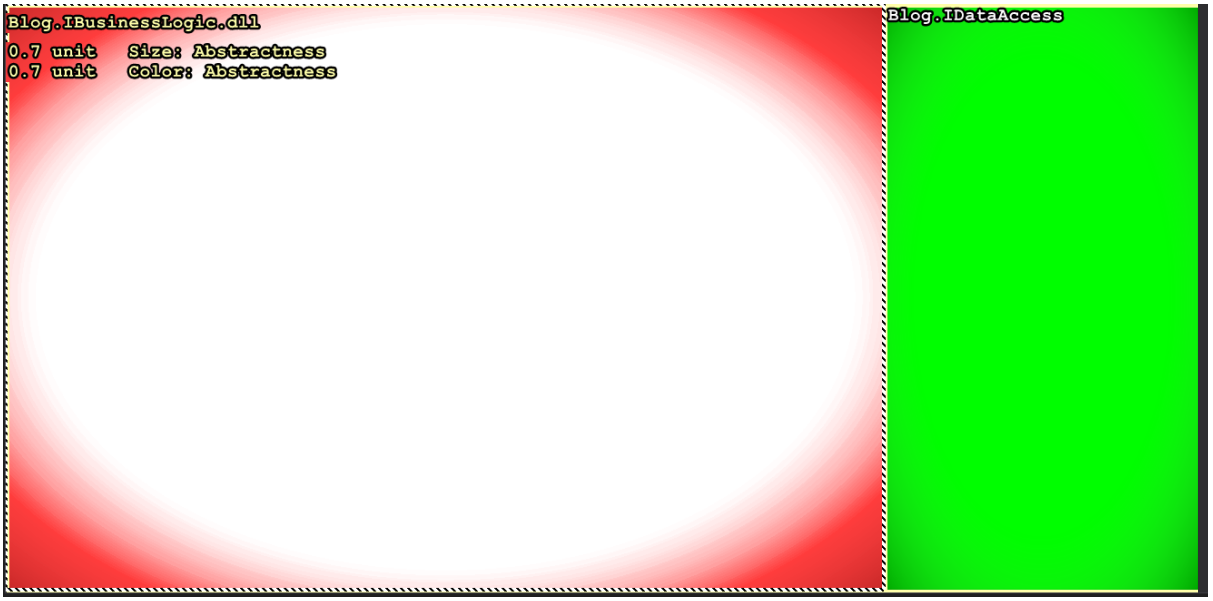
Relational Cohesion (H)

Se pueden ver valores de H bastante normales, casi todos están entre 1.5 ≤ H ≤ 4.0, por lo que parecen estar bien, el valor más alto es el del dominio con 2.77. Los paquetes de IDataAccess, IBussinessLogic y RegisterService se encuentran por debajo del 1.5 lo cual podría tener algo de sentido al tratarse de paquetes abstractos o utilizados para inyección de dependencias como lo es RegisterService.

Efferent coupling (Ce)



Abstractness



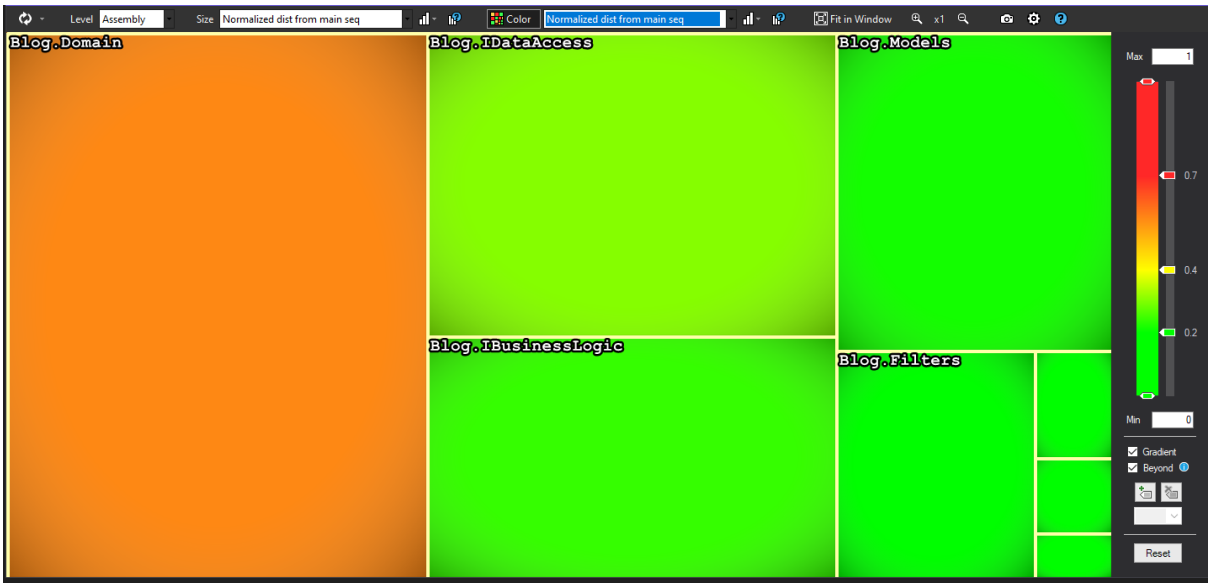
Se puede observar que IBusinessLogic cuenta con un 0.7 en abstracción por lo que indicaría que es un paquete bastante abstracto, en cuanto a IDataAccess marca un 0.25 lo que diria que el paquete es bastante concreto, este paquete contiene solamente IRespository la cual es una interfaz reutilizable para el paquete DataAccess por lo que no comprendemos el nivel tan bajo de abstracción. Suponemos que podría deberse a dependencias con paquetes de .net en sí.

Instability

En esta métrica tiene sentido que webApi sea lo más inestable ya que es la capa de mas arriba y la que depende del resto, cumpliendo así con el principio de dependencia de la estabilidad donde las capas superiores son más inestables que las inferiores.

Tratamos de guiarnos mucho por el principio de dependencias estables como se mencionó anteriormente esto hace que nuestro DataAccess sea el paquete que tiene más cosas dependiendo de él ya que es el de más bajo nivel. También tratamos de guiarnos en base al principio de clausura común. Al seguir el principio de clausura al agrupar los paquetes, también promovemos la reutilización de código. Al organizar las funcionalidades relacionadas en paquetes cohesivos. Tratamos de agrupar y organizar las funcionalidades de manera lógica y coherente, de modo que cada paquete sea autosuficiente y pueda cumplir con un propósito específico.

Abstracción e inestabilidad



En esta métrica se puede observar que todas se encuentran dentro de lo normal siendo Blog.Domain la más alta con 0.56, ningún paquete llega a los 0.7 donde empiezan a entrar o en la zona de dolor o en la zona de inutilidad, por ende se puede decir que el código no

necesita mayores cambio en este sentido. Tiene sentido que domain sea la más alta ya que un cambio allí podría afectar en otras partes del código, por eso sería doloroso, pero igualmente está fuera de la zona de dolor.

Resumen de mejoras

- Se realizaron mejoras importantes en el comportamiento del guardado de imágenes, guardandolas en la base de datos en base 64 mientras que anteriormente se guardaba como texto plano, también se agregó la carpeta wwwroot para que las imágenes se puedan obtener desde el browser.
- Se aplicó el patrón strategy para el reuso de la interfaz INotificationStrategy en función de si es un artículo o un comentario lo que se debe notificar.
- Se resolvieron todos los errores conocidos notificados en la anterior entrega.
- Se realizaron mejoras en la base de datos para la mejor obtención de algunos atributos. Hay que decir que gracias a que nuestro código de la primera entrega estaba abierto a la extensión no se vieron grandes impactos en tema de código, ya que la mayoría de cambios fueron de agregar código y no modificar.
- Se agregó un endpoint en users para obtener el usuario por username, esto nos ayudó a poder ver los perfiles públicos en el frontend haciendo click en el username en el post.
- Se agregó un endpoint para comentarios para obtener todos los comentarios ofensivos para así facilitar la pagina de moderación de comentarios y articulos en el frontend.
- Se agregaron nuevos paquetes para el uso de los importadores para así solo exponer el código necesario a usuarios terceros. Hay que decir que el archivo a importar está dentro del proyecto Blog.WebApi pero no en ninguna carpeta en especial, lo cual no es lo ideal, pero por temas de funcionamiento de nuestra solución tuvimos que crearlo de esa manera. Aun así hicimos un importador extensible tanto en el frontend como en el backend gracias a la implementación de los nuevos endpoints para esto, además de contar con los parameters y parameterType para poder extraer los diferentes parámetros que requieren los diferentes importadores que se quieran agregan en un futuro aparte del Json ya existente.

Anexo

Descripción de los recursos modificados de la API

URL Base

https://localhost:7105/api

Recursos

Artículos

- PUT - api/articles/approve/{id}
    - Permite aprobar los artículos que van a la moderación de palabras ofensivas
    - Devuelve un 201 en caso de aprobar el artículo
    - Se necesita estar logueado y ser un admin para poder llamar a este endpoint
- GET - api/articles/importers
    - trae los importadores que contiene el sistema en ese momento
    - Devuelve la lista de importadores
    - Se necesita estar logueado para utilizarlo
- POST - api/articles/import
    - nos permite importar los artículos a partir de un archivo json dentro de el proyecto de web api
    - devuelve la lista de artículos dentro del archivo mencionado anteriormente
    - Es necesario estar logueado para utilizarlo

Auth

No se agregaron nuevos endpoints

Comment

- GET - api/comments
    - Permite obtener todos los comentarios de la base de datos
    - devuelve un 200 OK
    - se necesita estar logueado y ser un admin para realizar la acción

User

- GET - api/users/rankingOffensive
    - Permite ver el ranking de usuarios con artículos con mas palabras ofensivas entre determinadas fechas
    - Se le envia por query la fecha de inicio y de fin para el filtrado
    - Devuelve un 200 OK con la lista de usuarios y el número de artículos y comentarios ofensivos que tuvo en ese tiempo.
- GET - api/users/user/{username}
    - Permite obtener a un usuario a partir de su username
    - Devuelve un 200 OK en caso de encontrarlo y un 404 en caso de no encontrarlo
    - Nos ayuda a buscar los perfiles públicos de los usuarios en el frontend

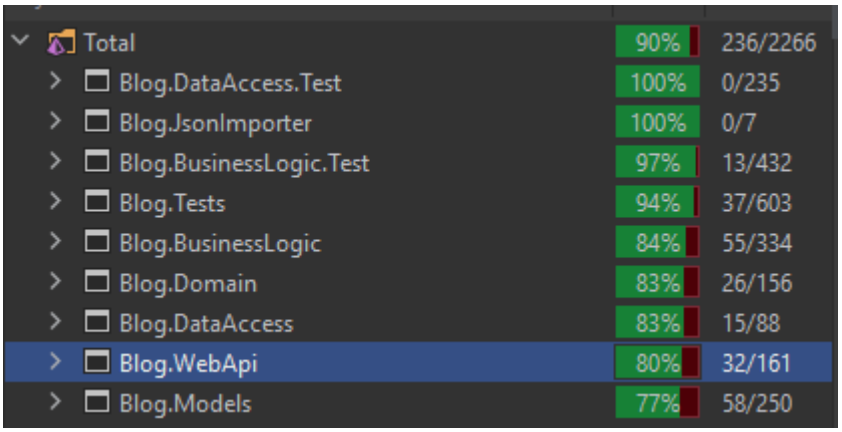
OffensiveWords

- GET - api/OffensiveWords
    - devuelve todas las palabras ofensivas agregadas por los administradores
    - Devuelve un 200 al responder la lista de palabras

    - Se le manda un header de Authorization ya que hay que estar logueado para usar el recurso

- POST - api/OffensiveWords
    - Permite agregar nuevas palabras ofensivas a la lista existente
    - Devuelve un 201 al agregar la nueva palabra a la lista
    - Se le envia por body la palabra a agregar
    - Se le manda un header de Authorization ya que hay que estar logueado para usar el recurso
- DELETE - api/OffensiveWords/{offensiveWord}
    - Se le envía por route la palabra a borrar
    - Permite borrar una palabra de la lista de palabras ofensivas
    - Devuelve un 200 al borrar una palabra de la lista
    - Se le manda un header de Authorization ya que hay que estar logueado para usar el recurso

Informe de cobertura



Obtuvimos un 90% en nuestros test de cobertura de nuestros métodos más importantes, por lo que se puede decir que nuestro código general cuenta con una buena cobertura, por temas de último momento no pudimos realizar test en algunos de los endpoint agregados a último momento y por eso es el bajo porcentaje en Blog.WebApi.