Universidad ORT Uruguay

Facultad de Ingeniería

Documentación de la segunda entrega del obligatorio de Diseño de Aplicaciones 2

Emiliano Yozzi - 230710

Franco Thomasset - 239611

Tutores:

Juan Irabedra, Santiago Tonarelli, Francisco Bouza

2022

Enlace al repositorio:ORT-DA2/239611-230710 (github.com)

Declaración de autoría:

Nosotros, Franco Thomasset y Emiliano Yozzi, 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 realizamos 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

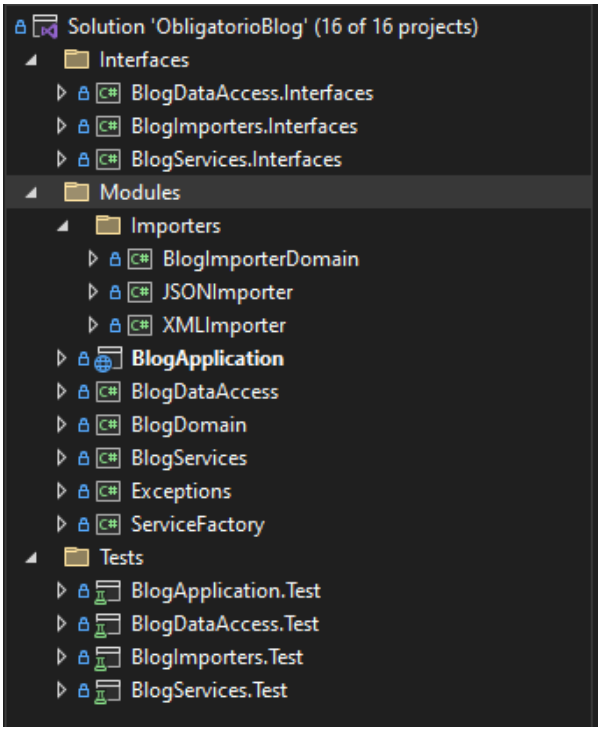
Índice

Declaración de autoría: **2**
Descripción general del trabajo **5**
Evidencia de arquitectura REST 5
Evidencia de aplicación de principios de Clean Code 6
Filtros y su funcionamiento: 8
Códigos de error utilizados: 9
Análisis de cobertura de código: 9
Errores conocidos **11**
Vista de Componentes **12**
Diagrama general de paquetes 12
Diagrama de componentes **12**
Vista de diseño **13**
Controllers: 13
Filters: 14
Models 15
BlogDomain 15
BlogServices 16
BlogServices.Interface 16
BlogDataAccess.Interfaces 17
BlogDataAccess 18
ServiceFactory 18
Importers 19
Exceptions 19
Vista de Procesos **19**
Modelado de tablas **20**
Justificación del diseño **22**
Análisis basado en métricas: 22
Principios de acoplamiento de paquetes: 23
Principios de cohesión de paquetes: 24
Decisiones de diseño: 25
Mejoras respecto a la primera entrega **26**
Set de datos de prueba **26**
Anexo: 27
Especificación de la API: 27
Obtener importadores de artículos 27
Importar artículos 27
Actualizar comentario 27
Eliminar comentario 27
Registrar palabras ofensivas 27
Eliminar palabra ofensiva 27
Obtener lista de palabras ofensivas 28

Ranking de actividad..... 28
Ranking de ofensas.....28
Buscar artículo por palabra..... 28
Obtener las notificaciones de un usuario..... 28
Obtener los artículos de un usuario.....28

Descripción general del trabajo

Para esta entrega expandimos nuestra solución agregando nuevos proyectos para cumplir con algunos de los nuevos requerimientos.



Este es el resultado final de nuestra solución. Los nuevos proyectos son BlogImporters.Interfaces, BlogIMporterDomain, BlogImporters.Test, JSONImporter y XMLImporter.

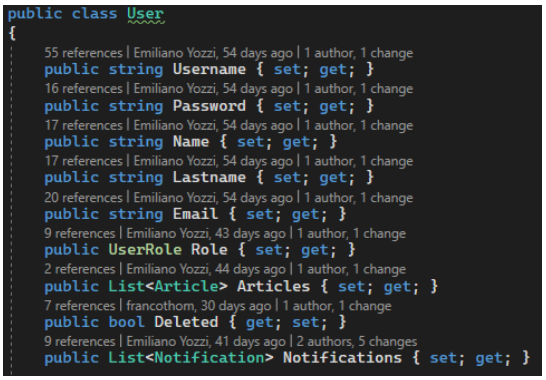
A nivel de clases, fue necesario agregar al dominio las clases: Image, DateLog, DateRange,OffensiveWordCollection. También tuvimos que agregar hacer cambios y crear controllers, servicios y repositorios para cumplir con los cambios pedidos por la consigna.

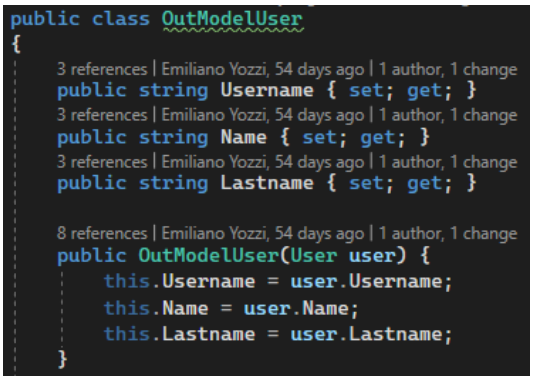
Esta solución contiene la implementación de todas las features pedidas por la consigna actual y la anterior.

Evidencia de arquitectura REST

Gracias al feedback de los docentes, podemos explicar detalles que nos faltaron para la entrega anterior. Por lo que, en este documento se harán menciones sobre apartados correspondientes a la documentación de la entrega anterior. Empezando con REST:

Basamos nuestra arquitectura en los elementos característicos de REST. Nuestro sistema contiene varios ‘recursos’ a los cuales se accede haciendo uso de ‘identificadores’ (muchas veces un campo ‘id’, pero no siempre, ej. User es identificado por ‘username’). Para transferir la información de los recursos hacemos uso de ‘representaciones’, estos son modelos de entrada y salida con los que podemos controlar exactamente qué información entra y sale en las peticiones.





A su vez, nuestra aplicación está dividida en tres capas principales.

- **API:** Donde se encuentran los controllers y filters que se encargan de recibir las peticiones y procesar la información inicial para enviarla a la capa de servicios.
- **Servicios:** En esta capa nos encargamos de aplicar la lógica de negocio correspondiente a cada acción sobre un recurso. Por ejemplo, controlar palabras inapropiadas a la hora de crear un nuevo artículo.
- **Acceso a Datos:** La responsabilidad de esta capa es persistir los recursos en una base de datos, proveyendo métodos a la capa de servicios para guardar y obtener información.

Esto se puede ver en el diagrama de componentes.

Evidencia de aplicación de principios de Clean Code

Durante el transcurso de este proyecto, siempre desarrollamos teniendo en mente las buenas prácticas planteadas en Clean Code. Entendemos que son indispensables para generar código de calidad que pueda ser fácilmente mantenible a lo largo del tiempo.

A continuación adjuntamos ejemplos de la aplicación de los mismos:

Nombres mnemotécnicos de variables y métodos:
Esto significa que se debe de poder entender el comportamiento de cada método y el propósito de cada variable sólo con leer su nombre.

```
14 references | 6/6 passing | Emiliano Yozzi, 6 days ago | 1 author, 2 changes
public List<string> CheckOffensiveWords(string text)
{
    List<string> wordsFound = new List<string>();
    List<string> offensiveWords = repository.Get().offensiveWords;
    string filteredText = RemoveNonAlphanumericChars(text).ToLower();

    foreach (string word in offensiveWords)
        if (filteredText.Contains(word.ToLower()))
            wordsFound.Add(word);

    return wordsFound;
}
```

El método ‘CheckOffensiveWords’ chequea las ocurrencias de palabras ofensivas en un texto pasado por parámetro. Queda bastante explícito en el nombre. A su vez, el nombre de las variables definidas dentro del método, representan el propósito de las mismas. Al leerlas, se entiende que ‘wordsFound’ es utilizada para almacenar las ocurrencias encontradas, ‘offensiveWords’ almacena las palabras consideradas como ofensivas y ‘filteredText’ es una versión filtrada del parámetro.

Responsabilidad única de métodos:
Cada método debe hacer una única cosa. Esto favorece la reusabilidad del código. Un ejemplo de esto es el siguiente:

```
11 references | 9/9 passing | francothom, 31 days ago | 2 authors, 3 changes
public User AddUser(User user)
{
    ValidateFields(user);
    return data.Add(user);
}

2 references | Emiliano Yozzi, 44 days ago | 1 author, 1 change
private void ValidateFields(User user) {
    CheckForEmptyFields(user);
    ValidateUsername(user.Username);
    ValidateEmail(user.Email);
}
```

Dada la necesidad de verificar que la información enviada a la hora de crear un usuario sea válida, hubo que escribir código que se encargue de validar dicha data. Nosotros decidimos no cargar con más responsabilidades al método ‘AddUser’ y creamos un nuevo método ‘ValidateFields’ que se encarga del comportamiento anteriormente mencionado.

Gracias a esto, pudimos reutilizar el código del método de validación cuando implementamos el método de ‘UpdateUser’.

```
public User UpdateUser(User user)
{
    ValidateFields(user);
    return data.Update(user);
}
```

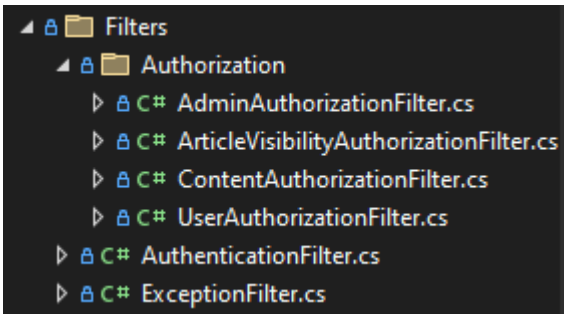
Mantener métodos cortos:
En nuestra entrega mantuvimos los métodos lo más corto posible en cuanto a cantidad de líneas. A excepción de algún método en las clases de testing, los métodos que hemos implementado oscilan entre 1 a 15 líneas como máximo (siendo 15 ya un caso muy raro). Las imágenes anteriores sirven como ejemplo para este punto.

Largo de líneas:
Intentamos mantener el largo máximo de las líneas alrededor de 80 caracteres. Puede que exista alguna excepción, pero en la gran mayoría de los casos, se cumple.

Cantidad de parámetros por método:
En nuestra solución no existen métodos que reciban más de 2 parámetros, a excepción de algún constructor. Siendo el constructor con más parámetros el de CommentService, ya que es necesario inyectarle dependencias para que se comunique con ICommentRepository, IArticleRepository, IUserRepository y IWordControl. Pero dejando de lado algún constructor, el resto de métodos no rompen esta regla.

Evitar comentarios:
Nosotros consideramos que un código que necesita comentarios para poder ser entendido, es un mal código. Un código limpio debería de ser capaz de ser entendido por sí mismo, ya que debería ser claro y fácil de leer. Por esto, no hicimos uso de los comentarios en ninguna parte de la solución.

Filtros y su funcionamiento:



En esta imagen se pueden ver los filtros que consideramos necesario desarrollar para este proyecto.

‘ExceptionFilter’ se encarga de “atajar” las excepciones que puedan ocurrir en el proceso, desde que la aplicación recibe una petición, hasta que envía la respuesta.

Dependiendo del tipo de excepción, modifica la respuesta para indicar un código de error que se ajuste al problema ocurrido.

‘AuthenticationFilter’ tiene como objetivo verificar que el token enviado en el Header de la petición sea válido (es decir, que pertenezca a una sesión ya iniciada). Además de dejar registrado en el contexto de la petición el usuario asociado a esa sesión.

Los filtros contenidos en la carpeta ‘Authorization’ son filtros que se encargan de verificar que el usuario registrado en el contexto por el filtro anteriormente mencionado, cumpla con ciertas condiciones. Más precisamente:

- **AdminAuthorization:** Verifica que el usuario tenga el rol de Admin.
- **ArticleVisibilityAuthorization:** Se usa al intentar acceder a un artículo. Verifica que, en caso de tratarse de un artículo privado, el usuario sea administrador o el dueño del artículo.
- **ContentAuthorization:** Se usa al intentar modificar/eliminar un contenido (artículo o comentario). Verifica que el usuario sea administrador o el dueño de dicho contenido.
- **UserAuthorization:** Se utiliza al intentar modificar la información de un usuario. Verifica que el usuario que hizo la petición sea el mismo usuario o un administrador.

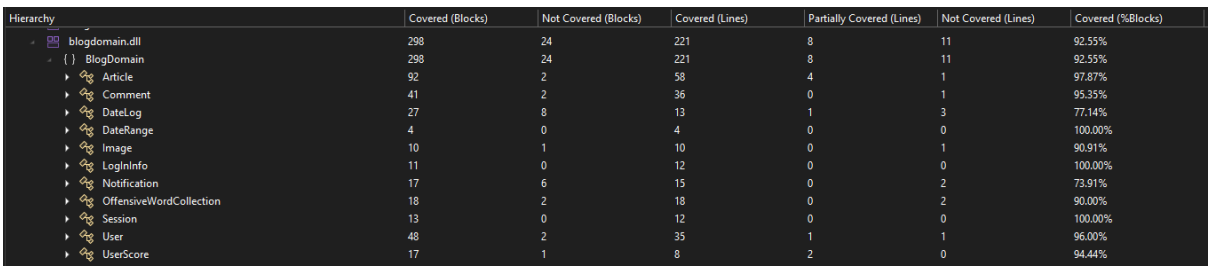
Códigos de error utilizados:

- **200 (OK):** Lo utilizamos en como respuesta siempre que una petición fue cumplida exitosamente.
- **400 (Bad Request):** Lo utilizamos cuando un parámetro enviado en la petición es incorrecto o la operación es inválida.
- **401 (Unauthorized):** Lo utilizamos cuando el usuario que hizo la petición carece de los permisos pertinentes.
- **404 (Not Found):** Lo utilizamos cuando una petición intenta acceder a un recurso que no existe.
- **500 (Internal Server Error):** Este código es mostrado cuando un error inesperado ocurre en el backend y no aplica ninguno de los códigos anteriores.

Es importante aclarar que no hacemos uso del código **201 (Created)** porque dificulta mucho a la hora de testear el código, ya que generaba la necesidad de mockear demasiadas clases.

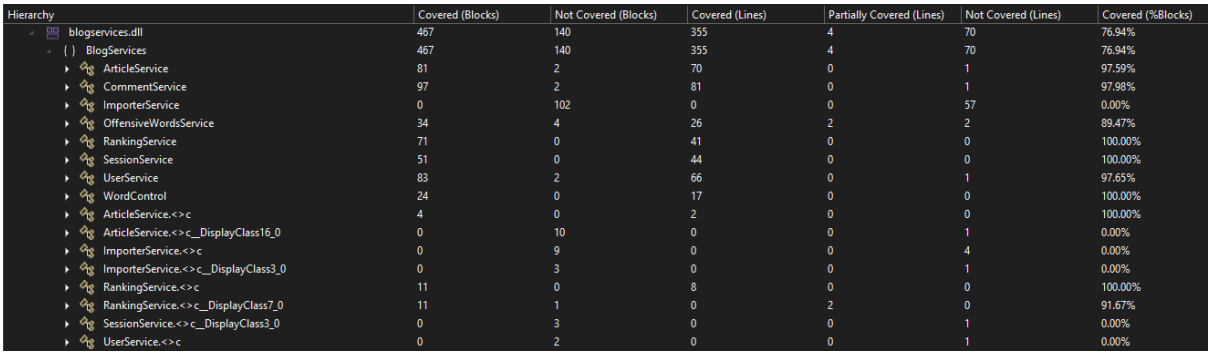
Análisis de cobertura de código:

En el proyecto de dominio tenemos un 92,55% de cobertura.



Hierarchy	Covered (Blocks)	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)	Covered (%Blocks)
blogdomain.dll	298	24	221	8	11	92.55%
{ } BlogDomain	298	24	221	8	11	92.55%
Article	92	2	58	4	1	97.87%
Comment	41	2	36	0	1	95.35%
DateLog	27	8	13	1	3	77.14%
DateRange	4	0	4	0	0	100.00%
Image	10	1	10	0	1	90.91%
LogInInfo	11	0	12	0	0	100.00%
Notification	17	6	15	0	2	73.91%
OffensiveWordCollection	18	2	18	0	2	90.00%
Session	13	0	12	0	0	100.00%
User	48	2	35	1	1	96.00%
UserScore	17	1	8	2	0	94.44%

En el proyecto que contiene los servicios tenemos un 76,94% de cobertura.



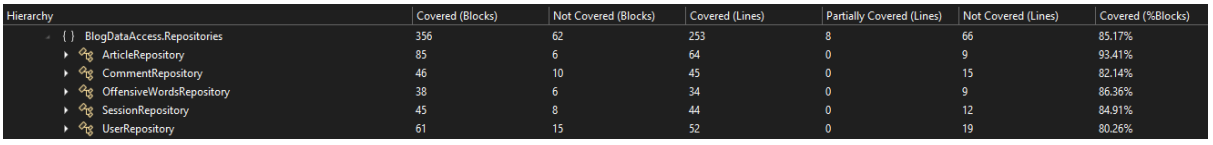
Hierarchy	Covered (Blocks)	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)	Covered (%Blocks)
blogservices.dll	467	140	355	4	70	76.94%
{ } BlogServices	467	140	355	4	70	76.94%
ArticleService	81	2	70	0	1	97.59%
CommentService	97	2	81	0	1	97.98%
ImporterService	0	102	0	0	57	0.00%
OffensiveWordsService	34	4	26	2	2	89.47%
RankingService	71	0	41	0	0	100.00%
SessionService	51	0	44	0	0	100.00%
UserService	83	2	66	0	1	97.65%
WordControl	24	0	17	0	0	100.00%
ArticleService.<>c	4	0	2	0	0	100.00%
ArticleService.<>c__DisplayClass16_0	0	10	0	0	1	0.00%
ImporterService.<>c	0	9	0	0	4	0.00%
ImporterService.<>c__DisplayClass3_0	0	3	0	0	1	0.00%
RankingService.<>c	11	0	8	0	0	100.00%
RankingService.<>c__DisplayClass7_0	11	1	0	2	0	91.67%
SessionService.<>c__DisplayClass3_0	0	3	0	0	1	0.00%
UserService.<>c	0	2	0	0	1	0.00%

Esto se debe a varios motivos:

1. Las clases que figuran como “NombreDeClase”.<>DisplayClassN_N son generadas al compilar las funciones lambda dentro de los métodos de las clases testeadas. Generalmente usamos estas funciones para indicarle a las clases “Repository” qué recursos obtener, pero debido al uso de Mocks para manipular el comportamiento de los repositorios, estas clases generadas quedan con una cobertura de 0%.
2. La clase ImporterService carece de pruebas unitarias porque nuestros docentes nos indicaron que podíamos optar por no generar tests para el desarrollo relacionado con Reflection. Esto es porque la complejidad aumenta en gran medida y escapa al propósito del proyecto.

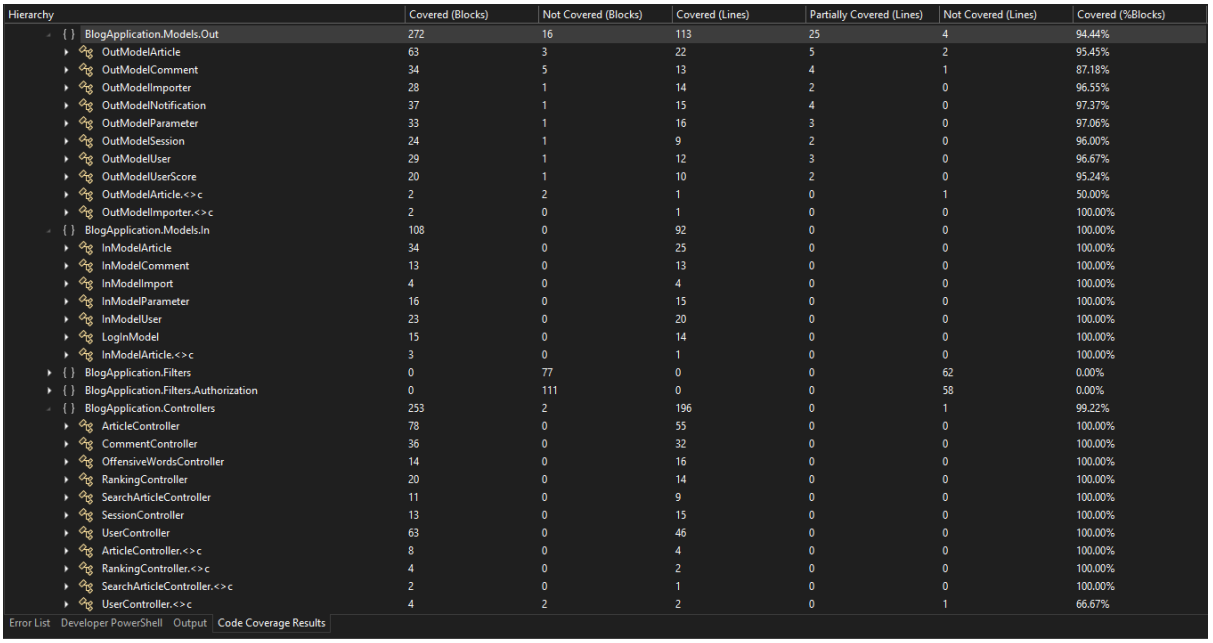
El resultado de calcular la cobertura, dejando de lado las clases mencionadas anteriormente, es de 97,5%.

En el proyecto que contiene las clases de acceso a datos, obtuvimos una cobertura del 85.4% (ignorando las migraciones y las clases generadas por el compilador, como sucedió en el caso anterior).



Hierarchy	Covered (Blocks)	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)	Covered (%Blocks)
{ } BlogDataAccess.Repositories	356	62	253	8	66	85.17%
ArticleRepository	85	6	64	0	9	93.41%
CommentRepository	46	10	45	0	15	82.14%
OffensiveWordsRepository	38	6	34	0	9	86.36%
SessionRepository	45	8	44	0	12	84.91%
UserRepository	61	15	52	0	19	80.26%

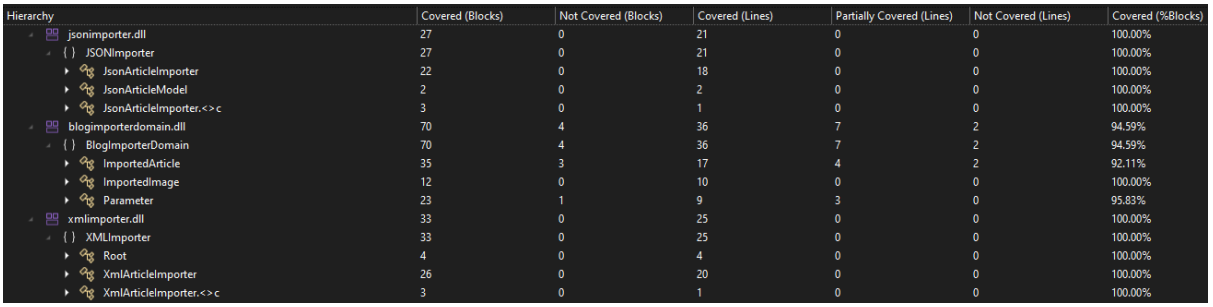
Para el proyecto que contiene los controllers y los distintos modelos de entrada y salida, logramos una cobertura de 97,6% (ignorando la cobertura de los filtros). Hicimos esto porque los docentes nos indicaron que podíamos optar por no testear los filtros.



Hierarchy	Covered (Blocks)	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)	Covered (%Blocks)
{ } BlogApplication.Models.Out	272	16	113	25	4	94.44%
OutModelArticle	63	3	22	5	2	95.45%
OutModelComment	34	5	13	4	1	87.18%
OutModelImporter	28	1	14	2	0	96.55%
OutModelNotification	37	1	15	4	0	97.37%
OutModelParameter	33	1	16	3	0	97.06%
OutModelSession	24	1	9	2	0	96.00%
OutModelUser	29	1	12	3	0	96.67%
OutModelUserScore	20	1	10	2	0	95.24%
OutModelArticle.<>c	2	2	1	0	1	50.00%
OutModelImporter.<>c	2	0	1	0	0	100.00%
{ } BlogApplication.Models.In	108	0	92	0	0	100.00%
InModelArticle	34	0	25	0	0	100.00%
InModelComment	13	0	13	0	0	100.00%
InModelImport	4	0	4	0	0	100.00%
InModelParameter	16	0	15	0	0	100.00%
InModelUser	23	0	20	0	0	100.00%
LogInModel	15	0	14	0	0	100.00%
InModelArticle.<>c	3	0	1	0	0	100.00%
{ } BlogApplication.Filters	0	77	0	0	62	0.00%
{ } BlogApplication.Filters.Authorization	0	111	0	0	58	0.00%
{ } BlogApplication.Controllers	253	2	196	0	1	99.22%
ArticleController	78	0	55	0	0	100.00%
CommentController	36	0	32	0	0	100.00%
OffensiveWordsController	14	0	16	0	0	100.00%
RankingController	20	0	14	0	0	100.00%
SearchArticleController	11	0	9	0	0	100.00%
SessionController	13	0	15	0	0	100.00%
UserController	63	0	46	0	0	100.00%
ArticleController.<>c	8	0	4	0	0	100.00%
RankingController.<>c	4	0	2	0	0	100.00%
SearchArticleController.<>c	2	0	1	0	0	100.00%
UserController.<>c	4	2	2	0	1	66.67%

Error List Developer PowerShell Output Code Coverage Results

Y finalmente los proyectos JSONImporter, XMLImporter y BlogImporterDomain con 100%, 100% y 94,6% respectivamente.



Hierarchy	Covered (Blocks)	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)	Covered (%Blocks)
jsonimporter.dll	27	0	21	0	0	100.00%
{ } JSONImporter	27	0	21	0	0	100.00%
JsonArticleImporter	22	0	18	0	0	100.00%
JsonArticleModel	2	0	2	0	0	100.00%
JsonArticleImporter.<>c	3	0	1	0	0	100.00%
blogimporterdomain.dll	70	4	36	7	2	94.59%
{ } BlogImporterDomain	70	4	36	7	2	94.59%
ImportedArticle	35	3	17	4	2	92.11%
ImportedImage	12	0	10	0	0	100.00%
Parameter	23	1	9	3	0	95.83%
xmlimporter.dll	33	0	25	0	0	100.00%
{ } XMLImporter	33	0	25	0	0	100.00%
Root	4	0	4	0	0	100.00%
XmlArticleImporter	26	0	20	0	0	100.00%
XmlArticleImporter.<>c	3	0	1	0	0	100.00%

Errores conocidos

- No es posible actualizar comentarios.
- No es posible actualizar artículos.
- No se puede traer el feed personalizado de artículos

Vista de Componentes

Diagrama general de paquetes

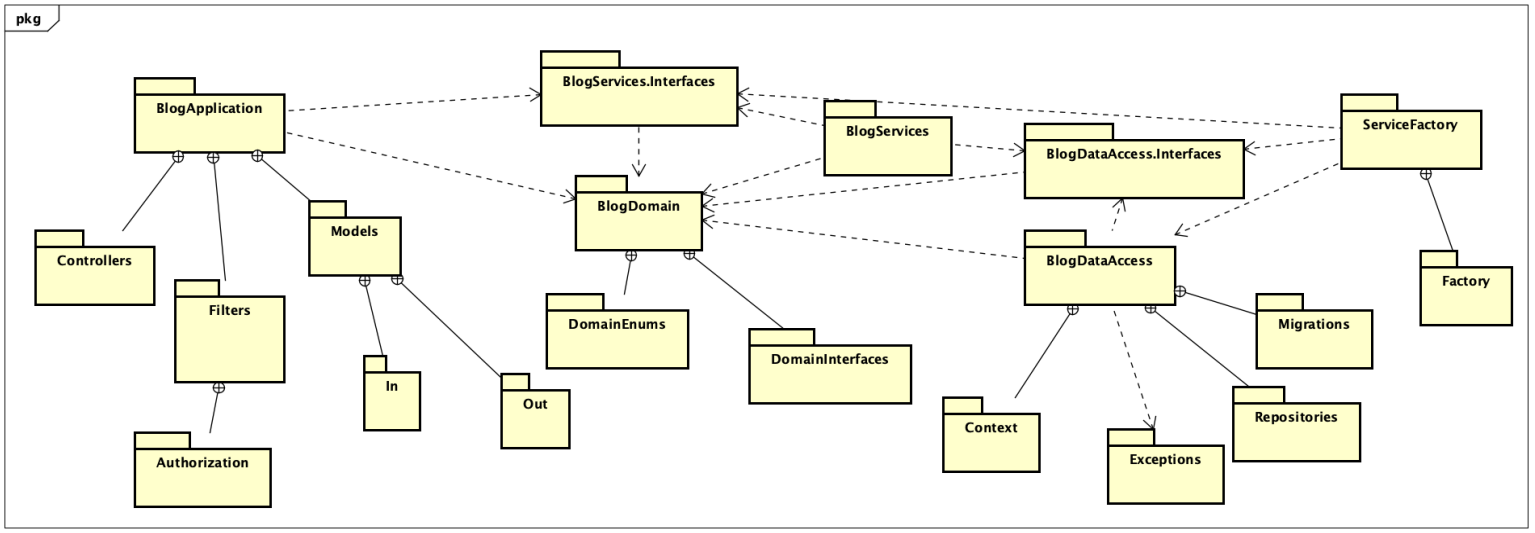
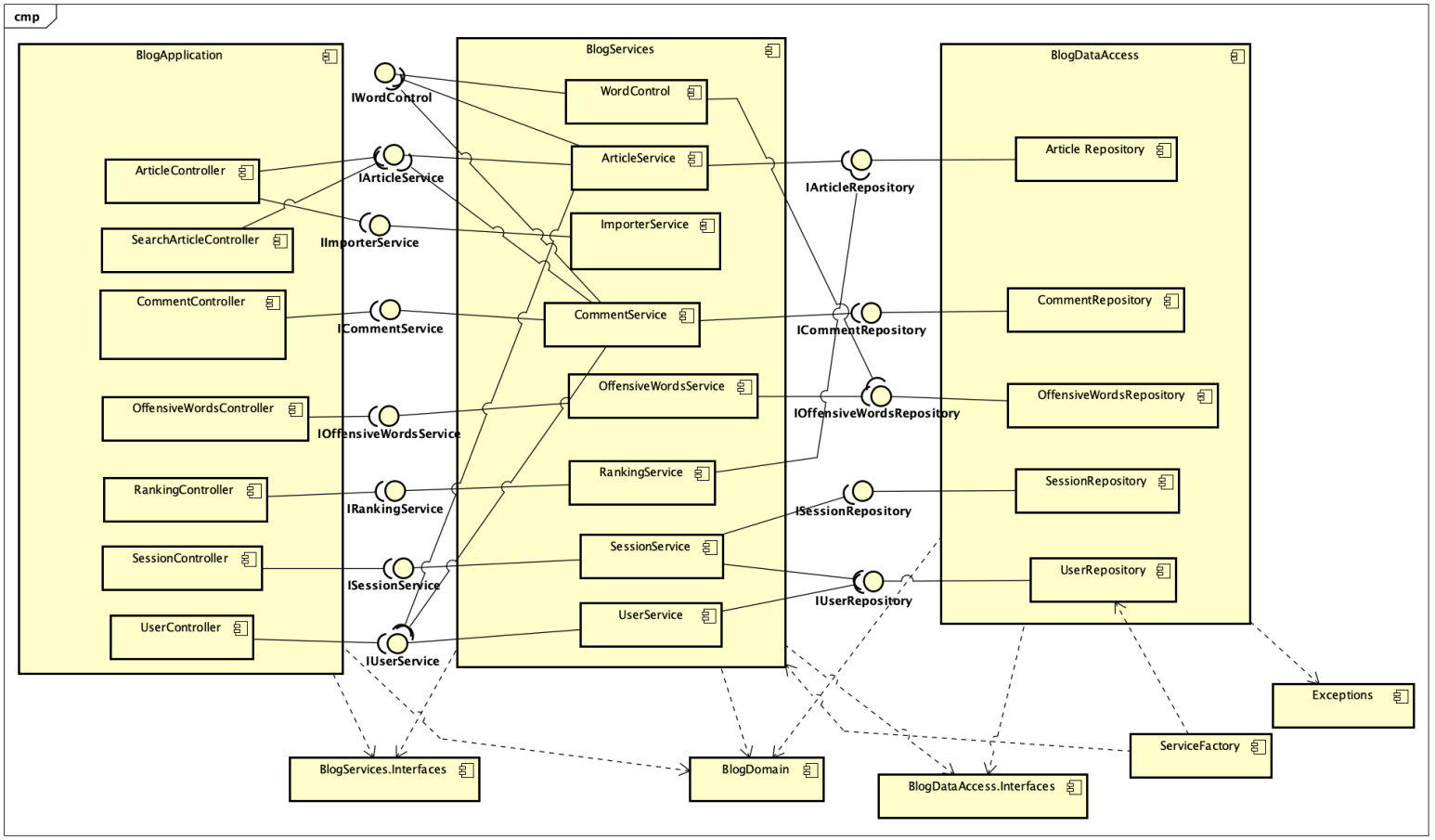


Diagrama de componentes



Los diferentes componentes se comunican a través de Interfaces. Esto nos proporciona:

- Separación de responsabilidades: Cada componente se ocupa de una única función o conjunto de funciones, lo que facilita la comprensión y el mantenimiento del código.

- Bajo acoplamiento: La conexión de componentes a través de interfaces minimiza el acoplamiento entre ellos. Esto significa que cada componente puede evolucionar y modificarse de forma independiente, sin afectar a los demás.

- Facilita la inyección de dependencias: El uso de interfaces permite implementar fácilmente la inyección de dependencias. En lugar de crear instancias de dependencias dentro de un componente, estas se suministran desde el exterior, lo que hace que el código sea más flexible y fácil de probar.

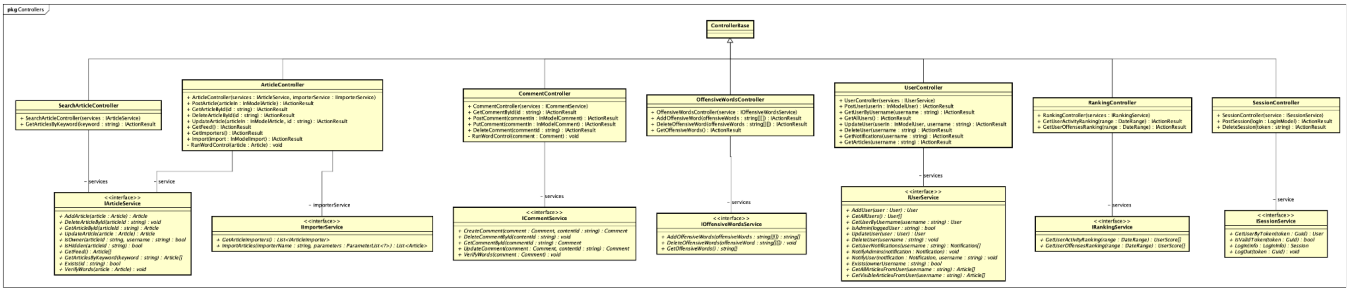
- Facilita la reutilización de código: Al aislar las funcionalidades en componentes separados e interconectarlos a través de interfaces, se fomenta la reutilización de código.

- Facilita la prueba unitaria: La utilización de interfaces para interconectar componentes facilita la creación de pruebas unitarias, ya que permite la sustitución de dependencias reales por objetos simulados (mocks). Esto simplifica el proceso de prueba y asegura que cada componente funcione correctamente.

Vista de diseño

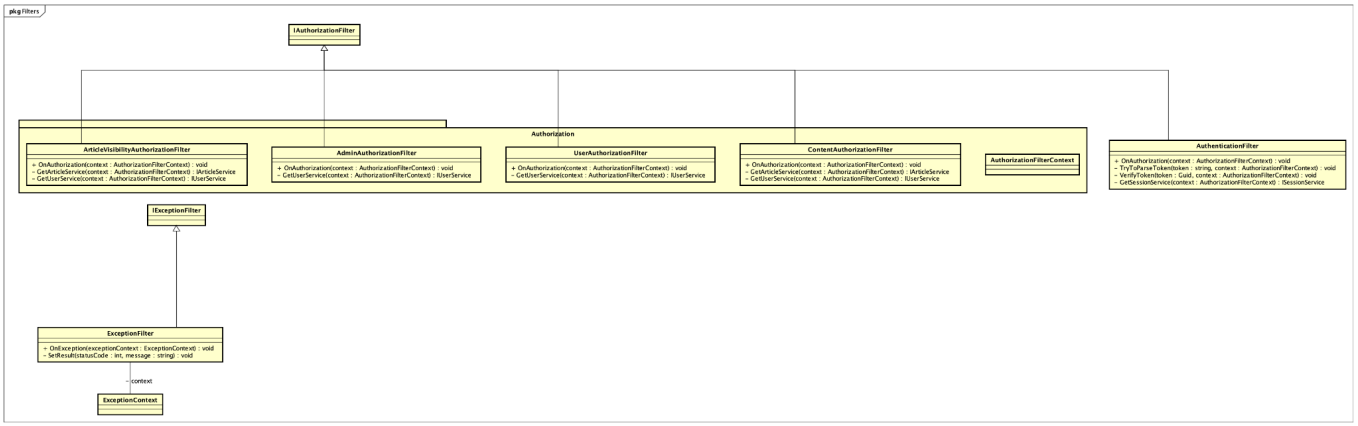
Dejamos los diagramas en una carpeta aparte para facilitar la legibilidad.

Controllers:



Paquete de controllers con los métodos para recibir las peticiones del usuario.

Filters:

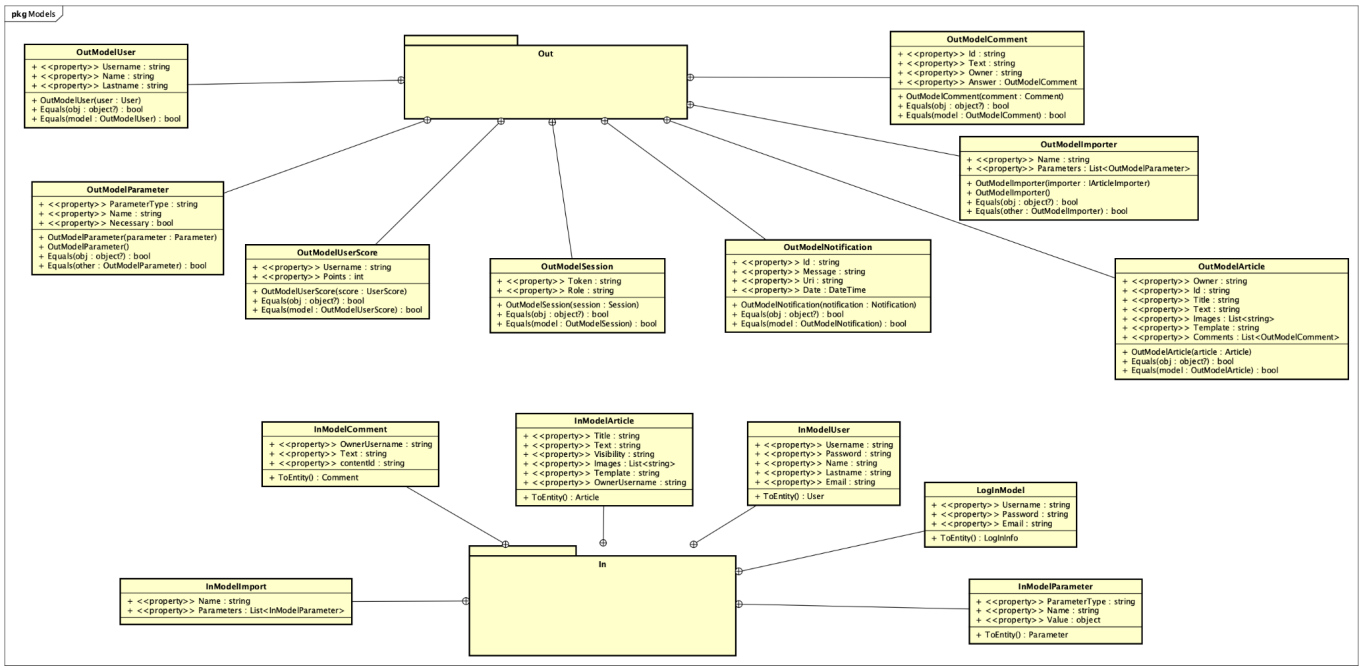


Hicimos uso de filtros con el objetivo de mejorar la calidad del código en los controllers. Para evitar escribir try y varios catch en cada controller, creamos el ExceptionFilter. Es una clase que implementa IExceptionFilter, generando un método que se ejecuta cada vez que una excepción es lanzada. En ese método escribimos los catch que consideramos necesarios para transformar la excepción en un mensaje útil como respuesta a la request. Luego creamos el filtro de autenticación “AuthenticationFilter”, este filtro se encarga de verificar que el recibido en el header sea válido.

Después están los filtros de autorización:

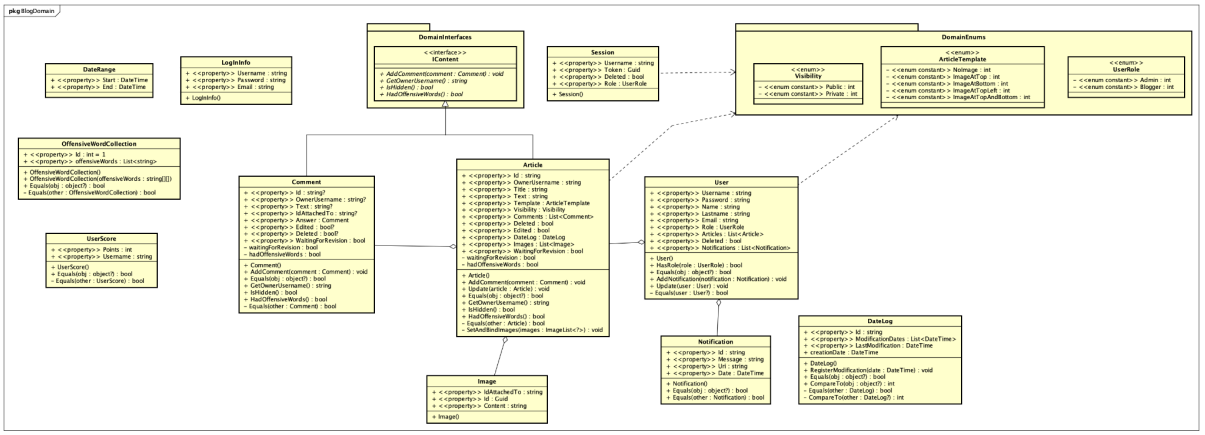
AdminAuthorizationFilter: Verifica que el token pertenezca a un administrador.
ArticleVisibilityAuthorizationFilter: Si el artículo es privado, verifica que el token pertenezca al dueño.
ContentAuthorizationFilter: Verifica que el token pertenezca al dueño del articulo.
UserAuthorizationFilter: Verifica que el token pertenezca al usuario o a un administrador.

Models



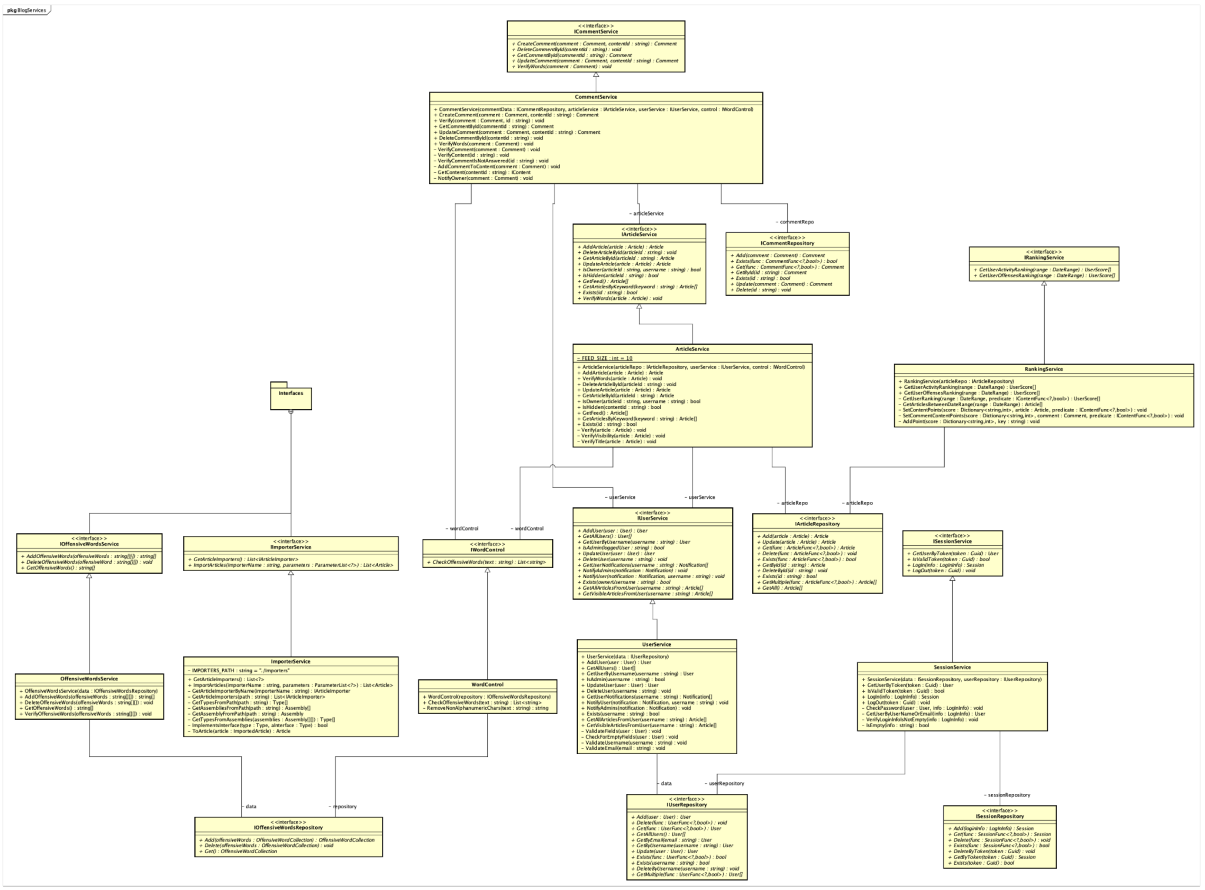
Objetos utilizados para controlar la entrada y salida de información para nuestros controllers, sin exponer la información completa de los objetos de nuestro dominio.

BlogDomain



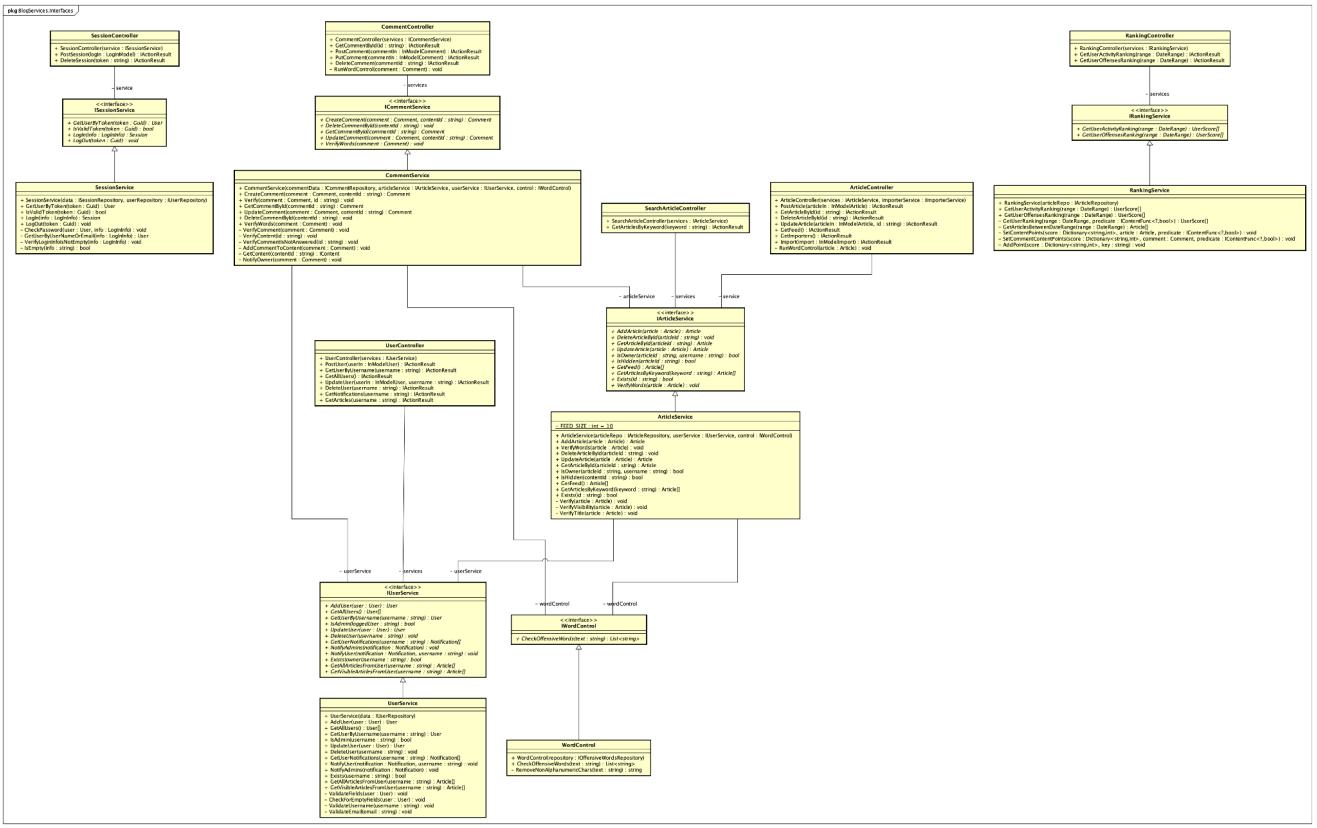
El dominio de nuestra aplicación tiene la información de cada objeto y sus relaciones entre sí. Es importante destacar que tenemos muy pocos métodos ya que la mayor parte de la funcionalidad se encuentra en BlogServices.

BlogServices



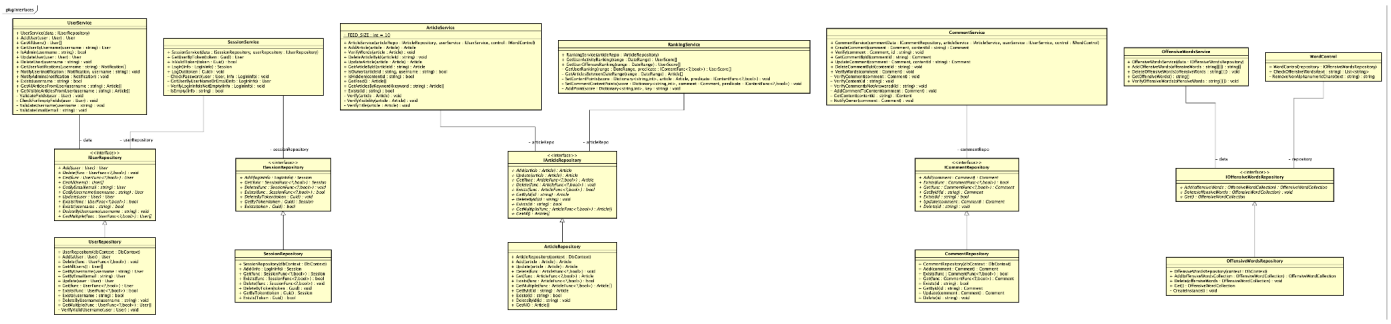
BlogServices contiene la mayor parte de la lógica de nuestra aplicación. Cada Service implementa un contrato dado por una interfaz por un lado y por otro requieren una interfaz que se encargue del manejo de datos.

BlogServices.Interface



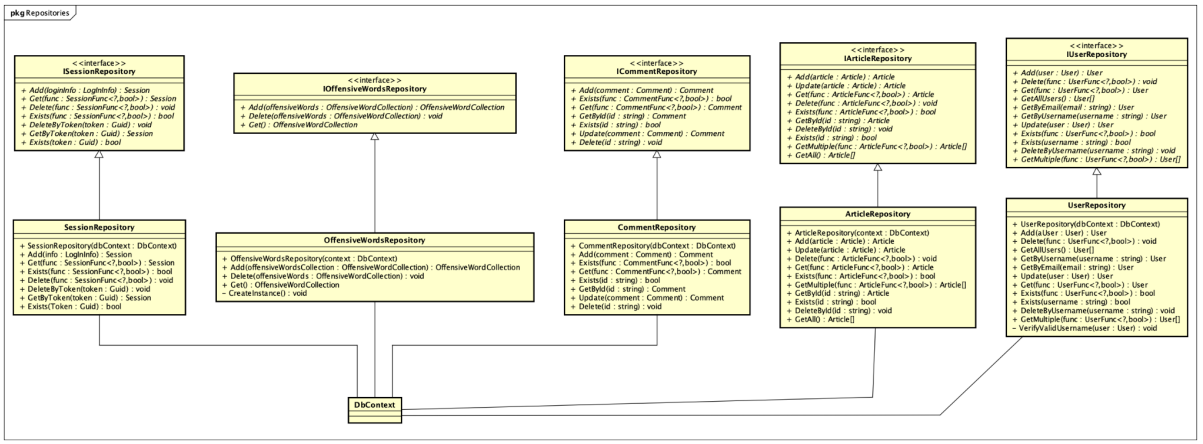
Contiene todos los contratos que necesite cada controller con su respectiva implementación

BlogDataAccess.Interfaces



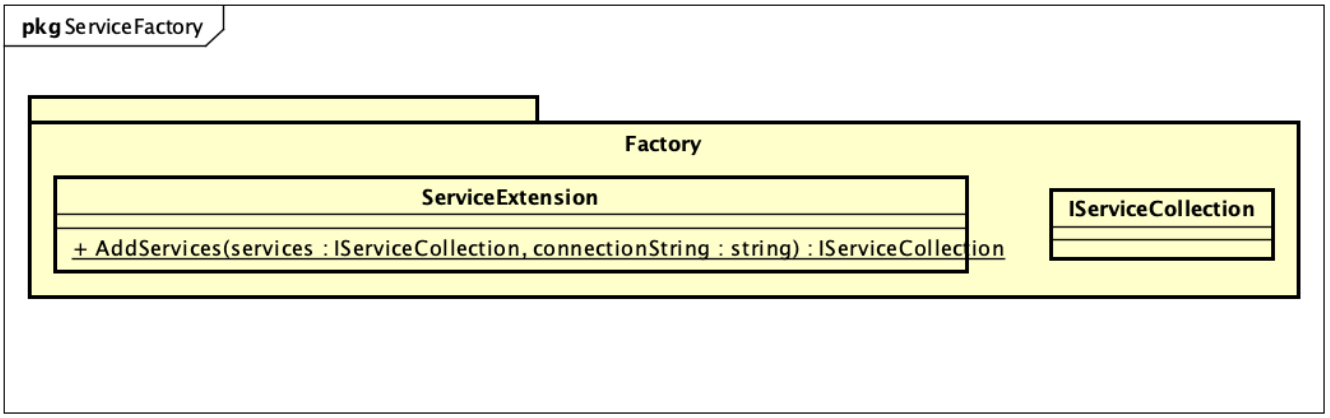
Capa encargada de desacoplar BlogService de la implementación de la base de datos, esto, sumando a la flexibilidad de Entity Framework Core nos permitiría cambiar de ORM

BlogDataAccess



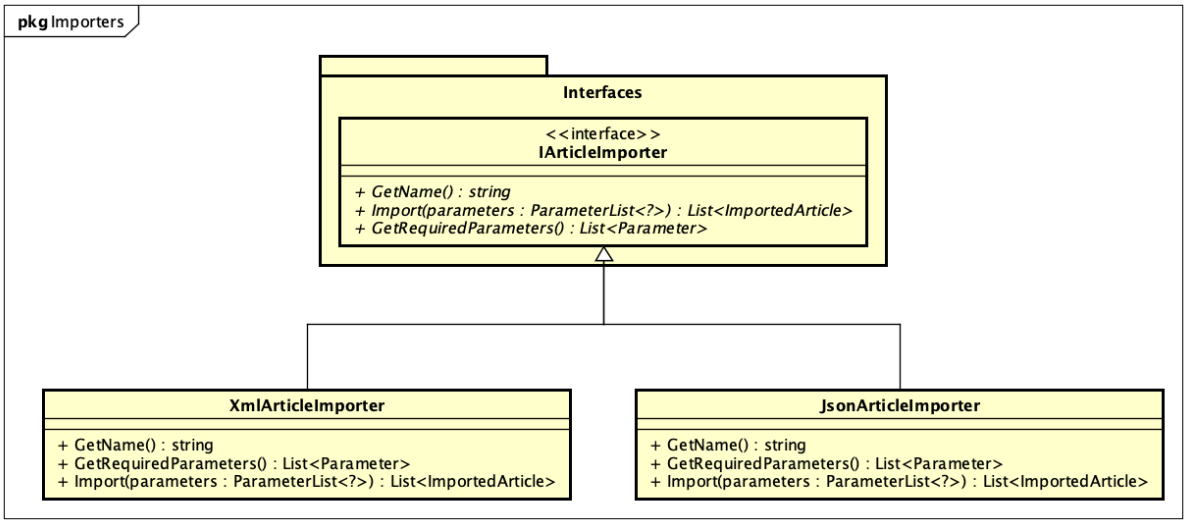
Tiene las implementaciones de cada interfaz repository y será donde se agregarán, modificarán y se consultarán datos de nuestra base de datos.

ServiceFactory



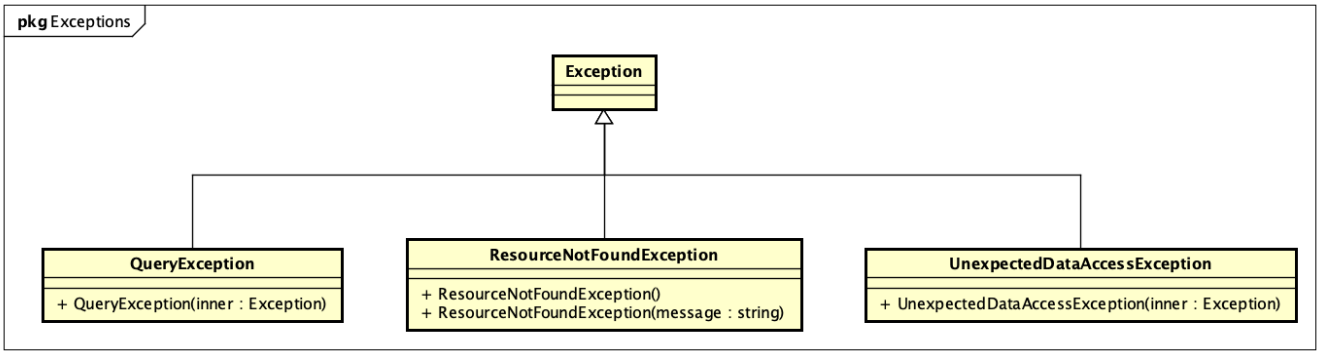
Paquete con la Service Extension, utilizado para la inyección de dependencias

Importers



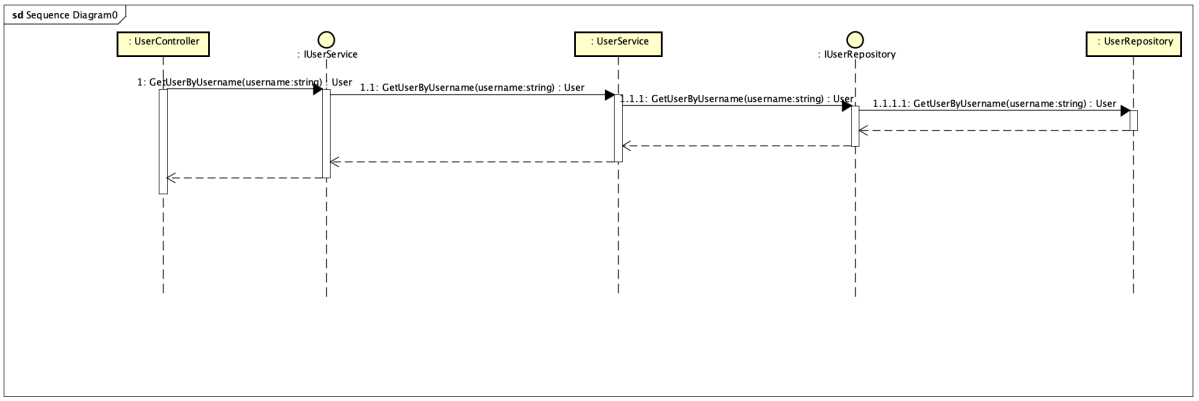
En este diagrama se puede observar cómo las distintas implementaciones de importers implementan la interfaz 'IArticleImporter'. Esta interfaz provee lo necesario para que un tercero pueda implementar un nuevo importador de artículos.

Exceptions

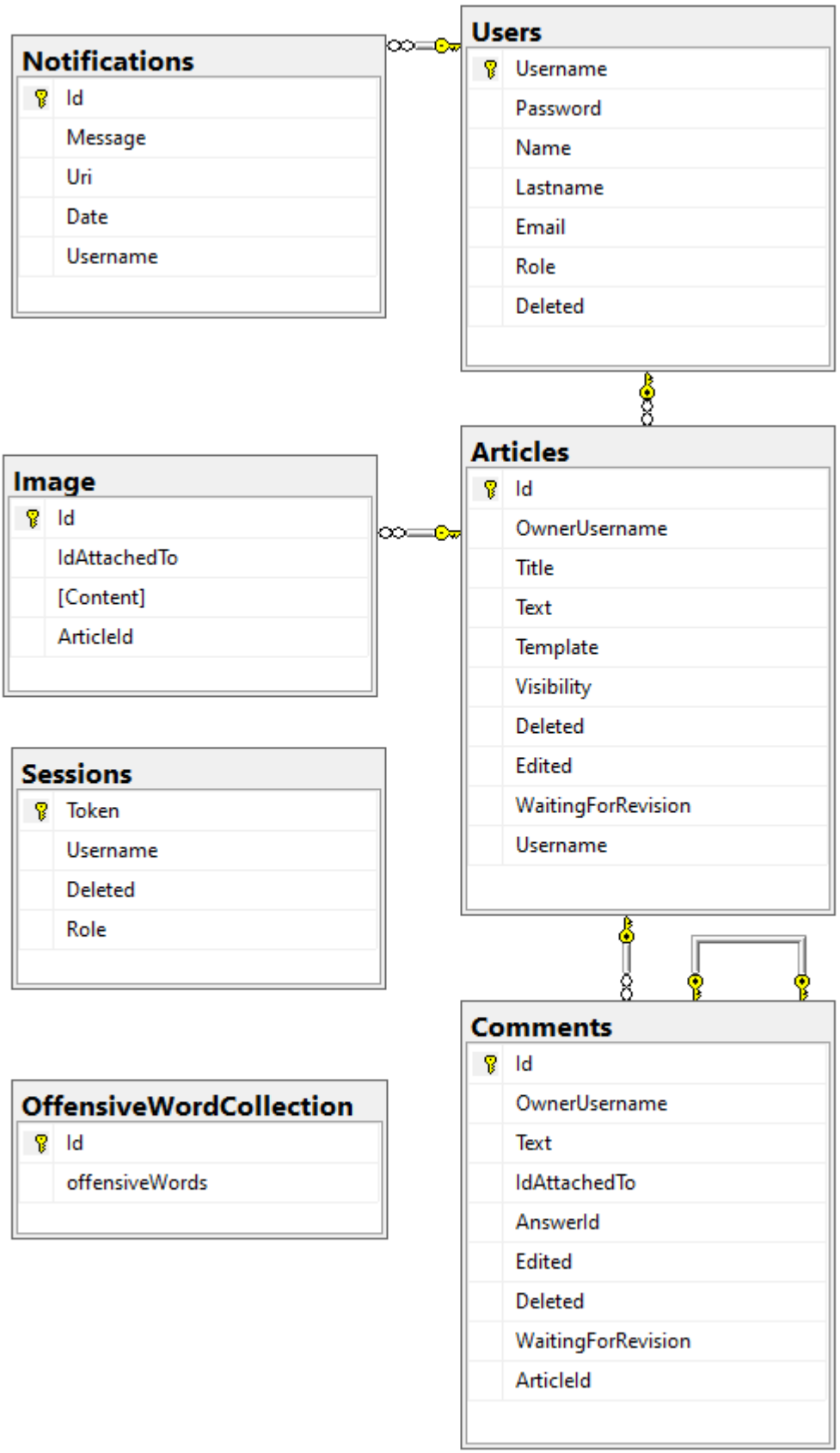


Vista de Procesos

Diagrama de secuencia de GetUserById



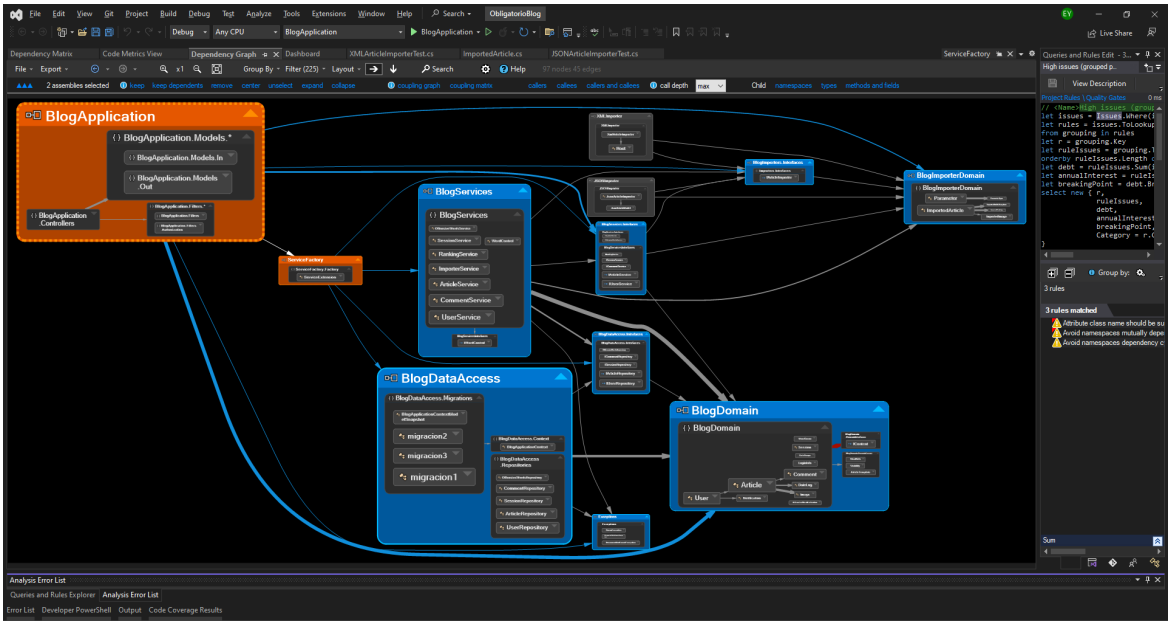
Modelado de tablas



Justificación del diseño

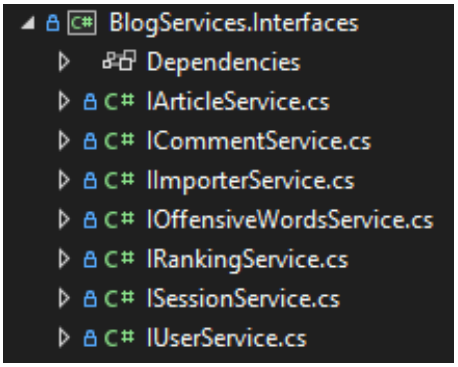
Análisis basado en métricas:

Para llevar a cabo este análisis, hicimos uso de la extensión NDepend. En el diagrama que se muestra a continuación es posible consultar las métricas (H, A, I, D) de los distintos paquetes que componen nuestra solución.



	H (Cohesión relacional)	A (Nivel de abstracción)	I (Nivel de inestabilidad)	D (Distancia normalizada)
BlogApplication	3.06	0	1	0
BlogServices	1.92	0.08	0.99	0.07
BlogDomain	2	0.05	0.31	0.63
BlogDataAccess	1.74	0	0.99	0.01
ServiceFactory	1	0	0.98	0.02
Exceptions	1	0	0.5	0.5
BlogServices.Interface	1	0.7	0.5	0.2
BlogDataAccess.Interface	1	0.62	0.55	0.18
BlogImporterDomain	1.44	0	0.59	0.41
BlogImporters.Interfaces	1	0.25	0.62	0.12
JSONImporter	1.6	0	1	0
XMLImporter	1.6	0	1	0

Es importante aclarar que los datos arrojados por el análisis de NDepend toman en cuenta assemblies de .NET y código generado en compilación. Por este motivo, las métricas se ven afectadas, generando casos extraños como que un paquete que posee únicamente clases abstractas, tenga un valor de A = 0.7.



Un ejemplo de esto, es el paquete 'BlogServices.Interfaces'. Está compuesto únicamente por interfaces, pero según el análisis de NDepend, su abstracción es de 0.7.

Otro ejemplo es el paquete 'BlogDomain'. Este paquete no tiene dependencias hacia otros paquetes (Ce = 0), sin embargo, la inestabilidad obtenida con NDepend es de 0.31.

Según la fórmula de inestabilidad, I debería ser 0:

$$I = Ce / (Ce + Ca) = 0 / (0 + Ca) = 0$$

Investigamos si es posible hacer el análisis sin tomar en cuenta los assemblies de .NET, pero debido a la poca información en internet sobre NDepend y el escaso tiempo, la investigación no dio frutos.

Si bien, las métricas no son exactamente correctas, aún es posible dar un análisis coherente acerca de la estructura de nuestra solución.

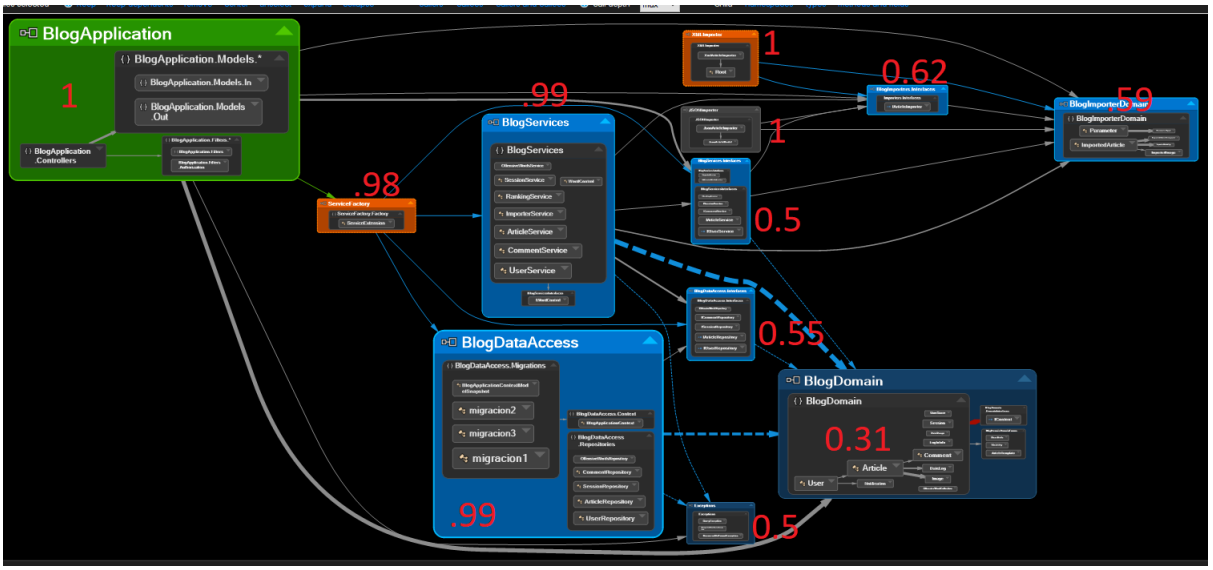
Para empezar, los valores de la métrica H se mantienen en entre 1.5 y 4.0 para todos los paquetes, a excepción de los paquetes utilizados para almacenar interfaces. Esto ocurre porque no tuvimos la necesidad de generar dependencias entre las interfaces dentro del paquete.

Los valores de distancia se mantienen razonables (menores a 0.4) excepto para BlogDomain, BlogImportersDomain y Exceptions. Esto es inevitable ya que la naturaleza de estos proyectos suele ser muy concreta (A bajo) y, dado que son la base de la solución, son requeridos por muchas clases dentro de otros paquetes además de que no dependen de clases externas (I bajo). Al tener un I y un A bajo, quedan situados en la "Zona de dolor", pero es natural, tratándose de proyectos de dominio y excepciones.

Principios de acoplamiento de paquetes:

ADP (Acyclic Dependencies Principle):
Como se puede ver en nuestro diagrama de paquetes, no existe ninguna dependencia circular entre los paquetes.

SDP (Stable Dependencies Principle):
Este principio indica que el sentido de las dependencias debe ser siempre hacia clases más estables.



Este es el diagrama dibujado por NDepend. Le agregamos el valor de inestabilidad en color rojo a cada paquete. Aquí se puede apreciar que las dependencias siempre van hacia paquetes con un valor de I menor. La única excepción es ServiceFactory, cuya inestabilidad es de 0.98 y depende de BlogService y BlogDataAcces (ambos con I = 0.99). Esto podría ser un error en las métricas generadas por NDepend, pero en caso de que no sea así, igual se puede justificar la existencia de ese proyecto ya que su único propósito es facilitar la inyección de dependencias.

SAP (Stable Abstraction Principle):
Este principio plantea que los paquetes estables deben ser abstractos.
Es difícil analizar este principio con las métricas obtenidas debido a que la inestabilidad de todos los paquetes (según NDepend) es mayor a la que deberían tener porque toma en cuenta las dependencias externas a la solución.

Sin embargo, podemos comparar qué tan inestables son los paquetes de interfaces en comparación a los paquetes de controladores, servicios y repositorios.

Si nos fijamos en la tabla, la inestabilidad de los paquetes abstractos varía entre 0.5 y 0.6 (dependiendo de que tantas interfaces contengan y que tantas clases del dominio requieran). Y la inestabilidad de los paquetes con lógica concreta varía entre 0.99 y 1.00. Entonces, teniendo eso en mente, podemos estimar que si NDepend no inflara la inestabilidad de los paquetes, los paquetes que contienen interfaces tendrían una inestabilidad cercana a 0 (no sería 0 porque dependen de clases del dominio).

Principios de cohesión de paquetes:

CCP (Common closure principle):

Este principio dice que las clases que cambian juntas deben estar en el mismo paquete.
Por la naturaleza de la arquitectura REST, no consideramos estar cumpliendo este principio ya que si es necesario cambiar una feature, probablemente haya cambios en el controlador, en el servicio y en el repositorio. Como estas clases están separadas en distintos proyectos (por el modelo de capas), no están juntas.

CRP (Common reuse principle):
Este principio indica que las clases que no se utilizan en conjunto, no deberían estar en el mismo paquete.
Nuestra solución cumple con este principio ya que, por ejemplo, todas las clases de 'BlogServices' se utilizan para controlar la lógica de negocio correspondiente a las peticiones que le comunica el proyecto 'BlogApplication'. Por lo tanto, todas las clases de servicios van juntas en ese proyecto.

Decisiones de diseño:

Acerca del manejo de importadores para los artículos, la consigna indica que se desea la posibilidad de permitir a terceros desarrollar distintos importadores, ya que no se conocen los distintos métodos de importación que puedan surgir a futuro.

Para cumplir con este requerimiento, aplicamos Reflection, pero además, generamos un nuevo proyecto llamado 'BlogImportersDomain'. El propósito de este proyecto es el de almacenar clases similares a las del dominio, pero conteniendo únicamente la información relevante para los desarrolladores de los importadores.

El objetivo es que los importadores no trabajen con clases del dominio principal directamente. Así no tenemos la necesidad de darle acceso a nuestro proyecto 'BlogDomain' a terceros. Lo único que necesita un tercero para desarrollar un importador es 'BlogImporters.Interface' y 'BlogImportersDomain'.

En la práctica, cuando un servicio instancia uno de estos importadores e importa una lista de 'ImportedArticles', esta lista es transformada en una lista de 'Articles' por el servicio mismo.

Si a futuro se quisiera importar un recurso distinto (ej. usuarios), bastaría con escribir la lógica correspondiente en la clase 'ImporterService', generar una clase similar a 'User' en 'BlogImportersDomain', desarrollar los action methods correspondientes y crear una interfaz para que un tercero sepa qué métodos debe implementar en su importador.

Mejoras respecto a la primera entrega

- Llevamos a cabo un ‘rework’ que consistió en un refactor de todos los tests que desarrollamos para la primera entrega. Esto lo hicimos con el objetivo de mejorar la calidad del código, haciéndolos más fáciles de entender. Más precisamente, lo que hicimos fue usar **[TestInitialize]** para reutilizar código y **[ExpectedException]** para verificar excepciones de manera más prolija.
- Completamos los requerimientos que nos quedaron pendiente para la primera entrega.
- El atributo de tipo string Image de la clase ‘Article’, pasó a ser una lista de un nuevo tipo llamado ‘Image’. Esto permite asignarle más de una imágen a un artículo y además facilita la persistencia de las imágenes en la base de datos.

Set de datos de prueba

Generamos un set de datos de prueba que contiene a 3 usuarios: admin, OniTheDemon y Titolandia22 (todos con contraseña: 123), 2 posts de diferentes usuarios, varios comentarios de los 3 usuarios y registramos algunas palabras ofensivas.

Anexo:

Especificación de la API:

Obtener importadores de artículos

URI: articles/importers
ACTION: GET
PARÁMETROS: Ninguno
PROPÓSITO: Retorna una lista con los importadores de artículos disponibles en el servidor.

Importar artículos

URI: articles/import
ACTION: POST
PARÁMETROS: InModelImport
PROPÓSITO: Importa los artículos indicados por los parámetros contenidos en el InModelImport recibido.

Actualizar comentario

URI: comments/{id}
ACTION: PUT
PARÁMETROS: InModelComment
PROPÓSITO: Actualiza el comentario.

Eliminar comentario

URI: comments/{id}
ACTION: DELETE
PARÁMETROS: InModelComment
PROPÓSITO: Elimina el comentario indicado.

Registrar palabras ofensivas

URI: offensiveWords
ACTION: POST
PARÁMETROS: string[]
PROPÓSITO: Agrega palabras al registro de palabras ofensivas.

Eliminar palabra ofensiva

URI: offensiveWords
ACTION: DELETE
PARÁMETROS: string[]
PROPÓSITO: Elimina una palabra del registro de palabras ofensivas.

Obtener lista de palabras ofensivas

URI: offensiveWords
ACTION: GET
PARÁMETROS: Ninguno
PROPÓSITO: Retorna el registro de palabras ofensivas.

Ranking de actividad

URI: ranking/activity
ACTION: GET
PARÁMETROS: DateRange
PROPÓSITO: Retorna una lista que representa el ranking de usuarios según su actividad.

Ranking de ofensas

URI: ranking/offenses
ACTION: GET
PARÁMETROS: DateRange
PROPÓSITO: Retorna una lista que representa el ranking de usuarios según sus ofensas.

Buscar artículo por palabra

URI: search/{keyword}
ACTION: GET
PARÁMETROS: string
PROPÓSITO: Retorna una lista de artículos que contienen al menos una ocurrencia de la palabra pasada por parámetro.

Obtener las notificaciones de un usuario

URI: users/{username}/notifications
ACTION: GET
PARÁMETROS: string
PROPÓSITO: Retorna una lista que contiene las notificaciones de un usuario.

Obtener los artículos de un usuario

URI: users/{username}/articles
ACTION: GET
PARÁMETROS: string
PROPÓSITO: Retorna una lista que contiene los artículos de un usuario.