

Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio 2 - Diseño de Aplicaciones 2
Descripción del Diseño

Entregado como requisito para la obtención del título de
Ingeniero en Sistemas

Milena dos Santos – 254813

Guzmán Dupont – 230263

Julieta Sarantes – 251105

Link a repo: <https://github.com/ORT-DA2/230263-251105-254813>

Tutores: Francisco Bouza, Juan Irabedra, Santiago Tonarelli

Declaración de Autoría

Nosotros, Julieta Sarantes, Guzmán Dupont, Milena Dos Santos, declaramos que el trabajo que se presenta en este obligatorio es de nuestra propia mano. Podemos asegurar que:

- El obligatorio fue producido en su totalidad mientras realizábamos Diseño de Aplicaciones 2.
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad.
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, el obligatorio es enteramente nuestro.
- En el obligatorio, hemos acusado recibo de las ayudas recibidas.
- Cuando el obligatorio 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.

Julieta Sarantes Machín

14/06/2023

Guzmán Dupont Wolcan

14/06/2023

Milena dos Santos

14/06/2023

Índice

Descripción general del sistema.....	4
Funcionalidades sin implementar.....	5
Descripción de paquetes.....	6
Vista de desarrollo.....	6
Vista lógica.....	8
Vista de Proceso.....	10
Vista física.....	11
Descripción de jerarquías.....	11
Mecanismo de acceso a la base de datos.....	12
Patrones de diseño y principios SOLID.....	13
Funcionalidades adicionales.....	14
Métricas de diseño.....	16
TDD.....	19
Clean code.....	21
Interfaz de usuario.....	22
Especificación de la API.....	24

ATENCIÓN: Por problemas con github, no pudimos subir la solución a la rama main, por lo que la versión final está en la rama dev.

Descripción general del sistema.

A partir de la entrega anterior y los nuevos requerimientos solicitados, extendimos y mejoramos nuestra solución. Se agregaron los nuevos requerimientos a la Web API que teníamos definida anteriormente, y se construyó la interfaz de usuario para esta.

Usamos nuevas tecnologías brindadas por los docentes, como Reflection y el framework de Angular.

Nuestra tarea era construir un sistema de Blogs. El cual debía tener variadas funcionalidades y dependiendo del rol que cada usuario tenía, lo que tenía habilitado hacer.

En cuanto al sistema, se puede tener tres tipos de roles: Administrador, Blogger y a su vez, también se puede agregar el de Moderador, a alguno de los anteriores. En este caso, nosotros optamos por ponerle un atributo "Rol" al usuario y si es administrador, toma el valor "Admin", en caso de que sea blogger, el valor es "Blog".

Los tres tipos de usuarios pueden agregar artículos, comentarios, buscar un artículo a través de un texto, loguearse, desloguearse, registrarse y modificar su perfil.

Así como también, puede agregar palabras ofensivas a una lista estática, la cual comparten entre ellos. Son los únicos habilitados para consultar sobre la distribución de los artículos publicados por mes para un año dado. A su vez, pueden listar el log de entradas al sistema. Con respecto a los nuevos requerimientos para el usuario administrador, se pueden ver las palabras detectadas en una articulo en especifico y también puede editarlo. Puede consultar el ranking de usuarios entre un rango de fechas en las que más publicaron artículos con palabras ofensivas.

En conclusión, creemos que construimos un buen sistema a pesar de la falta de tiempo y algunos requerimientos que no pudimos llegar a completar. Estamos al tanto de que siempre se puede mejorar, pero estamos satisfechos con el resultado.

Funcionalidades sin implementar

En esta sección aclaramos algunas funcionalidades del backend que no llegamos a implementar:

- Ocultar automáticamente artículo o comentario.
- Los administradores deben ser notificados de alguna forma visible que se ha detectado una palabra ofensiva en un artículo o comentario.
- Proporcionar una advertencia al usuario que ha publicado el artículo o comentario explicando la razón por la cual no ha sido publicado.
- En caso de ser rechazado el artículo o comentario, no se debe mostrar el artículo o comentario y dejar una marca que el mismo fue eliminado.
- En el caso del ranking de usuarios, nos faltó hacerlo por palabras ofensivas en comentarios sumadas a la de los artículos, solo tenemos las encontradas en estos últimos.
- No pudimos manejar bien las excepciones, teniendo una específica para cada tipo de error.

Aquí se aclaran las funcionalidades del frontend que no llegamos a implementar:

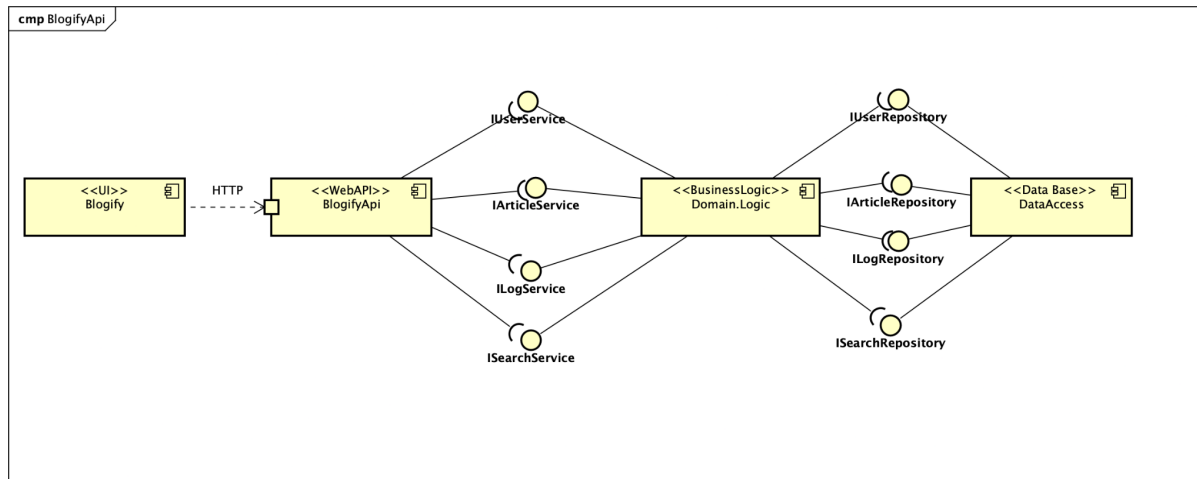
- Reflection.
- Ranking de usuario por año dado.
- Advertencia al usuario por palabras ofensivas.
- Listado de palabras ofensivas en comentarios o artículos.
- Listado de logs de entradas al sistema.
- Búsquedas realizadas por los usuarios.
- Buscar un artículo
- Modificar un artículo

Descripción de paquetes

Uso del modelo 4 + 1 para la arquitectura del sistema.

Vista de desarrollo

Para la vista de desarrollo se diseñó el siguiente diagrama de componentes.



A su vez, se diseñó el siguiente diagrama de paquetes (parte 1).

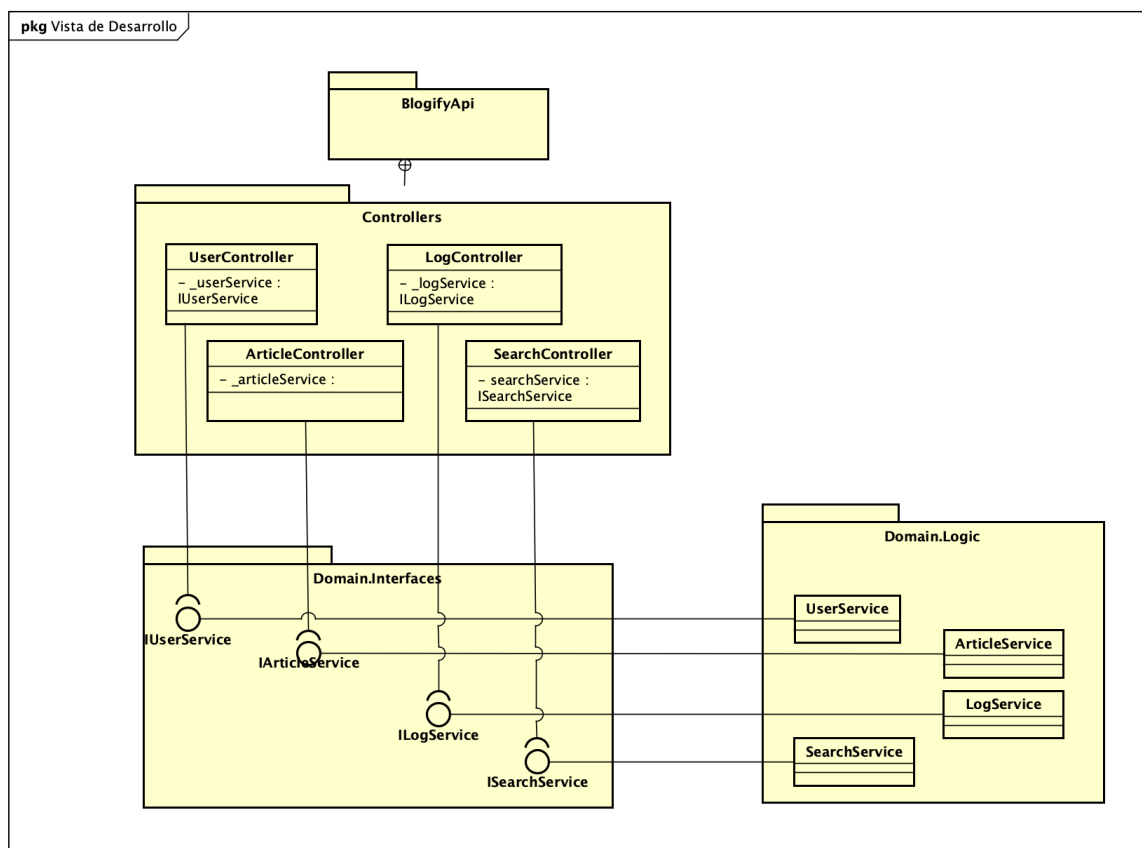
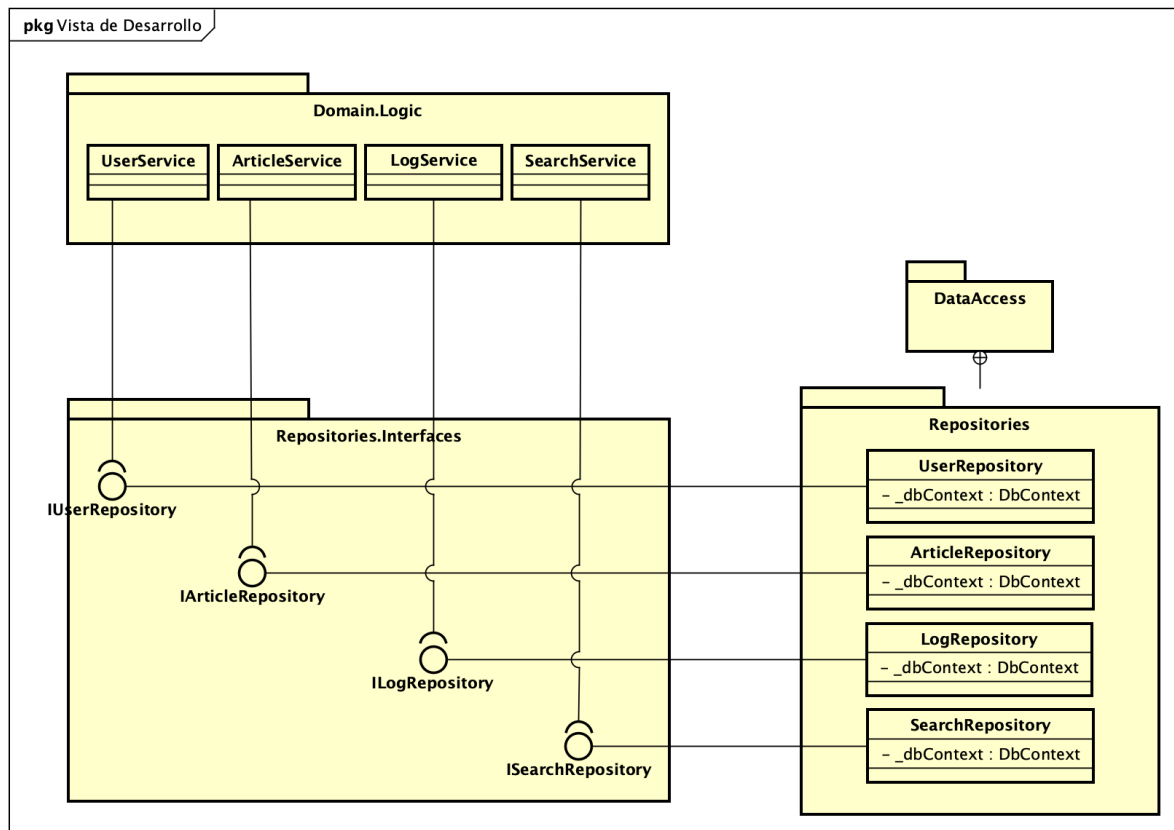
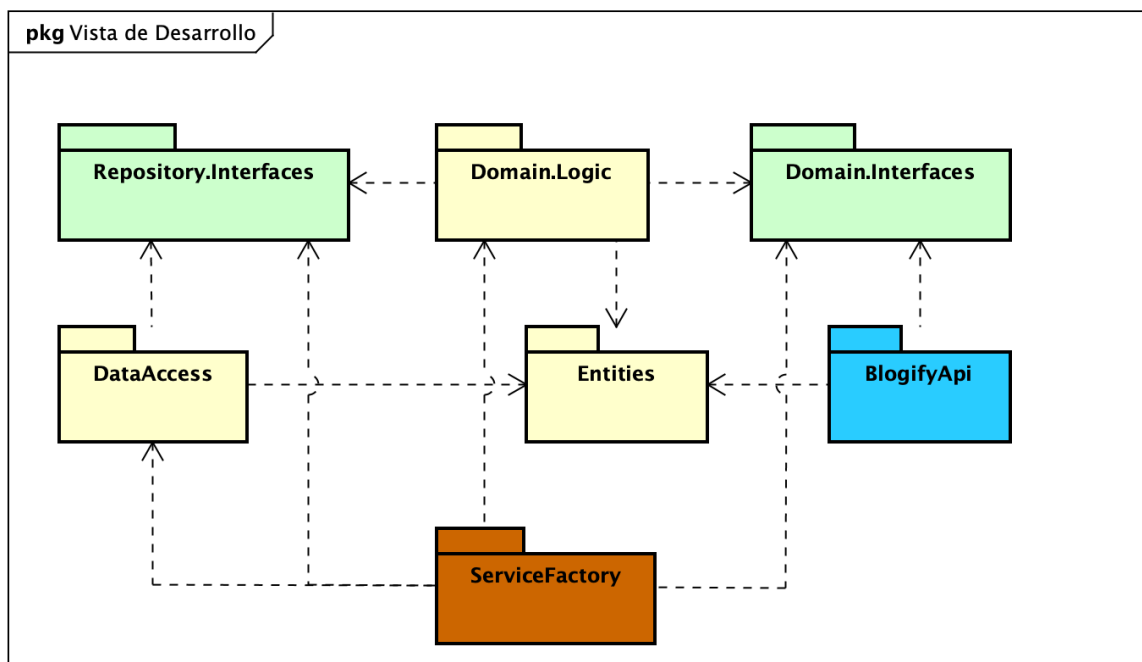


Diagrama de paquetes (parte 2).



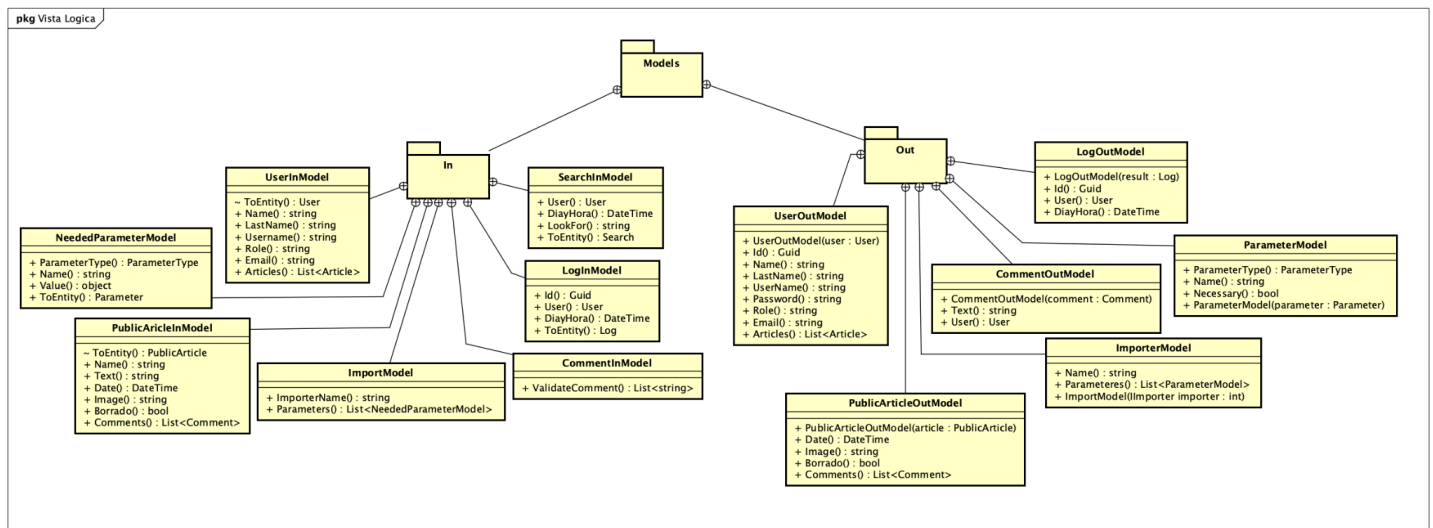
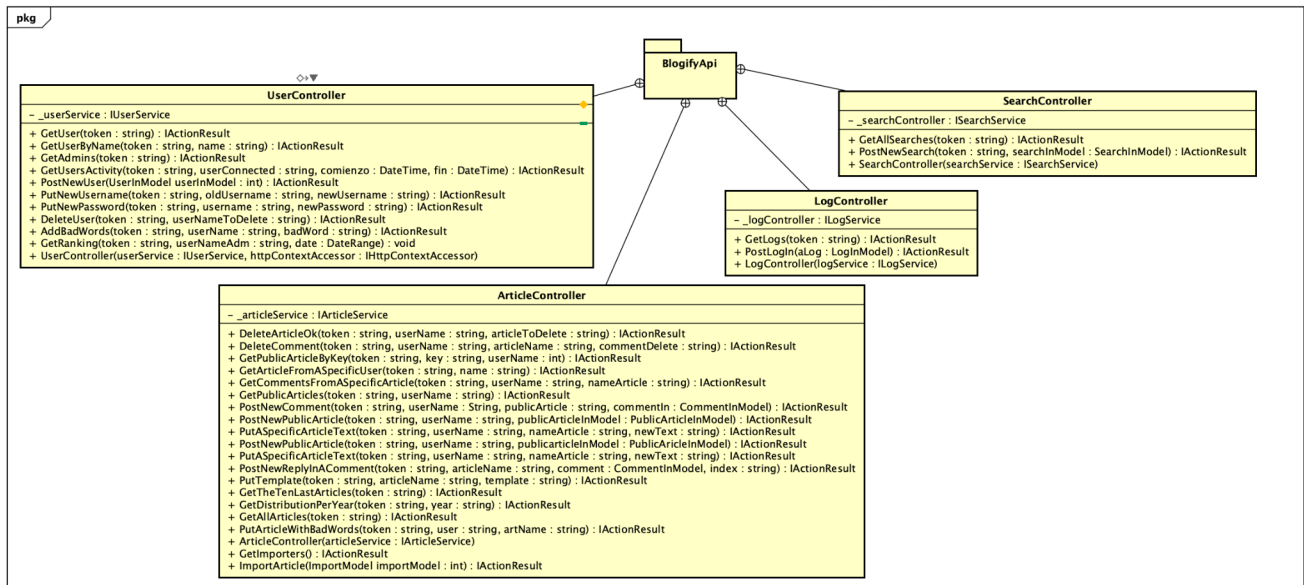
En este diagrama de paquetes se puede observar la inyección de dependencia realizada gracias a la herramienta provista por Visual Studio.

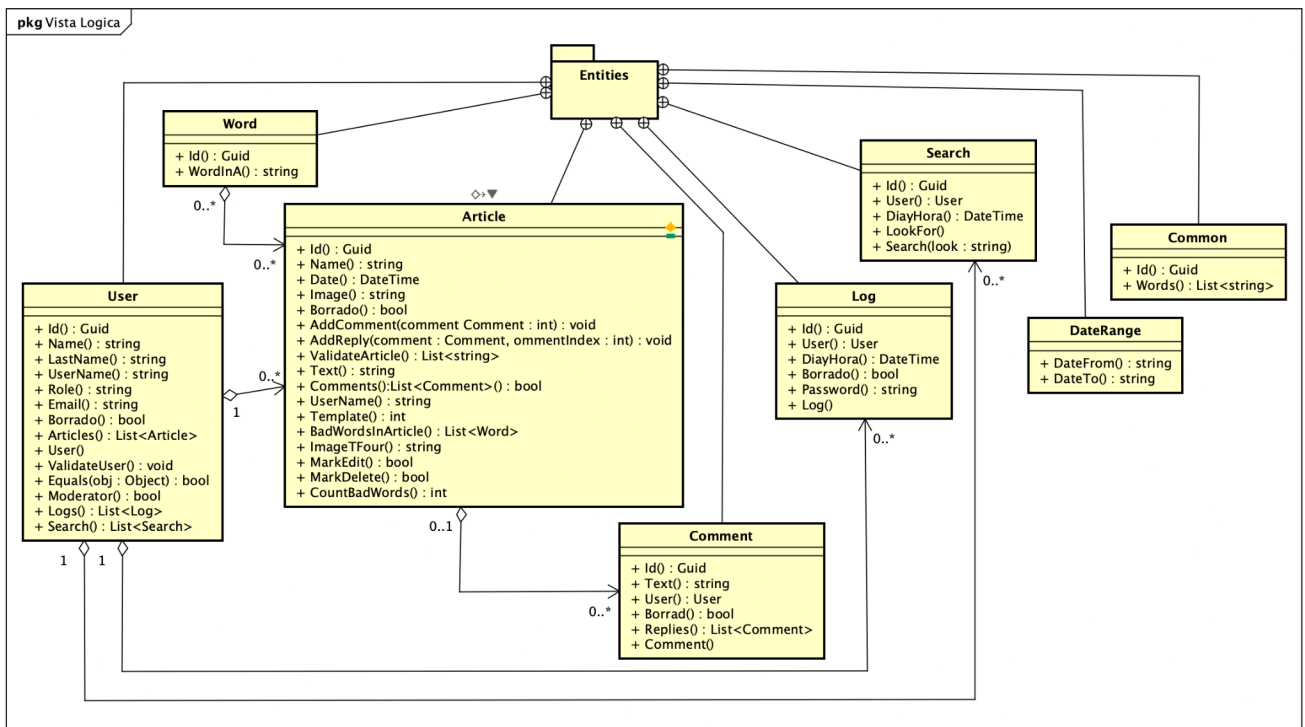
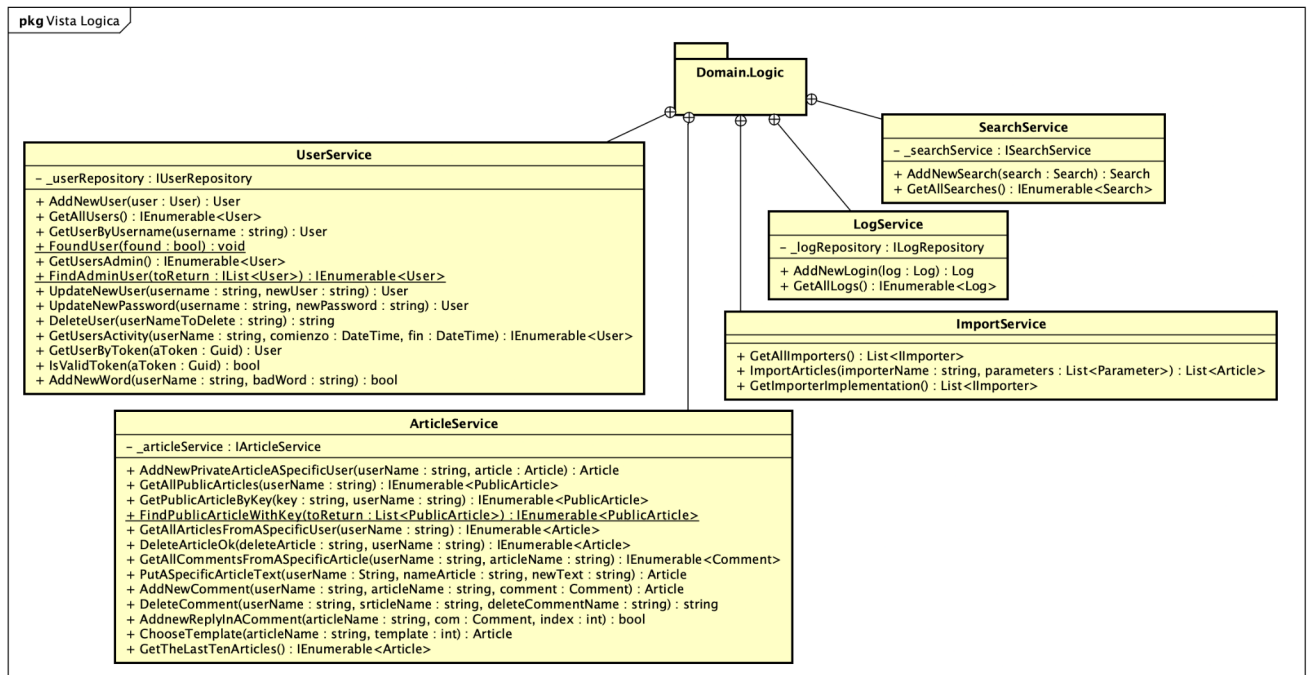


Aclaración: En el diagrama de paquetes (parte 1) no fue diagramado el *ImporterService* de la clase *Domain.Interface* ni el *ImporterService* de *Domain.Logic*, ya que son clases específicas de la funcionalidad del Importador.

Vista lógica

Para la vista lógica se crearon los siguientes diagramas:



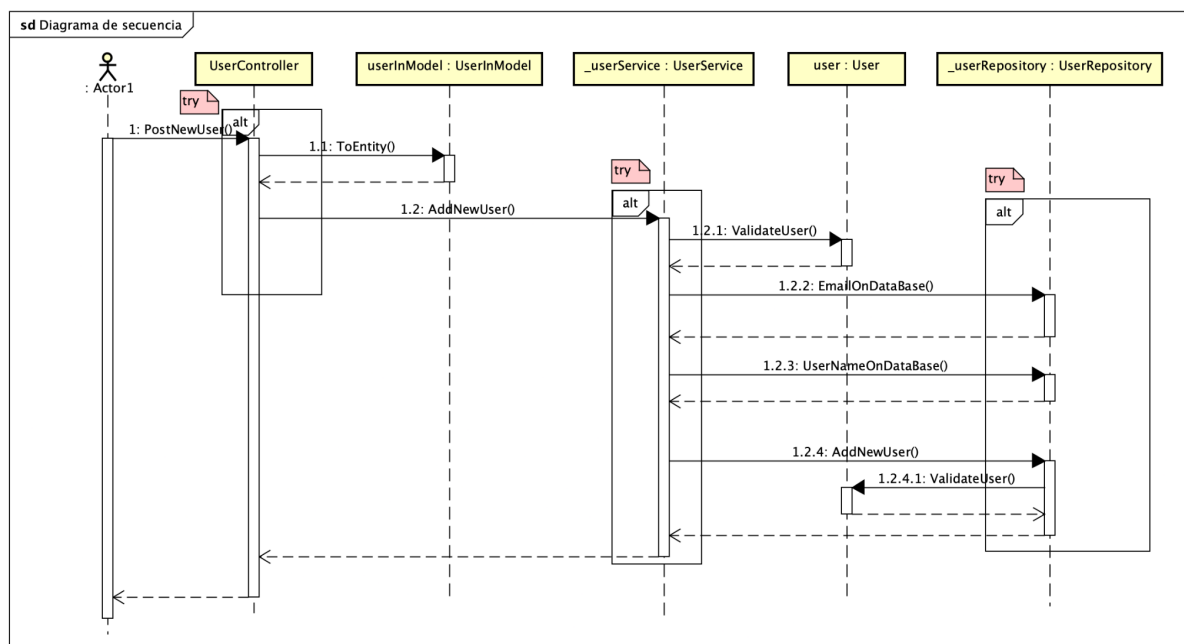


Podemos observar que cada uno de los diagramas utiliza el conector *nesting* para las clases que están contenidas en los respectivos paquetes.

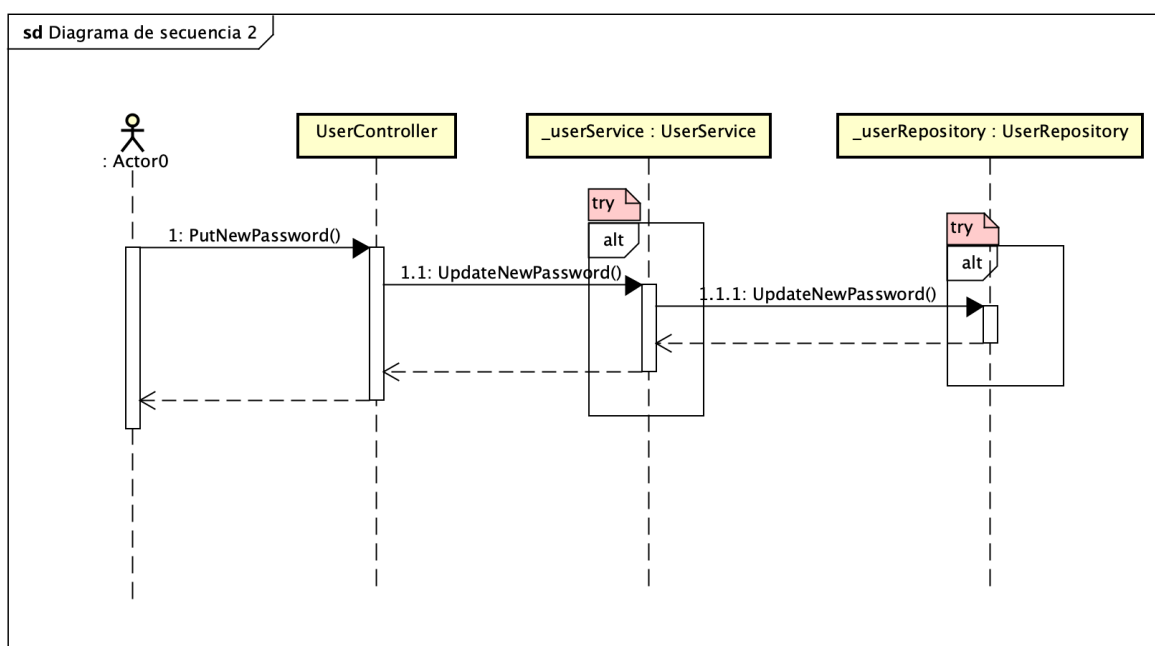
Vista de Proceso

Para la vista de proceso creamos un diagrama de secuencia que sigue los pasos que hace la aplicación al momento de crear un usuario y crear un artículo.

El actor comienza consultando al recurso users. Dentro del controlador UserController se llama a ToEntity de la clase UserInModel para simular un DTO. Luego se llama al AddNewUser de la clase UserService. Desde esa clase se llama a varios métodos para verificar que el usuario no exista en la base de datos previamente. Luego se retorna el usuario ingresado.



El otro diagrama que decidimos implementar es el actualizar contraseña que es un poco más sencillo que el anterior.



Vista física

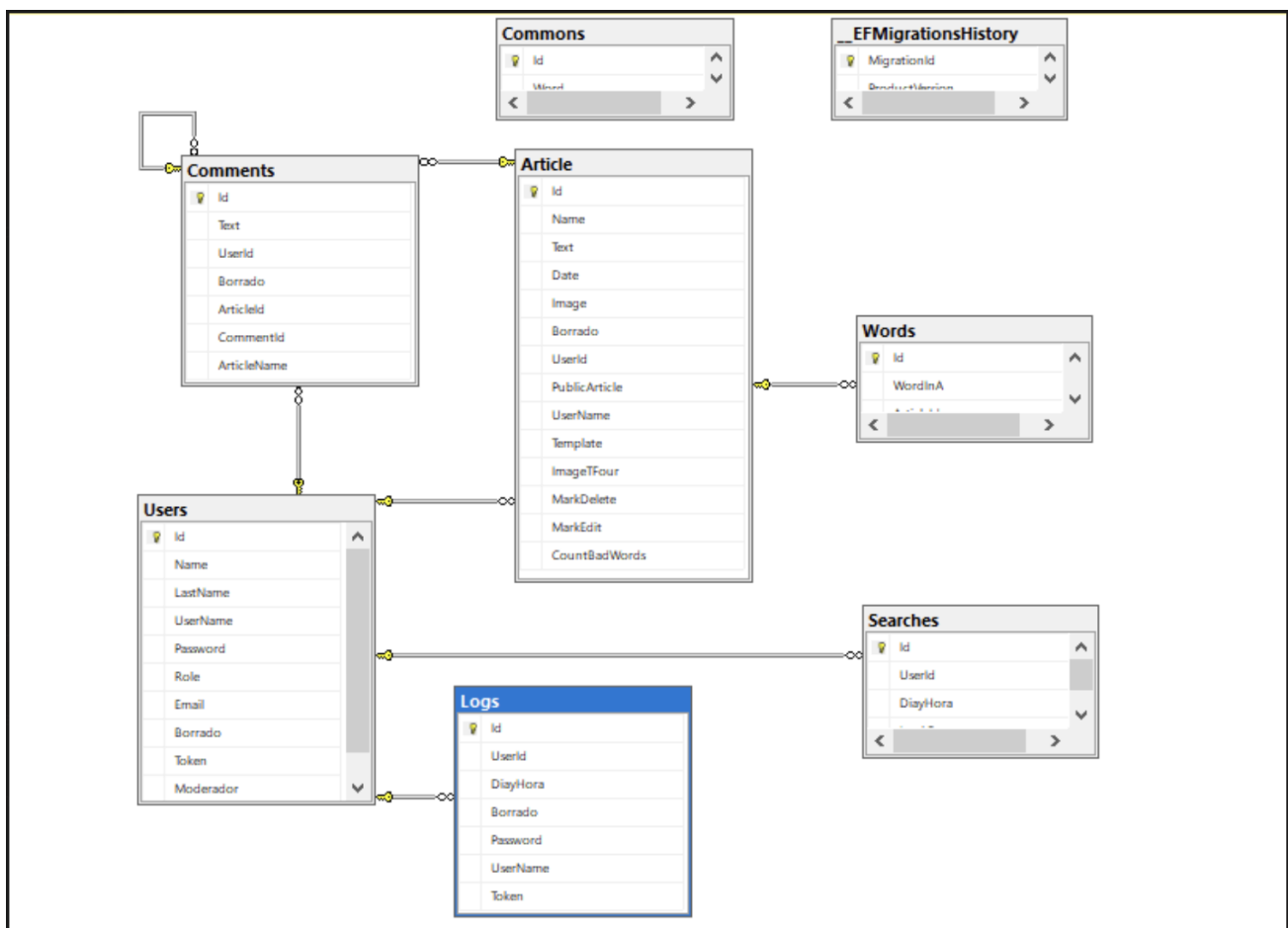
Esta vista no fue implementada dado que el alcance del Obligatorio no es suficientemente extenso como para especificarla.

Descripción de jerarquías

Para este proyecto decidimos eliminar la jerarquía que teníamos en el anterior obligatorio. Dicha jerarquía se encontraba en las clases Article y PublicArticle, como clase padre utilizamos Article y PublicArticle como la clase hija.

Modelo de tablas

Modelo generado con la aplicación Microsoft SQL Server Management Studio.



Mecanismo de acceso a la base de datos

Para el acceso a la base de datos utilizamos el ORM Entity Framework Core. Para ello tuvimos que instalar ciertos paquetes en el proyecto. Algunos de estos paquetes se pueden encontrar en la sección de Packages (NuGet) dentro de varios paquetes de la solución.

Para hacer la conexión con la base de datos de manera correcta, mantenible y escalable, se utilizaron varios proyectos, algunos de ellos son: de interfaces, capa lógica y de base de datos. De tal manera se aplica correctamente el principio DIP: Dependency Inversion Principle, que establece que: módulos de alto nivel no pueden depender de otros módulos de bajo nivel, deben depender de clases abstractas. Con esto en cuenta, se puede cambiar la base de datos sin afectar las implementaciones de la lógica.

Implementación de eliminar usuario en la base de datos utilizamos el mismo mecanismo que en la entrega anterior de borrado lógico.

Patrones de diseño y principios SOLID

Aplicación de principios SOLID de diseño.

Uno de los principios que respetamos dentro de nuestro proyecto es el Principio de Segregación de Interfaces (ISP). Este principio establece que se deben crear interfaces más pequeñas, específicas y cohesivas que se ajusten a las necesidades del cliente. En el obligatorio decidimos crear interfaces que permitan la comunicación entre la API y la lógica de negocio a través de las interfaces en *Domain.Interface*. A su vez la comunicación entre la lógica de negocio y el acceso a datos se da a través de interfaces, estas se encuentran en *Repositories.Interface*. Al hacer estas implementaciones buscamos reducir el acoplamiento y aumentar la cohesión de nuestros componentes.

A su vez buscamos respetar el Principio de Inversión de Dependencia (DIP). Este principio establece que módulos de bajo nivel no deben depender de módulos de alto nivel, sino que ambos deben depender de abstracciones. Para esto nos ayudamos con la inyección de dependencia, inyectando una instancia de la respectiva interfaz en las clases donde se necesita cierta implementación. Esto permite depender de abstracciones y no de las propias clases.

En las [páginas 6 y 7](#) de este documento se pueden encontrar los diagramas que describen las explicaciones anteriores.

No se aplicó ningún patrón GOF de diseño específico.

Funcionalidades adicionales

Para esta entrega se pedían ciertas funcionalidades adicionales del backend.

1. Los administradores pueden personalizar la lista de palabras ofensivas.
2. Mejora en el sistema de comentarios.
3. Buscar ranking de usuarios.
4. Rol de Moderador
5. Nueva plantilla
6. Reflection

Para la funcionalidad 1) se creó una nueva entidad *Common*, la cual contiene una lista de strings, en esta van a estar contenidas las palabras que tanto los administradores, como los moderadores agreguen. Se implementó con una lista *static*, ya que nos permite el acceso a métodos y variables de clase sin la necesidad de instanciar un objeto de dicha clase, y así poder agregar a la lista directamente. Cada vez que se quiera agregar una nueva palabra lo hacemos de la forma:

- `Common.Words.Add(badWord)`, siendo *Words* el nombre de la property en *Common*.

Para la funcionalidad 2) se agregó en comentarios una lista de comentarios llamada *Replies*, en estas van a estar las respuestas a los comentarios. A su vez, para agregar una respuesta, se le pasa el índice del comentario para así poder agregar las respuestas pertinentes.

- `Comments[commentIndex].Replies.Add(comment);`

Para la funcionalidad 3) en el controlador necesitábamos pasar dos fechas para buscar en este rango, por lo tanto se creó una nueva entidad llamada *DateRange* la cual contiene dos properties de tipo string, *DateFrom* y *DateTo*, las cuales se convierten a tipo *DateTime*, para así llamar a la función *GetRanking* y poder compararlas con la fecha de los artículos que tiene cada usuario correctamente. En esta funcionalidad, nos dimos cuenta de un error en la comprensión de la letra. Ya que era obtener un ranking de usuarios basándonos en la cantidad de artículos que tengan al menos una palabra ofensiva, sin embargo, nosotros, nos basamos en la

cantidad de palabras ofensivas que tenía cada artículo y haciendo la suma entre los diferentes artículos de las palabras ofensivas que contenía.

Para la funcionalidad 4) se agregó un nuevo atributo booleano en User, denominado *Moderador* el cual si se setea en true, puede tener las características de este. Sabemos que no es una forma muy amigable y que podríamos haber hecho tres entidades distintas (Administrador, Moderador, Blogger), las cuales heredarían de una clase User que contiene los atributos comunes, es un aspecto a mejorar a futuro.

Para la funcionalidad 5) se agregó un atributo ImageTFour en la clase Artículo, en el cual seteabamos la imagen con la ruta correspondiente si se elegía la plantilla 4.

Para la funcionalidad 6) una interfaz IImportService donde se definen las funciones GetName, GetParameters e ImportArticle. Esta interfaz es requerida por ImportService donde se obtienen los importers. Al mismo tiempo definimos la interfaz IImporter que define los métodos que serán utilizados por los distintos importadores como JSON o XML. Esto fue implementado de esa manera para permitir que en un futuro se pueda extender a otros importers sin necesidad de cambiar el proyecto en profundidad. Esta funcionalidad está implementada únicamente en el backend.

Para la parte de reflection nos basamos en el código utilizado por la clase del nocturno y lo adaptamos a nuestra solución.

Métricas de diseño

Para el análisis de métricas de diseño utilizamos la herramienta NDepend. Los resultados obtenidos se muestran abajo.

La cohesión relacional, H, es la siguiente:

Paquetes	Cohesión Relacional
Entities	1,94
Importer.Interfaces	1
Domain.Interfaces	1
Repositories.Interfaces	1
Domain.Logic	1,62
DataAccess	1,5
ServiceFactory	1
BlogifyAPI	2,62

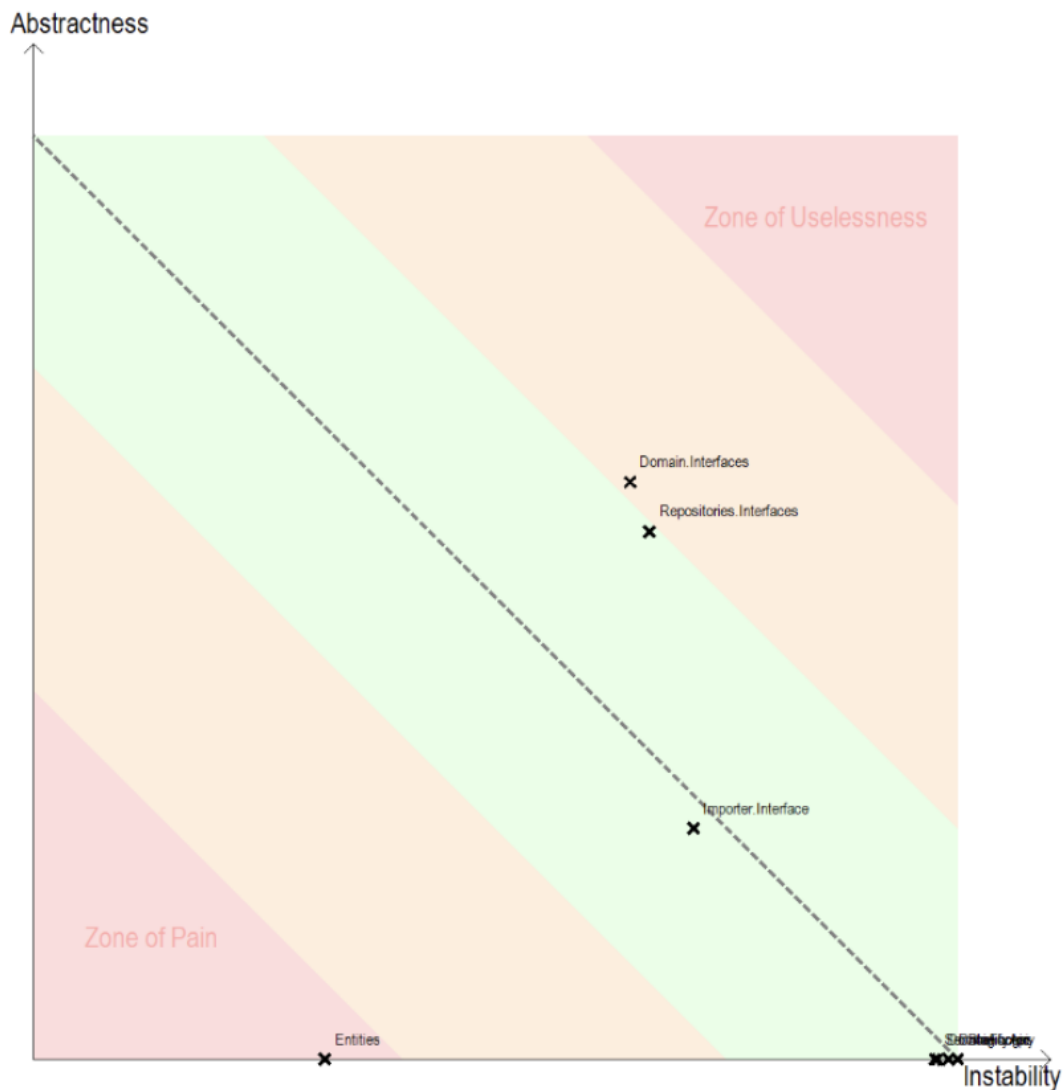
Vemos que algunos paquetes tienen una cohesión relacional baja. Esto se debe a que las relaciones que se están dando entre los paquetes es principalmente mediante interfaces e inyección de dependencias.

Aquí presentamos la tabla con los valores de Coeficiente aferente, Coeficiente eferente, Inestabilidad, Abstracción y Distancia.

Paquetes	Ca	Ce	Inestabilidad	Abstracción	Distancia
Entities	39	18	0,32	0	0,48
Importer.Interfaces	4	10	0,71	0,25	0,03
Domain.Interfaces	11	20	0,65	0,62	0,19
Repositories.Interfaces	9	18	0,67	0,57	0,17
Domain.Logic	1	45	0,98	0	0,02
DataAccess	1	103	0,99	0	0,01
ServiceFactory	1	39	0,98	0	0,02
BlogifyAPI	0	90	1	0	0

La Distancia está muy cercana a cero en gran parte de los casos, cosa que es recomendable

Matriz de Inestabilidad y Abstracción.



Como podemos ver, gran cantidad de las clases se encuentran en la zona recomendable, donde existe una relación aceptable entre la abstracción y la inestabilidad.

Por otro lado vemos que la clase Entities tiene abstracción cero, esto es normal ya que en esta clase se encuentran las clases concretas de nuestro proyecto y no posee ninguna clase abstracta o interfaz.

TDD

Al igual que en el obligatorio 1 se aplicó TDD (Test Driven Development), siguiendo el ciclo: rojo, verde y azul (refactor).

Como indica TDD, se crearon primero las pruebas y luego, a partir de esos test, el código de producción.

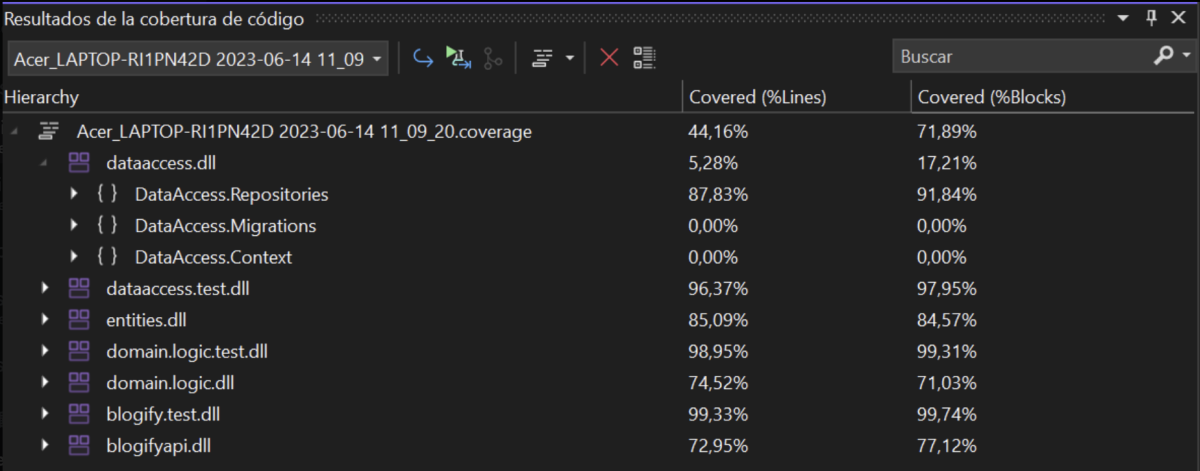
Las pruebas unitarias que desarrollamos nos permitieron detectar errores y problemas en el código de manera temprana, en lugar de que estos fueran detectados por los clientes.

Respetamos el principio F.I.R.S.T, de tests rápidos, independientes, repetibles, auto-validados y previamente escritos al código de producción.

Mantuvimos los 3 proyectos de pruebas como en el anterior obligatorio: *Domain.Logic.Test*, donde se prueban las clases del Dominio y la lógica de negocio; *DataAccess.Test*, donde se testea la base de datos; *Blogify.Test*, donde se prueban los controladores de la API.

Para la parte de importadores no se utilizó TDD dado que no tuvimos el tiempo por haberlo hecho muy cercano a la fecha de entrega.

Cobertura de código:



Hierarchy	Covered (%Lines)	Covered (%Blocks)
Acer_LAPTOP-RI1PN42D 2023-06-14 11_09_20.coverage	44,16%	71,89%
dataaccess.dll	5,28%	17,21%
{ } DataAccess.Repositories	87,83%	91,84%
{ } DataAccess.Migrations	0,00%	0,00%
{ } DataAccess.Context	0,00%	0,00%
dataaccess.test.dll	96,37%	97,95%
entities.dll	85,09%	84,57%
domain.logic.test.dll	98,95%	99,31%
domain.logic.dll	74,52%	71,03%
blogify.test.dll	99,33%	99,74%
blogifyapi.dll	72,95%	77,12%

Podemos ver que la cobertura de código baja por la carpeta de migraciones(DataAccess.Migration) y por la carpeta Context(DataAccess.Context)

que tiene a la clase del BlogifyContext. La cobertura de código de ambas carpetas es 0. Por lo que decidimos no tenerlos en cuenta para obtener el total de la cobertura de código.

Cálculo de cobertura de código:

- Blogify.Test → 99,74%
- BlogifyApi → 77,12%
- Entities → 84,57%
- DataAccess.Test → 97,95%
- DataAccess(DataAccess.Repositories) → 91,84%
- Domain.Logic.Test → 99,31%
- Domain.Logic → 71,03 %

La cobertura de bloques total es: 89%

Podemos ver que tenemos una cobertura cercana al 90%, tuvimos inconvenientes al hacer el test de los imports, por lo tanto, esa cobertura no está contemplada y es lo que nos baja significativamente la cobertura de bloques total.

Clean code

Aplicamos clean code durante todo el proyecto. Usamos este enfoque para asegurarnos de que nuestro código fuera fácil de leer, entender y mantener.

En principio, nos aseguramos de seguir una estructura clara y coherente para el código. Buscamos eliminar todo comentario innecesario en el código. Tratamos de la mejor manera de respetar el principio de que las variables deben ser descriptivas con los nombres, así como los nombres de las funciones y métodos. Respetamos los nombres de los atributos privados y las mayúsculas en los nombres de las funciones como indican las reglas de C#.

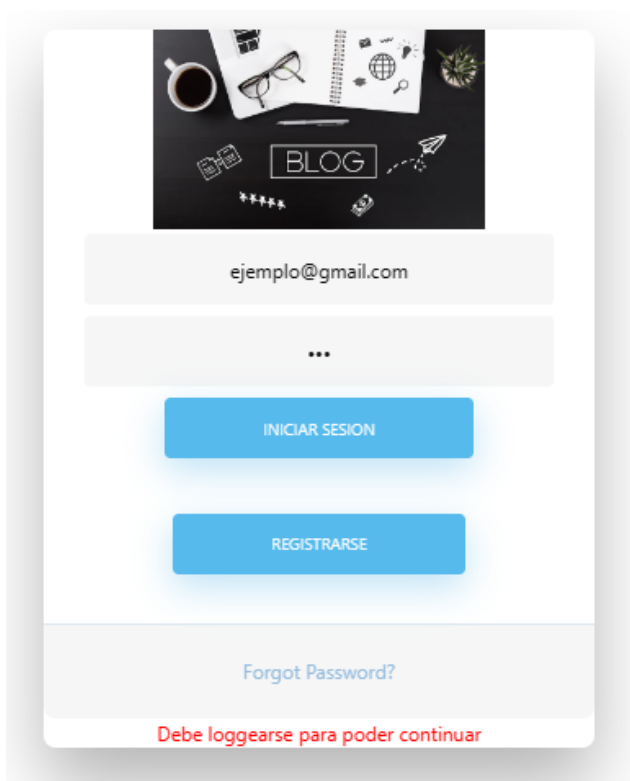
Así como también, utilizamos la modularidad del código como una forma de asegurarnos de que las diferentes piezas de nuestra solución estuvieran separadas y pudieran ser examinadas individualmente. Esto nos permite detectar errores de manera más ágil y realizar cambios con mayor facilidad en el futuro.

Interfaz de usuario

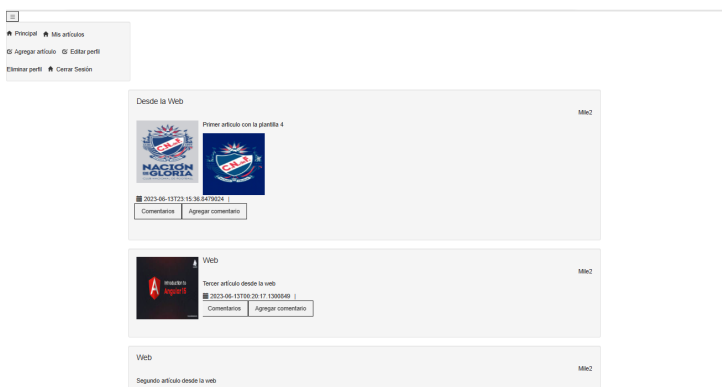
Para la interfaz de usuario utilizamos el framework Angular junto con la librería Bootstrap. Decidimos implementar una interfaz limpia y clara, con colores atractivos para el usuario.

Facilitamos algunas imágenes:

Esta primera imagen nos muestra el caso en que ingresamos un nombre de usuario o contraseña incorrecto.



Esta segunda imagen nos muestra el caso en que ingresamos correctamente el nombre de usuario y la contraseña. Este nos lleva a la página principal donde mostramos los últimos 10 artículos que fueron creados o modificados en el sistema.





Nombre

Texto

Ruta de la primera imagen

Ruta de la segunda imagen

Artículo Público: ☒ Si

☐ No

Template: ☒ 1

☐ 2

☐ 3

☐ 4

AGREGAR ARTÍCULO

EDITAR PERFIL



login

password

EDITAR PERFIL



ELIMINAR PERFIL

Especificación de la API

Para esta instancia, así como en la anterior, mantuvimos el uso de una API Rest, con controladores.

Los controladores son 4:

- UserController
- ArticleController
- LogController
- SearchController

En esta ocasión al UserController le agregamos el siguiente endpoint: AddBadWords, que nos ayuda a implementar la nueva funcionalidad de listado de malas palabras. Así como también, GetRanking.

Al ArticleController le agregamos los siguientes 8 nuevos endpoints: PostNewReply, PutTemplate, GetTheTenLastArticles, PutArticleWithBadWords, GetAllArticles, GetDistributionPerYear, GetImporters e ImportArticle. Estos surgieron de implementar la funcionalidad de comentarios anidados, agregar la foto a un artículo, obtener los últimos 10 artículos más recientes y la funcionalidad del importador.

Tanto el LogController como el SearchController no fueron modificados para este obligatorio.

Se implementaron la mayoría de las funcionalidades que nos habían faltado en el obligatorio anterior, para no seguir arrastrando errores y poder tener el sistema lo más completo y funcional posible.