Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio 2 de Diseño de Aplicaciones 2

Agustín Juárez - 236487, Agustín Campón - 233006

Tutor: Nicolas Fierro

Repositorio con la solución

2023

Descripción del Diseño

Nuestra aplicación se hizo siguiendo el requerimiento, en el cual el cliente necesitaba desarrollar un sistema de Blogs.

Un blog es una aplicación web en la cual sus usuarios pueden recopilar cronológicamente artículos, pudiendo otros usuarios realizar comentarios acerca de los mismos. En esta primera etapa del desarrollo, se comenzó por desarrollar la API REST, que consumirá nuestro front-end a posteriori, para ofrecer el servicio mencionado anteriormente.

La primera decisión de diseño comenzó en definir qué tipo de arquitectura íbamos a implementar en nuestra aplicación, y optamos por utilizar una arquitectura de tres capas.

La arquitectura de 3 capas es un patrón de diseño que divide la aplicación en tres capas principales:

1) Capa de Presentación
2) Capa de Lógica de Negocio
3) Capa de Acceso a Datos

Capa de Presentación:

La capa de presentación, en nuestra aplicación es representada por la Web API. La responsabilidad principal de nuestra Web API es recibir las solicitudes de los clientes y devolver la respuesta adecuada, utilizando JSON como formato.

Capa de Lógica de Negocio:

En nuestra aplicación es representada por los Servicios, que es donde generamos el procesamiento de los datos y aplicamos las reglas del negocio.

Capa de Acceso a Datos:

En nuestra aplicación es representada por los Repositorios, que son básicamente los encargados de interactuar con nuestra base de datos.

Descripción General del Trabajo

Nuestra API REST para el sistema de blogs requerido permitiría la interacción entre el cliente (navegador web o postman) y el servidor para llevar a cabo las diversas acciones y funcionalidades requeridas. La API sería responsable de procesar las solicitudes del cliente y devolver los datos necesarios en formato JSON.

Las principales funcionalidades de la API REST para este sistema de blogs son:

1) **Autenticación y autorización**: Permitir el registro, inicio y cierre de sesión de los usuarios, así como validar sus credenciales y roles para acceder a diferentes recursos y acciones.

2) **Gestión de perfiles de usuario**: Proporcionar endpoints para que los usuarios puedan actualizar su información personal, como nombre, apellido, nombre de usuario, contraseña y correo electrónico.

3) **Gestión de artículos**: Permitir a los usuarios crear, modificar y eliminar sus propios artículos, además de establecer la visibilidad de los mismos (público o privado).

4) **Comentarios y respuestas:** Facilitar la posibilidad de que los usuarios realicen comentarios en los artículos públicos, así como responder a los comentarios de otros usuarios en sus propios artículos.

5) **Notificaciones**: Enviar notificaciones a los usuarios cuando haya nuevos comentarios en sus artículos y facilitar la posibilidad de responderlos de manera sencilla.

6) **Búsqueda de artículos:** Proporcionar la posibilidad de filtrar artículos basándose en un texto de búsqueda, devolviendo aquellos que contengan el texto en su nombre o contenido.

7) **Módulo de administración de usuarios:** Permitir a los administradores crear, modificar y eliminar usuarios, así como acceder a información y estadísticas sobre la actividad de los usuarios en el sistema.

8) **Listado de artículos recientes:** Ofrecer la posibilidad de obtener los últimos 10 artículos creados.

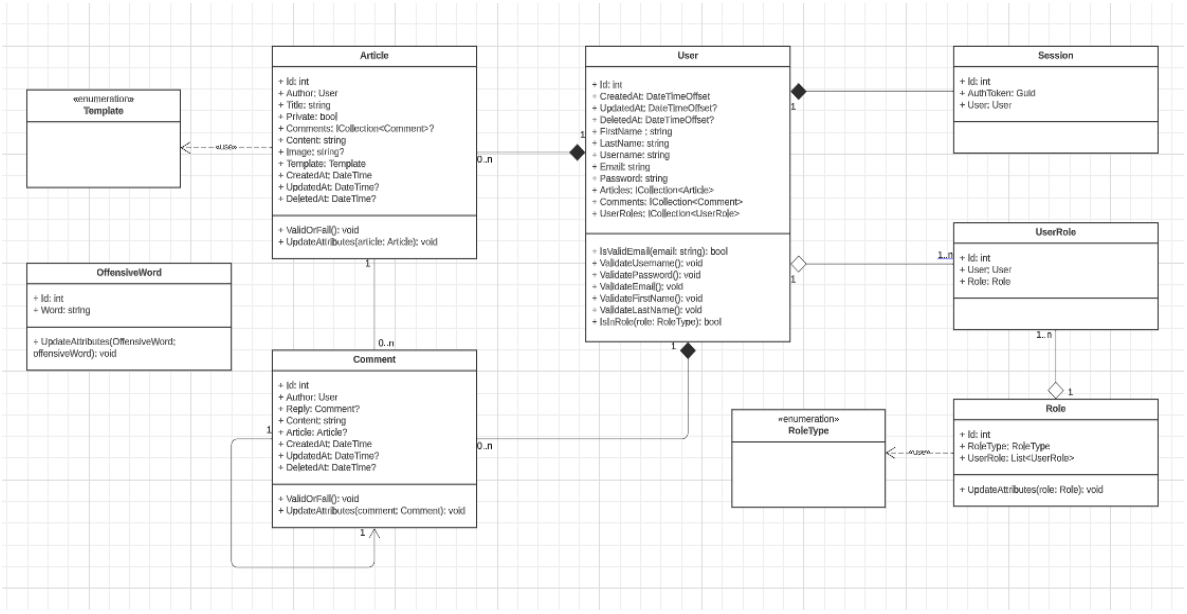
Al implementar una API REST para nuestro sistema de blogs, se facilita la escalabilidad y la integración con diferentes clientes y plataformas, garantizando así una arquitectura más flexible y mantenible.

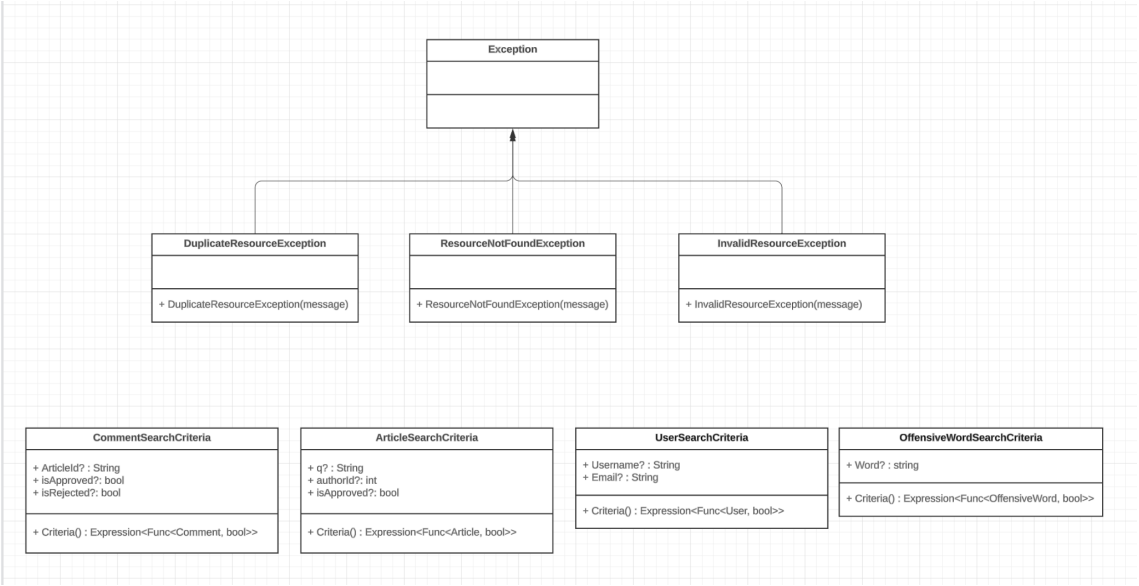
Diagramas de clases

(Vista Lógica/Diseño)

En este sector de la documentación nos enfocaremos en la vista lógica, aquí nos enfocaremos en describir cómo se soportan los requerimientos funcionales del sistema, enfocándonos en el diseño y en la colaboración de sus entidades.

Domain

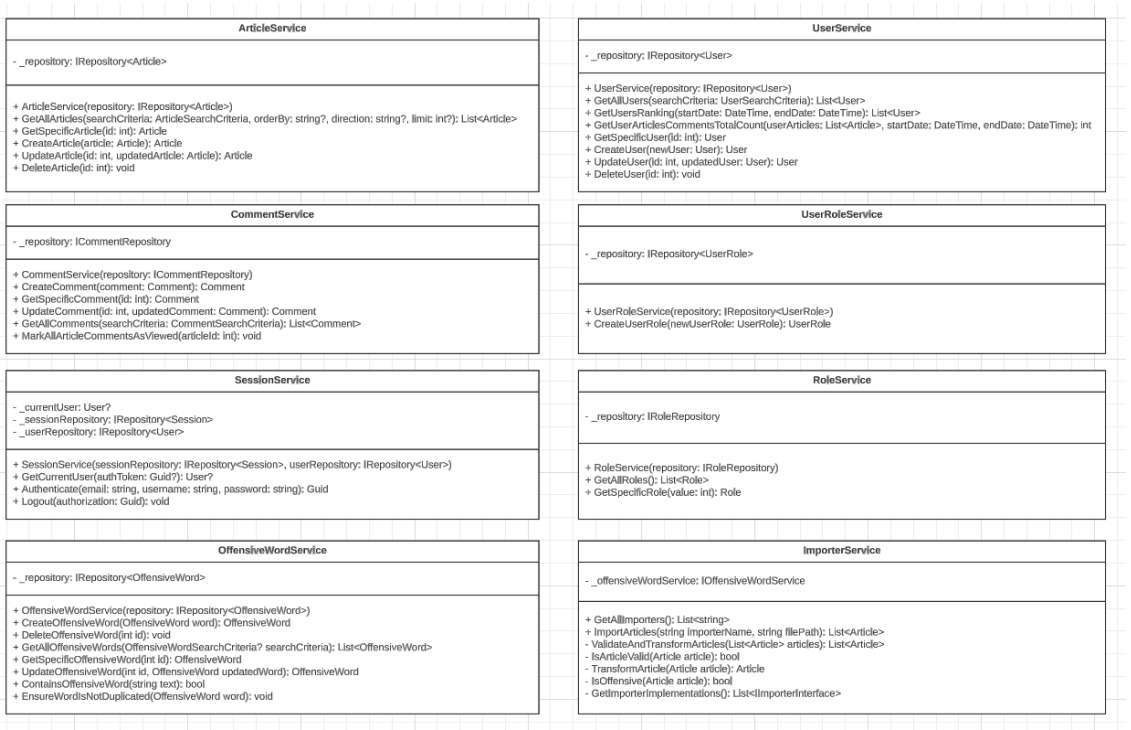




Nuestro paquete de Dominio es el encargado de representar las entidades de nuestro dominio (de nuestro mundo) y gestionar las relaciones entre estas entidades. También es responsable de asegurarse que cada entidad sea correcta en todo momento, esto lo logra siguiendo la lógica de que cada entidad es responsable de validarse a sí misma.

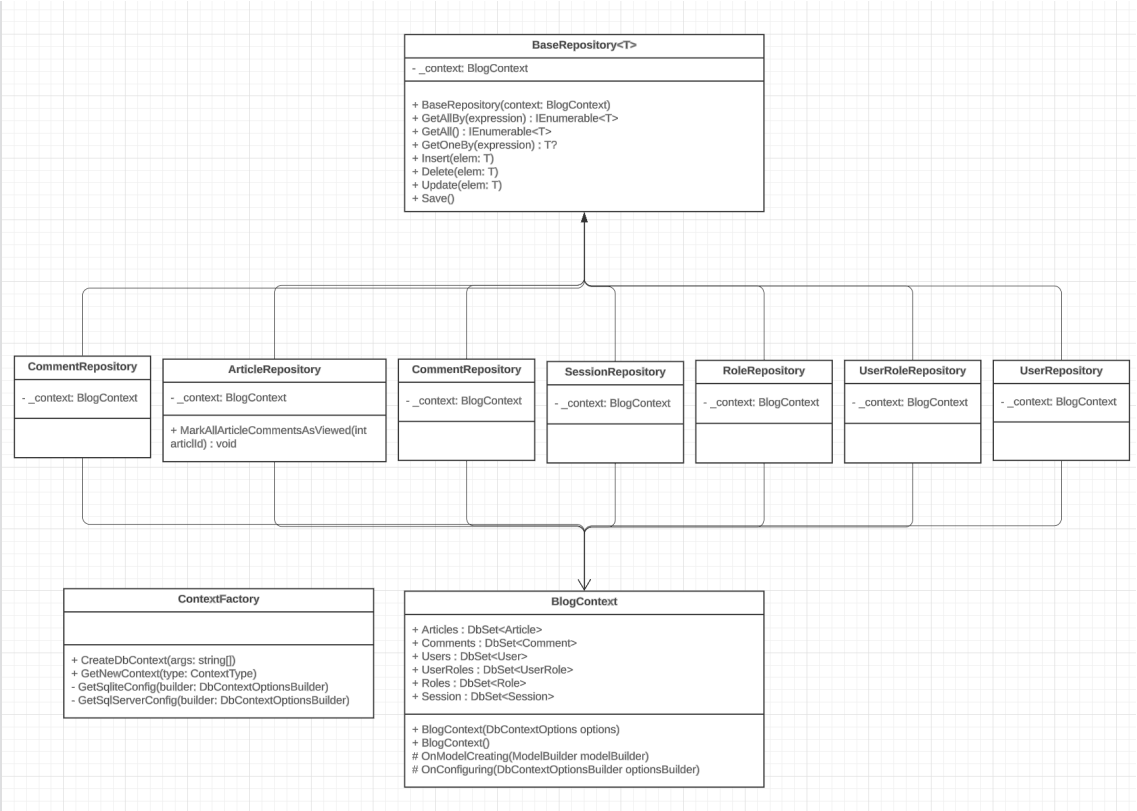
Es importante recalcar que las excepciones se encuentran dentro de nuestro paquete de Dominio, porque una de las cualidades de este paquete, es que todos necesitan conocerlo, por lo tanto resulta conveniente colocar las excepciones aquí.

Services



El paquete "Services" tiene la responsabilidad de manejar la lógica de negocios de la aplicación, en este paquete cada clase de servicio se centra en una entidad específica del dominio y sus operaciones relacionadas, como la creación, actualización, eliminación y consulta de datos. Estas clases de servicios también pueden implementar validaciones y reglas de negocio específicas para mantener la integridad de los datos y garantizar el correcto funcionamiento de la aplicación.

DataAccess



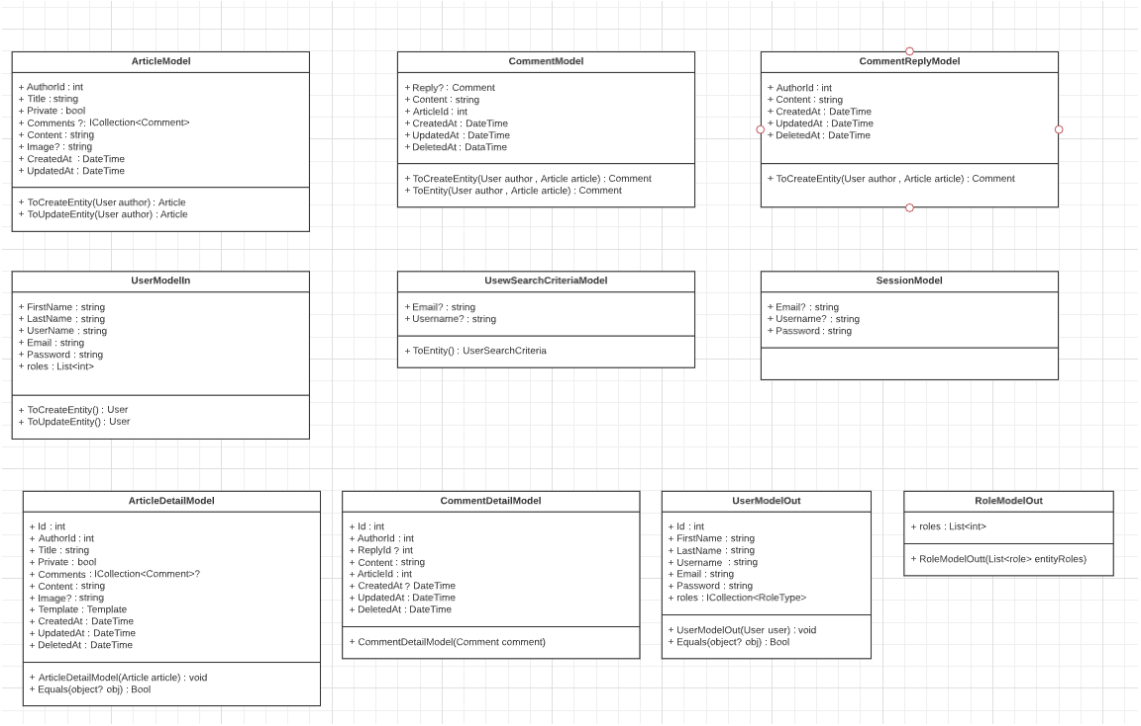
El paquete de DataAccess tiene la responsabilidad de conocer cómo persistir los datos en SQLServer.

Como se ve claramente en el diagrama, utilizamos herencia para favorecernos de las ventajas que nos provee. En este caso nos permite evitar repetirnos (DRY) y nos ayuda a mantener las cosas simples (KISS).

Factory

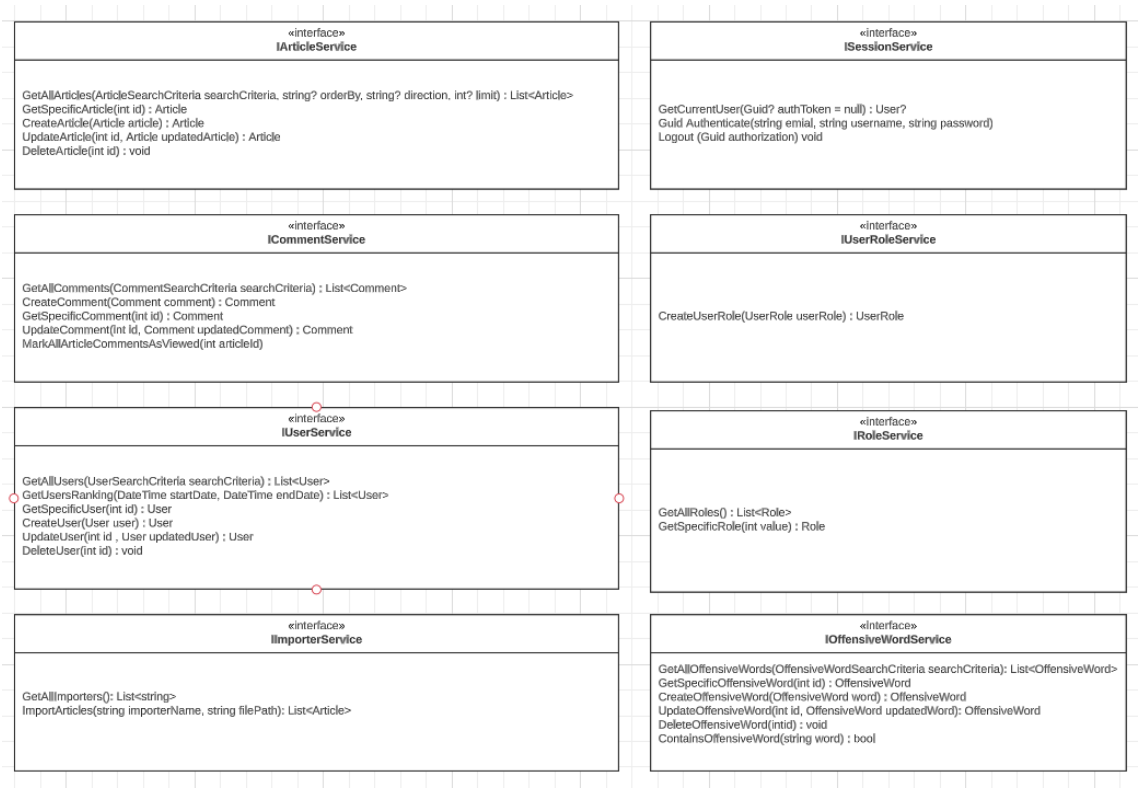
Nuestro paquete de Factory es responsable de conocer qué implementación deberá proveer para cada interfaz requerida en el sistema.

Models



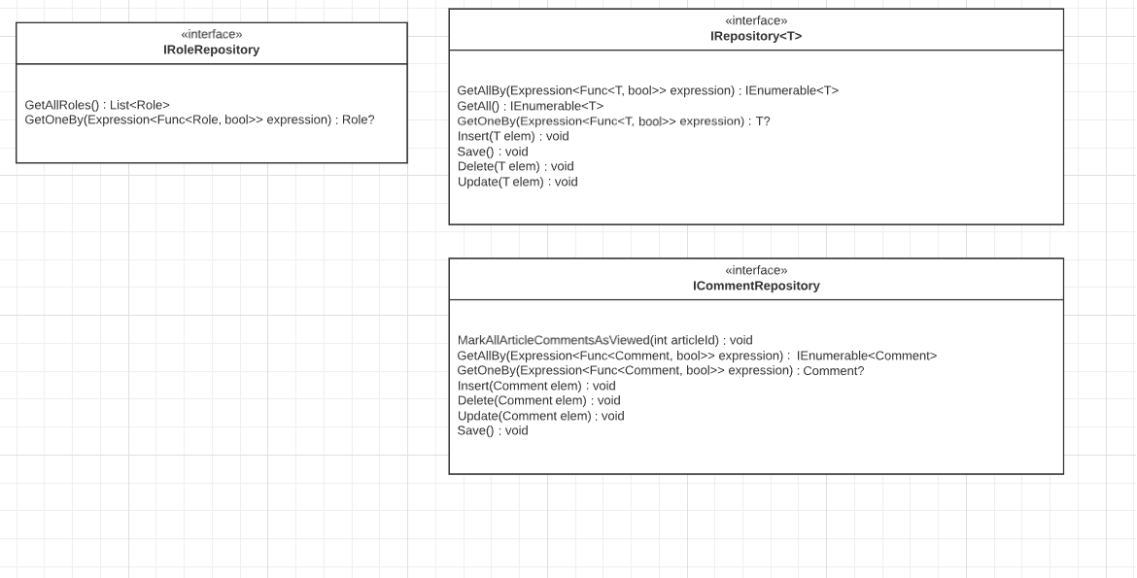
Este paquete es responsable de facilitar la transferencia de datos entre la capa de presentación (API) y la capa de lógica de negocio.

IServices



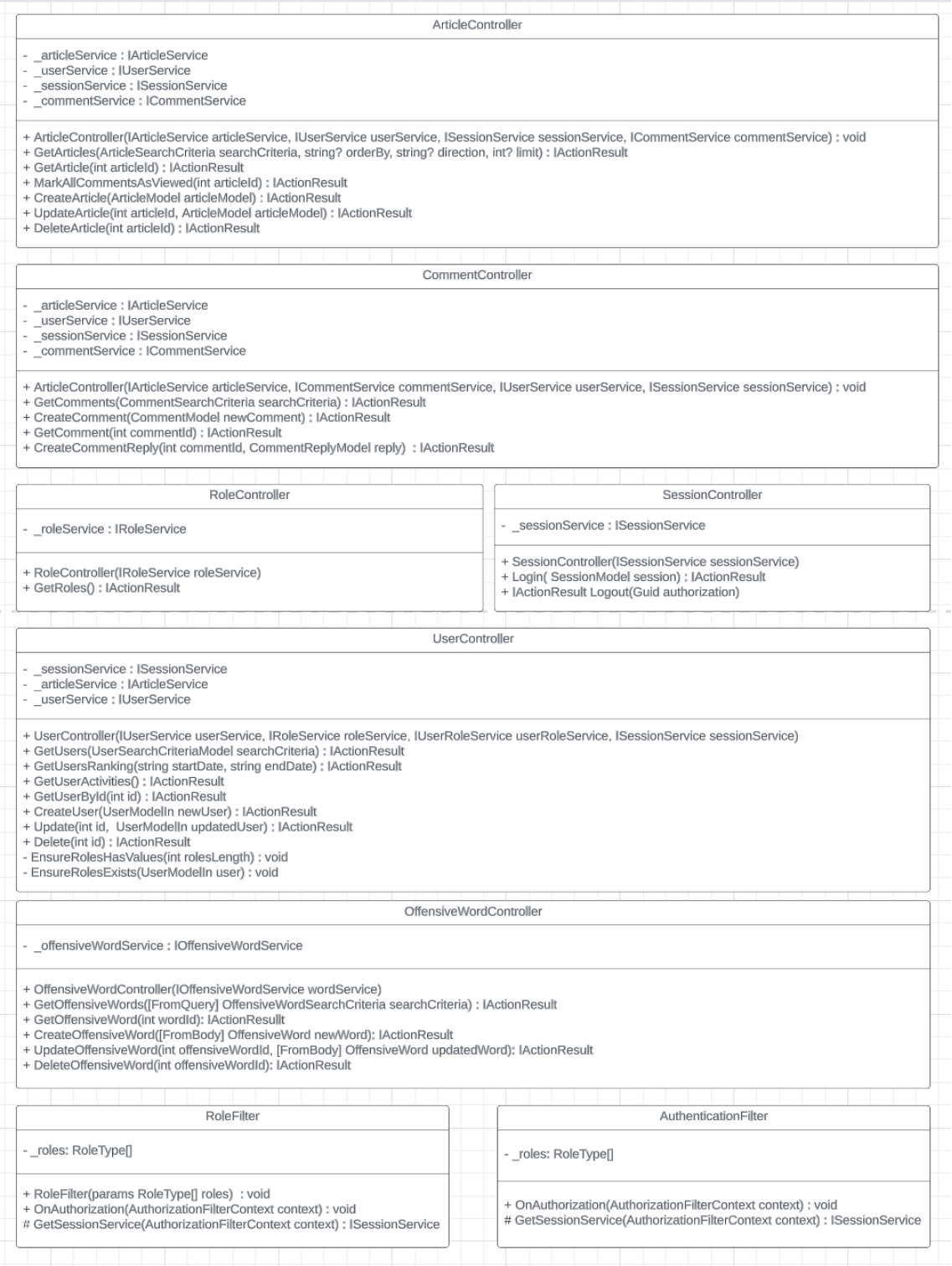
La responsabilidad de IServices fue aclarada en el diagrama de componentes, explicando en profundidad DIP y DI.

IDataAccess



La responsabilidad de IDataAccess fue aclarada en el diagrama de componentes, explicando en profundidad DIP y DI.

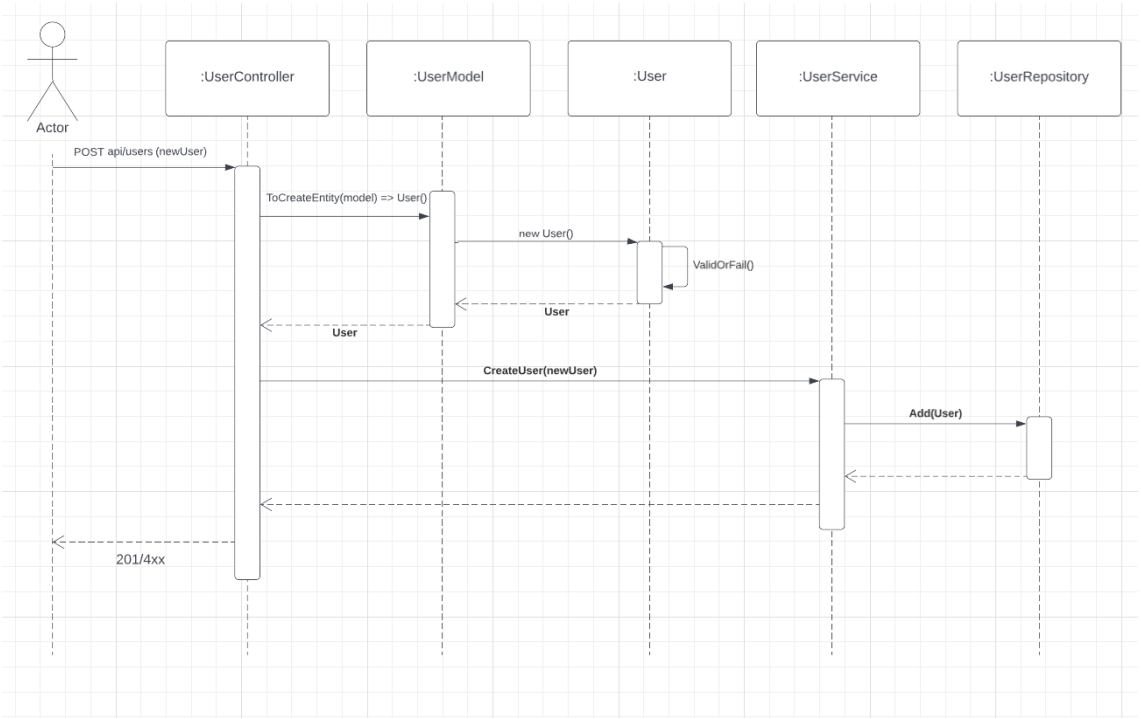
WebAPI



El paquete de WebAPI es responsable de recibir las solicitudes de los clientes y devolver la respuesta adecuada. Responsable de la interacción con los clientes.

Interacciones

En el siguiente diagrama de interacción, se podrá ver en qué forma es que se relaciona nuestro sistema, para lograr el objetivo común de crear un nuevo usuario.



El siguiente es un diagrama de interacción en el que se puede ver la forma de relacionarse de las clases para lograr el objetivo común de eliminar un usuario.

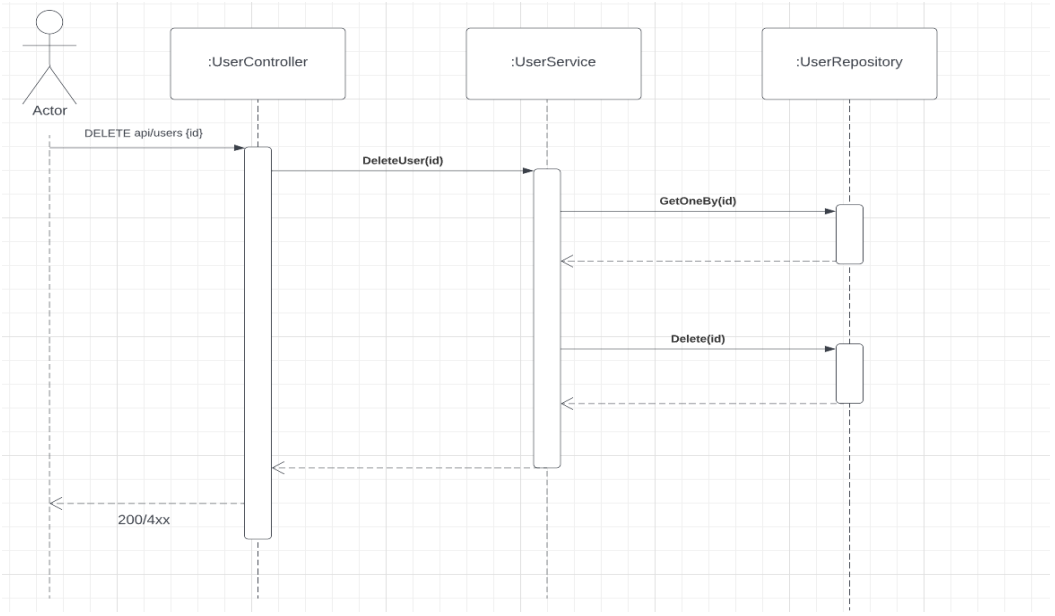


Diagrama general de paquetes

(Vista Implementación)

En este sector de la documentación nos enfocaremos en mostrar cómo se relacionan nuestros diversos componentes de software (nuestros paquetes y nuestros componentes)

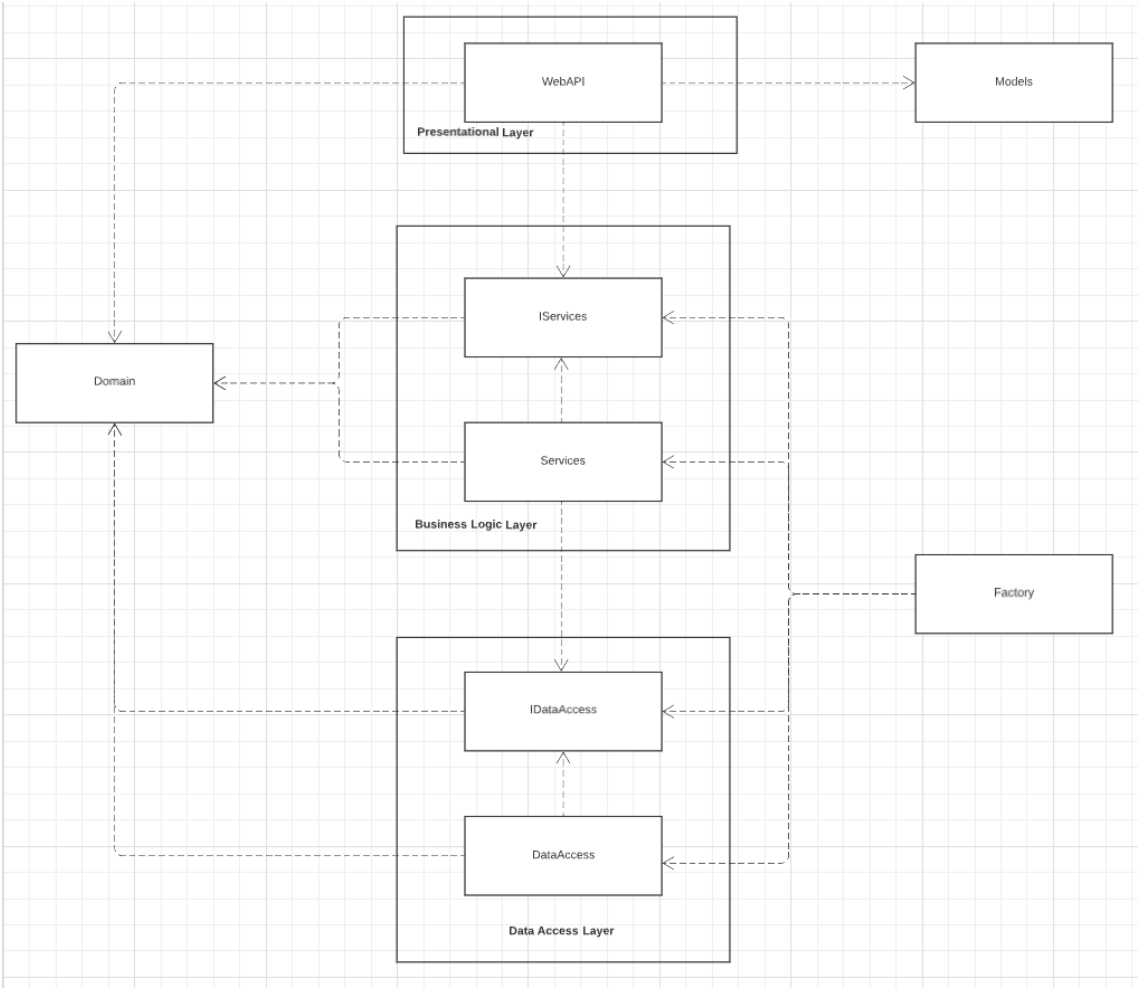
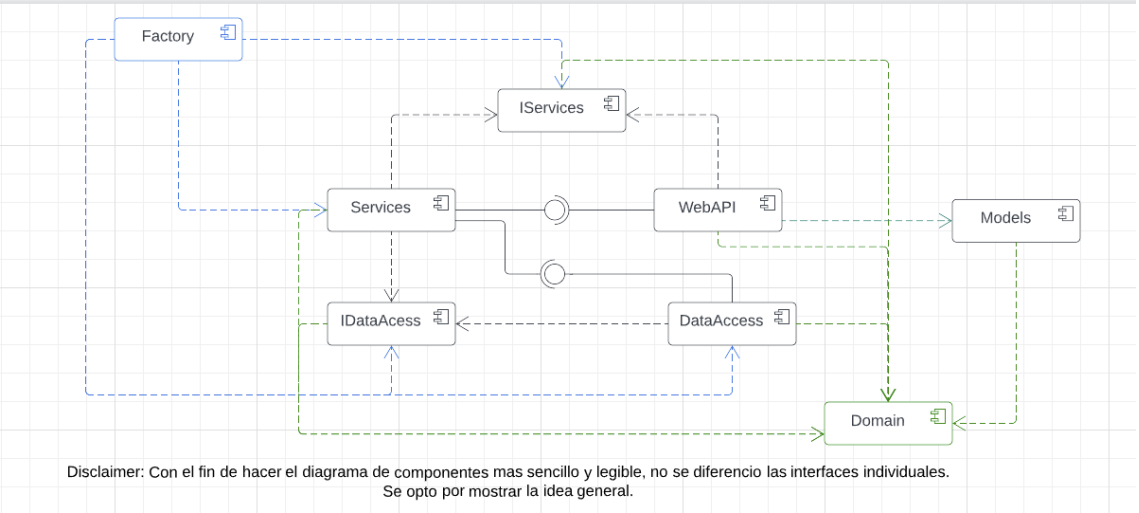


Diagrama de Componentes



Dividimos la aplicación en estos componentes, debido a que tomamos como premisa seguir las mejores prácticas de diseño con el fin de implementar un sistema robusto y extensible.

Ejemplo :

Todos nuestros Software Components aplican el “Single Responsibility Principle”. Es decir todos nuestros componentes tienen una única responsabilidad y una única razón por la cual deberían de cambiar en un futuro.

A continuación listamos las responsabilidades de todos nuestros componentes:

1) **WebAPI**: Es responsable de recibir las solicitudes de los clientes y devolver la respuesta adecuada. Responsable de la interacción con los clientes.
2) **Services**: Es responsable de conocer e implementar la lógica de negocio.
3) **DataAccess:** Es responsable de conocer cómo acceder a nuestra base de datos, y cómo persistir la información.
4) **Factory** : Es responsable de conocer qué implementación deberá proveer para cada interfaz requerida en el sistema.
5) **Models:** Es responsable de facilitar la transferencia de datos entre la capa de presentación (API) y la capa de lógica de negocio.

6) **Domain**: Tiene la responsabilidad de modelar las entidades que representan nuestro problema.

Notaran que no describimos SRP para IDataAccess y IServices, esto es porque ambos paquetes fueron implementados para poder aplicar **DIP** y **DI**.

Principio de Inversión de Dependencias

Aplicar DIP nos permite obtener una arquitectura más desacoplada y de ser necesario poder intercambiar módulos sin generar impacto en la aplicación.

El Principio de Inversión de Dependencias es un principio de diseño que busca reducir las dependencias directas entre módulos de alto y bajo nivel. En su lugar, se propone que ambas partes dependen de abstracciones (en nuestro caso representamos las abstracciones con interfaces).

En la definición de DIP, utilizamos los términos “módulo de alto nivel” y “módulo de bajo nivel”.

Módulo de alto nivel : Un módulo de alto nivel se encarga de interactuar con otros módulos de bajo nivel para llevar a cabo su trabajo. En general, los módulos de alto nivel se centran en "qué" debe hacerse, en lugar de "cómo" debe hacerse. ej: Servicios

Módulo de bajo nivel: Un módulo de bajo nivel se refiere a la parte de nuestro sistema que se ocupa de los detalles de implementación y de cómo se llevan a cabo las tareas. Estos módulos proporcionan funcionalidades que los módulos de alto nivel utilizan para que puedan cumplir con sus responsabilidades. ej: DataAccess

Aplicar DIP nos ofrece las siguientes ventajas :

1) **Modularidad**: Al separar las responsabilidades en módulos bien definidos y con abstracciones, se fomenta la modularidad.
2) **Flexibilidad**: Al depender de abstracciones en lugar de implementaciones concretas, se permite que el sistema sea más fácilmente adaptable a cambios.

Por ejemplo, si se necesita reemplazar un componente con otro, basta con que el nuevo componente cumpla con la abstracción requerida, sin necesidad de modificar el resto del sistema.

3) **Facilita las pruebas**: Al depender de abstracciones, es más fácil crear pruebas unitarias para cada módulo, ya que se pueden utilizar mocks que implementen la abstracción requerida, en lugar de depender de implementaciones concretas que podrían ser difíciles de configurar o controlar.
4) **Reducción de acoplamiento:** El principio DIP ayuda a reducir el acoplamiento entre módulos, lo que a su vez disminuye la probabilidad de que un cambio en un módulo tenga efectos no deseados en otros módulos.

Inyección de Dependencias

La Inyección de Dependencias es un patrón que utilizamos para desacoplar la creación y la asignación de dependencias de una clase. En lugar de crear las dependencias dentro una clase, estas se "inyectan" desde el exterior, esto es responsabilidad de nuestro componente de Factory.

Aplicar Inyección de Dependencias nos ofrece las siguientes ventajas:

1) **Desacoplamiento**: Ayuda a reducir el acoplamiento entre módulos al permitir que las dependencias sean proporcionadas externamente. Esto facilita el intercambio de implementaciones y la adaptación del sistema a cambios.
2) **Flexibilidad y adaptabilidad:** Al permitir la inyección de dependencias, el sistema se vuelve más fácilmente adaptable a cambios. Por ejemplo, si es necesario reemplazar una implementación con otra, basta con inyectar
3) **Facilita las pruebas**: Al permitir la inyección de dependencias, es más fácil crear pruebas unitarias y de integración, ya que se pueden reemplazar las dependencias reales con mocks durante las pruebas. Esto simplifica el aislamiento de las partes del sistema que se están probando y la simulación de diferentes condiciones.

4) **Separación de responsabilidades**: La Inyección de Dependencias promueve la separación de responsabilidades en el diseño del software. Cada clase se enfoca en su propia funcionalidad y delega la gestión de sus dependencias a componentes externos, como "factories".

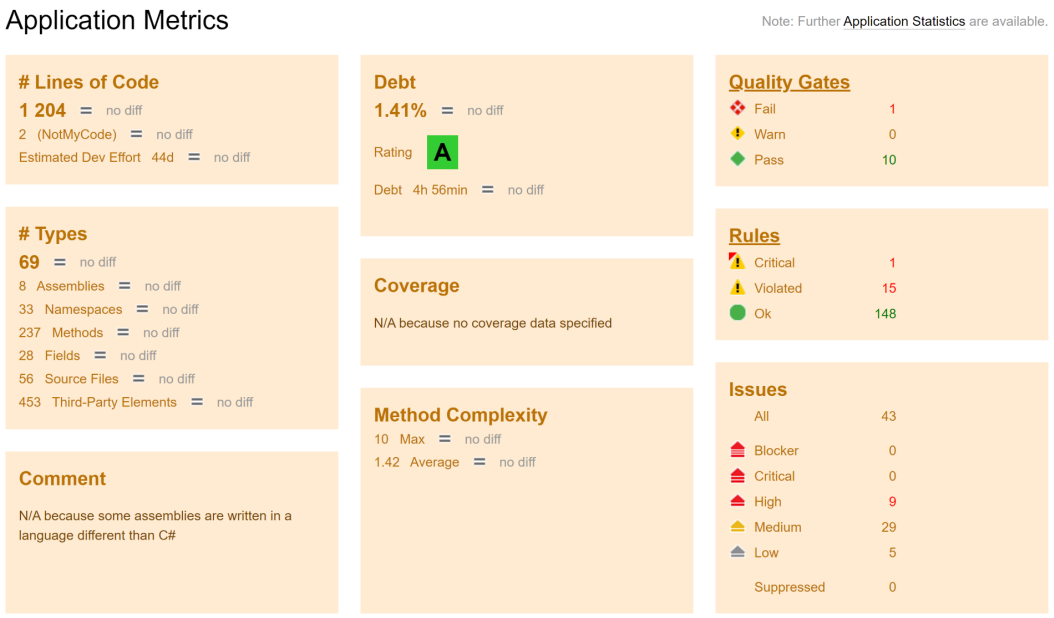
Inversión de Dependencias + Inyección de dependencias

Al combinar estos dos patrones, potenciamos los beneficios que ofrecen ambos patrones por separado (enumeramos las principales ventajas) :

1) **Mayor desacoplamiento**: Aplicar DIP promueve el desacoplamiento entre módulos al depender de abstracciones en lugar de implementaciones concretas. La Inyección de Dependencias facilita aún más este desacoplamiento al permitir que las dependencias sean proporcionadas externamente, en lugar de ser creadas o instanciadas dentro de los propios módulos.
2) **Mayor flexibilidad y adaptabilidad:** Al depender de abstracciones y permitir la inyección de dependencias, el sistema se vuelve más fácilmente adaptable a cambios. Por ejemplo, si es necesario reemplazar una implementación con otra, basta con inyectar la nueva implementación sin necesidad de modificar el código de los módulos que la utilizan.
3) **Facilita la configuración del sistema**: Al utilizar la Inyección de Dependencias, la configuración de las diferentes partes del sistema puede realizarse de manera centralizada y más fácilmente (en nuestra Factory), lo que simplifica la gestión y el mantenimiento del software.

Principios del diseño de paquetes

En esta sección nos enfocaremos en realizar un extenso análisis basado en las métricas aprendidas en el curso.



Podemos analizar que nuestro proyecto posee una deuda técnica muy baja, siendo esta de **1.41%.**

Ndepend evalúa una deuda clase “A” a soluciones cuya deuda técnica esté entre 0% y 5%, colocando nuestro proyecto mucho más cerca del 0% que del 5%. Simplemente observando esto, podemos inferir que estamos ante un repositorio en el cual se usaron las mejores prácticas de desarrollo.

Teniendo claro esto, comenzaremos a analizar si nuestra solución cumple con los principios de diseño de paquetes escritos por Robert C. Martin.

Common Reuse Principle

Comenzaremos por hacer un análisis del Common Reuse Principle (**CRP**). Este principio sugiere que las clases que se usan juntas deben agruparse juntas en el mismo paquete. En otras palabras, si una clase en un paquete cambia de manera que afecta a otras, todas las clases afectadas deben estar en el mismo paquete.

Nuestro obligatorio incumple con este principio, esto es porque escogimos una arquitectura basada en capas, favoreciendo así la agilidad (puesto que requerimos de muy poco tiempo para cumplir con los requerimientos del cliente).

Para mantener la agilidad en el desarrollo a la vez que aplicamos CRP, deberíamos de generar un cambio en la arquitectura de nuestro repositorio, este sería básicamente transformar la estructura monolítica a una arquitectura basada en microservicios.

Este cambio permitiría implementar cada funcionalidad como un microservicio independiente, siendo cada microservicio responsable de encapsular una funcionalidad específica del sistema. Pudiendo así cumplir fácilmente con CRP.

Un ejemplo practico seria tener el proyecto Blog.User :
Blog.User:

- Blog.User.Services
- Blog.User.Domain
- Blog.User.Model
- Blog.User.DataAccess

Common Closure Principle

Este principio sugiere que las clases que cambian por las mismas razones y en los mismos momentos deben estar en el mismo paquete. Esto ayuda a minimizar el trabajo cuando se realizan cambios y a prevenir errores que podrían ser introducidos por cambios en otras partes del sistema.

Nuevamente nuestro obligatorio incumple con este principio, nuevamente porque favorecemos la agilidad en el desarrollo, basándonos en una arquitectura basada en capas. La manera más sencilla de cumplir con este principio sería básicamente migrar a una arquitectura basada en microservicios, como describimos en el punto anterior.

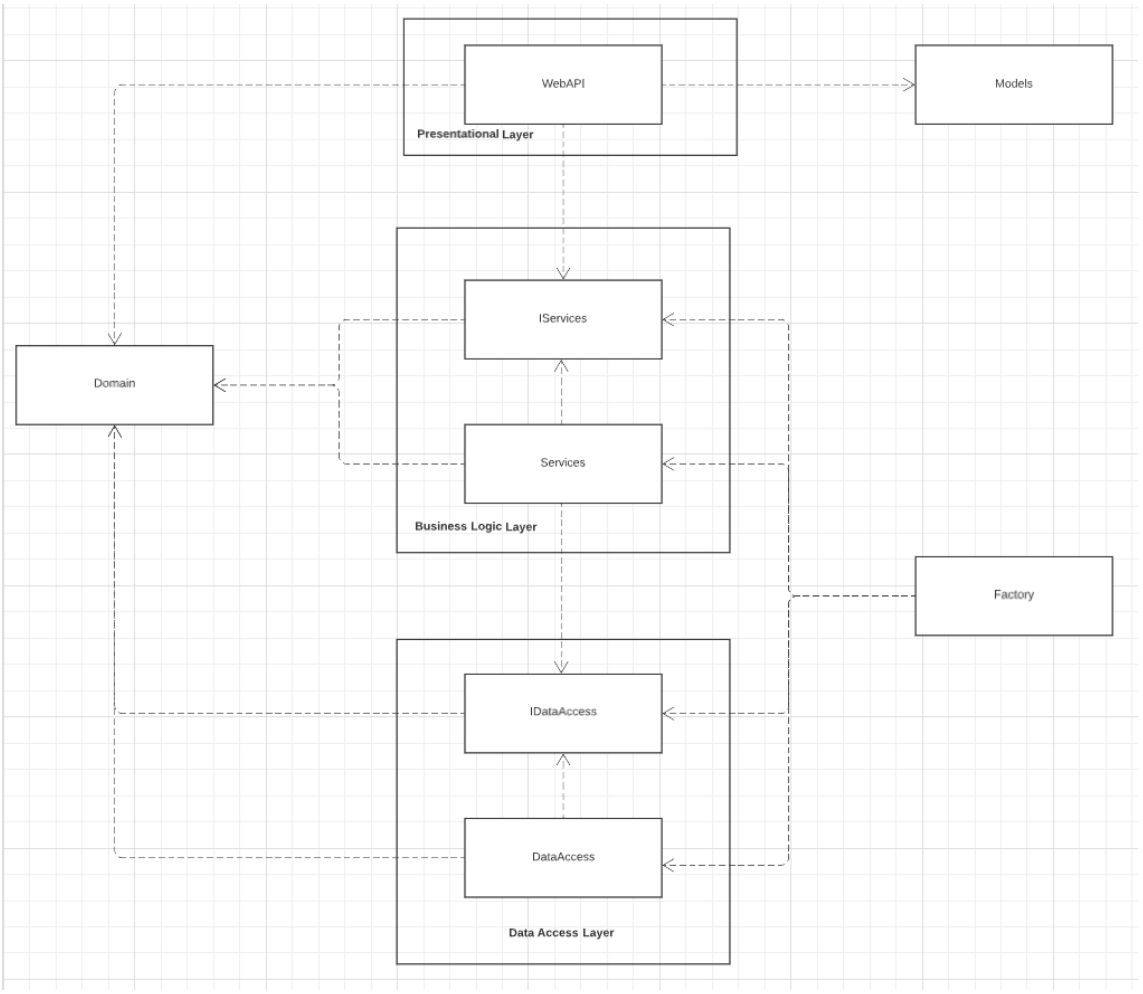
Reuse/Release Equivalence Principle

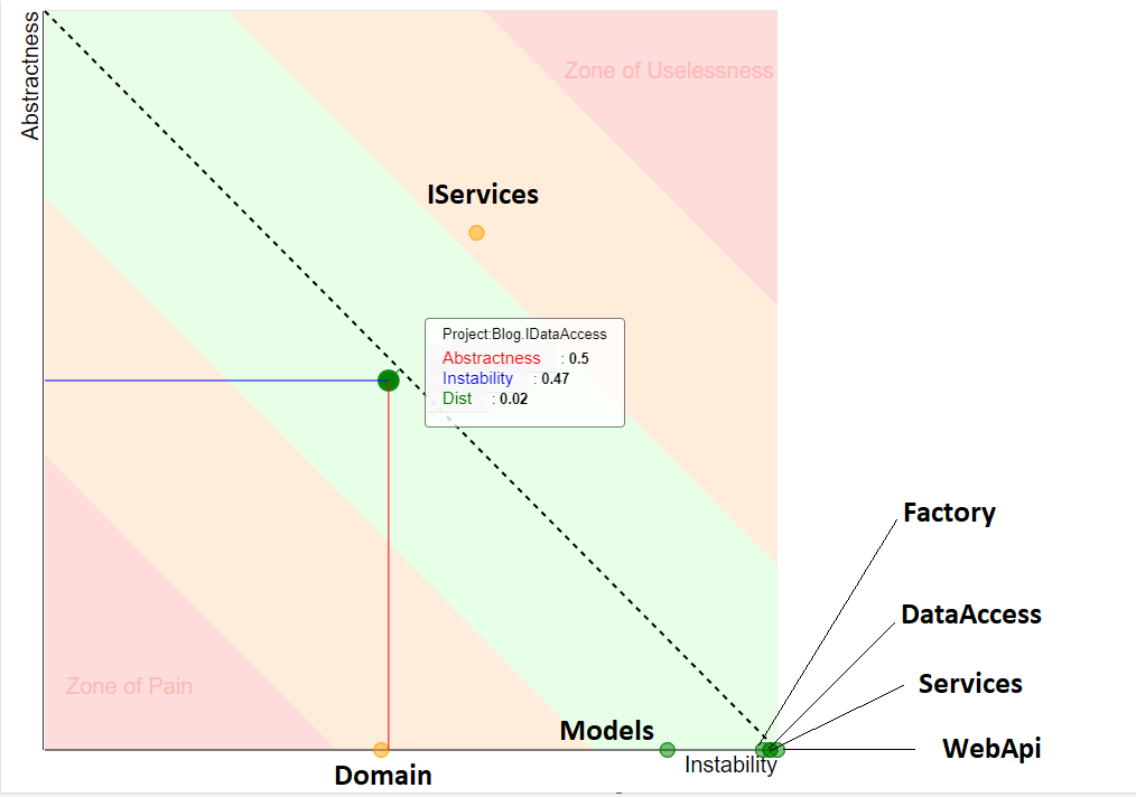
Este principio establece que el granulado de reuso y el granulado de liberación son equivalentes. Es decir, las clases y módulos que se utilizan juntos se liberan juntos. En un contexto práctico, esto significa que por cada proyecto solo debe haber 1 y solo 1 namespace.

En este caso cumplimos estrictamente con este principios en todos los proyectos que forman parte de nuestra solución. Respetando siempre tener solo un namespace por proyecto.

Stable Dependencies Principle

Este principio establece que un paquete debe depender solo de paquetes que sean más estables que él. Un paquete estable es aquel que es poco probable que cambie, lo que significa que las dependencias hacia este son seguras.


WebAPI
Models
Presentational Layer
IServices
Domain
Services
Business Logic Layer
Factory
IDataAccess
DataAccess
Data Access Layer

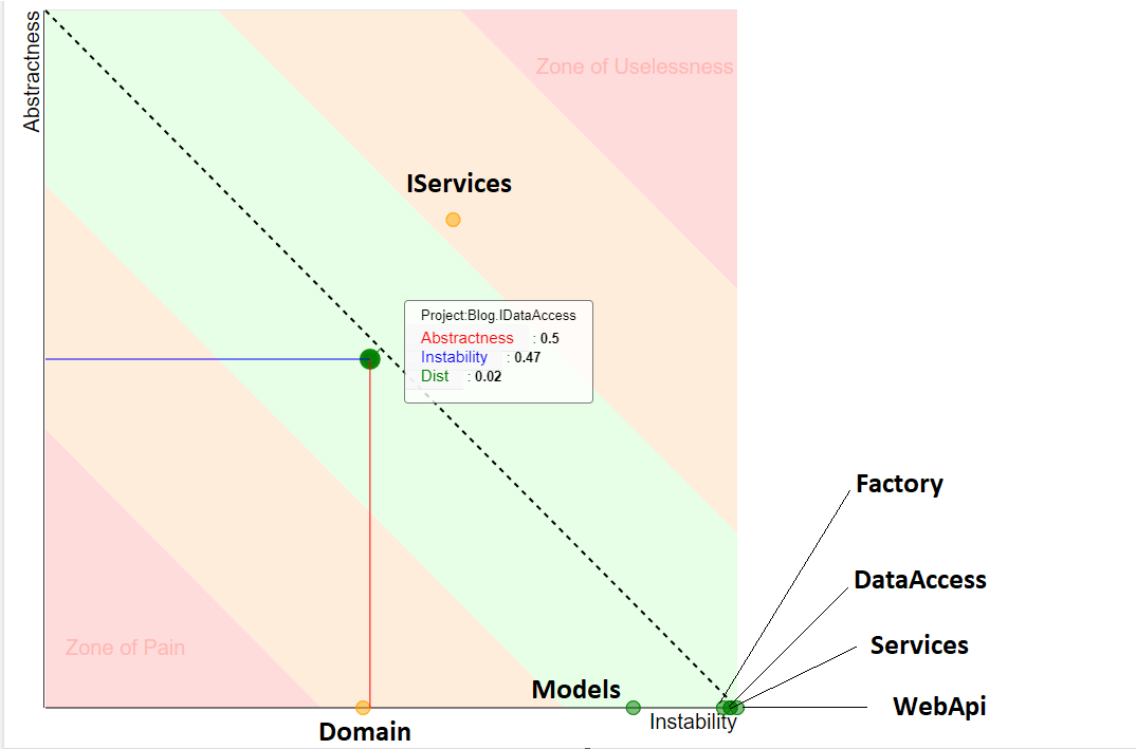


Como podemos ver en nuestro obligatorio nos adherimos estrictamente a este principio, dado que todos nuestros paquetes dependen pura y exclusivamente de paquetes más estables.

El único caso que no cumple, (por 0,1) es Factory, que depende de DataAcces y Services, ambos paquetes siendo 0,1 más inestables que Factory. (Pero no es algo que nos preocupe dada la naturaleza del paquete Factory)

Stable Abstractions Principle

Este principio establece que un paquete debería ser tan abstracto como sea su estabilidad. Es decir, los paquetes que son altamente estables y tienen muchas dependencias deberían ser altamente abstractos para permitir que su implementación pueda variar sin afectar a los paquetes que dependen de él



Como podemos ver en nuestro obligatorio cumplimos parcialmente con este principio.

Podemos decir que tanto en Services, DataAccess, WebApi, Factory y IDataAccess cumplimos con este principio (siendo 0,2 la distancia más grande entre el punto donde se encuentra el paquete y la secuencia principal.

Solo en tres paquetes podemos decir que no cumplimos, en Domain IServices y Model, siendo estos paquetes los únicos que se encuentran visiblemente lejos de la secuencia principal

Cohesión Relacional (H)

Assemblies	# lines of code	# IL instruction	# Types	# Abstract Types	# lines of comment	% Comment	% Coverage	Afferent Coupling	Efferent Coupling	Relational Cohesion
Blog.Factory v1.0.0.0	15	84	4	0	3	16.67	-	1	42	1
Blog.IServices v1.0.0.0	0	30	10	7	-	-	-	16	23	1.1
Blog.IDataAccess v1.0.0.0	0	30	6	3	-	-	-	16	14	1.5
Blog.Services v1.0.0.0	190	1773	10	0	44	18.8	-	1	68	1.7
Blog.Models v1.0.0.0	310	1632	17	0	1	0.32	-	6	33	2.06
Blog.WebApi v1.0.0.0	384	2788	20	0	33	7.91	-	0	124	2.2
Blog.DataAccess v1.0.0.0	86	1327	17	0	9	9.47	-	1	77	2.59
Blog.Domain v1.0.0.0	219	1455	19	0	2	0.9	-	46	39	2.68

A nivel de cohesión relacional, podemos decir que un paquete es cohesivo cuando su cohesión relacional se encuentra entre [1.5 , 4.0].

Paquetes cuyo H (cohesión relacional) sea menor a 1.5 decimos que es un paquete poco cohesivo, si la H es mayor a 4.0 decimos que el paquete está demasiado acoplado.

Como podemos observar solo tenemos dos paquetes cuyo nivel de cohesión no es aceptable. “Factory” e “ IServices”. Dada la naturaleza del paquete Factory, lo esperado es que no sea un paquete cohesivo.

Por otra parte, IServices es un paquete poco cohesivo, lo cual no es sorprendente, dado que se encarga de tener abstracciones para generar desacople entre la WebApi, que es quien consume nuestros servicios y el servicio actual.

Mecanismo de extensibilidad / Reflection

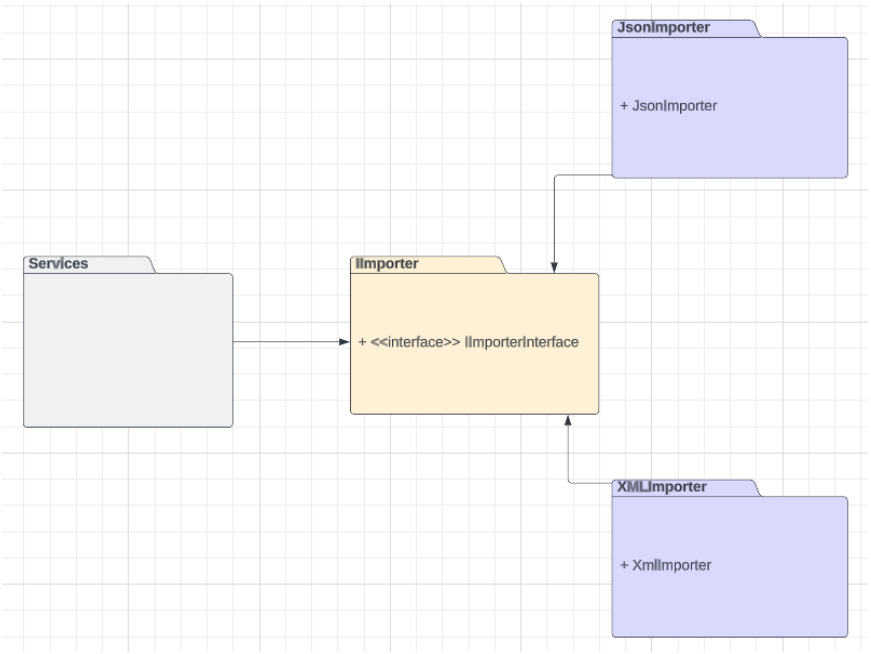
Para esta entrega se pide implementar la funcionalidad de importación de artículos sin comentarios para publicarlos desde diferentes importadores, pero no se quiere desarrollarlos internamente. Para esto hicimos uso de Reflection, que nos da la posibilidad que el sistema se pueda auto examinar y poder extender funcionalidades de forma dinámica y en proceso de ejecución del sistema sin necesidad de reiniciarlo.

Definimos una interfaz pública que será nuestro contrato con los desarrolladores que deseen extender y agregar un nuevo importador a nuestro sistema.

Para el ejemplo del obligatorio implementamos un JSON Importer, encargado de importar los artículos desde un archivo formato json usando una librería externa llamada Newtonsoft.Json que deserializa el contenido del archivo para poder interpretarlo y cargar en nuestro sistema los artículos a importar.

Luego de esto, buildeamos el importer, generamos la ddl y dejamos una copia en la carpeta /Importers de nuestro paquete WebAPI que va a ser el encargado mediante el Importer Service de poder leer las ddls de los importadores existentes en nuestro sistema e importar los artículos correspondientes.

Diagrama de paquetes utilizado para la implementación:



En el paquete de IImporter definimos la interfaz pública IImporterInterface que será el contrato con los desarrolladores externos. Por otro lado tenemos los importers; JsonImporter y XMLImporter lo cual extienden esta interfaz para implementar su lógica de importación. Para nuestro obligatorio implementamos solamente JsonImporter.

Por último tenemos el paquete de servicio que se encargará de obtener las implementaciones realizadas usando la interfaz definida IImporterInterface.