Universidad ORT Uruguay

Obligatorio 1

Diseño de Aplicaciones 2

Descripción del diseño

Agustina Disiot 221025

Ivan Monjardin 239850

https://github.com/ORT-DA2/Disiot-221025-Monjardin-239850

2021

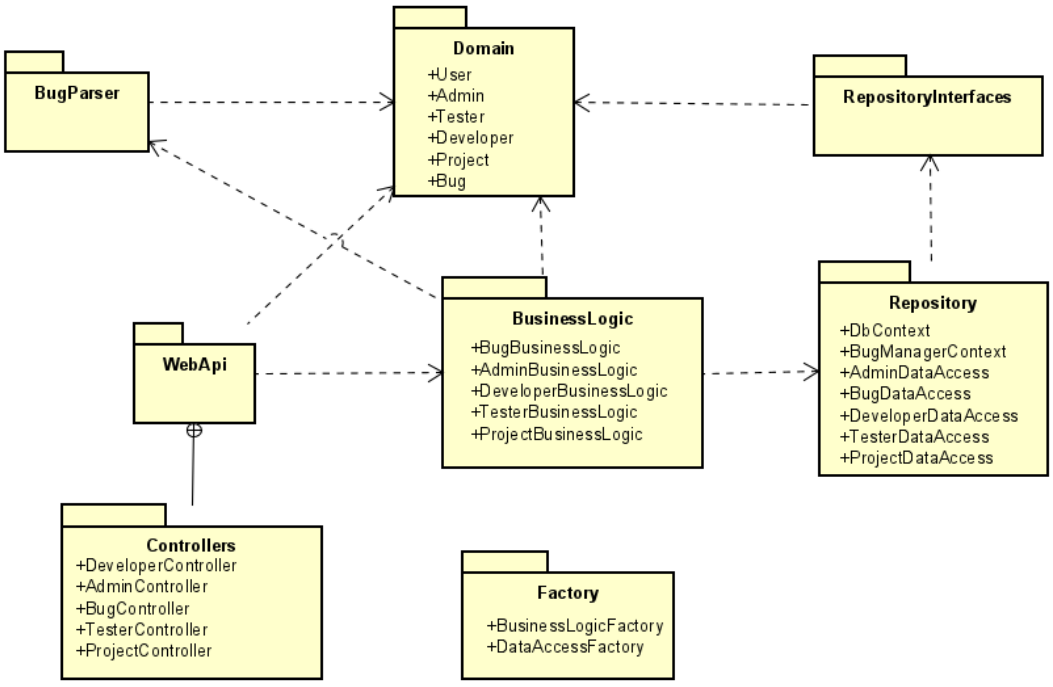
Índice:

Descripción general del trabajo

Se diseñó una aplicación para administrar los incidentes en los proyectos de software. Para esta primera entrega se implementó el backend, que incluye una API para acceder a los recursos y persistencia de una base de datos.

Para implementar los requerimientos se decidió dividir el proyecto en diferentes capas, Web Api, Business Logic, Domain y DataAccess. Un requerimiento debe atravesar todas las capas para poder completarse por lo tanto en cada capa no están solo las responsabilidades de la capa en sí, sino además, las responsabilidades relacionadas a los requerimientos. Es por esto que decidimos separar cada capa en función responsabilidades más acotadas, apuntando que cada clase de cada capa tenga una única responsabilidad, pensando siempre en el Single Responsibility Principle.

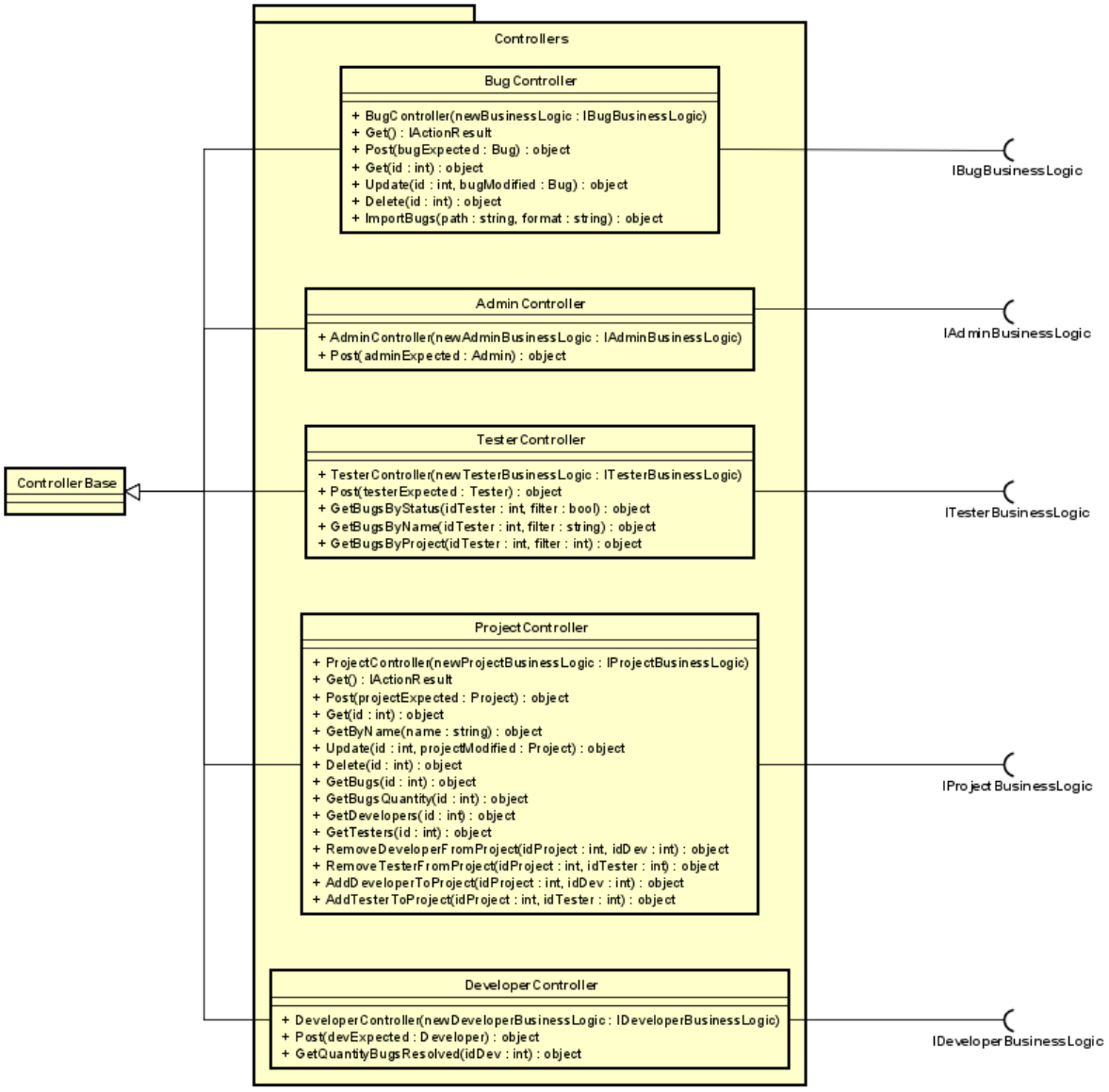
Tomando en cuenta la vista de implementación, un diagrama de las clases de cada capa/paquete es el siguiente:



Acá se puede ver la separación que hicimos principalmente tomando en cuenta la división que hicimos en el dominio. En cada capa hay una clase responsable de Bug, de Developer, de Project etc.

Pasando a la vista lógica vamos a poder ver las clases de cada proyecto

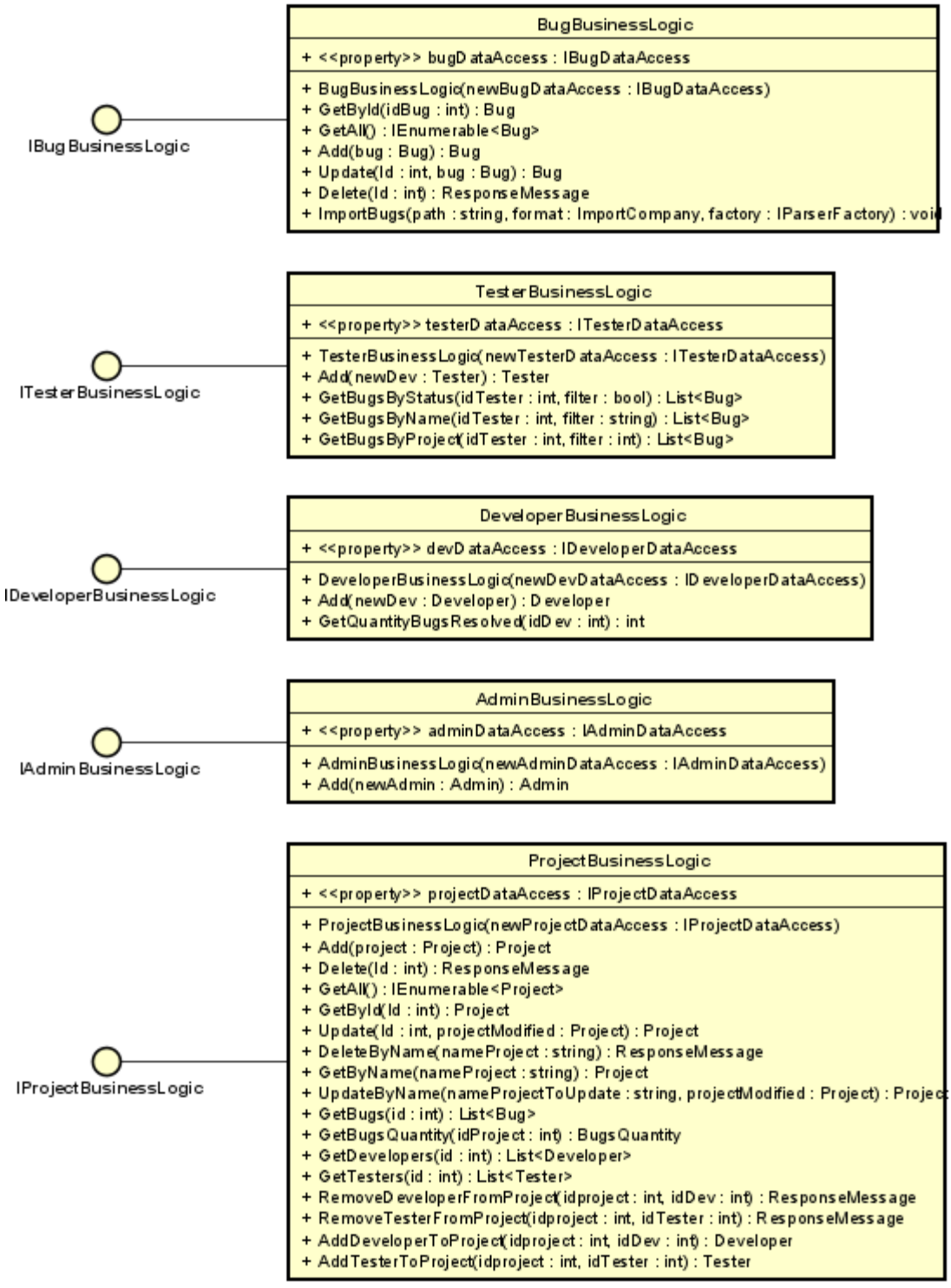
WebApi



Es la clase encargada de los controladores que administran los endpoints mediante el cual se comunicará el front-end para mandar los recursos. Más adelante se detallan todas las rutas pero lo importante es saber que cada controlador tiene su propio principio de URI, ejemplo ProjectController administra todas las request a las URI que comienzo con /projects/, idem para todos los controllers.

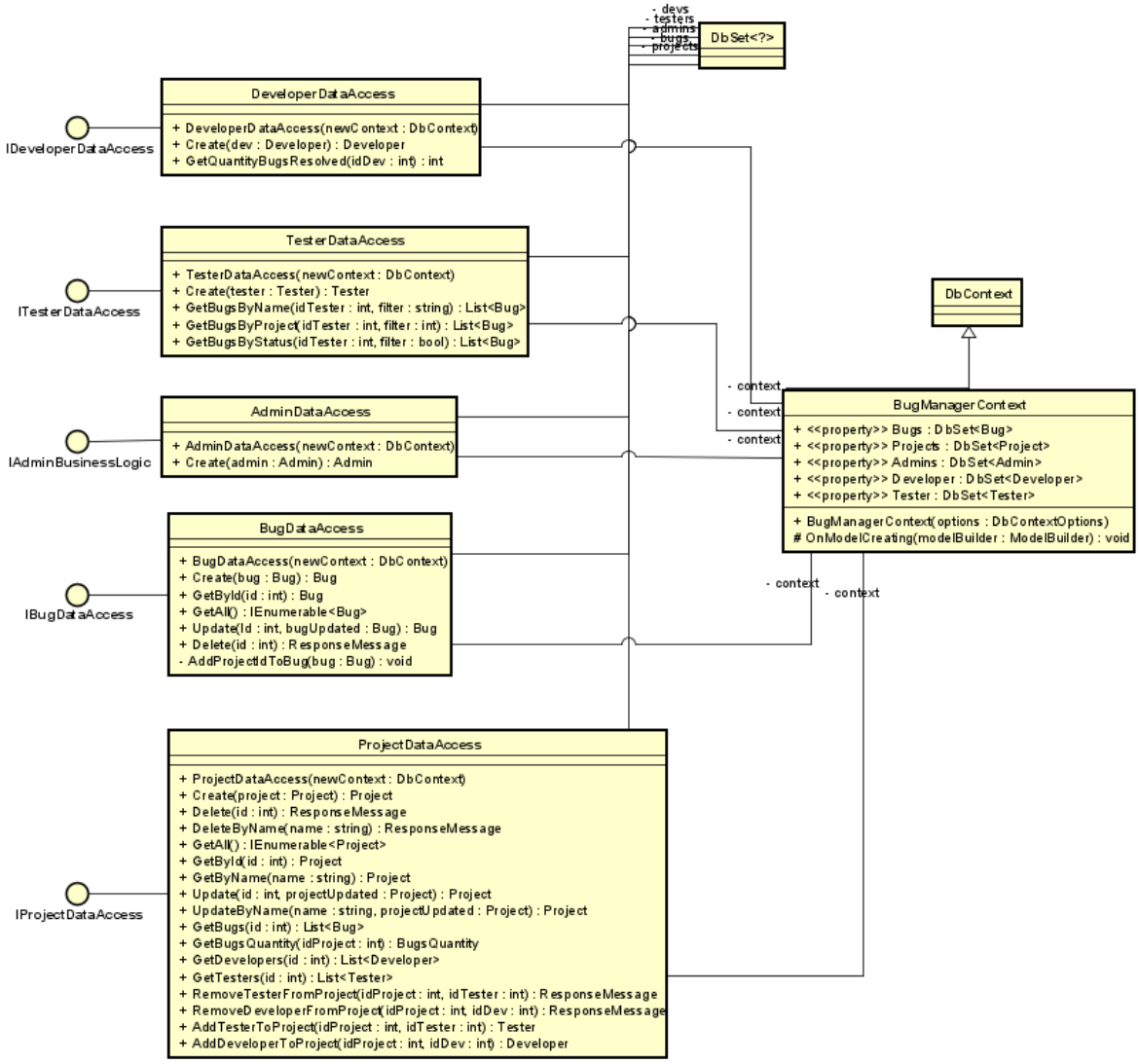
Esto nos permite hacer una separación de responsabilidad y no dejar todo en un solo controller.

BusinessLogic



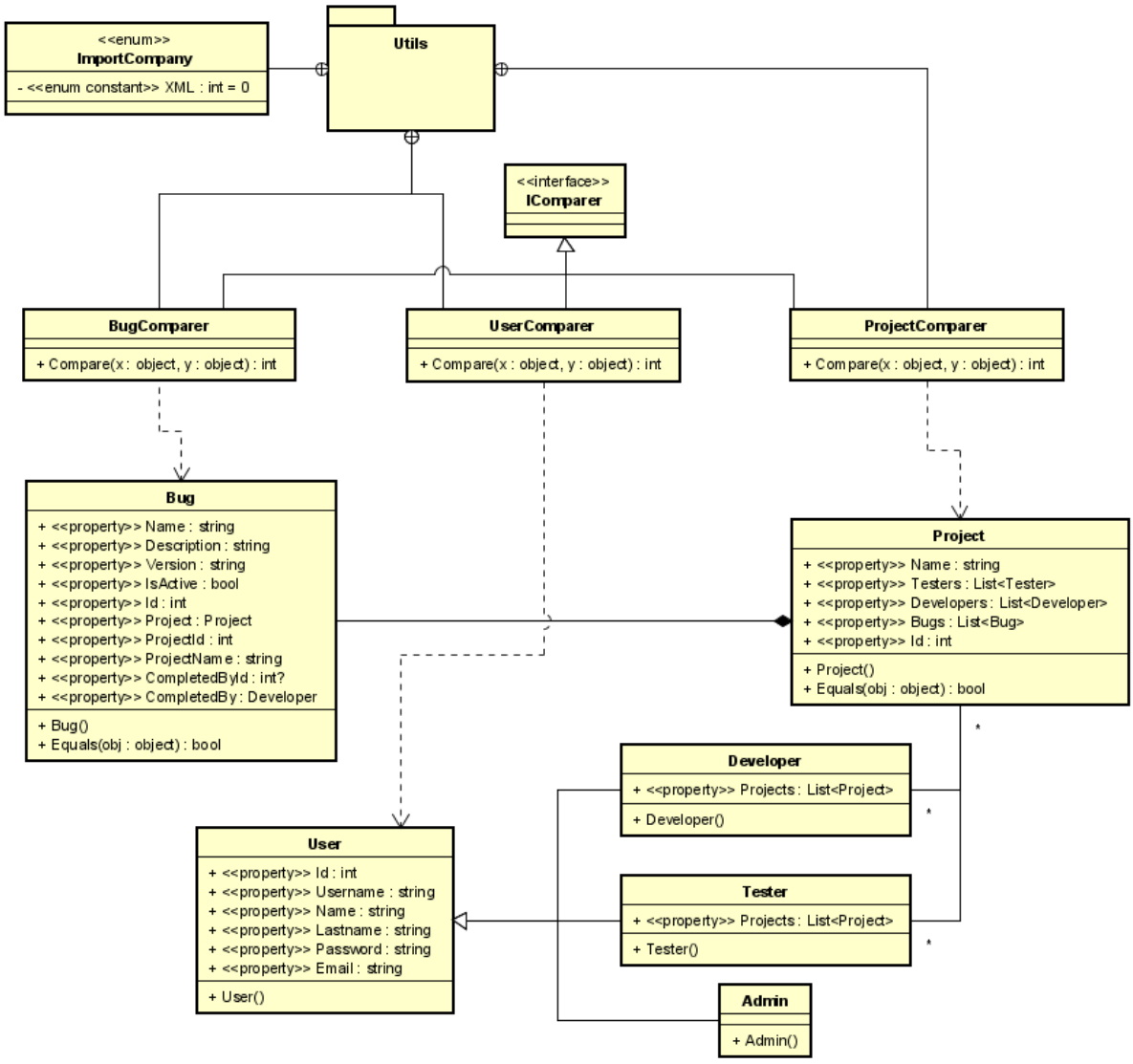
Es la capa encargada de recibir las peticiones de WebApi, aplicar las reglas de negocio necesarias y utilizar el Repository para guardar y recibir los datos.

Repository



Se encarga de guardar los datos del dominio en una base de datos persistente que fue creada utilizando Code First. Si bien Business Logic es el principal encargado de la lógica, operaciones como CRUD y otras que consideramos como responsabilidad de la base de datos se encuentran acá. Una de estas otras responsabilidades sería por ejemplo contar la cantidad de bugs resultados por desarrollador, ya que si bien tiene algo de lógica, se puede ver como una petición básica a la capa de acceso a datos.

Domain

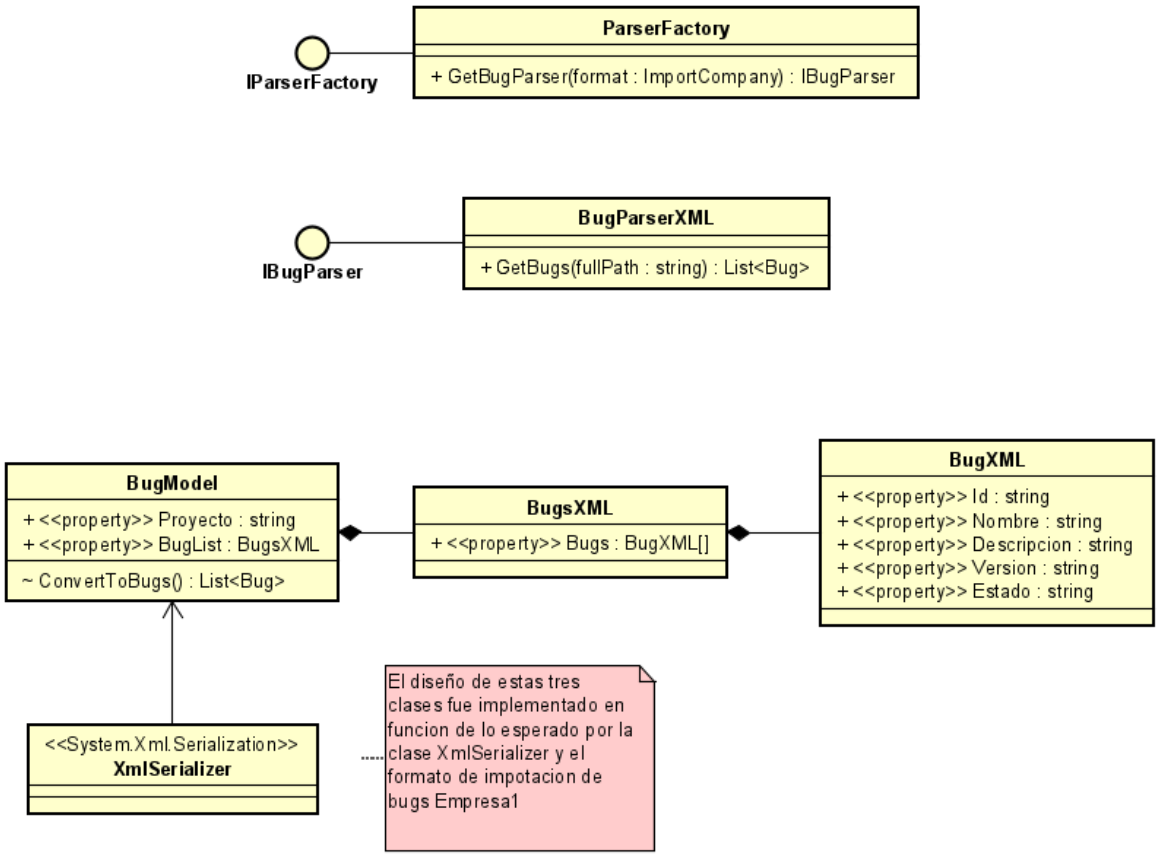


Esta capa no hace referencia a ninguna, es el paquete de las entidades básicas, las de dominio. Cada una incluye un constructor y propiedades relacionadas a esa clase. Sin bien no tienen ninguna lógica, originalmente se iba a poner las validaciones siguiendo el patrón experto y evitando “AnemicDomain”

Básicamente, cada clase es la encargada de guardar las propiedades relacionadas a ella misma. Esto incluye relaciones con otras entidades del domino. Se incluyeron ambos lados de la relación (ej. Proyecto con bugs y Bug con Proyecto) ya que facilita la utilización de EntityFramewok. Por esa misma razón se incluyeron propiedades como ProjectId que se usa en conjunto con ef core.

Además de los comparadores también se incluyeron las excepciones en el dominio. Originalmente estaban en businessLogic pero tomando en cuenta que todos hacen referencia al dominio y que una excepción se podría ver como una entidad del problema, se decidió mover a dominio.

BugParser



Para la implementación de la importación de bugs, se decidió separar del resto de los proyectos la lógica de parsear los archivos. De esta manera es más extensible si en el futuro se quiere agregar una nueva empresa no es necesario cambiar, por ejemplo el business logic.
El business logic utiliza un factory donde le pasa el tipo de archivo de la empresa que quiere importar y la factory ya le devuelve la clase correcta a utilizar.
La idea de esta clase era de hacerla lo más extensible posible. Cumpliendo el principio de abierto al cambio y cerrado a la modificación. Es fácil crear una nueva clase, y no es necesario modificar el resto de los paquetes.
Se incluyó en el diagrama las clases utilizadas para parsear los archivos a xml para lo cual se utiliza la clase de sistema XmlSerializer.

Funcionalidades implementadas y Errores Conocidos

Si bien intentamos implementar todas las funcionalidades, algunos problemas de la capa de data access nos complicaron y no llegamos a implementar transversalmente todas las funcionalidades. Algunas fueron implementadas en todas las capas pero no terminaron funcionando o tuvieron algún error en la capa de acceso a datos.

Se implementaron los siguientes funcionalidades y sus respectivos endpoints:

- Creación de usuarios:
    - Admin
    - Desarollador
    - Testers
- Creación, modificación, eliminación de proyecto y sus listas:
    - Lista de testers
    - Lista de desarolladores
    - Lista de Bugs
- Importación de bugs mediante archivo XML diseño de manera extensible para poder agregar más en el futuro (más adelante se muestra en más detalle)
- Calcular la cantidad de bugs por proyecto
- Mantenimiento de un bug, lo cual incluye: crear, modificar, eliminar sus propiedades (nombre, descripción, versión, estado)
- Para un bug en particular se muestra la información del mismo
- Además de estas funcionalidades se incluyeron otras que no son necesariamente requerimientos:
    - Conseguir todos los bugs
    - Conseguir los desarrolladores de un proyecto
    - Conseguir los testers de un proyecto
    - Conseguir los bugs de un proyecto

No se llegaron a implementar del todo o contienen errores los siguientes requerimientos:

- Login o identificación del usuario que utiliza la web api
- Importación de archivo mediante texto posicional (si se incluyeron factories, interfaces y otras clases que podrán facilitar su desarrollo en un futuro)
- Cantidad de bugs resueltos por un desarrollador, no funciona siempre
- El filtrado de bugs por proyecto, nombre etc. fue implementado en todas las capas pero no logramos que funcione en la capa de data access
- Conseguir la lista de los proyectos de un desarrollador
- La modificación de bugs permite modificar de activo a resuelto, pero consideramos mejor práctica hacer un endpoint más específico para esto.
- Validación del dominio o de gran parte de los datos/ agregar más excepciones. El manejo de excepciones si está y se va a explicar más adelante.

Independencia de Librerías

A lo largo de las clases se ve nuestro uso de interfaces tiene dos principales ventajas:

1. Nos permite utilizar inyección de dependencias, la cual hablaremos más adelante
2. De la mano al anterior, las interfaces nos permiten desacoplar cada capa una de la otra. Ninguna capa depende de una implementación concreta de la otra. Además esto nos permitió realizar tests utilizando mocks (algo que también hablaremos más adelante)

El proyecto se podía haber realizado igual sin utilizar todas las distintas clases que mencionamos anteriormente,por ejemplo, teniendo todo en WebApi. Sin embargo eso iba a traer grandes problemas de extensibilidad, mantenimiento y romper muchos principios como single responsibility, bajo acoplamiento y alta cohesión. Todo eso nos dio la pauta de diseñar un sistema donde los componentes tengan dependencias lo más estables posible. Por ejemplo, dependiendo de interfaces, un "contrato" pensado para que dure aunque se modifique las clases concretas que lo implementan.

Inyección de dependencias

Se utilizó inyección de dependencias para poder evitar dependencias circulares y teniendo una forma fácil de modificar las clases concretas que utilizamos en nuestra aplicación. Esto quiere decir que:

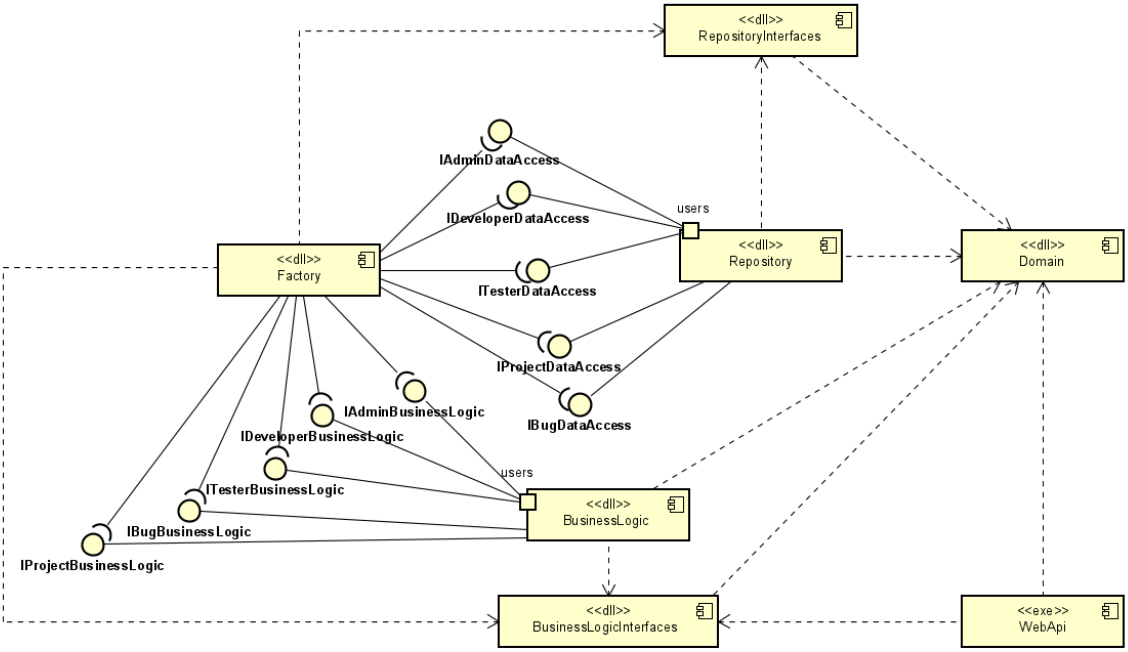
Las clases de business logic no crean una clase concreta de, por ejemplo, data access, para poder acceder a la base de datos. Business logic solo utiliza la interfaz de la clase del repository y sus funciones. Se utiliza inyección de dependencias para especificar a la hora de correr la webApi, que implementación concreta de dicha interfaz utilizar.

Cómo implementamos inyección de dependencias fue mediante una clase que usa el patrón factory en el proyecto web api (ya que ahí es donde se corre el back end). Está factory conecta cada interfaz con su clase concreta que deseamos utilizar.

```
public void AddCustomServices()
{
    serviceCollection.AddScoped<IBugBusinessLogic, BugBusinessLogic>();
    serviceCollection.AddScoped<IProjectBusinessLogic, ProjectBusinessLogic>();
    serviceCollection.AddScoped<IAdminBusinessLogic, AdminBusinessLogic>();
    serviceCollection.AddScoped<IDeveloperBusinessLogic, DeveloperBusinessLogic>();
    serviceCollection.AddScoped<ITesterBusinessLogic, TesterBusinessLogic>();
}
```

Código de la factory de business logic

En el siguiente diagrama de componentes podemos ver como funciona la factory y la inyección de dependencias. Vemos como hay muchas clases que no se utilizan directamente pero que pasan por la Factory para recibir las implementaciones concretas.


<<dll>>
RepositoryInterfaces
IAdminDataAccess
IDeveloperDataAccess
users
<<dll>>
Factory
<<dll>>
Repository
<<dll>>
Domain
ITesterDataAccess
IProjectDataAccess
IBugDataAccess
IAdminBusinessLogic
IDeveloperBusinessLogic
users
ITesterBusinessLogic
IBugBusinessLogic
IProjectBusinessLogic
<<dll>>
BusinessLogic
<<dll>>
BusinessLogicInterfaces
<<exe>>
WebApi

Nuestras decisiones de diseño

A lo largo del obligatorio fuimos tomando varias decisiones de diseño en función de patrones o principios que nos parecieron importantes. Acá están las más importantes, en ningun orden en particular:

- Cada interfaz (IBusinessLogic, IDataAccess, etc) van en proyectos a parte y no en una carpeta adentro de la clase concreta. Esto habilita a no depender de la clase concreta en caso que usen el mismo namespace y hace la solución más extensible. Se puede agregar y eliminar clases concretas sin modificar la interfaz tanto de lugar como sus metodos. Lo único que hay que cambiar es el factory de la inyección de dependency en caso que se quieren cambiar las clases concretas.
- Cuando hubo interfaces similares, por ejemplo en IBusinessLogic donde muchas de las clases van a implementar métodos de CRUD, se agruparon los métodos en comun en una interfaz usando Generics (IBuisnessLogic<T>) permitiendo que se hagan nuevas interfaces específicas relacionadas a cada clase del dominio. Esto primero era para no repetir código y permitiendo en un futuro se quiere cambiar algún método, por ejemplo el retorno de Delete o similar, se puede hacer fácilmente y no haya que modificar varias clases. Más que nada esto da una coherencia a la hora de utilizar métodos básicos de los recursos. De todos modos, siempre es mejor una interfaz más específica que una generica que después tengan que implementar las clases concretas. Los métodos de bugs no deberían pertenecer a la interfaz de proyecto y vice versa, apuntamos siempre a que las clases usen las interfaces más especificas que puedan, siguiendo los principios de diseño.
- Al igual que no tenemos las interfaces dentro de una clase concreta, tampoco tenemos los test dentro de la clase. Nos pareció mejor tener proyectos separados.Una razon es que así hay responsabilidades claras y no hay un mismo proyecto que hace dos cosas (ejecución y pruebas).
- Se investigó y se decidió que el responsable de hacer, por ejemplo, SaveChanges() a la base de datos es el data Access y no , por ejemplo, la businessLogic. Nosotros vimos cómo su “singular responsibility” administrar todo lo relación al almacenamiento de datos. Similar a como podría suceder en una base de datos no code first donde se le manda consultas SQL posiblemente complejas para que ejecute la BD y retorne algún valor. Esto permitió no incluir operaciones específicas de data access al resto en los proyectos. Si bien a veces podría ser útil ejecutar dos operaciones
- El bug Parser es el responsable de abrir el archivo y devolverme su lista de bugs del archivo. La idea es que la única responsabilidad de esa clase es devolverte una lista de bugs de un path, como lo hace no debe importarle a las otras clases. Por lo tanto, no nos pareció bien pasarle un Stream File o hacer parte de lo que se consideraría importar bugs de archivos en otra clase.
- Se utilizaron archivos xml de pruebas para los tests de Bug Parser y métodos de mstest para moverlos a la carpeta de Debug para que estén disponible a la hora de ejecutar los tests. No nos pareció bien ni:
    - Poner el archivo en el código y pasarlo de alguna manera, ya que ocuparía mucho espacio en el código y dificulta la lectura.

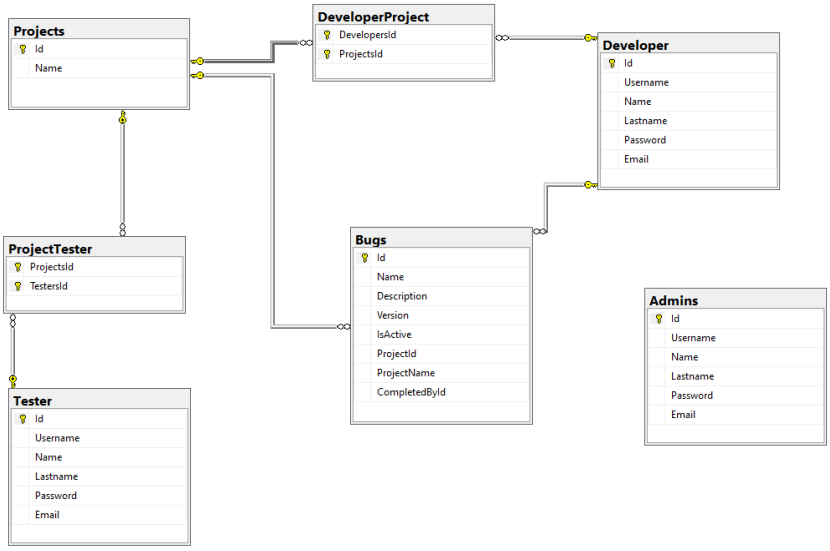
    - Armar los archivos durante los tests. Sería poner mucha lógica innecesaria en los tests
    - Utilizar algún tipo de Mock para files. Primero porque creo que no se puede mockear al ser estática y segundo porque, como dijimos antes, la responsabilidad de la clase incluye poder utilizar el archivo que le pasen.
- Entendemos de la letra que si en el futuro se agregan más formas de importar bugs para otras empresas, que van a ser siempre archivos. Y no, por ejemplo, pasarle un binario mediante la web api sin antes guardarlo en un archivo. De todos modos, se diseñó para que se pueda agregar cualquier tipo de archivo en el futuro.
- Utilizamos dos factories de servicios para cada capa de la inyección de dependencias de businesslogic y data access. Poner junto la inyección de dependencias de business logic y de data access, rompía el single-responsibility principle. Tampoco está bueno, poner todas las capas en un misma clase, de la forma que lo implementamos se podría agregar una tercera capa más fácilmente (sin modificar los archivos ya existentes).
- Si bien sólo los llegamos a usar entre web api y el cliente, hubiera estado bueno utilizar DTO entre business logic y web api. Esto nos permitiría más control de que se le manda el cliente y que no. Por ejemplo, muchas propiedades que existen para facilitar el uso con EntityFramework no son necesarias mandarlas al cliente. Esto se va a notas más en el Swagger que utiliza todos los parámetros.
- Dejamos el default de EF sobre la primera Id de las entidades siendo 1. Permitiría diferenciar entre un valor posiblemente válido y el 0 que seguro no es válido.
- En la api de importación de bugs, elegimos que el usuario mande el path del archivo por el header y no por la route, así se soluciona el problema de mandar algunos caracteres no válidos en la URI.
- Con respecto al obligatorio y GitFlow:

Se establecieron algunos estándares para trabajar en el repositorio, como por ejemplo:

- El nombre de las ramas feature es "feature-NombreDeLaRama". Se utiliza CamelCase y la primera letra del nombre va con mayúscula.
- Los commits se escriben en presente (Ej. “Add**s** modify button”).
- Para arreglar una funcionalidad se crea una rama “fix-NombreDeLaRama”
- Si se generó un problema al mergear a develop, se puede arreglar y crear una rama fix.
- Utilizamos pull request donde la otra persona tiene que aceptar la pull request
- Para hacer que git detecte carpetas vacias se agrego un archivo .gitkeep según la convención
- Los commits automaticos de Github (ej, Merge pull request) no los modificamos
- No se puede mergear a develop directamente ni sin la aprobación de la otra persona

Mecanismo de Acceso a datos

La capa de acceso a datos se utilizó para persistir los objetos del dominio. Es posible cerrar la aplicación y volverla a abrir y continuar teniendo acceso a los mismos datos.
Para la creación de la base de datos se utilizó la técnica de Code First y EntityFramework.
En función de nuestro dominio generamos la siguiente estructura de base de datos.



Descripción del manejo de excepciones

Se crearon algunas excepciones que se tiran en su mayoría en la capa de data access, pasando por business logic hasta llegar a la capa de web api.
En dicha capa, se utilizó un filtro (ExceptionFilter) para detectar las excepciones y que sea él el responsable de manejar y mandar los errores a los clientes.
La utilización del filtro tiene varias ventajas:

- Los controlleres de webApi no tienen que preocuparse por las excepciones, solo se encargan de enviar las respuestas exitosas. Por lo tanto mantienen una única responsabilidad y simplifican el código ya que cada función de un controlador suele ser de 1 o 2 líneas.
- Nos permite tener una única clase con responsabilidad de los errores y cualquier cambio que se quiera hacer con respecto a los mismos se hace en un solo archivo. No, por ejemplo, en cada controlador. Esto hace la solución abierta a la extensión y cerrada al cambio.
- Nos permite enviar códigos http de error específicos a cada excepción. Por ejemplo si un recurso no se encontró tiramos 404. Si hay una excepción que no chateamos, tenemos un “else” que devuelve el código de error 500 significando un error por parte del servidor.

Si bien se intentó realizar este filtro con TDD, no se logró del todo y las pruebas fueron eliminadas ya que no controlan correctamente si las clases tiraban las excepciones o no.

Deploy

Para poder conectarse con una base de datos a elección, hay que modificar la connection string en el archivo .appsettings del proyecto WebApi.
También hay un .env en el proyecto Repository pero en principio no sería necesario cambiarlo para ejecutar la web api.
Para poder utilizar la api, hay que conectarse a
http://localhost:5000/ ejemplo:
http://localhost:5000/projects/2/bugs

Conslusión

Si bien tuvimos problemas a la hora de implementar algunas funcionalidades decidimos enfocarnos en la documentación y en el diseño de la solución. Haciéndola extensible e intentando que sea sencilla de modificar.