Universidad ORT Uruguay

Obligatorio 1

Diseño de Aplicaciones 2

Docente: Gabriel Piffaretti

Agustín Ferrari 240503

Francisco Rossi 219401

ORT-DA2/240503_219401_ (github.com)

2021

Índice:

1. Descripción General:

En este proyecto el equipo se propuso como objetivo, desarrollar código profesional, fácil de entender, consistente y extensible. Para llevar a cabo esto, se implementaron varias técnicas aprendidas durante el curso, como, Test Driven Development (TDD), recomendaciones de Clean Code, entre otras.

Se diseñó una solución, teniendo en mente desarrollar de una forma profesional y simple, que contenga componentes extensibles a futuros cambios y pueda soportar mantenimiento si es necesario, pudiendo reutilizar código ya existente.

El equipo cuenta con 2 integrantes, por lo que se decidió en conjunto, trabajar utilizando GitHub y Git Flow, creando ramas individuales para cada funcionalidad y/o refactors necesarios. Se planteó de esta manera para tener la posibilidad de avanzar sin tener que ser necesario la disponibilidad horaria de los otros integrantes todo el tiempo. Para que el código pueda ser consistente y todos estemos al tanto de los avances, se priorizó el uso de los pull request y reunirse al menos 1 o 2 veces al día para discutir y sacar dudas entre todos. Todos los pulls request fueron revisados por el otro integrante y se hicieron comentarios con sugerencias o cambios necesarios, los cuales fueron arreglados antes de realizar los merge a develop.

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. “Adds modify button”).
- Para arreglar una funcionalidad se crea una rama “fix-NombreDeLaRama”
- Si se generó un problema al mergear a develop, no se puede arreglar y commitear directo a develop. Se debe crear una rama Fix para arreglar ese problema puntual.
- Regla para los pull request, el otro integrante debe aprobar los cambios

Continuamos trabajando con la herramienta Notion como lo hicimos para DA1. Nos ayudó a manejar las tareas necesarias, y a planear una mejor solución.
Cada vez que se nos presentaba una situación o se agregaba una modificación en el foro, se agregaba en la columna de Not Started y se le asignaba una prioridad del 1 al 5, siendo 1 lo más importante. Luego cada vez que algún integrante se ponía a desarrollar, elegía una tarea, la marcaba como In progress y creaba la rama para trabajar en ellas. Finalmente se pasaba a la columna de Complete y se podía avanzar con otra tarea.

Diseño

Se comenzó a diseñar la solución final, por el dominio, más específicamente por el proyecto de DomainTest, el cuál gracias a TDD nos generó los objetos del dominio como Bug, Project y User. En este proyecto surgieron las validaciones de ciertos campos, como por ejemplo el email del usuario.
Luego se continuó con DataAccessTest para poder crear los repositorios necesarios, nuestra solución cuenta con un repositorio por entidad del dominio y además un para las sesiones activas de los usuarios logueados.

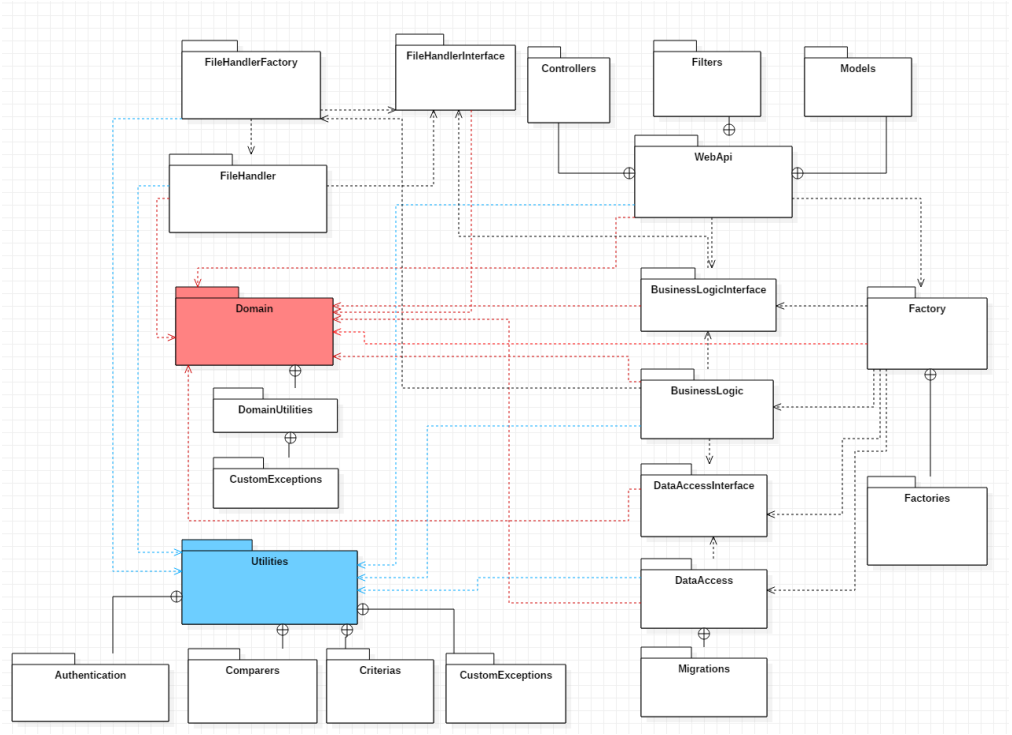
La lógica de negocio se implementó a partir de los test también dividiendo las clases, para que se ocuparan de un recurso en particular y sus responsabilidades, por ejemplo BugLogic se encarga de comunicarse con el repositorio de bug para cumplir las acciones requeridas por el usuario interpretadas por la WebApi. En este último paquete se procesan e interpretan las request por parte del front-end para determinar con qué lógica de negocio se debe comunicar para satisfacer el pedido.

Pudimos implementar todas las funcionalidades requeridas e incluir algunas que consideramos importantes para mejorar la usabilidad del sistema. Como por ejemplo obtener los users del sistema, esta acción solo la puede realizar el Admin.Las decisiones de cómo y cuáles funciones implementar fueron tomadas en conjunto por todo el equipo.

2. Diagramas:

Todos los diagramas mostrados a continuación se encuentran en formato svg publicados en GitHub y adjuntos en el archivo .zip entregado en la carpeta Documentation/Diagrams.

Diagrama de Paquetes



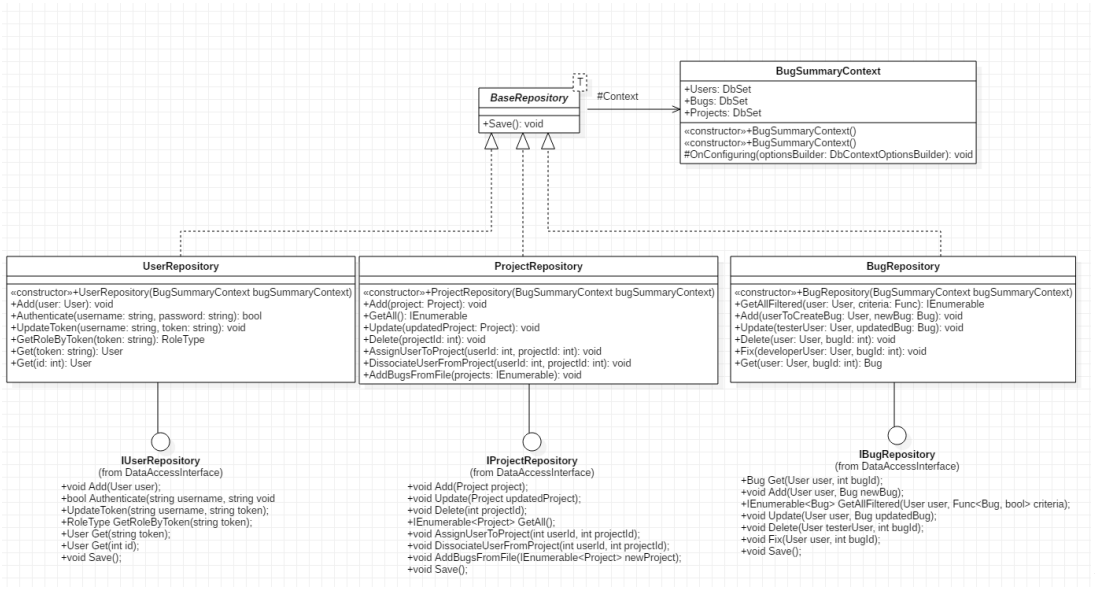
[1]

Se pintó de colores los paquetes más estables del sistema, aquellos los cuáles la mayoría dependen de ellos, pero sin que estos dependan de nadie.
Agregamos varios paquetes a la solución, pero 3 de los más importantes fueron WebApi, BusinessLogic y DataAccess.WebApi se implementó de forma que interprete todos los pedidos del cliente o front-end a través de APIs y rutas determinadas para resolver un acción específica comunicándose con otras capas del sistema. El segundo fue una decisión de diseño para poder utilizar controladoras encargadas de la lógica de la aplicación y el último para crear una capa de acceso con la base de datos (BD). En esta última, está el contexto de la aplicación y la interacción con la BD, creando allí las tablas y restricciones necesarias. También están las operaciones de búsqueda y modificación de datos, que finalmente se terminó usando en la WebApi.
En particular los últimos dos paquetes mencionados implementan interfaces que exponen sus métodos hacia los otros paquetes, sin necesidad de que se conozca su implementación. Los paquetes que contienen Interface en su nombre tienen la responsabilidad de exponer los métodos, para que la capa que quiera usar esos métodos se acople solamente a las interfaces.

[1] Diagrama encontrado en Documentation/Diagrams/PACKAGE.SVG

Diagramas de clases:

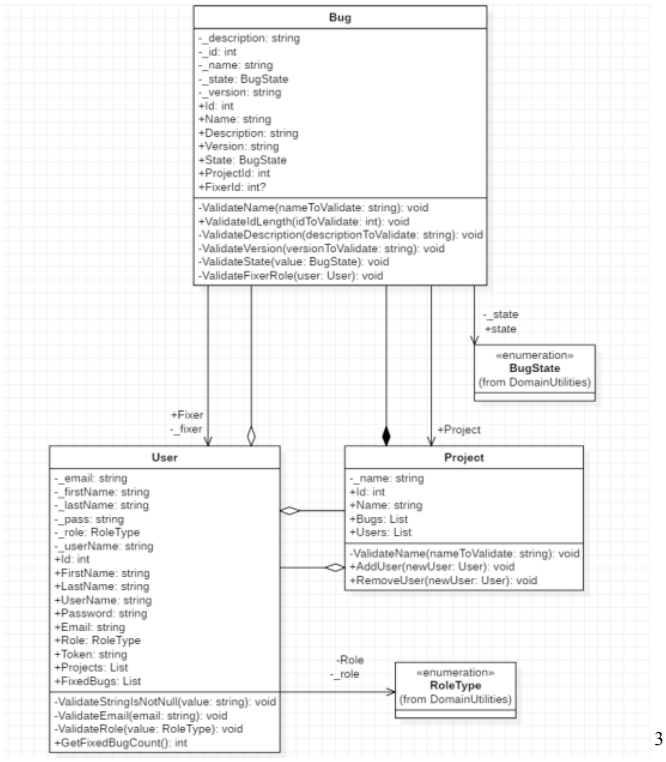
Paquete DataAccess



[2]

Las responsabilidades de este paquete son, interactuar con la base de datos para distintas acciones, tanto de búsqueda, como de modificación, de agregar o borrar objetos del sistema según corresponda. También contiene el contexto del sistema, el cuál guarda las entidades como DbSet para poder realizar las acciones ya mencionadas.

Paquete Domain

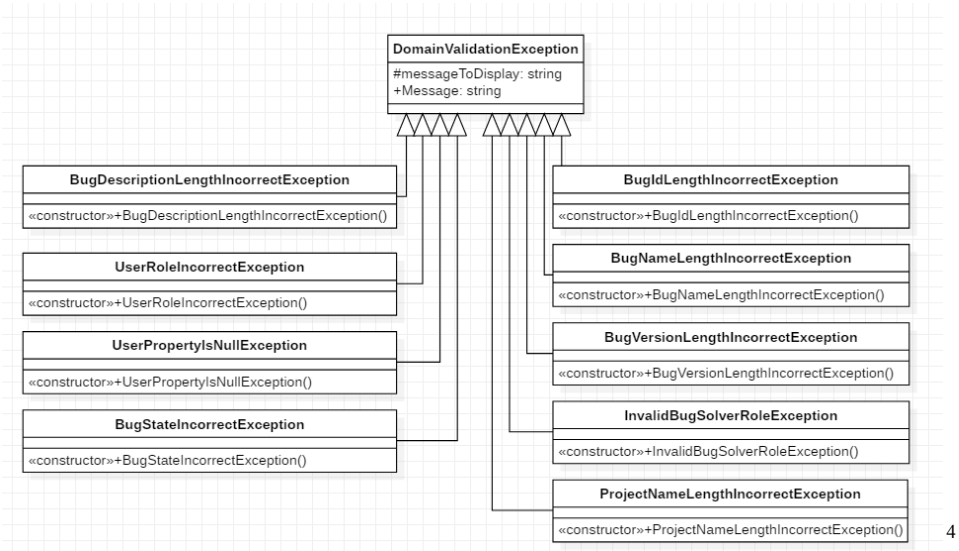


[3]

[2] Diagrama encontrado en Documentation/Diagrams/DATAACCESS.SVG
[3] Diagrama encontrado en Documentation/Diagrams/DOMAIN.SVG

Este paquete tiene la responsabilidad de contener los objetos del sistema, Bug, User y Project. Siguiendo las recomendaciones de Martin Fowler “The logic that should be in a domain object is domain logic...” en el artículo AnemicDomainModel, implementamos validaciones de las properties en las clases del dominio mismo, por ejemplo que el largo del nombre del proyecto no supere los 30 caracteres.

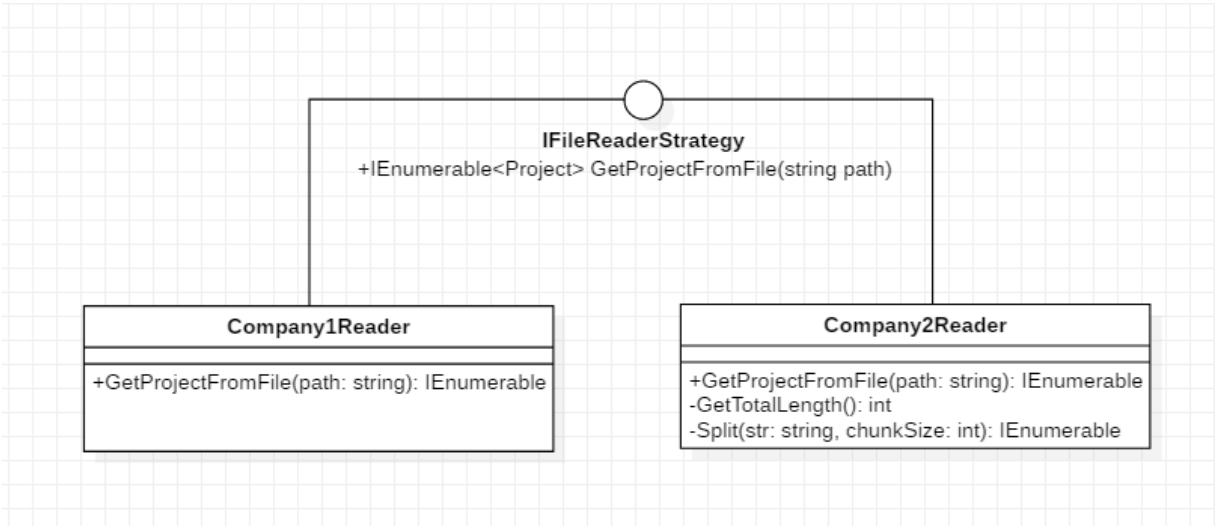
Excepciones de dominio:



4

Para el manejo de errores, decidimos crear excepciones personalizadas que fueron utilizadas a lo largo de las diferentes capas. Las mismas heredan de una clase padre, se planteó de esta manera para que a la hora de capturarlas fuera más sencillo y seguir las recomendaciones de Clean Code.

Paquete FileHandler

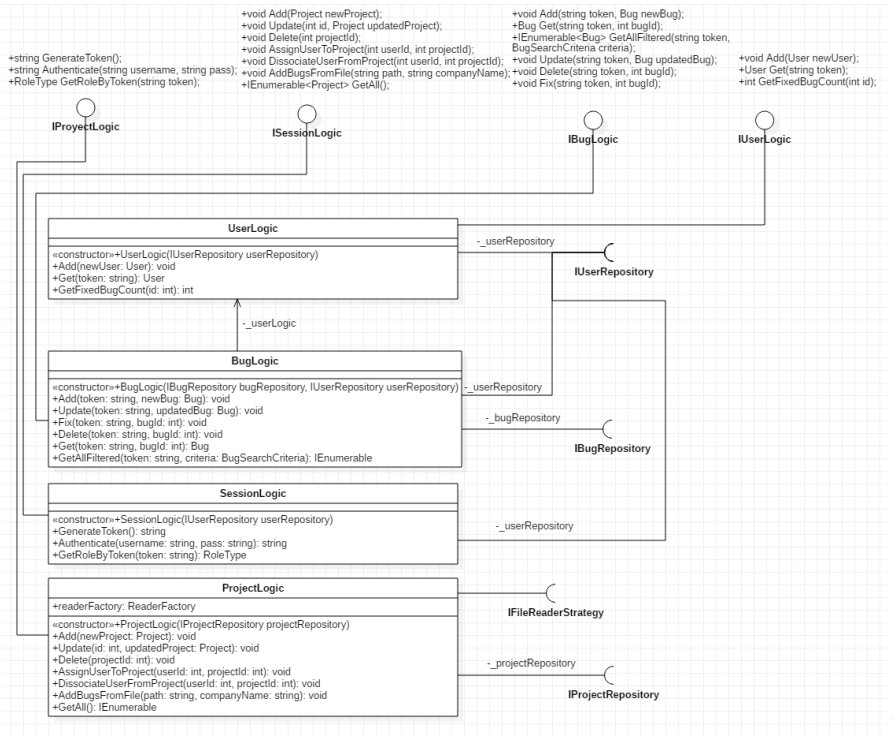


5

[4] Diagrama encontrado en Documentation/Diagrams/FILEHANDLER.SVG
[5] Diagrama encontrado en Documentation/Diagrams/DOMAINEXCEPTIONS.SVG

Este paquete se encarga de convertir los archivos de carga de bug enviados por las empresas en objetos del dominio que puedan ser almacenados fácilmente en la base de datos. Para poder hacer una solución extensible, se implementó el patrón strategy, creando una interface IFileReaderStrategy requiriendo la implementación del método que a partir de un link devuelve una colección de proyectos a agregar al sistema. Para cada empresa que tenga un método de carga de bugs, se crea una clase que herede de esta interfaz y se implementa la lectura de los datos. En este caso solo tenemos dos empresas, Empresa1 y Empresa2, por lo cual se crearon dos clases; Company1Reader y Company2Reader. Para obtener la strategy correcta, se creó una Factory, que dependiendo del string recibido, devuelve una instancia de la clase que se encarga de manejar la carga de bugs para esa empresa. Con esta solución se permite añadir nuevos métodos de cargas y/o empresas sin problema, únicamente se necesita crear una nueva clase que herede de la interfaz strategy y agregarlo al factory.

Paquete BusinessLogic

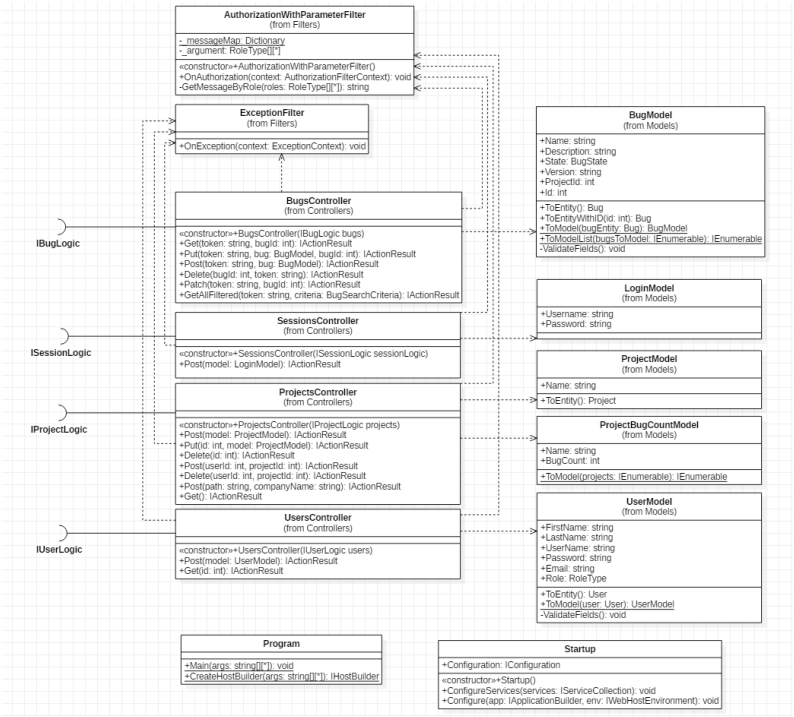


6

La responsabilidad de este paquete es a partir de los pedidos que se realizan en la WebApi, poder consumir los métodos expuestos por los DataAccess según corresponda para impactar los datos del sistema.

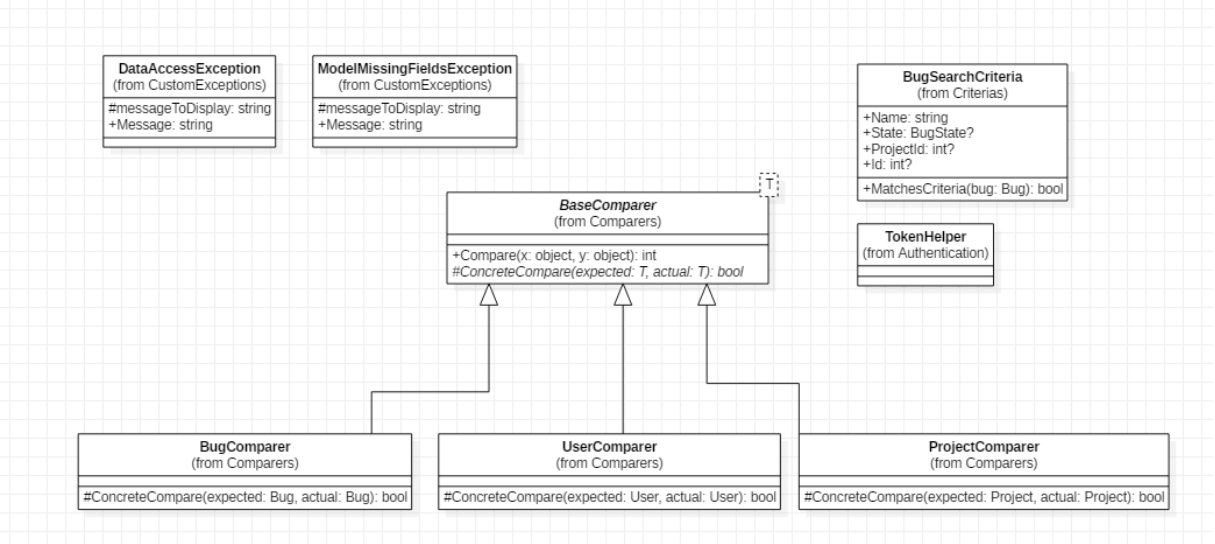
[6] Diagrama encontrado en Documentation/Diagrams/BUSINESSLOGIC.SVG

Paquete WebApi



Este paquete tiene distintas responsabilidades. La principal es la interpretación del front-end en un endpoint concreto, para realizar la acción correspondiente. También se encarga de convertir a modelos o DTOs, los objetos JSON del Postman para mapearlos a una entidad concreta del dominio, también se realiza la operación inversa de convertir objetos a modelos para enviarlos en un Get por ejemplo. Por último tiene la responsabilidad de aplicar filtros a los endpoints, en este sistema solo se implementan dos, pero muy útiles. Uno es sobre las excepciones que tiran a lo largo del sistema, ExceptionFilter para poder devolver un mensaje de error al cliente. El otro es para validar que el usuario tenga los permisos correspondientes para la request solicitada.

Paquete Utilities



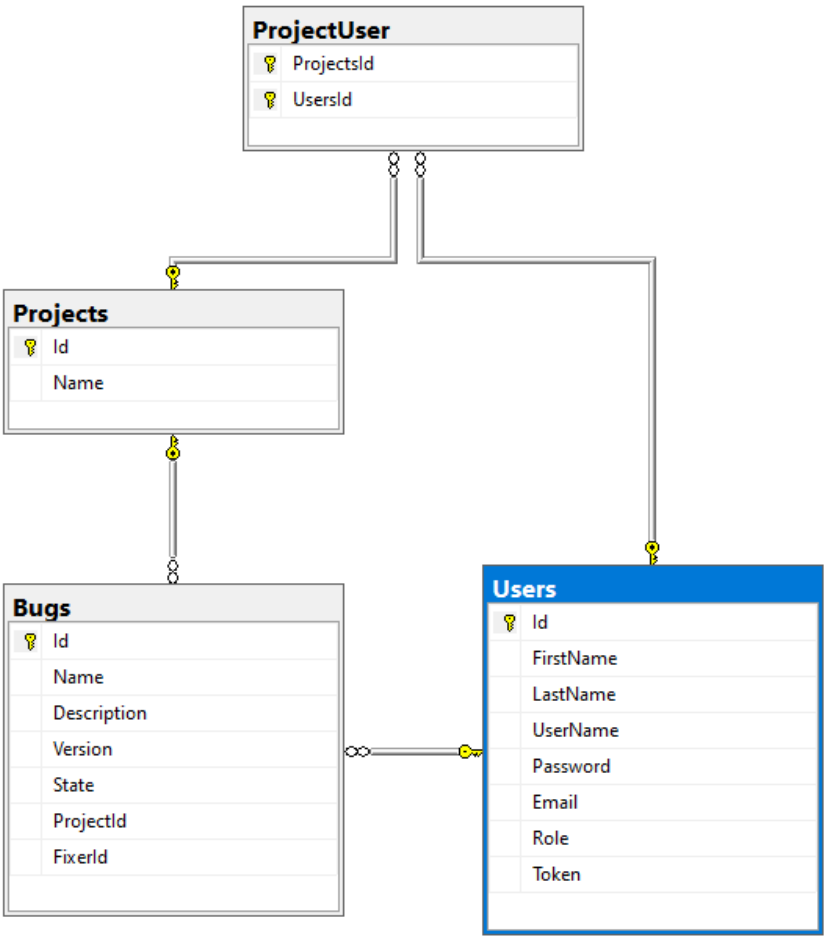
Este paquete contiene clases que utilizan distintas capas de la solución. Los comparers son para comparar dos objetos del dominio, o dos listas de los mismos.

A la hora de hacer los tests, nos encontramos con algunos problemas, entre ellos estaba la forma de comparar objetos individuales para saber si todas las propiedades de los mismos, una solución pudo haber sido crear comparadores de objetos, pero como a diferencia de cuando queríamos testear una lista de objetos, que interesa testear algunos campos únicamente (para lo cual utilizamos comparators). El problema de crear comparadores de objetos para todas las propiedades es que a medida que avanzamos con el proyecto podrán aparecer nuevas propiedades y eso implicaría tener que actualizar estos comparadores constantemente. Es por esto que decidimos utilizar CompareLogic, que nos permite resolver este problema, recibiendo dos objetos cualquiera y comparando todas sus propiedades.

Sin embargo utilizamos una combinación de ambos para casos relevantes.

El paquete también contiene una clase BugSearchCriteria que se utiliza para el filtrado de los bugs según distintas propiedades.

Estructura de base de datos:



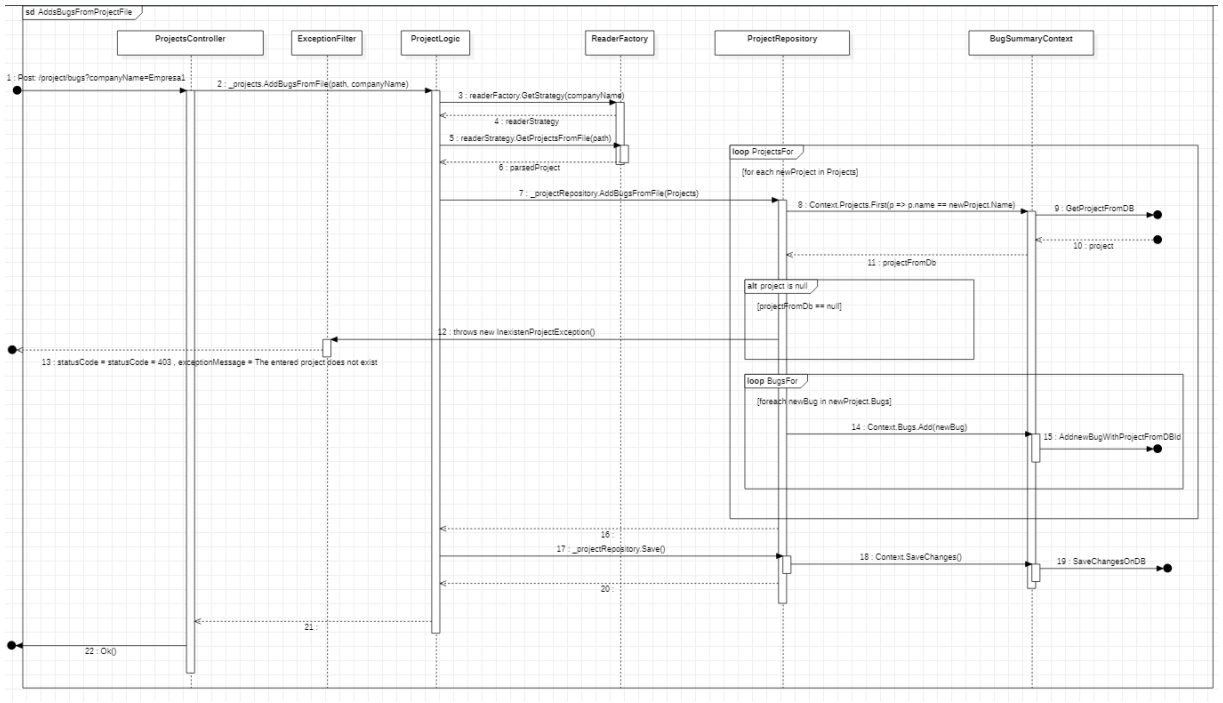
El modelo de tablas de la base de datos fue creado con el método code-first, creando primero las clases de dominio y a medida que actualizamos las mismas, creando migrations para actualizar el modelo. En total realizamos 5 migrations.

En Utilities, se añadieron clases que utilizamos como soporte para funcionalidades de otras clases. Como por ejemplo Validator que valida los largos generales para distintas properties, también se encuentra PasswordReportByColor, la cual es fundamental para el reporte de fortaleza de contraseñas. Otro ejemplo es la encriptación y hash que son las encargadas de encriptar las password del usuario y hashear la master password respectivamente. Esto es para que la aplicación tenga un nivel de seguridad mayor.

Diagramas de Interacción

A continuación se mostraran diagramas de interacción a modo de ejemplo del funcionamiento de la WebApi de algunas de las funcionalidades más importantes. Se decidió no realizar los mismos con el máximo detalle, priorizando la legibilidad y un entendimiento más simple. Por ejemplo, no se agregaron las interacciones con filtros de autenticación o file handlers para reducir la complejidad.

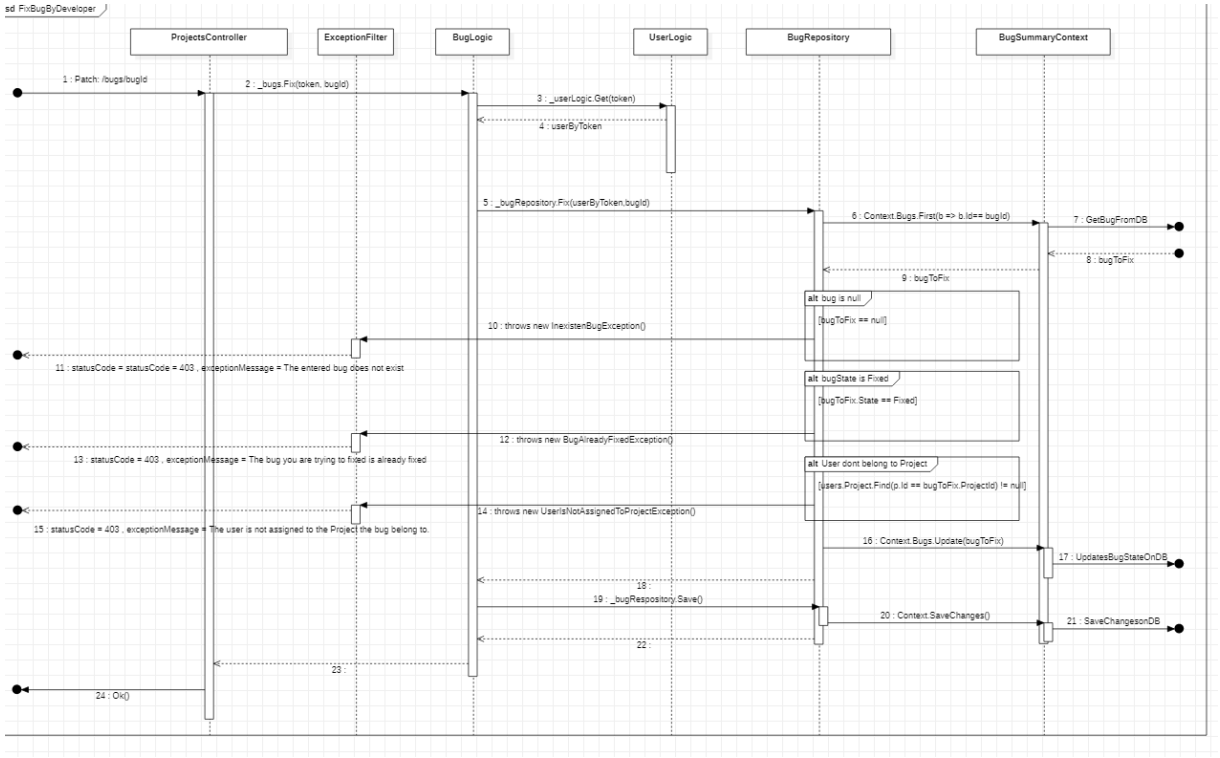
En este diagrama, se buscó reflejar las clases involucradas en la acción de cargar un archivo de datos con bugs, ya sea por medio de un archivo txt, o un XML. La acción comienza en WebApi y termina en la base de datos, guardando los datos breacheados.



7

En el siguiente diagrama se quiere reflejar la acción de agregar una hacer un fix de un bug, en esta acción participa un usuario developer, que envía su token y el id del bug que fixeó a la WebApi. También se muestran los casos de error en los que la acción no puede ser realizada por algún motivo.

[7] Diagrama encontrado en Documentation/Diagrams/ADDSBUGSFROMPROJECTFILE.SVG



8

Este diagrama representa las interacciones entre los diferentes componentes para arreglar un bug, osea cambias su estado a Fixed.

Se omitió el tema de filtros de autorización de roles, asumiendo que el usuario es un developer, para poder realizar la acción.

[8] Diagrama encontrado en Documentation/Diagrams/FIXBUG.SVG

Justificación de diseño

Algunas de las decisiones de diseño ya fueron comentadas anteriormente, sin embargo queríamos resaltar otros puntos relevantes.

Roles

Para el manejo de roles decidimos implementar un enum, de esta manera nos aseguramos de que en caso de crearse más roles, es sistema sea extensible, solo necesitando agregar un campo más al enum. Otra alternativa que contemplamos era crear una herencia de user, donde los hijos definan sus roles, es decir 4 clases, una para developer, tester, admin y otra para user, lo cual implicaría que cada vez que se quiera agregar un nuevo rol, se deba agregar un user nuevo al dominio y por ende actualizar también la estructura de la base de datos. Esto genera una inestabilidad mucho mayor tanto en el dominio como en la base de datos, por lo cual decidimos hacer un enum, que nos permite resolver este posible problema a futuro sin complicaciones.

Principio SRP

Para almacenar el token de sesión de un usuario, se decidió crear un campo en la clase usuario y de esta forma actualizarlo cuando el usuario inicie sesión. Si bien hacer que el usuario sea el responsable de guardar su token podría verse como una violacion de SRP, creemos que el usuario debe ser el experto en información, contiene toda la información necesaria para poder realizar el inicio de sesión. Otra razón por la se tomó esta decisión, fue que el único requerimiento sobre el manejo de sesión era poder hacer un login, y crear una nueva clase únicamente conteniendo el usuario y su token, implicando que se guarde en una tabla diferente nos pareció complejizar demasiado el problema.

Principio DIP

En todas las capas del proyecto se intentó seguir el principio DIP, para poder obtener una solución con bajo acoplamiento entre clases concretas. Además de eso implementamos inyección de dependencias, lo que nos permite delegar la responsabilidad de crear y eliminar las instancias de las interfaces al Framework. Otro beneficio de esta decisión de diseño, fue la facilidad que brindó a la hora de hacer el unit testing, ya que para hacer un mock de una clase de la cual dependemos, únicamente necesitamos instanciar la clase que testeamos con la dependencia mockeada.

A contrario de como veniamos nombrando los proyectos, creamos TestUtils con Test al principio para diferenciar que no es un test, sino un proyecto con funciones de utilidades para usar en los test.

Exception testing

Como ya mencionamos anteriormente hicimos un manejo de excepciones personalizadas.
Un problema que tuvimos fue el testeo de excepciones custom, ya que las herramientas proporcionadas por MSTest no nos permitían validar los mensajes, decidimos crear una clase para poder hacer esto en test utils, la cual usamos en los tests unitarios para resolver este problema.

Listas de user en proyectos

Para el manejo de usuarios asignados a un proyecto se decidió crear una lista única de usuarios en proyecto, donde puedan agregarse usuarios de tipo tester o developer. Se consideró que manejar una lista única de usuarios iba a ser suficiente, ya que se tiene un manejo estricto de los roles a la hora de modificar esta lista. Además al manejar una sola lista de usuarios, el esquema de la base de datos es considerablemente mas simple y nos permite extender los roles, ya que si hubiésemos elegido manejar diferentes listas por cada rol, a la hora de agregar un rol al sistema, se tendría que modificar la clase proyecto y el esquema de datos.

Validaciones

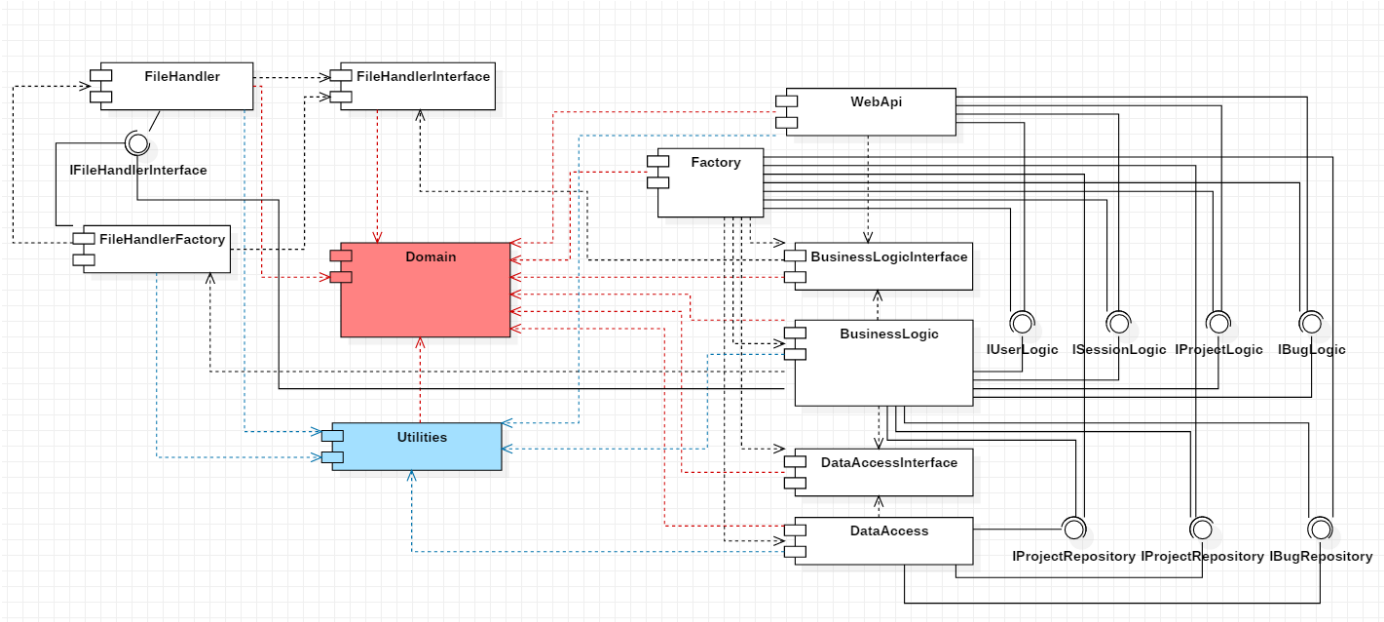
Para las validaciones de los objetos de dominio, se decidió crear excepciones custom y manejarlas a nivel de dominio, ya que los requerimientos de las mismas permiten hacerlo de esta forma sin comprometer la estabilidad del paquete. Si por ejemplo se hubiese querido validar más campos de la contraseña (en esta instancia solo se necesita verificar que la misma no sea nula) como por ejemplo chequear el largo o que contenga cierta cantidad de caracteres, hubiésemos implementado la validación en business logic en lugar de el dominio ya que los requisitos para una contraseña segura cambian constantemente y esto implicaría reducir la estabilidad de nuestro dominio.

Estándares de codificación

Al inicio del proyecto entendimos que era necesario establecer en grupo estándares de codificación para poder llegar a una solución clara y con buena calidad de código.
Algunos estándares establecidos fueron:

- El código y commits se escribe en inglés
- La interfaz gráfica se hace en inglés
- El nombre del atributo privado de las clases es _nombreAtributo
- No utilizar el "this." para referirse a atributos al menos que sea:
    - Para evitar errores de compilación
    - Para ayudar a la comprensión del código
    - Para referirnos a una property adentro de una misma clase
- Los nombres de los métodos comienzan en minúscula.
- Variables con un formato en camelCase.
-

Diagrama de composición



El diagrama de componentes muestra todos los componentes que comprenden al sistema. Se utilizaron colores para identificar las dependencias con los componentes más estables, el de dominio y el de utilities. Luego cada componente que interactúa con una interfaz se representa esta relación, si se conecta directamente con el círculo significa que implementa los métodos de dicha interfaz, sino que solamente la consuma desacoplando el componente que la

implementa. La Factory depende de las interfaces de BusinessLogicInterface para poder hacer que la inyección de dependencia funcione correctamente.