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, agregando el uso del framework Angular para desarrollar el frontend..

Se diseñó una solución, simple, que cumpla los requerimientos funcionales solicitados agregando algunos extras que consideramos importantes y que contenga componentes únicos para soportar futuros cambios pudiendo realizar mantenimiento si es necesario a 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:

- 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.
- Las ramas que apuntan a cambiar en el frontend deben indicarlo en el nombre, por ejemplo “feature-pageNotFoundFrontend”.

Continuamos trabajando con la herramienta Notion como lo hicimos en el primer obligatorio de DA2. 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.

Para el frontend, nos involucramos con el framework Angular. Se investigó bastantes funcionalidades por nuestra cuenta para poder obtener una interfaz de usuario, simple y a la vez moderna. Nos basamos en una template de Creative-Tim que mencionamos en el Anexo.

Errores conocidos o mejoras en la solución

En nuestra solución tuvimos un problema con respecto a la configuración de startup en WebApi. Luego de investigación sobre los permisos del navegador para acceder desde diferentes puertos, relacionado a el Intercambio de Recursos de Origen Cruzado (CORS (en-US)), descubrimos que debíamos agregar permisos a las controladoras correspondientes. El problema es que tuvimos que comentar una línea de configuración sobre HTTPS (app.UseHttpsRedirection()) que no nos permite que el Swagger se active en una ruta segura del protocolo HTTP, y nos permite funcionar la WebApi con el framework de Angular.

Tenemos otro problema con respecto a los DTOs o modelos de las entidades del sistema. Utilizamos únicamente un modelo tanto para enviar, como para recibir información por clase del dominio. Esto nos trajo problemas a la hora de crear un objeto desde el frontend, ya que todos estos objetos cuentan con un id para poderlos identificar concretamente, el cúal es un número autogenerado por Entity Framework y la base de datos para mantener un orden y evitar repetidos.
Dado esta situación, debemos completar todos los campos del modelo, tanto para recibir como para enviar información entre la solución del frontend y la de backend, pero al crear un bug en la web por ejemplo, no queremos setear el id manualmente, sino que EF se encargue. En nuestro caso, antes de usar los servicios correspondientes para hacer los pedidos HTTP, seteamos un id random que luego se descarta en el backend.
Una solución que pensamos fue crear un modelo para enviar y otro para traer información entre la WebApi y el backend, no lo realizamos de esta manera porque priorizamos otros aspectos del obligatorio y no creemos que fuera de tanta prioridad dado que la funcionalidad la consideramos de mayor importancia.
El último error que detectamos fue que los acciones muestran un mensaje de error o de éxito cuando se realiza la http request y se obtiene una respuesta suscribiéndote al pedido de acción. El problema es que no limpiamos los mensajes, luego de un tiempo o dar la opción de limpiarlo con un botón, es algo que consideramos podríamos mejorar en el futuro para una mejor experiencia de usuario.

Diseño

Backend

Se comenzó a diseñar la solución final, comenzando 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.

Frontend

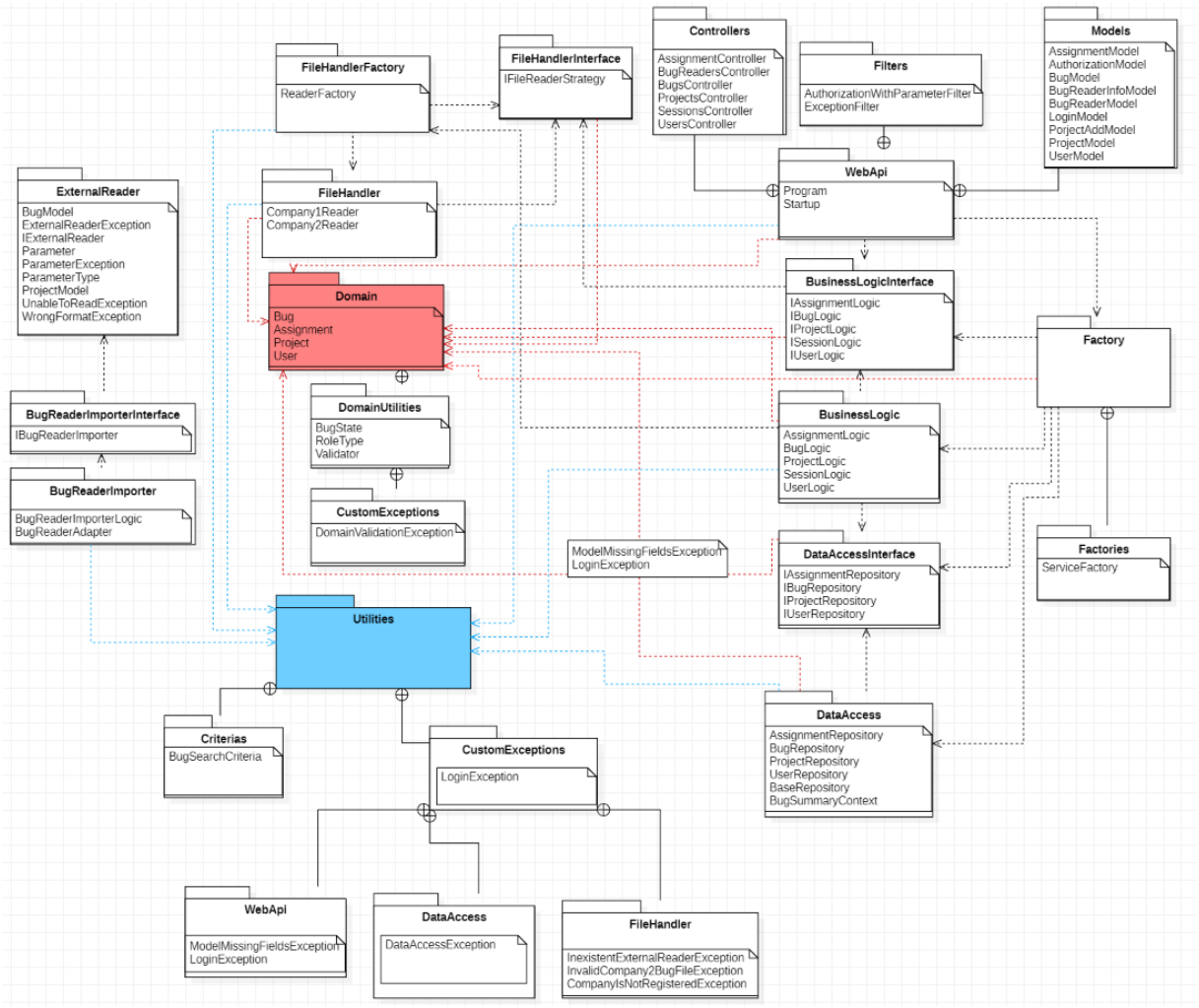
Nuestro proyecto de frontend se basa básicamente en dos grandes áreas o componentes. layouts y pages. Los layouts fueron creados para distinguir ciertos componentes o funcionalidades por cada tipo de rol (admin-layout, developer-layout, tester-layout, auth-layout) agregando uno extra para el inicio de sesión en el sistema.
Luego las pages, son las páginas que se visualizan concretamente en el navegador, donde algunos incluyen otros componentes anidados, como también redirecciones a otras pages. Algunos ejemplos de estos son projects, bugs, bugs-editor que es el componente que permite solo a los administradores y tester editar un bug en particular.

Se intentó disminuir al máximo el preguntar constantemente el rol del usuario que ingresó al sistema para habilitar funcionalidades. Para esto en un componente en común como es el sidebar, se le asignan permisos por rol a las páginas que se muestran en el sidebar y cuando el usuario ingresa las credenciales en el login, solo puede ver a las páginas según su rol. Se intentó reutilizar lo más posibles los componentes como tablas y formularios, para que los distintos roles puedan utilizar el componente, sin tener que copiar y pegar el html y sus métodos en los archivos de typescript. Por ejemplo en un principio se había diseñado un componente de proyecto orientado únicamente a un usuario con rol de administrador. Se decidió extender esta funcionalidad para poder que todos los usuarios del sistema puedan ver los proyectos a los que pertenecen, y se modificó el componente original con botones específicos para el admin, así no se repetía código copiando lo que si nos servía para otros usuario. Algo muy similar se implementó para la tabla de bugs, donde los usuarios pueden ver los bugs asignados a sus propios proyectos o ver todos si este es admin, pero pueden realizar diferentes acciones sobre estos (actualizarlo o arreglarlo) apretando un único botón dependiendo su rol que el routing maneja.

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 ellos no dependen de casi ningún otro. Se hará un análisis mas detallado de esto en la sección de métricas.

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

[1] Diagrama encontrado en Documentation/Release2/Diagrams/Package.svg

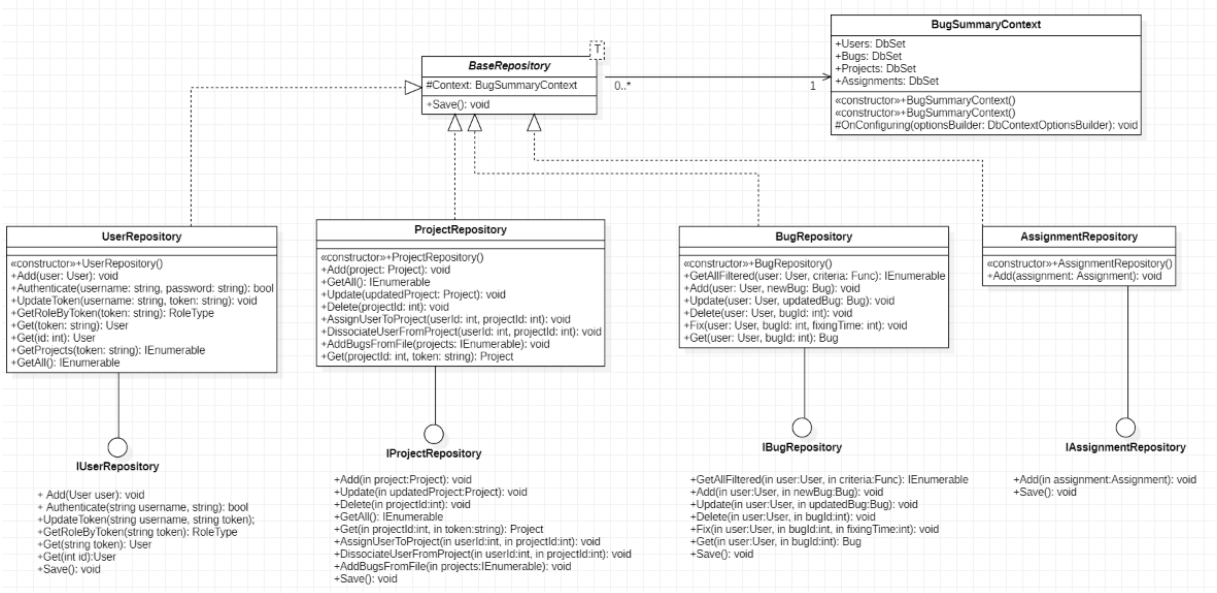
crear una capa de acceso con la base de datos (BD). En esta última, está el contexto de la aplicación y la interacción con la BD, creando allí las tablas y restricciones necesarias. También están las operaciones de búsqueda (gets) y modificación de datos (update), que finalmente se terminó usando en 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.

En esta segunda entrega se agregaron tres nuevos paquetes para el manejo de importadores con Reflection, el más importante siendo ExternalReader, que actúa como contrato para las empresas, ofreciendo una interfaz con métodos a implementar, DTOs y excepciones custom. Entraremos más en detalle sobre esto en la justificación de diseño. También se agregaron las clases en cada paquete para el diagrama de paquetes para un mejor entendimiento de la estructura de nuestra solución.

Diagramas de clases:

Paquete DataAccess y DataAccessInterface



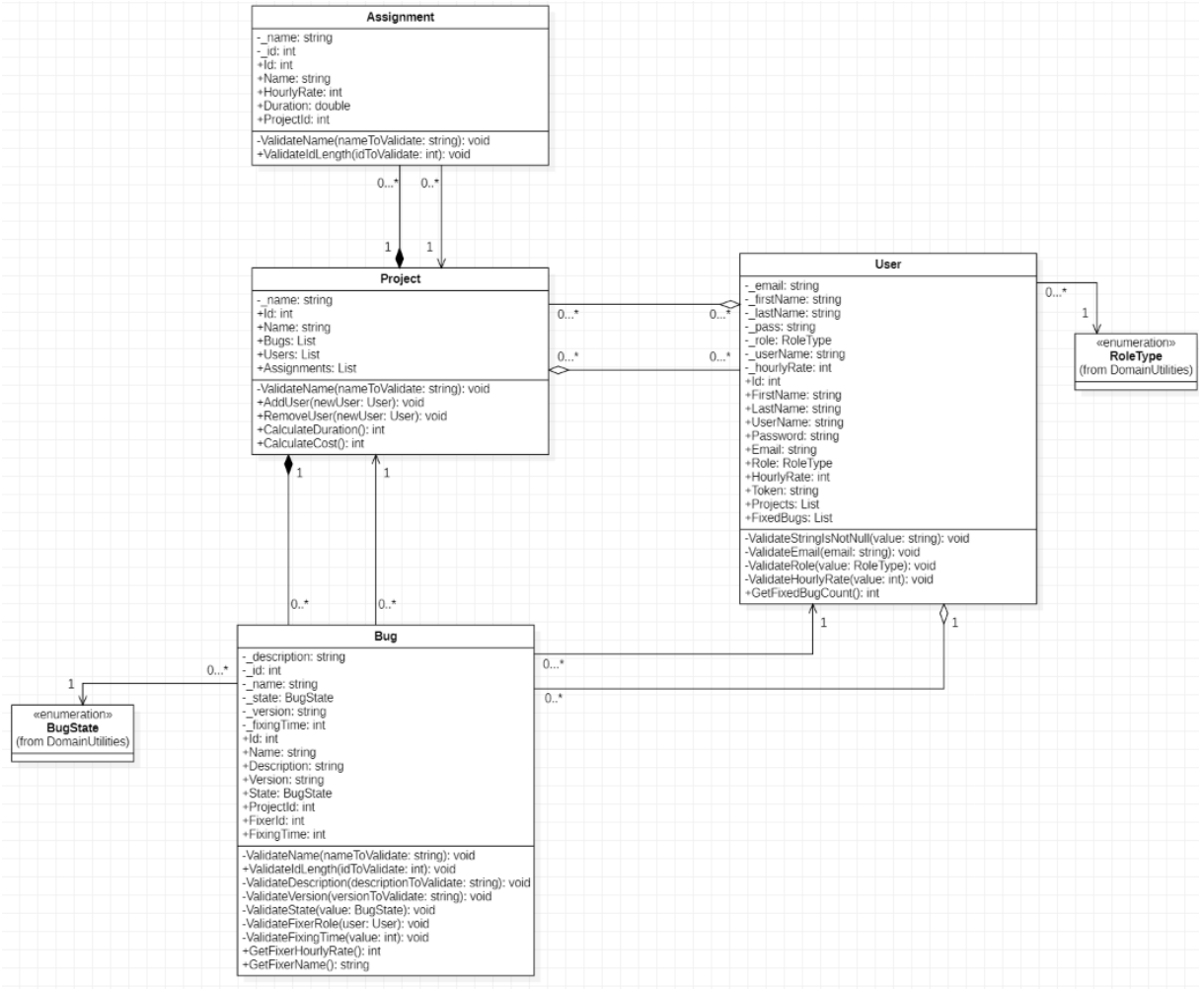
2

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.

[2] Diagrama encontrado en Documentation/Release2/Diagrams/DataAccess.svg

Para esta segunda entrega, se agregó un repository que es el encargado de manejar los datos de los Assignments pertenecientes a un proyecto. Este cambio, nos influyó en cambiar varios paquetes, agregando interfaces y clases para brindar funcionalidad sobre este objeto.

Paquete Domain



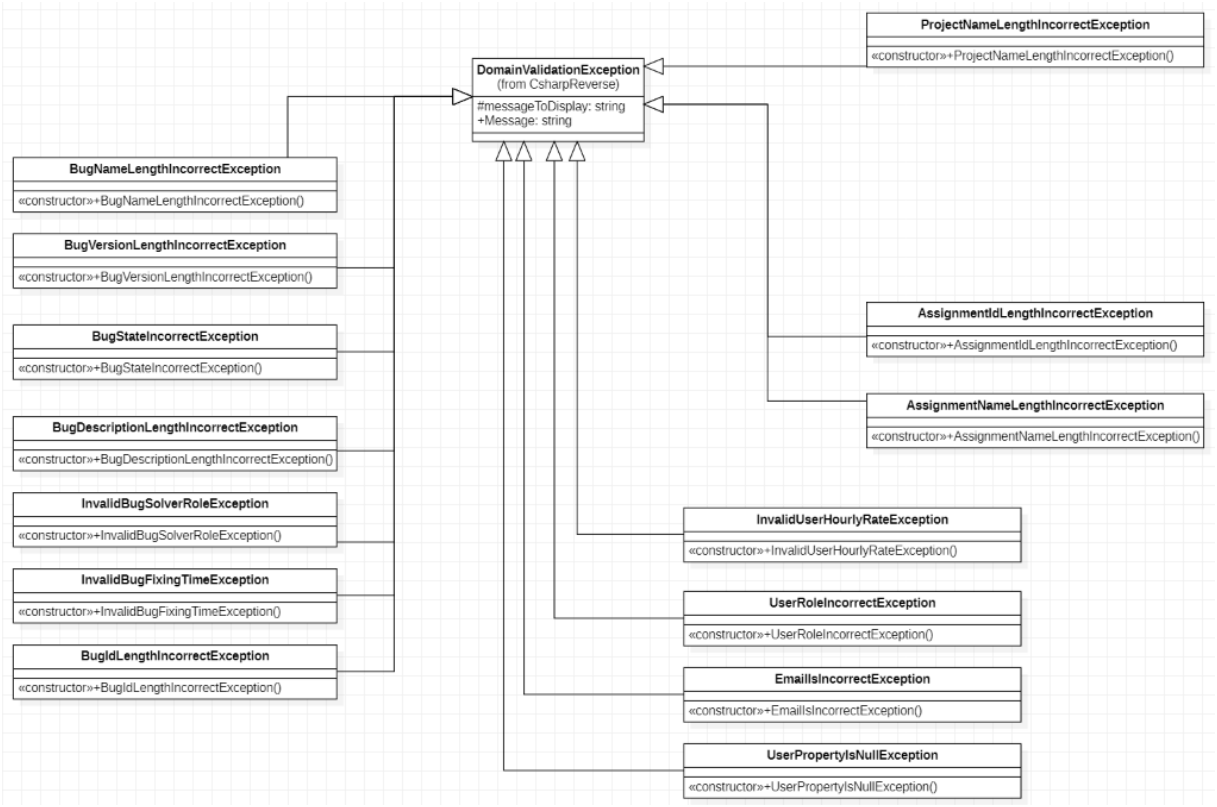
3

Este paquete tiene la responsabilidad de contener los objetos del sistema, Bug, User y Project. Ahora también se agregó un objeto Assignment, lo cuál trajo varios cambios necesarios dado que es el paquete más estable en nuestra solución, al cambiarlo necesitamos modificar varios puntos del sistema.

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:

[3] Diagrama encontrado en Documentation/Release2/Diagrams/Domain.svg

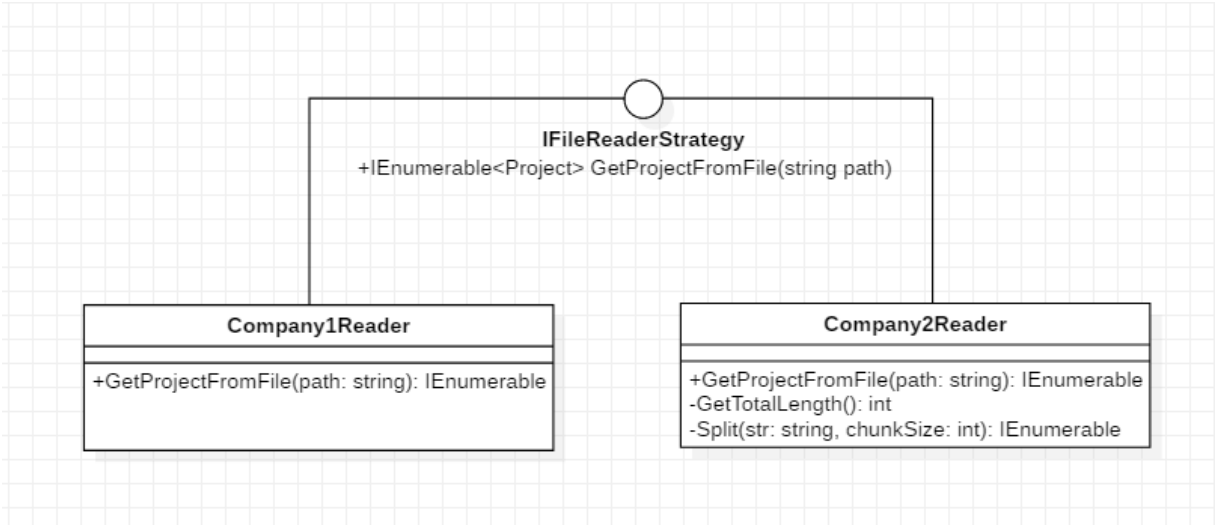


4

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. Al contar con un paquete especializado para agrupar las excepciones de dominio, nos perjudica alguna métrica en particular, pero al agregar nuevas excepciones como las de assignments, ganamos en solo tener que agregar una excepción particular y no tener que modificar ningún punto del sistema.

[4] Diagrama encontrado en Documentation//Release2/Diagrams/DomainExceptions.svg

Paquete FileHandler

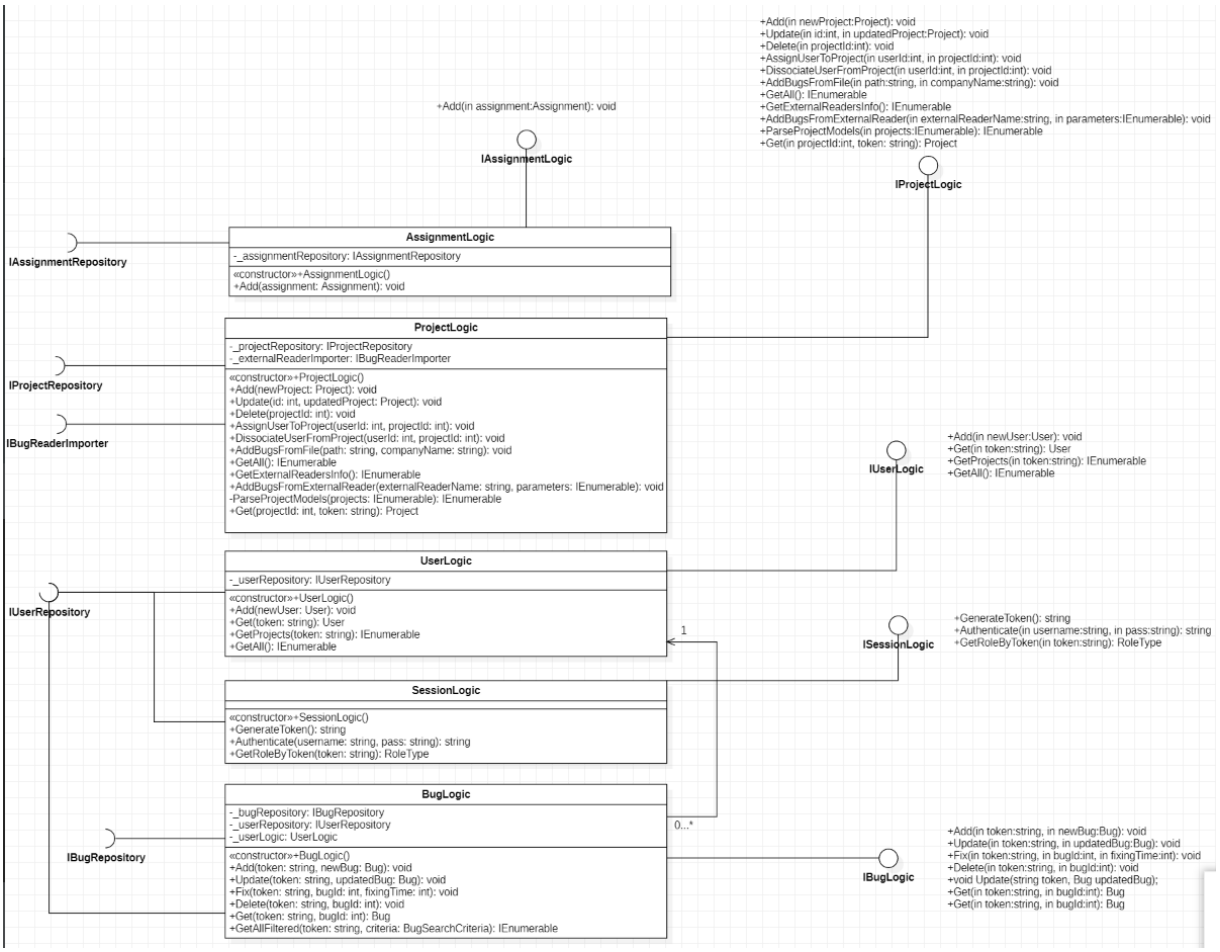


[5]

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.

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

Paquete BusinessLogic y BusinessLogicInterfaces

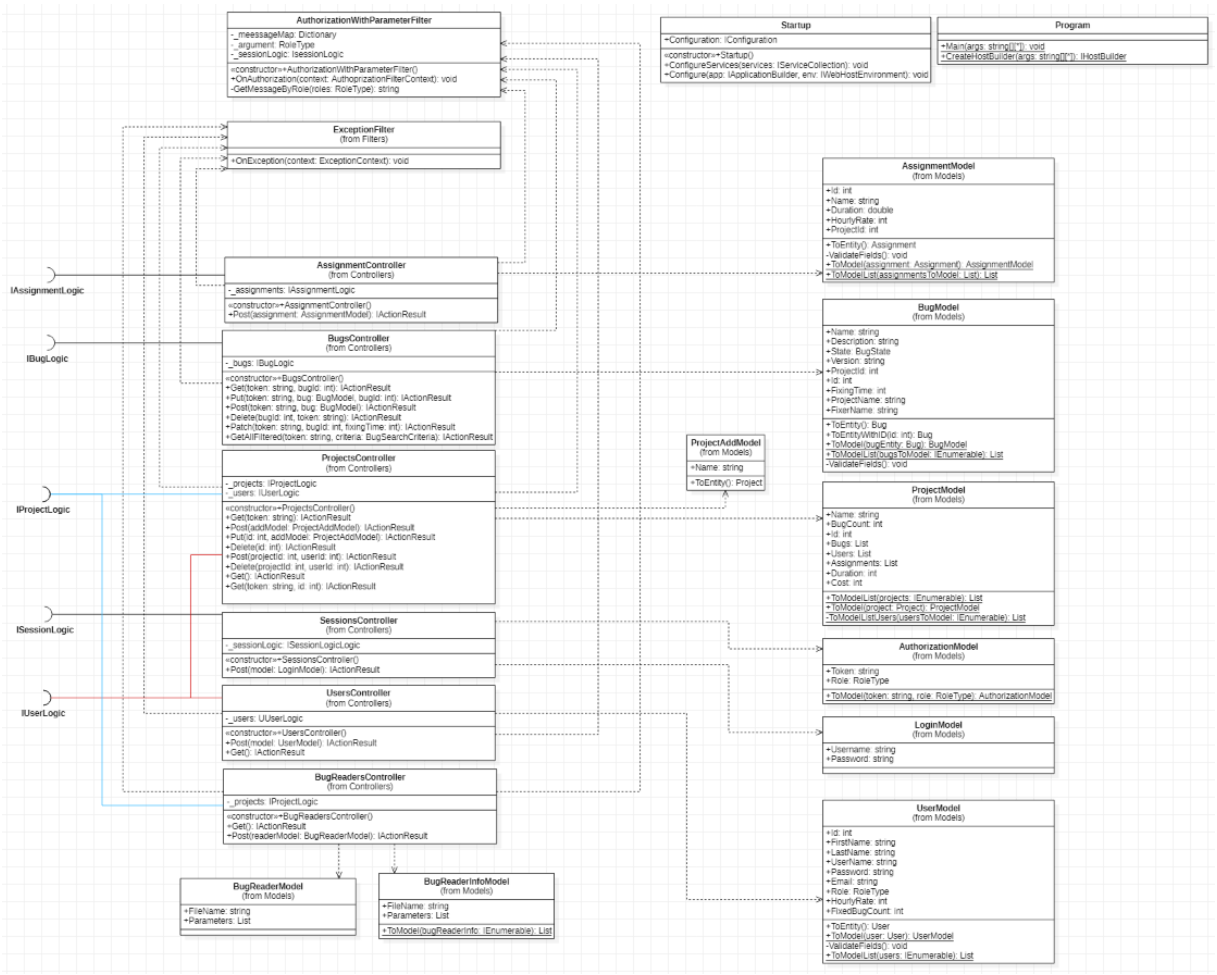


6

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, utilizando las interfaces correspondientes para la comunicación entre capas. Para el manejo de assignments se creó una nueva clase encargada de la BL de assignments.

[6] Diagrama encontrado en Documentation/Release2/Diagrams/BusinessLogic.svg

Paquete WebApi



7

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 frontend para mapearlos a una entidad concreta del dominio. También se realiza la operación inversa de convertir objetos concretos a modelos para enviarlos en una respuesta HTTP, por ejemplo un Get de bugs. 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.

Para esta nueva entrega, se agregó una controladora especializada para crear o realizar un Post sobre assignments, no se creó un Get específico en este controller, dado que para obtener un assignment siempre debía en conjunto con un proyecto, porque decidimos incluirlo en el modelo de proyecto en vez de tener un endpoint especializado.

Varios modelos fueron actualizados, por ejemplo AuthorizationModel, para cuando un usuario inicia sesión, se obtiene un token y también el rol para poder tener un mayor control del mismo. También se agregó el FixingTime a los BugModel para saber cuánto tiempo tomó

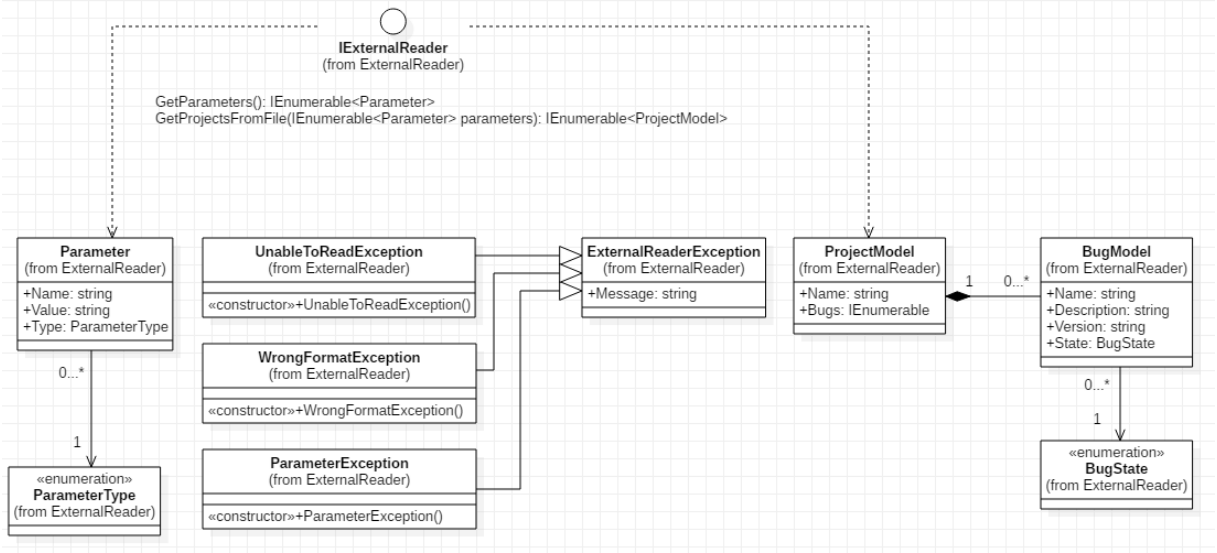
[7] Diagrama encontrado en Documentation/Release2/Diagrams/WebApi.svg

resolverlo en particular si este tiene estado Fixed, y un FixerName para indicar quién fue el encargado de resolver ese Bug y mostrarlo en el frontend.

Paquete Utilities

Este paquete se creó con el fin de agrupar clases que puedan ser utilizadas en todo el proyecto, como interfaces, criterios de búsqueda, etc. En esta entrega alguna de las clases que tenía antes fueron movidas TestUtilities, ya que antes no hacíamos tanta diferencia entre las mismas pero ahora se decidió separar las clases de utilidad para el dominio de la de los tests.
El diagrama de este paquete se puede encontrar en el anexo, mayoritariamente son excepciones con herencias.

Paquete ExternalReader

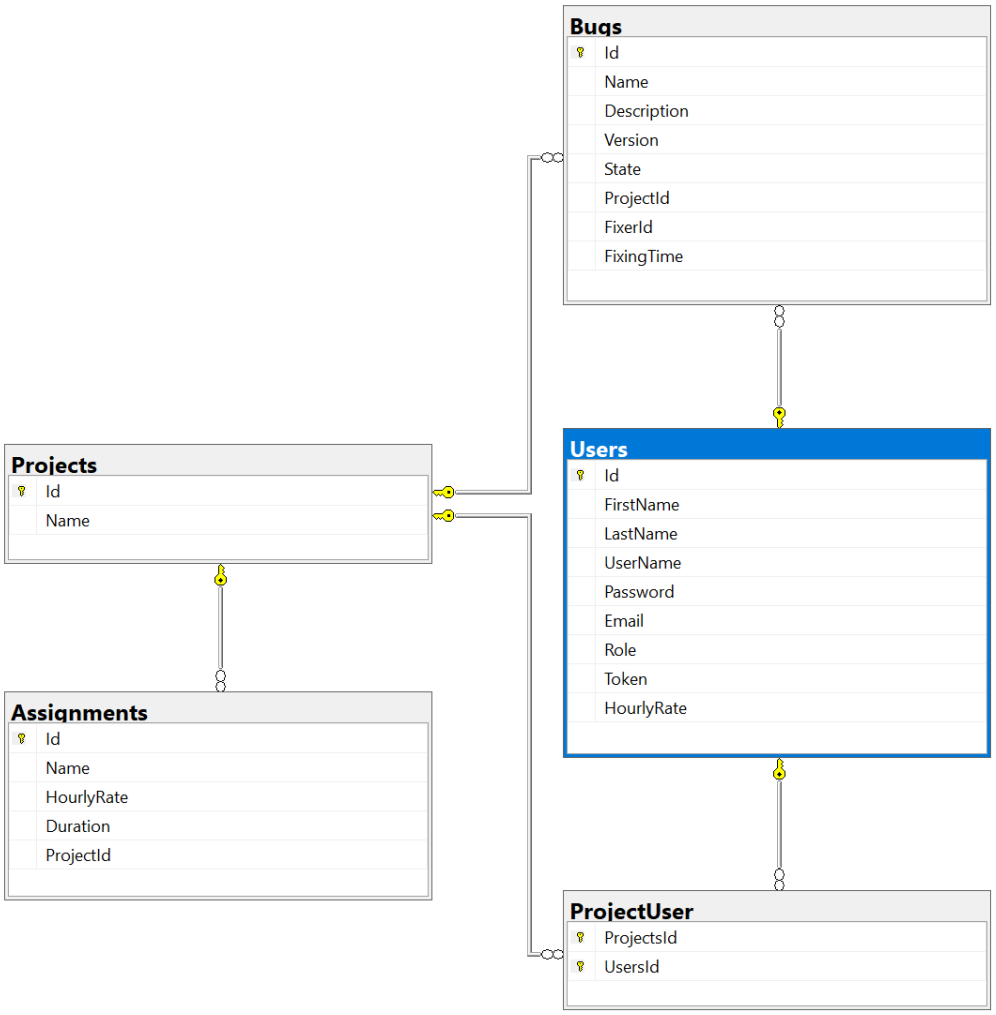


8

La responsabilidad de este paquete es servir como un contrato que podamos compilar y enviar a las empresas que quieran implementar su external reader, de esta forma nos aseguramos de que implementen los parámetros que solicitamos y usen los tipos de datos y excepciones que están dentro de la clase.
Para ver más detalle acerca del diseño y este paquete ver Justificación de diseño -> Extensibilidad de importadores.

[8] Diagrama encontrado en Documentation/Release2/Diagrams/ExternalReader.svg

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. Para esta segunda entrega, se agregó properties requeridas para clases ya existentes como el tiempo que llevó arreglar un bug (FixingTime) o el salario por hora de un usuario (HourlyRate). También se añadió una clase Assignments correspondientes a las tareas de un proyecto, las cuáles sólo un administrador puede crear.

La relación entre estas tablas, muestra nuestro diseño de solución para este obligatorio. Comenzando con proyectos, es una clase que diseñamos y consideramos como principal al tener además de properties de caracterización propia como el nombre y id, listas de: usuarios, assignments y bugs. El nombre lo utilizamos más frecuentemente como forma de identificación en el frontend, siendo un valor más user-friendly que un número autogenerado como lo es el id. Los bugs, contienen información propia de sus existencia, como su estado (Active o Fixed), características de identificación como las del proyecto y también existe una relación con Users dado que puede tener un FixerId, o dicho de otra forma, un Usuario que arregló un bug en particular siendo este un developer.

Los usuarios, abarcar todos los usuarios del sistema, sin importar sus roles. Esta decisión nos benefició en algunos aspectos y en otros no tanto, por ejemplo debemos preguntar el rol al acceder a los métodos en el DataAccess y realizar funcionalidades especializadas. También cuenta con una property Token la cuál es usada para autenticar al usuario y su rol y una lista de bugs para saber cuáles bugs arregló una persona sin tener que recorrer todos los proyectos buscando esta información..
Existe la posibilidad de que un usuario este en varios proyectos a la vez, a diferencia de los bugs. Esto genera una relación N a N entre proyectos y usuarios, generando esta tabla que se muestra como ProjectUser, la cuál guarda la información correspondiente para mejorar la performance de búsqueda por ejemplo.

Ciclo de vida del contexto

Nuestra solución utiliza un ServiceFactory.cs encargado de hacer la inyección de dependencias del proyecto en general. Para crear un contexto único y común a la ejecución del programa se crea un BugSummaryContext, que contiene los DbSet de Users, Bugs, Projects, y Assignments para poder modificar, agregar y obtener información de la base de datos. Desde el startup del paquete de WebApi, se inyecta esa dependencia mencionada, con una configuración (connectionString) para poder acceder a la base de datos correspondiente.
C# nos proporciona un garbage collector para poder disponer del mismo, una vez no se utilice más, sin tener que preocuparnos de realizar esta acción manualmente.

Justificación de diseño

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

Extensibilidad de importadores

Para este obligatorio se solicita que las empresas que utilizan el software sean capaces de agregar importadores de bugs (BugReaders) al sistema en tiempo de ejecución. Para esto se utilizó Reflection, permitiendo que se puedan agregar dll en una carpeta preestablecida y que el proyecto pueda hacer uso de los mismo en tiempo de ejecución.
Dicha carpeta se especifica en el archivo de app config de la WebAPI, pudiendo cambiarla a cualquier otra sin tener que cambiar el código (en caso de que la hubiésemos hardcodeado), lo que evita también tener que recompilar la solución.

Para poder leer y utilizar los assemblies de las empresas, es necesario que ambas partes conozcan las clases necesarias para poder guardar los bugs/proyectos, que métodos implementar y cómo manejar los errores para que puedan ser entendidos por nuestra solución.

- Esto puede traer complicaciones, en el caso de las clases que deseemos compartir con las empresas, pensamos en usar las clases que ya teníamos creadas en nuestro dominio, pero rápidamente descartamos esta opción, ya que estaríamos exponiendo el mismo a todas las empresas que quieran crear su propio importador de bugs. Como solo nos interesa exponer 2 clases de dominio (Bug y Proyecto) y además solo necesitamos utilizar algunas de las properties con algunos métodos particulares, se decidió crear dos DTO que sirvan para transportar los datos entre las implementaciones de las empresas y nuestra solución.
- Para mostrar los métodos que se deben implementar, se creó una interfaz que hace uso de las clases antes mencionadas. Esta sirve como contrato mostrando la forma en la que solicitaremos y recibiremos los datos en cada método. También se decidió que cada implementación del importador de bugs debe tener únicamente una clase que se encargue de implementar la interfaz, es decir por cada ensamblado, debe haber una única clase que se encargue de comunicarse con nuestra solución, aunque pueden haber más clases auxiliares obviamente. Las implementaciones de los importadores se asumen que serán cargadas en la carpeta definida en el app config por un administrador del servidor (es decir no deben cargarse por el frontend) y se seleccionará un nombre de archivo (EJ: TataJSONReader), el cual será el que deba seleccionarse en el frontend para hacer uso de esta implementación.
- También se crearon tres tipos de excepciones (UnableToReadException, ParameterException, WrongFormatException) para poder manejar los errores y asegurarse de que nuestra solución pueda entenderlos y enviar una respuesta apropiada como respuesta HTTP para que pueda ser consumida y mostrada a el usuario que intenta importar los bugs.

- Por último, pero no menos importante, también se tomó en cuenta el caso en que un importador necesite más de un parámetro, ya que si bien con los importadores que venimos trabajando solo se necesita el path a el archivo a leer, posiblemente alguna empresa quiera leer archivos de una base de datos por ejemplo, lo que probablemente requiera de mas parametros como una IP, puerto, pass, etc. Como nuestra solución busca ser lo más extensible posible, decidimos crear una clase parameter, que nos permita enviar una lista de los mismos y de esta manera no limitar a las empresas a poder usar un parámetro únicamente.

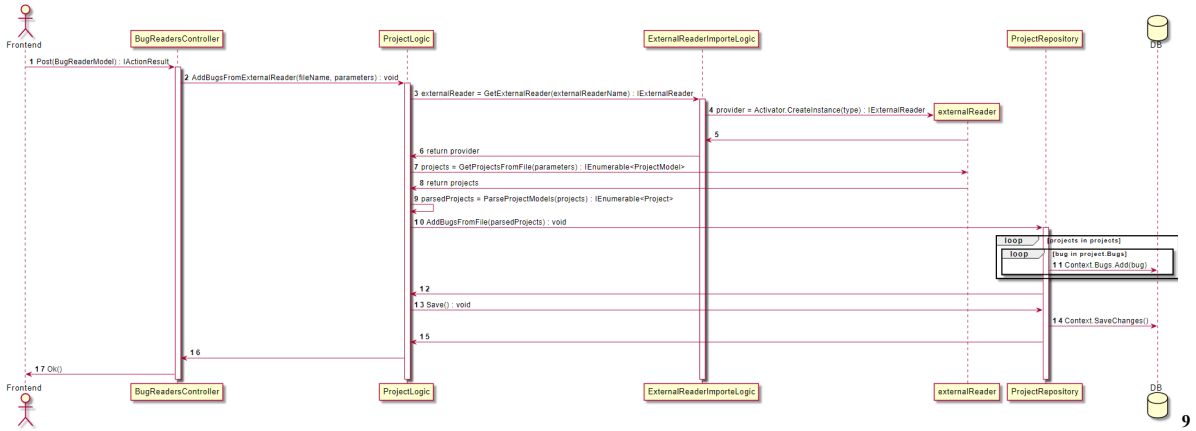
```
public class Parameter
{
    public string Name { get; set; }
    public string Value { get; set; }
    public ParameterType Type { get; set; }

}
```

```
public enum ParameterType
{
    String = 0,
    Int = 1,
    Date = 2
}
```

Además como se puede ver en la foto, se decidió permitir diferentes tipos de datos, como se puede ver en la clase ParameterType, para no restringir a las empresas (ya que podrían querer agregar bugs que fueron avistados a partir de cierta fecha por ejemplo) y favorecer la extensibilidad.

En nuestro repositorio se puede encontrar la implementación de un importador de bugs en formato JSON, este permite importar n bugs para un proyecto por archivo y se encuentra en la carpeta ExternalReaderImplementations, en el caso de que se usen más proyectos, si uno de los proyectos a los que se quiere cargar no pertenece al sistema se cancela la carga y se tira una excepcion.

A continuación se adjunta un diagrama de secuencia mostrando el flujo normal de agregar un bug por medio de un assembly de una empresa agregado con reflection, se buscó mostrar el funcionamiento de alto nivel, por lo cual no se entra en detalle acerca de la creación del assembly y el chequeo de que importe nuestra interfaz por ejemplo.



[9]

[9] Diagrama encontrado en Documentation/Release2/Diagrams/ImportBugsReflection.svg

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.

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 más 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.

Mejoras de diseño

Reflection

Para seguir con la estructura general del proyecto y la utilización de el principio DIP al modificar el proyecto para permitir leer assemblies y usar sus métodos, decidimos hacer aplicar el principio DIP, de esta manera en lugar de que ProjectLogic dependa del paquete encargado de leer los importadores (ExternalReaderImporter) que es un paquete de menor nivel, crear otro paquete con una interfaz para poder aplicar DIP e inyección de dependencias. Además habiendo modularizado el proyecto permitimos que en un futuro otra controladora haga uso de el bug reader en caso de ser necesario, si hubiésemos agregado la lógica de leer el assembly y usarlo dentro de ProjectLogic esto no sería posible.
Esto además facilitó realizar las pruebas, ya que no tuvimos que crear nuevos archivos para hacer una prueba de integración de la clase concreta, ya que nos parece innecesario testearlo directamente, habiendo creado la interfaz pudimos testear haciendo un mock de la misma.

Tarea

Para esta entrega, se solicitó añadir una entidad más al sistema una tarea que contenga un nombre, un costo por hora y una duración en horas. Se comenzó desarrollando esta funcionalidad como cualquier otra desde los DataAccessTest escalando hacia la capa de WebApi, exponiendo un endpoint de creación de tareas.
Un desafió que presentaba esta funcionalidad era estructurar la solución de tal manera que se permitiera extender la lógica con la que ya contábamos y poder calcular el costo de un proyecto, involucrando todas las tareas, con los costos de arreglar los bugs que necesita la información de valor por hora de un tester o desarrollador. Nos beneficiamos de nuestro diseño previo para poder calcular el costo de un proyecto, ya que contabamos con una relación bidireccional entre Proyectos y Bugs, por lo que podíamos acceder a los datos necesarios de las diferentes entidades para realizar un cálculo simple que decidimos implementar en el dominio del Proyecto. Esta implementación se realizó en esta clase ya que nos basamos en el principio de experto y la programación orientada a objetos, donde el objeto es el encargado de ser responsable de sus validaciones y de sus propiedades como objeto. Se realizó esto sin romper la ley de demeter, ya que se contaba con estas relación con Bugs y Assignments para calcular lo solicitado.

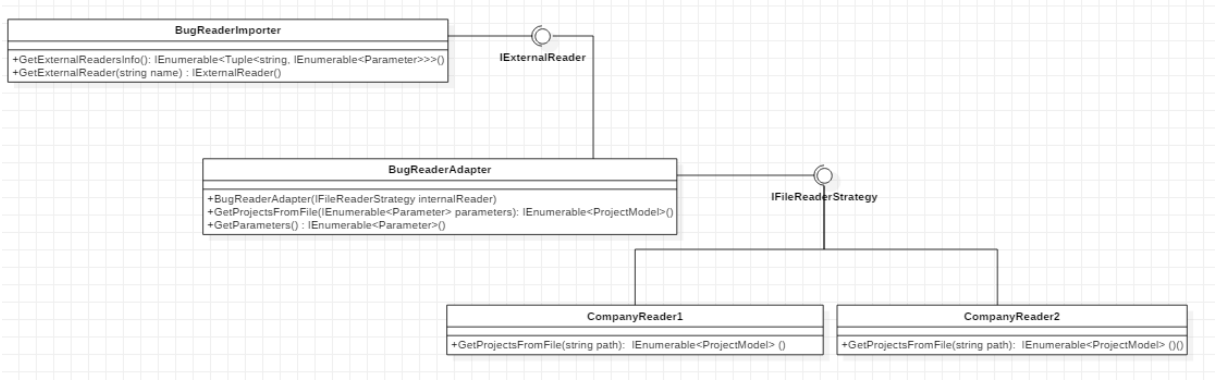
Refactor de endpoints

Tener que crear una interfaz gráfica para esta entrega, nos hizo ver los endpoints de otra manera, ya que no los usamos únicamente para hacer solicitudes de postman como en la entrega pasada, donde era más difícil imaginarse cómo podría usarse en una implementación de frontend y esto nos impedía ver claramente que tan bien teníamos separados los endpoints.

Una vez empezamos a utilizar los endpoints en la frontend nos dimos cuenta que la existencia de muchos de ellos no tenía sentido por sí sola, por lo cual decidimos mergearlos, como por ejemplo obtener la cantidad de bugs resueltos por un developer, que estaba separado del get normal de un usuario. Esto también nos ayudó en esta segunda entrega, mergeando el costo de un proyecto a el get de proyectos por ejemplo. Otra mejora que tuvimos que hacer a los endpoints fue creas un getter de proyecto por id, ya que antes devolvemos toda la información de los proyectos en el get all, algo que no tenía mucho sentido, pues si se agregan muchos proyectos, el overhead de traer todos junto con sus usuarios, tareas y bugs sería gigante para consultar algunos pocos en el mejor caso. Por esto decidimos hacer que el get all traiga la información básica necesaria para poder identificarlos en la lista y una vez se seleccione el proyecto se use el nuevo endpoint de get por id y obtener todos los datos que se necesiten.

Adapter

Dentro de las mejoras más importantes del obligatorio, está la implementación del patrón adapter, el mismo fue utilizado para resolver el problema de los importadores de bugs, ya que en un principio nos concentramos en la implementación de los importadores externos de la segunda entrega, una vez habíamos terminado la parte de importar bugs de importadores externos en el frontend, nos dimos cuenta que no tendría mucho sentido hacer una página separada para usar los importadores externos, ya que al cliente final le ocasiona confusión y podría no entender por qué se separan de esta forma. Por esto decidimos reutilizar la interfaz que ya teníamos, combinando los endpoints de importer externos e internos a la solución, creando un adapter que permite adaptar la interfaz de IExternalReader a la de IFileReaderStrategy.



Análisis de métricas

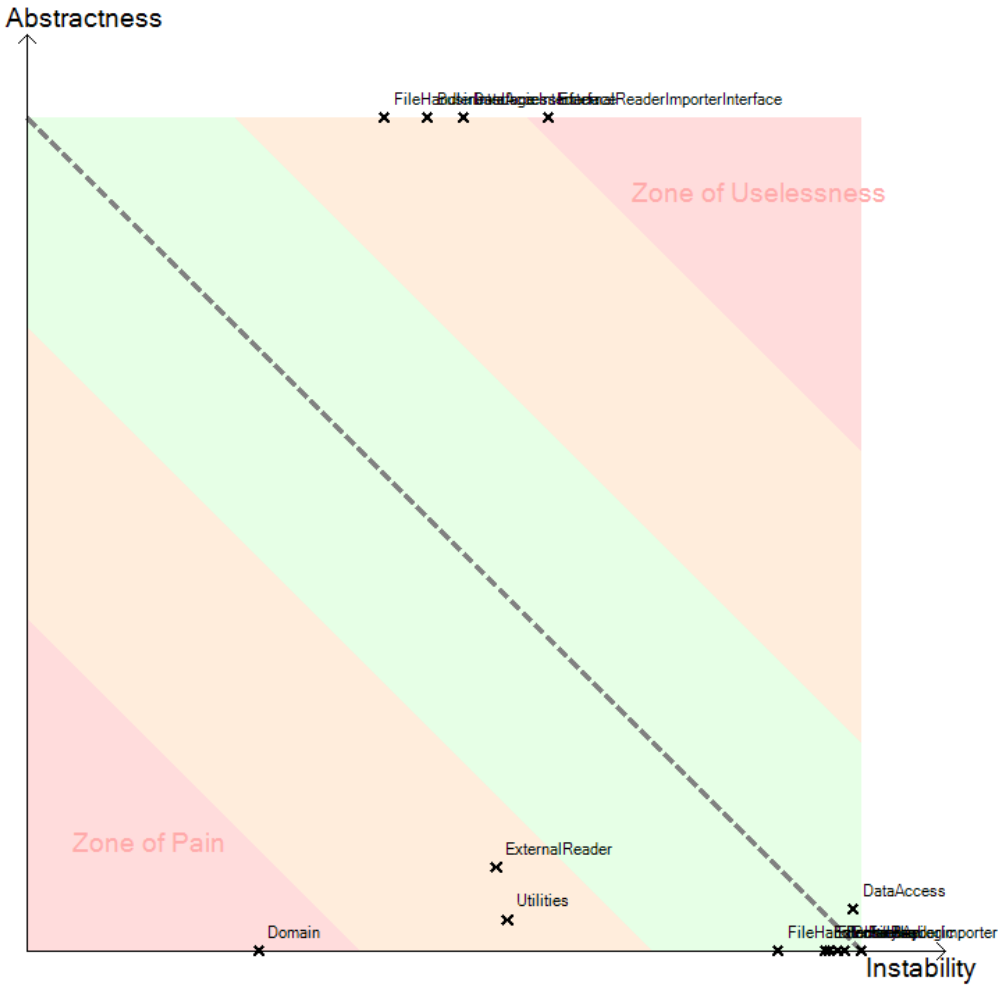
Para el análisis de métricas se usó NDepend, que facilitó el cálculo de las mismas ya que con solo importar la solución ya calcula todos los datos. Decidimos excluir los proyectos de test ya que pensamos que no aportan al análisis ya que no forman parte directamente del diseño sino que complementan la solución.

A continuación analizaremos las métricas vistas en clase:

Abstracción (A)

Para ver la abstracción se puede ver la gráfica de abstractness vs instability, a partir de la cual podemos ver que las clases más abstractas son las de Interfaces (ExternalReaderImporterInterface, DataAccessInterface, BusinessLogicInterface, FileHandlerInterface) teniendo todas una A = 1. Esto tiene sentido ya que estas clases son usadas únicamente para declarar interfaces para evitar utilizar el patrón DIP y evitar que las clases de alto nivel dependan de las implementaciones concretas de más bajo nivel.

El resto de los proyectos tienen una abstracción mucho menor, teniendo todos A = 0, excepto ExternalReader con A = 0.1, DataAccess con A = 0.05 y Utilities = 0.04.
Esto se debe a que estas clases son muy concretas (la mayoría no tiene o tienen pocas interfaces y/o clases concretas).
A partir de esta métrica por sí sola no podemos sacar demasiadas conclusiones, por lo cual vamos a evaluar la inestabilidad para luego poder hacer una comparación entre ellas y los principios con los que se relacionan



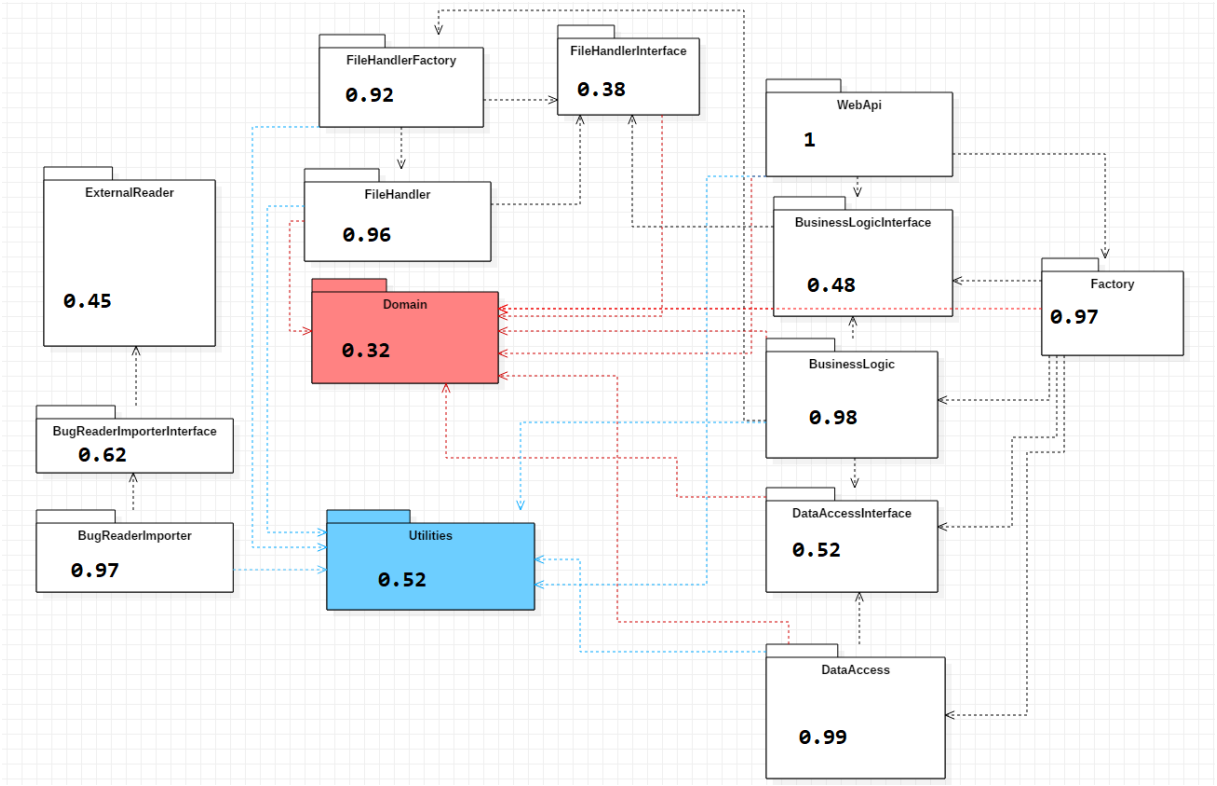
Inestabilidad (I)

En cuanto a la inestabilidad, como esperábamos, el proyecto más estable del sistema es Domain con un I = 0.28, esto se debe a que la mayoría de los paquetes dependen de él, pero solo tiene dependencias internas. Analizando ambas métricas (I = 0.28 y A = 0), podemos concluir que no se cumple el principio de abstracciones estables *"Los paquetes deben ser tan estables como abstractos son"* (Martin), ya que si bien la clase de dominio es estable como esperábamos, es totalmente concreta por ende si A = 0.
El problema que trae esto es que al ser un paquete estable, muchas clases dependen de él, pero al ser concreto puede cambiar sus clases, afectando a todas las clases que dependan de él (por esto se encuentra en la zona de dolor). Para resolver este problema se podrían crear clases abstractas para que los demás proyectos dependen de las mismas, reduciendo la frecuencia de los cambios y por ende los problemas que pueden generar los mismos en todas las clases que utilicen el dominio. El problema de hacer el dominio abstracto es que se agrega más complejidad al proyecto y al fin y al cabo lo que queremos representar con el mismo son objetos de la vida real, por lo cual tiene sentido que las clases sean concretas también, por lo cual es un tradeoff entre complejidad y el cumplimiento del principio de abstracciones estables.

Las demás clases que se encuentran con una inestabilidad cercana a I = 0.5 son las interfaces, ya que no dependen de muchos proyectos, tal vez de dominio o external reader, pero también dependen de él pocos proyectos (los que usan inversión de dependencias), por lo cual el acoplamiento aferente y eferente es similar y terminan aproximándose a I = 0.5.
La interfaz que hay que destacar es la de ExternalReaderImporterInterface, ya que es la que tiene más inestabilidad de los proyectos de interfaces (I = 0.62), ubicándolo en la zona de inutilidad. Esto se debe a que esta interfaz es utilizada pocas veces, teniendo un Ca = 3 y
Ce =5.
Para solucionar este problema se podría mover esta interfaz a otro paquete más dependido, el problema de esto es que ya no cumpliría su objetivo inicial, que fue utilizar DIP. Otra solución podría ser utilizar más la interfaz, pero al estar usando DIP, la estamos usando en todos los lugares donde se requiere usar la clase concreta, ya que dependemos de la interfaz, la cual a través de inversión de dependencias se carga en la clase controladora que la utiliza (no dependemos del proyecto de implementación concreta sino de la interfaz).
Otros proyectos que sin ser proyectos de interfaces tienen una inestabilidad cercana a I = 0.5 son el de Utilities y ExternalReader, esto se debe a que tienen aproximadamente la misma cantidad de Ca y Ce como pasa con los anteriores, además en el caso de ExternalReader, este proyecto se pensó para que sirva como un contrato para entregar compilado a las empresas para que puedan utilizar las interfaces y clases que les brindamos para reflection, por lo cual tiene sentido que no sea tan utilizado dentro de nuestra solución ya que su fin es el descrito anteriormente.

Las demás clases (FileHandler, WebApi, BusinessLogic, DataAccess, Factory, ExternalReaderImporter) cuentan con una inestabilidad cercana a I = 1. Esto tiene mucho

sentido ya que a través de la utilización del patrón DIP, logramos que los paquetes no dependan de éstos sino de sus interfaces, haciendo que su acoplamiento aferente sea de Ca = 1, ya que la única clase que depende de ellos es la Factory, para poder hacer la inyección de dependencias en la clase que use sus interfaces.
Estos paquetes cumplen el principio de abstracciones estables, ya que poseen un nivel de abstracción muy cercano a A = 0 y una inestabilidad de I = 1, por lo cual si debemos realizar cambios en alguno de esto paquetes, el posible daño ocasionado en otras clases será menor ya que al usar DIP, las clases dependen de las interfaces y no las clases concretas, que son las que se encuentran en esto paquetes.



Otro principio que podemos analizar a partir de la inestabilidad es el principio de dependencias estables (*"Las dependencias entre paquetes deben ir en el sentido de la estabilidad" (Martin)*), arriba se muestra una foto de los proyectos junto con su inestabilidad.
Como se puede apreciar en el diagrama, nuestra solución cumple el principio de dependencias estables en un 93 % aproximadamente. Únicamente dos de las dependencias no cumplen este principio, Factory 0.97 -> BusinessLogic 0.98 y Factory 0.97 -> BusinessLogic 0.99, con valores tan pequeños que podrían ser despreciables incluso.

Distancia Normalizada (D')

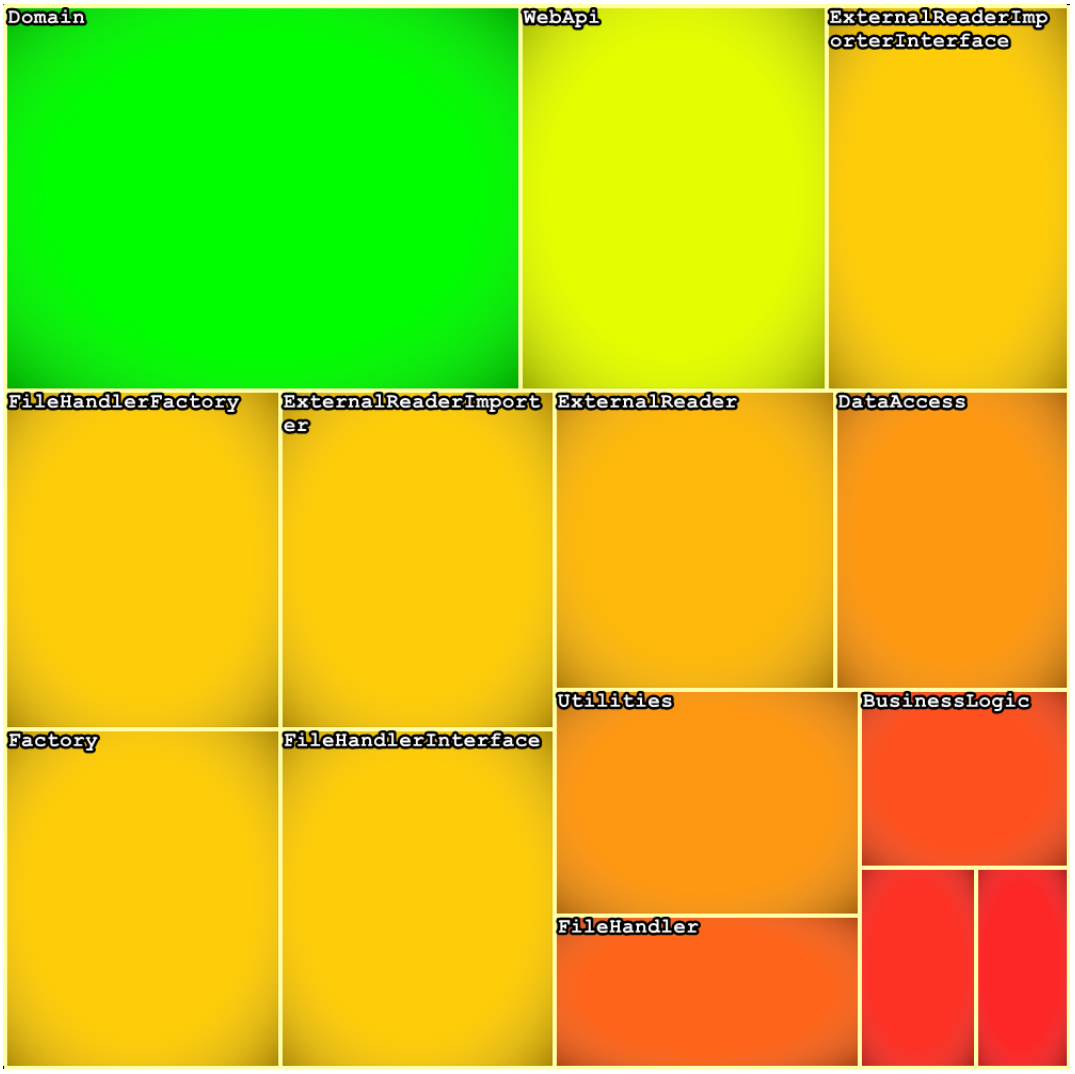
La distancia normalizada también puede observarse en la gráfica mostrada al inicio de esta sección, esta nos indica que tan buena es la relación es entre A e I, mientras más se acerque D' a 0, mejor será la relación entre A e I y de también nos asegurará que se cumpla mejor el

principio de abstracciones estables, ya que de la forma que se calcula D' (D' = |A + I -1|) mientras más cerca de 0 se encuentre el resultado mejor será la relación según Martin.

Es interesante ver en la gráfica las franjas, ya que también se puede ver como que tanto cumplen éste principio, siendo las clases que se encuentran en la franja verde las que lo cumplen, D' cercano a 0 y las demás con una peor relación entre las métricas A e I, que no cumplen el principio. Las peores en este sentido son las de Domain y ExternalReaderImporterInterface, confirmando lo que discutimos en la sección de inestabilidad.

Al ser una métrica que depende de A e I, ya habiendo discutido la relación entre ambas métricas para todos los proyectos, las posibles soluciones para mejorar D' van de la mano con las que planteamos en la sección de inestabilidad también.

Cohesión Relacional (H)



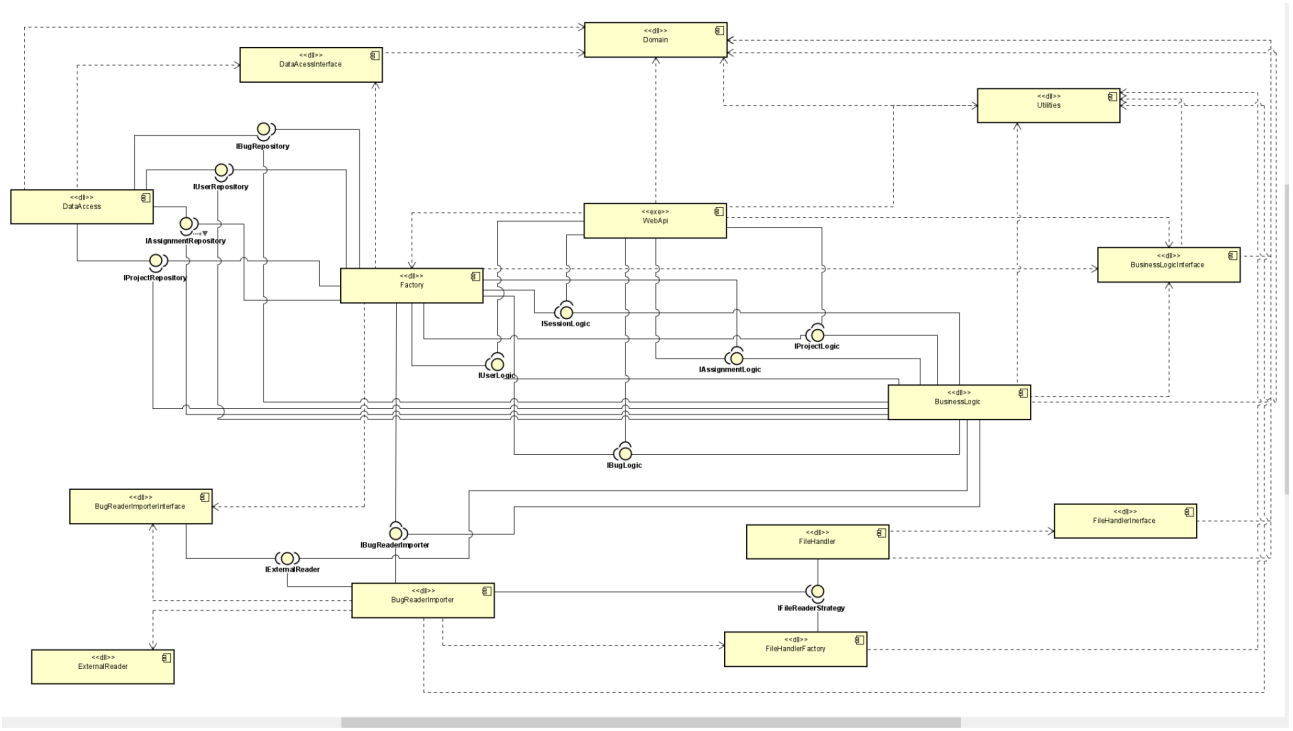
Para medir la cohesión relacional se utilizó NDepend también, como se puede ver en la foto, la clase con más cohesión es la de dominio como es de esperar ya que sus clases se relacionan

mucho entre sí, un bug tiene un proyecto, proyecto tiene lista de bugs, un usuario tiene bugs fixeados, etc. Además también decidimos poner las excepciones de dominio dentro del mismo proyecto, por lo cual todos los que usen excepciones generan más relaciones dentro del paquete, haciendo que este sea más cohesivo. Este es el único paquete con H entre 1.5 y 4. (H =2.13) Por lo cual tiene una buena cohesión.

Para el resto de las clases, no se obtuvo un cohesion muy buena, la que más se acerca a tener una buena cohesión es WebApi con H = 1.26, esto se debe a que las controllers no interactúan entre ellas, pero los modelos y filtros estan en el mismo proyecto y se relacionan con la mayoría de controladoras y entre sí, sino la cohesión sería mucho más baja.
Los proyectos en amarillo (hasta ExternalReaderImporterInterface) tiene H = 1, que es una cohesión baja, pero esto se debe a que todos estos proyectos tienen unicamente una clase, pues son Factories o interfaces para proyectos que solo necesitan una interfaz, por lo cual, si bien la metrica nos da un resultado negativo, es esperable porque el objetivo de estos proyectos implica obtener este resultado negativo. Si se movieran las interfaces a la clase que las implementa no estaríamos cumpliendo el patrón DIP por el cual las creamos y su existencia perdería sentido.
Algo similar ocurre con los proyectos con menos cohesión, los que no se ve el nombre, que son DataAccessInterface (H = 0.25) y BusinessLogicInterface (H = 0.2), donde tenemos aún peor cohesión, debido a que son proyectos de interfaces igual que los anteriores pero tienen más interfaces la cohesión es menor (obviamente no hay relaciones). Pero al ser una decisión de diseño usar el patrón DIP, es esperable también la cohesión baja y no tendría sentido mover las interfases a otro proyecto.

Los proyectos restantes (DataAccess H = 0.75, FileHandler H = 0.74, Utilities H = 0.5 y BusinessLogic H = 0.4) fueron creados pensando en una distribución de capas más que por entidades, la cohesión relacional de éstos es muy baja (no se buscó que las clases dentro de los mismos se relacionen entre sí, sino que agrupen una determinada capa).
Para solucionar esto se podrían haber creados los paquetes usando un approach más hacia clausura común *"Las clases pertenecientes a un paquete, deben cambiar por el mismo tipo de cambio. Si un cambio afecta a un paquete, afecta a todas las clases del mismo y a ningún otro paquete"* (Martin), esto implicaría crear un paquete bug por ejemplo, que contenga bug repository, bug logic, bug, etc. De esta forma se podría mejorar esta métrica considerablemente ya que las clases dentro del paquete se relacionarían entre sí.

Diagrama de componentes



10

El diagrama de componentes muestra todos los componentes que comprenden al sistema y como se dan las dependencias entre estos. 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. Esto hace que algunas dependencias no existan directamente entre paquetes, por ejemplo Factory depende de BusinessLogic, pero lo mostramos por medio de interfaces que BusinessLogic implementa.
En este diagrama no utilizamos el recurso de puertos, dado que entendemos que no comparten rasgos específicos para agruparlos, por ejemplo, no consideramos que se debería agrupar la lógica de un user con la lógica de una tarea. Si se realizaran cambios en el futuro para que la lógica de una tarea esté dentro de un proyecto por ejemplo, podríamos agregar un componente que refleje este agrupamiento que indica ciertos puntos de coincidencia.

[10] Diagrama encontrado en Documentation/Release2/Diagrams/Components.svg

Anexo

Cobertura de las pruebas unitarias

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
agust_DESKTOP-PFVBI7S 2021-11...	0	0.00%	2346	100.00%
businesslogic.dll	0	0.00%	139	100.00%
dataaccess.dll	0	0.00%	666	100.00%
domain.dll	0	0.00%	309	100.00%
externalreader.dll	0	0.00%	25	100.00%
externalreadertest.dll	0	0.00%	89	100.00%
filehandler.dll	0	0.00%	105	100.00%
filehandlerfactory.dll	0	0.00%	16	100.00%
testutilities.dll	0	0.00%	31	100.00%
utilities.dll	0	0.00%	71	100.00%
webapi.dll	0	0.00%	570	100.00%
webapitest.dll	0	0.00%	325	100.00%

Se logró llegar a el 100% de cobertura de las pruebas, se siguió el objetivo de seguir utilizando TDD en el backend y llegar a un nivel alto de cobertura de los mismos. Cabe destacar que también fue muy importante realizar pruebas de integración, ya únicamente con pruebas unitarias no se pueden cubrir todos los casos y hubieron veces que si bien pasaban todas las pruebas, realizamos una prueba de integración y descubrimos errores nuevos.

Templates

Se utilizaron principalmente 2 templates para trabajar nuestro obligatorio para resolver el frontend.
Como template principal, se utilizó Argon de Creative Tim, donde se dejó referencias al mismo directamente en el footer, indicando esta acción. El link de este template es el siguiente: https://www.creative-tim.com/product/argon-dashboard-angular

También utilizamos un template para la página que se muestra al ingresar una ruta erronea, mostrando el componente de page-not-found. Este template lo obtuvimos de esta página: https://codepen.io/JuliaSS/pen/ZMaXQV

Datos de prueba

En la carpeta Database ubicada en root se pueden encontrar dos sets de datos de prueba, uno vacío, conteniendo solo un admin para no tener que agregarlo a mano y otro con proyectos, tareas, bugs y más usuarios cargados.
En el último se cargaron 5 proyectos, se intentó mostrar la mayoría de los casos de los mismos, el primero tiene bugs y usuarios, el segundo tiene bugs usuarios y tareas, el tercero

usuarios, el cuarto está vacío y el quinto se llama “Nombre de Proyecto” ya que todos los tests de archivos JSON, XML y TXT que tenemos intentan agregar bugs a éste.
También se crearon todo tipo de usuarios, se crearon bugs pendientes, arreglados por otros usuarios, creados por admin, para mostrar también los costos de proyecto.

	Id	FirstName	LastName	UserName	Password	Email	Role	Token	HourlyRate
1	19	Pepe	Gonzales	admin	admin	admin@gmail.com	3	/VvLha6q2UhZRJEXCnC0Q6xm9YVEHhUE	0
2	20	Juan	Sanchez	jaunito243	pass	jaun@gmail.com	2	K561U66q2UgKlcJE15tfSrePv9CHzBik	24
3	21	Mario	Ramirez	mmario3	pass	marito@gmail.com	1	NULL	27
4	22	Maria	Paz	marim	pass	mari@mail.com	2	NULL	32
5	23	Pedro	Melnik	pedrete	pass	lego@gmaicol.com	1	NULL	23
6	24	Braian	Sosa	sosaaaa	pass	sosita@gmail.com	3	NULL	0

	Id	Name
1	23	Proyecto Tienda Inglesa
2	24	Proyecto ORT
3	25	Proyecto 1
4	26	Empty Project
5	27	Nombre de Proyecto

	Id	Name	HourlyRate	Duration	ProjectId
1	2	Terminar pruebas de dominio	14	23	24
2	3	Finish diagrams	44	23	25

Evidencia del diseño y especificación de la API

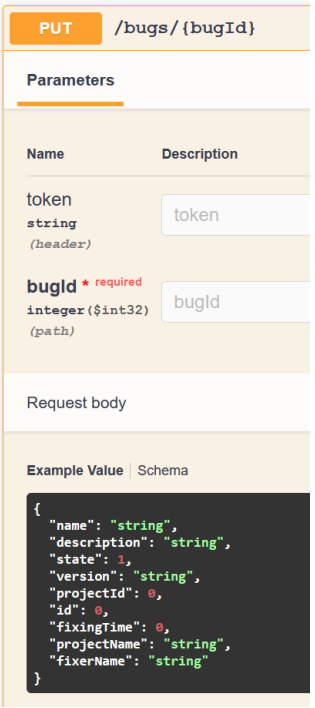
Descripción General:

Para la creación de los endpoints en el proyecto del WebApi se siguió los criterios de REST, por ejemplo no usar verbos en la URI dado que cada acción ya tiene un verbo que indica que tipo de resultado se espera al ejecutar el endpoint.

Pudimos implementar todas las funcionalidades requeridas generando un endpoint asociado para poder ejecutar el endpoint desde el frontend.

Descripción y justificación de diseño de la API:

Criterios REST



Creamos una interfaz uniforme, basando nuestros endpoints, en los diferentes recursos que cuenta el sistema en forma de objetos de dominio del sistema.
Para todos los casos que se necesite mandar un body o recibir objetos, está indicado como se debe parsear (JSON) para mapearlo en el sistema a una entidad concreta.

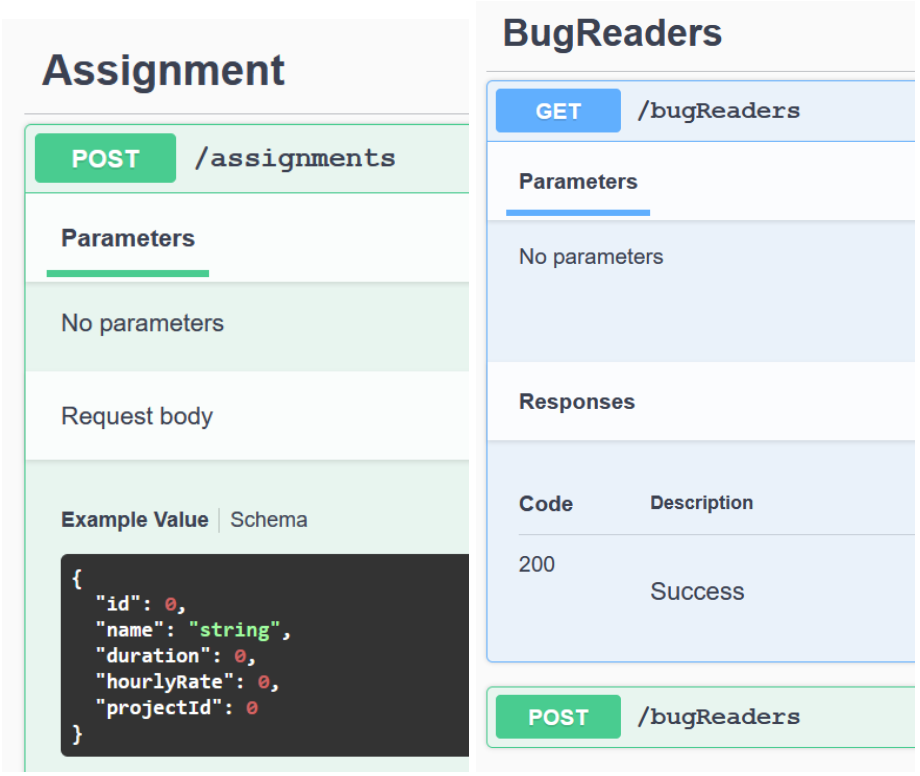
No se acopla a la tecnología del código .NET para que si cambia la implementación las URIs ya estén preparadas para aplicarse en otras diferentes tecnologías o requieren la menor cantidad de cambios posibles..

Para continuar con criterios de REST, no se maneja el concepto de estados, esto indica que toda la información solicitada por un request, debe estar contenida en sí misma, por diferentes mecanismos (body, header, query, etc.) y también habilita que dos llamadas consecutivas se puedan realizar sin problemas.

Se simuló la relación entre cliente servidor, porque siempre se trabajó desde localhost, pero el sistema debería soportar realizar una comunicación entre cliente y servidor, cambiando algunas configuraciones. Para esta entrega se cambió el cliente de PostMan a una aplicación desarrollada en Angular y se pudo utilizar correctamente los endpoints, luego de configurar los CORS.

Se utilizaron los principales métodos HTTP, Post, Put, Patch, Get y Delete, para indicar que tipo de acción realiza el endpoint, evitando agregar verbos a las URIs ya que los métodos

mencionados ya deberían indicar que acción realizan sin necesidad de incluir otro verbo. También se siguió la recomendación de utilizar los recursos en plural en toda acción.
Algunos ejemplos pueden ser estos endpoints:



Mecanismos de autorización

Para resolver este problema de permisos dentro del sistema, utilizamos básicamente dos herramientas que provee .NET sin recurrir a JWT que no tuvimos tiempo de analizarlo.

La primera fue GUID, el cuál genera un número random y es utilizado en la industria para crear un token de sesión al usuario. Nos pareció chequear además que no pueda ser repetible este número, por lo que le concatenamos como string al número generado, la fecha actual del sistema. Esto nos da una certeza de que no se va a repetir el token dentro del sistema. Para utilizar este token, guardamos este GUID en la tabla User, donde lo creamos a partir del endpoint Post de SessionsController. En el caso de que un administrador haya creado un usuario con esas credenciales, se crea un token y se lo asigna a ese usuario en particular. Se puede obtener el rol a partir del token, ya que es único e identifica a los usuarios, por lo que lo devolvemos al cliente para poder saber que rol y que token tiene un usuario que ingrese sesión. No se implementó un LogOut en el backend, por lo que la única forma de eliminar el token, sería eliminar el usuario desde la base de datos.

La segunda, fue IAuthorizationFilter, una interfaz que provee AspNetCore para poder aplicar un filtro en cada request que se realice si el filtro está activado. En nuestro caso se utilizó para confirmar que el cliente realizando la request, tenga permisos para la misma, según su rol.

Códigos de estado

Nuestro sistema maneja distintos códigos de estado según la situación de cada acción.
Los códigos se manejaron en los filtros, que son los encargados de asegurarse que el usuario tenga el rol correspondiente, pero también de cachear las excepciones de algún método en específico. Para no preguntar por cada excepción del dominio, por ejemplo que el nombre de proyecto sea válido, el mail del usuario, etc. se creó una clase padre que abarca todos los casos pero sigue mostrando los mensajes específicos.
En este filtro se devuelve un 500 por defecto si hubo un error y el sistema no lo supo manejar.
409 se devuelve en el caso de que se quiera crear un Proyecto, pero el nuevo nombre ya se encuentra en la base de datos. Si la sintaxis que el cliente envía no es correcta se muestra un 400, por ejemplo si faltan campos de algún objeto en el JSON.
El último código que maneja este filtro es 403, indicando que el servidor entendió pero se rehúsa a realizar la acción. Creemos que este código debió ser 409 pero no nos dió el tiempo de cambiarlo.

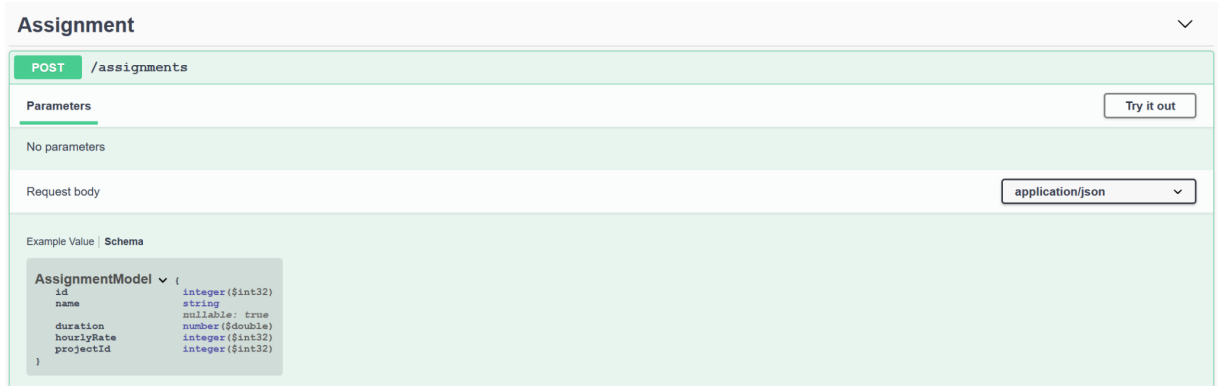
El filtro de autorización, maneja dos códigos, 401 y 403, uno para indicar que no tiene permisos, mostrando como mensaje los roles autorizados, y el otro para indicar que no tiene permisos, y no se puede autenticar.

Resources

Se presentarán los cambios en los endpoints según la primera entrega, por recurso.

Assignments

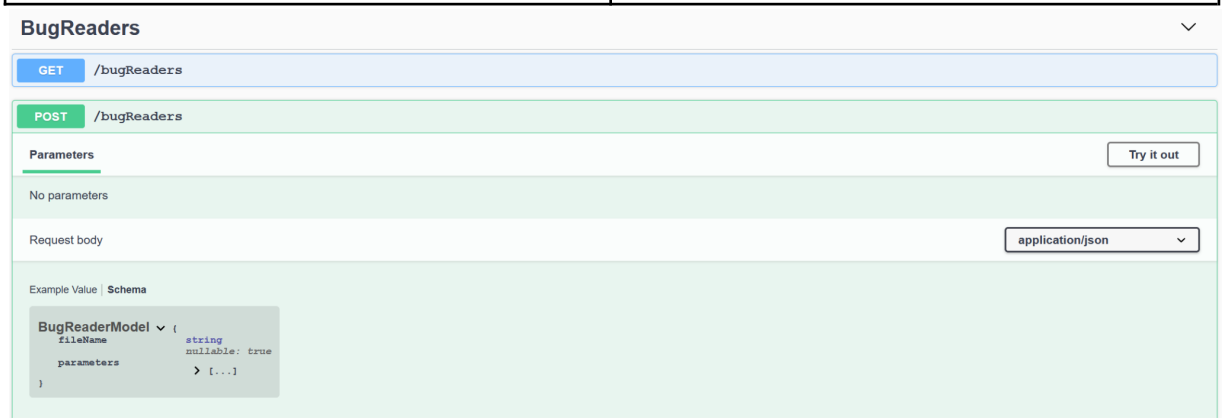
POST /assignments	Se crea una tarea en el sistema



La tarea solo puede existir en un proyecto, por lo que requiere el id de esa entidad para ser creada.

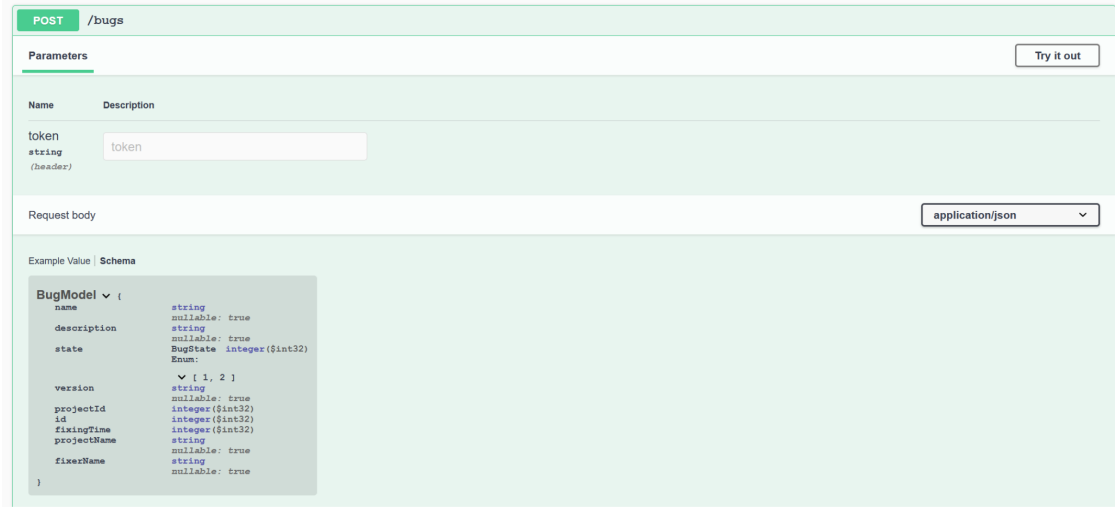
BugReaders

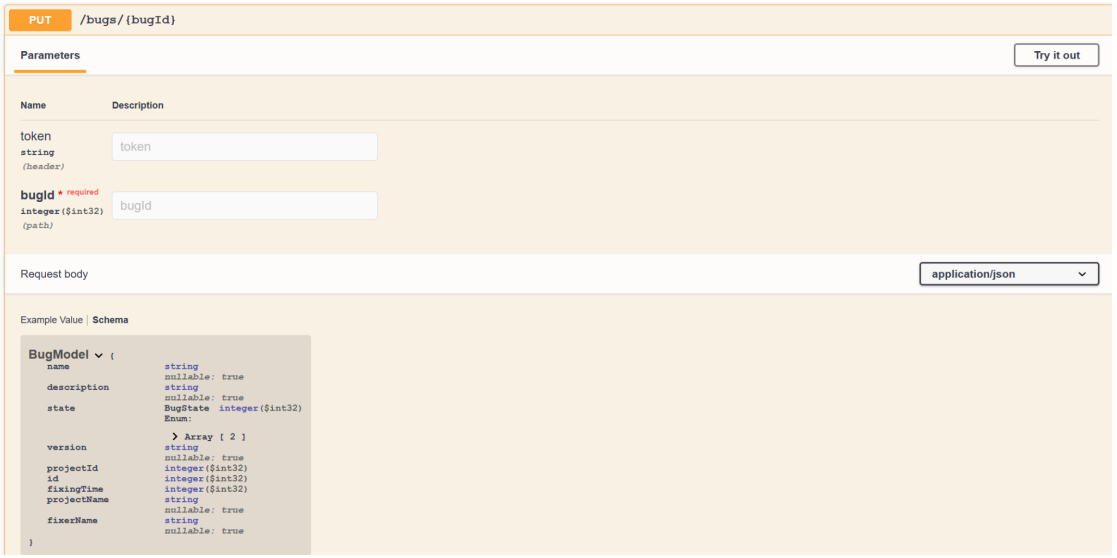
GET /bugReaders	Trae la información de parámetros y nombres de los external readers
POST /bugReaders	Carga los bugs de un external reader



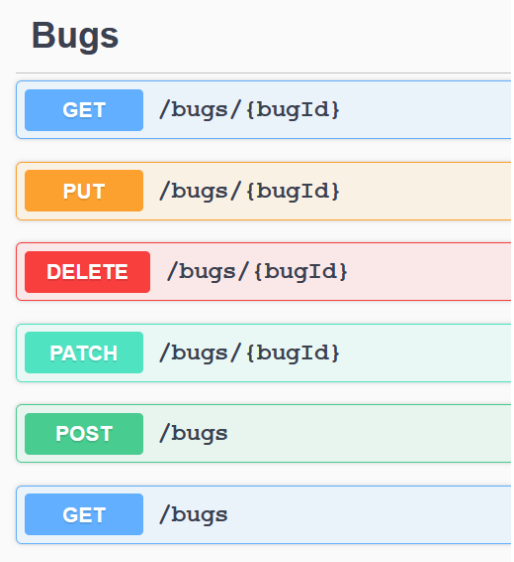
Bugs

PUT /bugs/{bugId}	Modifica un bug en específico. Se modificaron los campos.
POST /bugs/{bugId}	Crea un bug en específico. Se modificaron los campos requeridos.





Se modifica el modelo que interviene en estos endpoints, agregando fixingTime, projectName y fixerName. El projectName lo traemos para saber a que proyecto pertenece sin necesidad de traer el objeto.

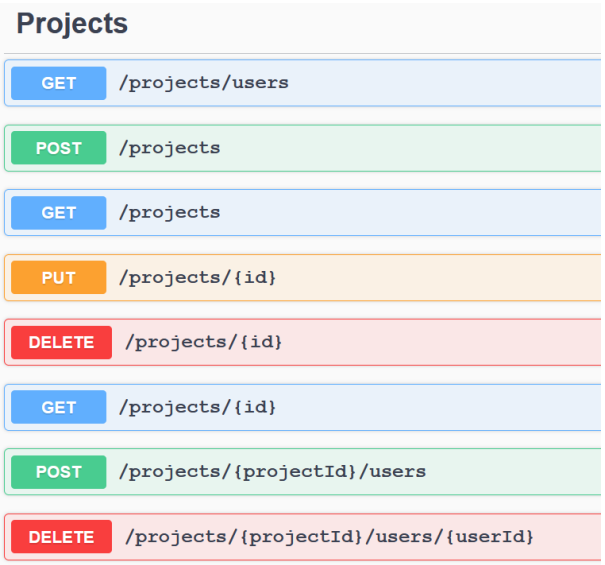


Projects

GET /projects/users	Trae todos los proyectos del usuario ingresado en el sistema
GET /projects/{id}	Trae un proyecto en particular para un usuario
DELETE/projects/{projectId}/users/{userId}	Desasocia un usuario del proyecto

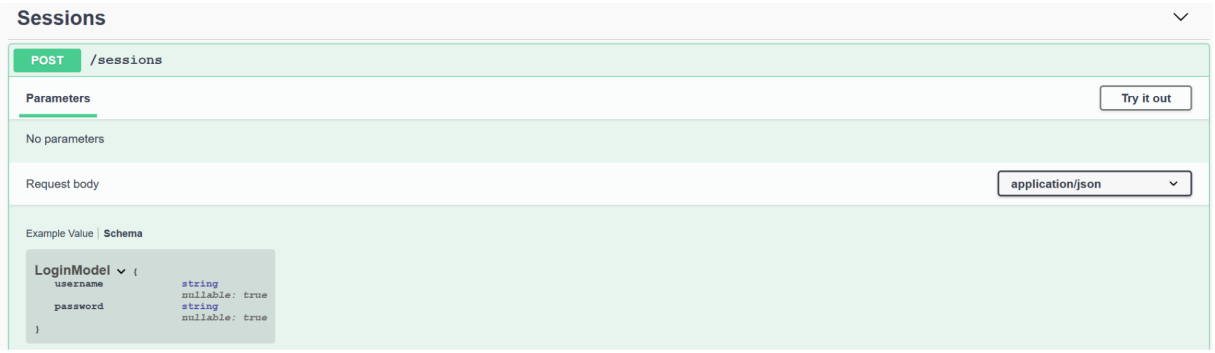
Los get se agregaron como funcionalidades extras para que la interfaz tenga una usabilidad mejor a lo planteado inicialmente. El delete ya existía, pero tuvimos que agregar el id del

usuario a desasociar, dado que no podíamos utilizar el body como hacemos en el POST, dado que angular no permite utilizar body con métodos HTTP Delete.



Sessions

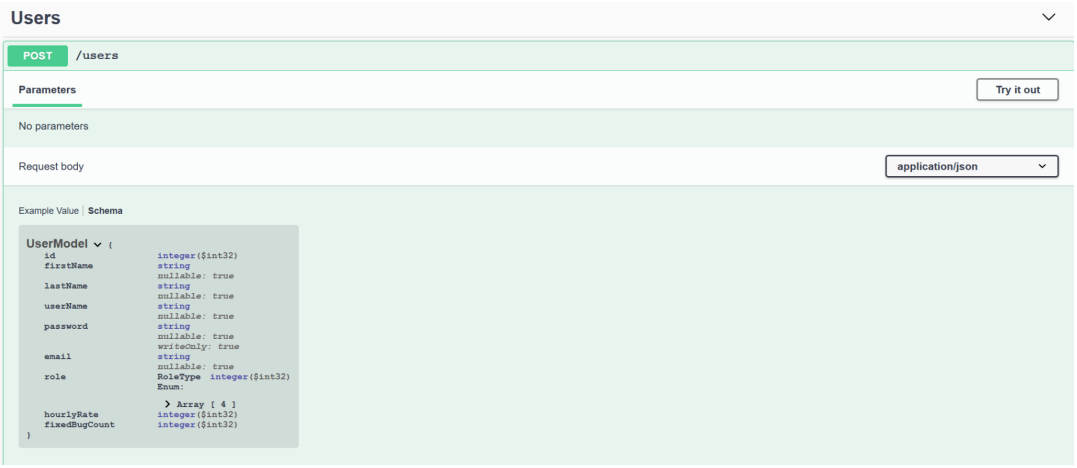
POST/sessions	Es el endpoint de LogIn, para asignar un token al usuario.



Devuelve un token y un rol para poder utilizarlos en el frontend.

Users

POST /users	Crea un usuario en el sistema



Se cambiaron los campos del usuario en el dominio, por lo tanto se cambió el modelo también. Por ejemplo, se agregó hourlyrate.

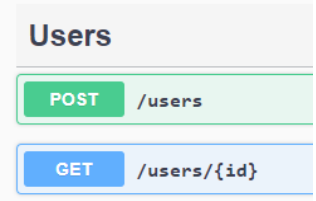
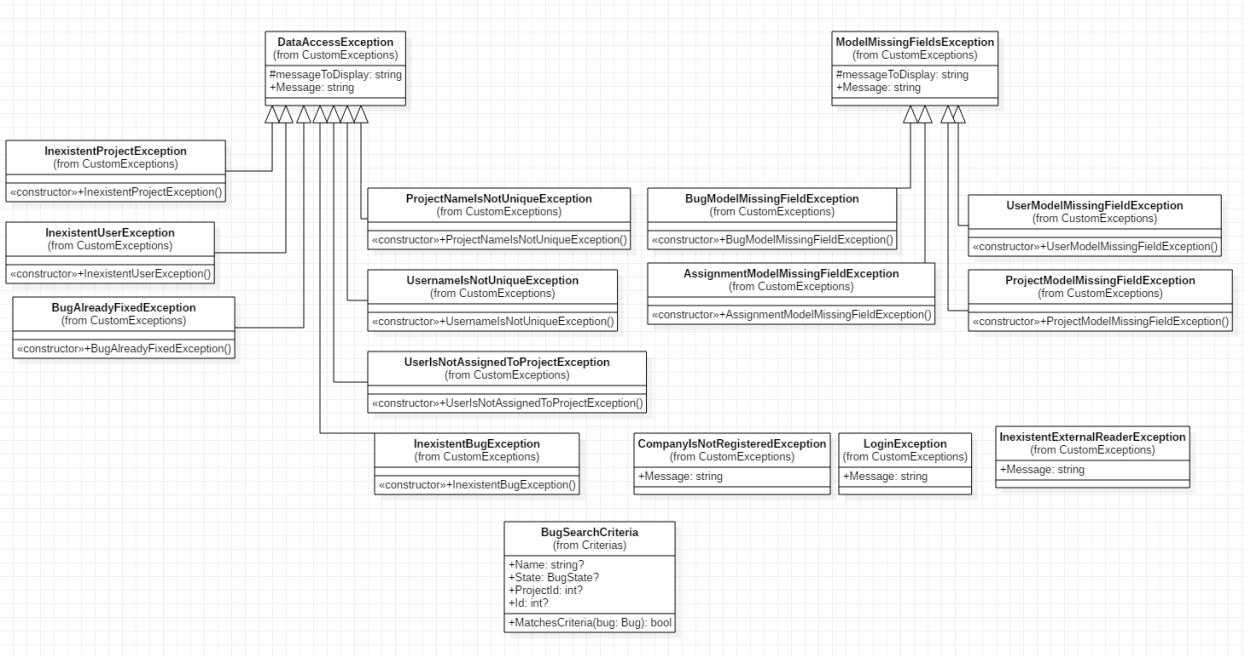


Diagrama de clase de paquete utilities



11

[11] Diagrama encontrado en Documentation/Release2/Diagrams/Utilities.svg