Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2 - Obligatorio 1

Entrega como requisito de la materia

Diseños de aplicaciones 2

https://github.com/ORT-DA2/OBL-Asadurian-Reyes.git

Diego Asadurian 198874 - Hernán Reyes 235861

Tutor: Gabriel Piffaretti, Nicolas Blanco, Daniel Acevedo

07 de Octubre de 2021

Índice

Descripción General del trabajo/sistema

En esta primera instancia se desarrolló el BackEnd del sistema “penguin”, cuya finalidad es ayudar a las empresas de desarrollo de software a llevar un control sobre los bugs asociados a sus proyectos, como también los desarrolladores y testers involucrados en cada bug y proyecto.

El mismo cuenta con distintas funcionalidades, unas de sus principales es poder crear proyectos y poder generar bugs correspondientes a los mismos. Así como también, poder asignar developers y/o testers entre otras.

Este desarrollo se llevó a cabo utilizando el framework de .Net Core.

Alcance del sistema

En esta primera entrega pudimos cumplir con todos los requerimientos no funcionales especificados en la letra.

Por otro lado, dentro de los requerimientos funcionales no se llegó con el cumplimiento de las funcionalidades del rol tester de filtrar un bug por proyecto, nombre o estado del mismo por temas de tiempo.
Las mismas no fueron implementadas ya que no era un requerimiento tan excluyente para el funcionamiento de nuestra aplicación, ya que en el sistema ya podemos filtrar bugs por id, dado esto pusimos nuestro enfoque a otras funcionalidades pudiendo así llegar con un producto más maduro, en términos de funcionamiento y calidad, en los requerimientos excluyente de la aplicación.

Dichos requerimientos no implementados ya fueron analizados y serán llevados a cabo en una segunda entrega del obligatorio.

A su vez se tomaron algunas decisiones en distintas funcionalidades donde se especificarán en detalle en la sección “Principales decisiones - Funcionalidades”.

- Mejoras a tener en cuenta

- **Mejor cumplimiento de TDD:**

  En algunas ocasiones no pudimos cumplir estrictamente con TDD, sumando que en algunas funcionalidades no se implementaron pruebas y en otras no se logro cumplir el 100% de cobertura del método.

  Creemos que aumentando nuestras pruebas podemos generar mayor valor a nuestra aplicación, sin garantizar la ausencia de errores, pero generando un mayor respaldo a la calidad de nuestros requerimientos.
  Esto va ser tenido en cuenta para una segunda entrega.

- **Mayor estudio de lo desconocido:**

  Al comenzar nuestro obligatorio comenzamos por nuestro dominio y lógica de negocio. El mismo fue implementado de una determinada manera, como por ejemplo con el desarrollo de una clase AdministratorLogic. En su momento veíamos necesaria la implementación de dicha clase dado nuestro conocimiento y una visión de diseño 1. Al momento de implementar la WebApi de la manera que fue pensada en su principio vimos que no era necesaria la clase implementada, ya que con los filtros de autenticación podríamos validar qué funciones podría realizar un determinado usuario. Dado el desconocimiento de los filtros, como también de apis, realizamos implementaciones inecesarias que luego debieron ser eliminadas.

Principales decisiones

Funcionalidades

- Alta de bugs, solo se podrán crear bugs siempre y cuando exista el proyecto asociado al mismo previamente.
- Importación de bugs, solo se podrá realizar la importación de los bugs asociados a un proyecto existente al sistema.
- Un bug puede ser resuelto sin un usuario asociado a su resolución. Es decir, si un tester cambia el estado del bug a resuelto el bug no asocia a quien lo resolvió. En cambio si un desarrollador actualiza el bug a resuelto si aparece que dicho desarrollador lo resolvió.
- El nombre de un proyecto debe ser único.
- El email de un usuario debe ser único.
- Asumimos que para la importación de archivos se le debe enviar una ruta donde se encuentre el mismo, aunque el servidor realiza distintas verificaciones para verificar que sea un archivo válido. Lo que queremos decir es que no recibimos archivos de tipo byte sino la ruta del archivo que se encuentra en el servidor.

Persistencia

Para la persistencia de los datos utilizamos la herramienta Entity Framework Core de .NET basándonos en la técnica Code First.

De esta manera, generamos nuestra base de datos a partir de nuestro código implementado previamente contando así una mayor flexibilidad a la hora de poder realizar cambios de estructura en nuestros modelos.

Nuestra configuración de relaciones, claves, mapeos, etc fue realizada a través de fluent api.

Nos basamos en la interfaz “IEntityTypeConfiguration<ENTIDAD>”, la cual nos permitió configurar cada entidad en clases separadas en vez de tener en nuestro contexto toda la configuración de todas las entidades en nuestro método OnModelCreating.

Esto lo creímos necesario para cumplir con el patrón SRP, de esta manera nuestro contexto sólo se encarga de definir nuestro modelo de tablas y configurar su conexión.

Por otro lado, en las clases que implementan la interfaz IEntityTypeConfiguration realizamos todas las configuraciones necesarias a cada entidad/tabla, contando así una clase por cada configuración de entidad y en cada clase se está manejando una sola entidad de configuración.

Para acceder a nuestros datos contamos con distintos repositorios.
Primero nos basamos en un repositorio genérico para realizar las operaciones basicas para cualquier entidad. A medida que veíamos necesario funcionalidades que solo aplicaban a una dicha entidad realizamos repositorios por entidades para las mismas.

Nos terminó generando repositorios para las entidades de Bug, Project y User. Las cuales tienen funciones específicas para cada entidad, de igual manera dicho repositorio extiende del repositorio genérico por tanto utilizamos las funcionalidades genéricas de acceso en el repositorio genérico para no duplicar código entre los repositorios por entidades y el genérico.

El repositorio genérico lo vimos necesario para no duplicar código ya que muchas de las funcionalidades se replicaban en cada entidad.

Web API

El punto de entrada a nuestro sistema es a través de la API la cual se realizó cumpliendo el estándar REST, ya que de esta manera contamos con una web api estandarizada más fácil de consumir por un tercero. Además de ser un requerimiento no funcional

En la misma se trató de contar con la menor lógica posible, utilizando las delegaciones correspondientes a nuestra BussinessLogic.

Volviendo al tema de nuestra API, su rol principal es atender peticiones y devolver respuestas a distintos clientes que la consuman.

- Principales verbos REST utilizados

    - GET
    - POST
    - PUT
    - DELETE

Esto logró que cada Controller tenga entre 3 y 7 endpoints, dependiendo del mismo.

- Retornos utilizados

Los retornos que manejamos en nuestra API contienen mensajes de éxito o error, esto va de la mano con los Status Code que brindamos para indicarle al usuario más información sobre la petición que realizó el cliente al servidor.

Los StatusCode que manejamos en nuestra API fueron.

Status Code	Representación
200 Ok	La solicitud fue realizada correctamente y la respuesta incluye datos.
201 Created	Utilizada en solicitud POST, realizada correctamente.
204 NoContent	Solicitud realizada correctamente. Utilizada para métodos Update y Delete.
404 Error	Manejo de distintos errores en el sistema.
403	Identificar que no tenemos permisos
401	Identificar que no estamos logueados

- Transferencia de datos

Para la transferencia de datos se utilizaron archivos JSON.

- Modelos

Decidimos que cada entidad del dominio tiene sus diferentes modelos en nuestra api (EntryModel, OutModel, UpdateModel).

El modelo Entry se utiliza para especificar que dicho modelo va a ingresar al sistema, por otro lado el modelo Out, se utiliza para enviarle al usuario su solicitud. También el modelo Update para poder manejar qué datos son editables y cuáles no.

Algunos modelos a parte de los mencionados también tienen un modelo extra que es para brindarle al usuario una determinada información, como es el ejemplo del modelo ProjectReportModel, el cual únicamente lo utilizamos para enviarle al usuario el reporte del proyecto.

El propósito de dichos modelos fue limitar la información de estos mismos ya que la creación de un objeto puede no ser igual al retorno del mismo, por ejemplo.

- Clase base

Implementamos una clase base denominada ApiBaseController que hereda de ControllerBase. La misma se implementó para que todos nuestros controller hereden de ella.

De esta manera solo debemos especificar que todos los controllers son [ApiController] en una única clase y no en cada controller. Es así que no estamos repitiendo código.

De esta misma manera se le asignó nuestro filtro de excepciones a dicha clase base para que a todos nuestros controller se le aplique dicho filtro.

- Filters Authorization

Para la autenticación utilizamos el AuthorizationFilter el cual recibe por parámetro un string el cual es el rol al cual se quiere autorizar o no, con esto evitamos la duplicación de código.

Este filtro se encarga de tomar el token desde el Header y validarlo, se encapsula la lógica en un solo lugar y este se ejecuta antes de llegar a los métodos del controller. Por lo que si el token no es válido o inexistente no se accede a los métodos del controller que lo implementen.

Otro detalle a tener en cuenta es que generamos una clase estática “Authorization”, la misma tiene los distintos permisos que puede tener un endpoint.
Es decir, nuestros token de autorización están conformados con un identificador (nuestro rol) y luego un guid generado. En esta clase se identificaron las posibles combinaciones de autorizaciones (un método puede ser accedido por dos tipos de roles, por ejemplo) por tanto se definieron las constantes necesarias para realizar las distintas autorizaciones.

```
public const string AllAutorization =
    Rol.administrator + "," + Rol.developer + "," + Rol.tester;

public const string Administrator =
    Rol.administrator;

public const string Developer =
    Rol.administrator;

public const string Tester =
    Rol.administrator;
```

Viendo nuestro código podemos ver la constante “AllAutorization” la cual tiene los identificadores de todos los roles. Pudiendo ser consumido en nuestro filtro de la siguiente manera.

```
[HttpGet]
[AuthorizationFilter(Autorization.AllAutorization)]
1 reference
public IActionResult GetAllBugs()
{
```

- Filters Exception

Por otro lado utilizamos un filter de excepción (ExceptionFilter), el cual lo denominamos en la clase ApiBaseController ya que es desde la cual todos los controller heredan de ella, con el fin de evitar duplicar código. Este filter lo que hace es obtener las excepciones de los métodos que son ejecutadas en los controllers y devolverle a nuestro cliente el status code con un mensaje de error correspondiente.

Dominio

En tema de entidades, realizamos las validaciones de sus datos en las clases de las mismas ya que cada entidad es el experto y conoce toda su información por tanto vimos viable agregar cada validación del lado del dominio.

Cuando realizamos un análisis de nuestras entidades en el negocio definimos como entidades estables a la entidad Project y User ya que vemos que su estructura/información no tendrán importantes cambios.

Por otro lado, los datos de rol en usuario y estado en bug eran muy inestables.
Creemos que los mismos pueden ser muy alterados y querer contar con más estados de mis bugs como también nuevos roles para User.
Con esto concluimos que deberían separarse en clases, de esta manera al momento de querer agregar un nuevo estado de los bugs o rol de usuario estamos cumpliendo con OCP.

Y más que cumpliendo con ocp, se podría agregar nuevos estados sin tocar el código mediante un script de insert a mi tabla correspondiente (rol y estado, en nuestro caso) con los nuevos datos que queremos manejar. Siempre y cuando no se quiera una lógica especificada para un estado de bug específico, por ejemplo.

Lógica de negocios

Tal como se comentaba anteriormente se implementó una clase “Rol”.
Para continuar con el cumplimiento de OCP del lado de la lógica de negocios se implementó una clase en businessLogic por cada rol, como es el ejemplo de “DeveloperLogic” y “TesterLogic”. De esta manera al querer ingresar un nuevo rol solo debemos crear una nueva clase lógica con las funcionalidades para dicho rol.
Por otro lado, estamos cumpliendo con SRP, dado que cada clase tiene una única responsabilidad.
En nuestro caso la responsabilidad de un desarrollador y por otro lado las responsabilidad de un tester.

Sobre la importación de bugs decidimos realizar un único endpoints que pueda recibir distintas extensiones de archivos, para esto tuvimos que generar una interface “IBugsImport” que implementa nuestra clase “BugsImport”, la misma fue implementada como una fachada donde su responsabilidad principal es preguntar qué tipo de extensión es el archivo recibido.
Una vez preguntado el tipo de archivo la misma clase delega a la clase correspondiente como la de “BugsImportTxt” o “BugsImportXml”.
De esta manera si queremos extendernos a un nuevo tipo de archivo no debemos cambiar el contrato de nuestra web api, solo debemos generar una nueva clase BugsImportTipoArchivoNuevo desde la lógica de negocio y agregar nuevo código en nuestra clase fachada (BugsImport) haciendo la delegación correspondiente a la clase de la nueva extensión.

Bugs o defectos conocidos

No detectamos ningún error en nuestros ciclos de prueba pero no garantizamos la ausencia de los mismos. Ya que es casi imposible.

Descripción de la API

Como se comentó anteriormente nuestra API cumple con el estándar REST, por lo cual utiliza los verbos HTTP. También como se mencionó anteriormente utilizamos diferentes URIs para poder acceder a distintos recursos mediante el mismo verbo HTTP.

Para poder acceder a los métodos de la lógica de una forma correcta, lo que hicimos fue inyectar las dependencias de la lógica necesarias para cada controller para así poder acceder a los métodos de los mismos de un forma correcta y evitar manejar instancias de clases.

El mismo proceso se realizó para las clases de los repositorios que acceden a la base de datos.

Dichas clases de dominio se mapean con los modelos implementados para cada entidad como los mencionados anteriormente (EntryModel, OutMode, UpdateModell).

El propósito general de los modelos es limitar la información que se recibe o envía, ya que no siempre se desea enviar toda la información al cliente, asi como también no estamos relevando nuestra estructura del dominio ni la de nuestra base de datos.

Endpoints y Justificación

UserController(penguin/users)

- POST : crea un usuario, recibe nombre, apellido, username, password, email y rol en el body.
- GET: retorna todos los usuarios que hay en el sistema.
- GET /{userId}: retorna la información de un usuario específico con igual id al enviado.

TesterController(penguin/testers)

- GET /{testerId}/bugs: devuelve todos los bugs asociados al tester con igual id al enviado.
- POST {testerId}/project/{projectId}: asigna un tester con igual id al enviado, a un proyecto con igual id al enviado.
- DELETE /{idTester}/project/{projectId}: elimina un tester con igual id al enviado, de un proyecto con igual id al enviado.

DeveloperController(penguin/developers)

- GET /{developerId}/bugs: devuelve todos los bugs de los proyectos en los cuales el desarrollador con igual id al enviado pertenece.
- POST /{developerId}/project/{idProject}: asigna un developer con igual id al enviado, a un proyecto con igual id al enviado
- DELETE /{developerId}/project/{idProject}: elimina un developer con igual id al enviado, de un proyecto con igual id al enviado.
- GET /{developerId}/countBugs: devuelve la cantidad de bugs resueltos por el desarrollador con igual id al enviado.
- PUT /{developerId}/bugState: actualiza el estado de un bug, enviando el id del developer, y el body recibe el nuevo estado y el id del bug.

ProjectController(penguin/projects)

- POST: crea un nuevo proyecto, recibe nombre en el body.
- GET: devuelve todos los proyectos existentes en el sistema.
- GET /bugs: devuelve todos los bugs de los proyectos
- GET /{projectId}: devuelve el proyecto con igual id al enviado.
- GET /{projectId}/bugs: devuelve todos los bugs que tiene el proyecto con igual id al enviado.
- DELETE /{projectId}: elimina del sistema el proyecto con igual id al enviado.
- PUT /{projectId}: actualiza el proyecto con igual id al enviado, recibiendo el nuevo nombre por body.

BugController(penguin/bugs)

- POST: crea un bug, recibe projecto, id, nombre, dominio, version, estado en el body.
- GET: devuelve todos los bugs del sistema.
- GET /{bugId}: devuelve el bug con igual id al enviado y en el body se le pasa el id del usuario que está realizando el request.
- DELETE /{bugId}: elimina del sistema el bug con igual id enviado y en el body se pasa el id del usuario que está realizando el request.
- PUT /{bugId}: actualiza el bug con igual id al enviado, y le agrega los datos proyecto, nombre, dominio, versión, estado enviamos en el body y el id del usuario que está realizando el request.

ImportsController(penguin/imports)

- POST : agrega al sistema el bug importado con el nombre de la ruta del archivo enviados en el body

SessionController(penguin/sessions)

- POST /login: sirve para la creación de una sesión de un usuario. Si los datos enviados en el body(email y password) son correctos le es devuelto al cliente un token, con un identificador dependiendo su rol.
- POST /logout: sirve para cerrar la sesión de un usuario. Si el token enviado en el body es correcto, se cierra la sesión y se le elimina el token de la base de datos.

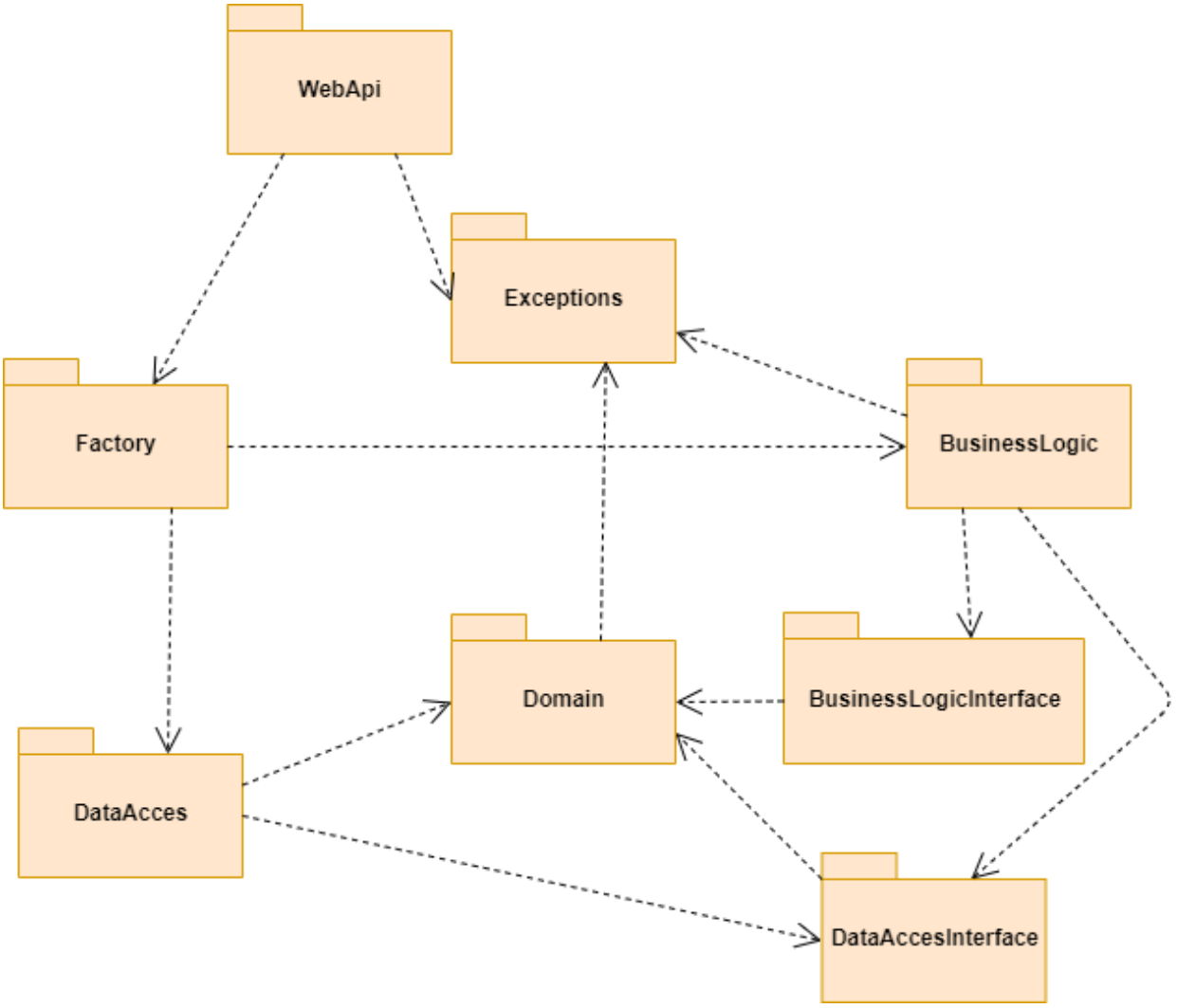
Descripción y justificación de diseño

Para llevar a cabo de forma correcta los diseños nos basamos en las vistas del modelo 4+1, dicho modelo sirve para describir la arquitectura de un sistema en varias vistas concurrentes como son: Vista Lógica, Vista de proceso, Vista de desarrollo, Vista de despliegue.

Vista Lógica

Dicha vista describe el modelo desde los elementos de diseño que componen el sistema y también cómo interactúan entre ellos.

En el diagrama que mostraremos a continuación se podrá observar la división de capas y las dependencias en nuestro proyecto, en este diagrama no se mostrará el paquete de Test.



A continuación haremos una breve descripción de la responsabilidad de cada paquete que maneja nuestro sistema.

Paquete	Responsabilidad
WebApi	Su responsabilidad es recibir las peticiones del usuario, se realiza mediante los controller y cada controller deriva a la lógica de negocio correspondiente. La entrada y salida de los datos es a través de los modelos.
BusinessLogicInterface	Dicha paquete posee las interfaces de nuestra lógica de negocio. Estas interfaces son utilizadas por las clases.
BusinessLogic	Dicha paquete maneja toda la lógica de cada entidad del dominio. La misma tiene la responsabilidad de manejar los datos de forma correcta. El mismo posee la implementación del paquete BusinessLogicInterface.
Domain	Dicho paquete posee las entidades que se manejan en todo el proyecto. Alguna clase en particular como por ejemplo User y Project tienen sus propios métodos de validación de datos, ya que conoce sus datos.
DataAccess	El mismo maneja el contexto de nuestra base de datos, además de implementar el paquete DataAccessInterface.
DataAccessInterface	Posee la interfaz con los métodos relacionados a las distintas peticiones de datos.
Factory	La misma tiene la responsabilidad de inyectar todas las instancias interfaces de las clases del negocio como también las del DataAccess en la WebApi, mediante la interfaz IServiceCollection. La finalidad de la misma es mantener un código más prolijo y ordenado.
Exceptions	Definimos nuestras propias excepciones que manejaremos en todo el sistema.

Para realizar los diseños nos basamos en el principio SOLID DIP(Inversión de Dependencia) que hace énfasis en que los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones.

Es por esto como se puede observar en el diagrama presentado anteriormente que las capas de bajo nivel como DataAccess dependen de las capas de alto nivel como es DataAccessInterface.

Esto cumple con DIP y es muy importante ya que las capas de bajo nivel están más dispuestas a cambios, por tanto un tercero que la consuma deberá cambiar su lógica para poder consumirla, pero al contar con un contrato la lógica que la consuma no va a cambiar porque solo se modificaría nuestra capa de bajo nivel que la implemente.

Otros aspectos que se tuvieron en cuenta a la hora de llevar a cabo dicha solución es que la misma cumple con los siguientes requisitos propuestos por Clean Architecture:

- La aplicación no debe depender de la interfaz de usuario.
- Todas las capas deben poder probarse de forma independiente.
- No se debe depender de frameworks específicos.

En nuestras palabras lo que propone el Clean Architecture es que las dependencias deben ser a capas que sean poco probables al cambio. Porque si dependemos de capas que pueden cambiar nos generaría un gran impacto en el sistema.

Diagrama de clases BusinessLogic

Este paquete como se comentó anteriormente es el encargado de manejar toda la lógica de negocio. Es el encargado de realizar todas las validaciones del negocio que son por fuera de cada entidad, como también es la encargada de utilizar el acceso a datos. Es por esto que dicho paquete posee las implementaciones de la clase de BusinessLogicInterface, utiliza la interfaz del IRepositorio (o IUserRepository, etc) para poder interactuar con la base de datos.

Siguiendo el principio Single Responsability de SOLID es que se creó una clase de Business Logic por entidad. Esto nos garantiza que solamente se realicen acciones sobre cada entidad mediante su propia lógica, ésta será modificada solamente si la entidad es afectada por cambios. Además de esto, cada lógica implementa una interfaz que expone las funciones requeridas para los respectivos controllers. Con esto estamos invirtiendo las dependencias y además la segregando las interfaces ya que el cliente implementa lo mínimo indispensable para su uso

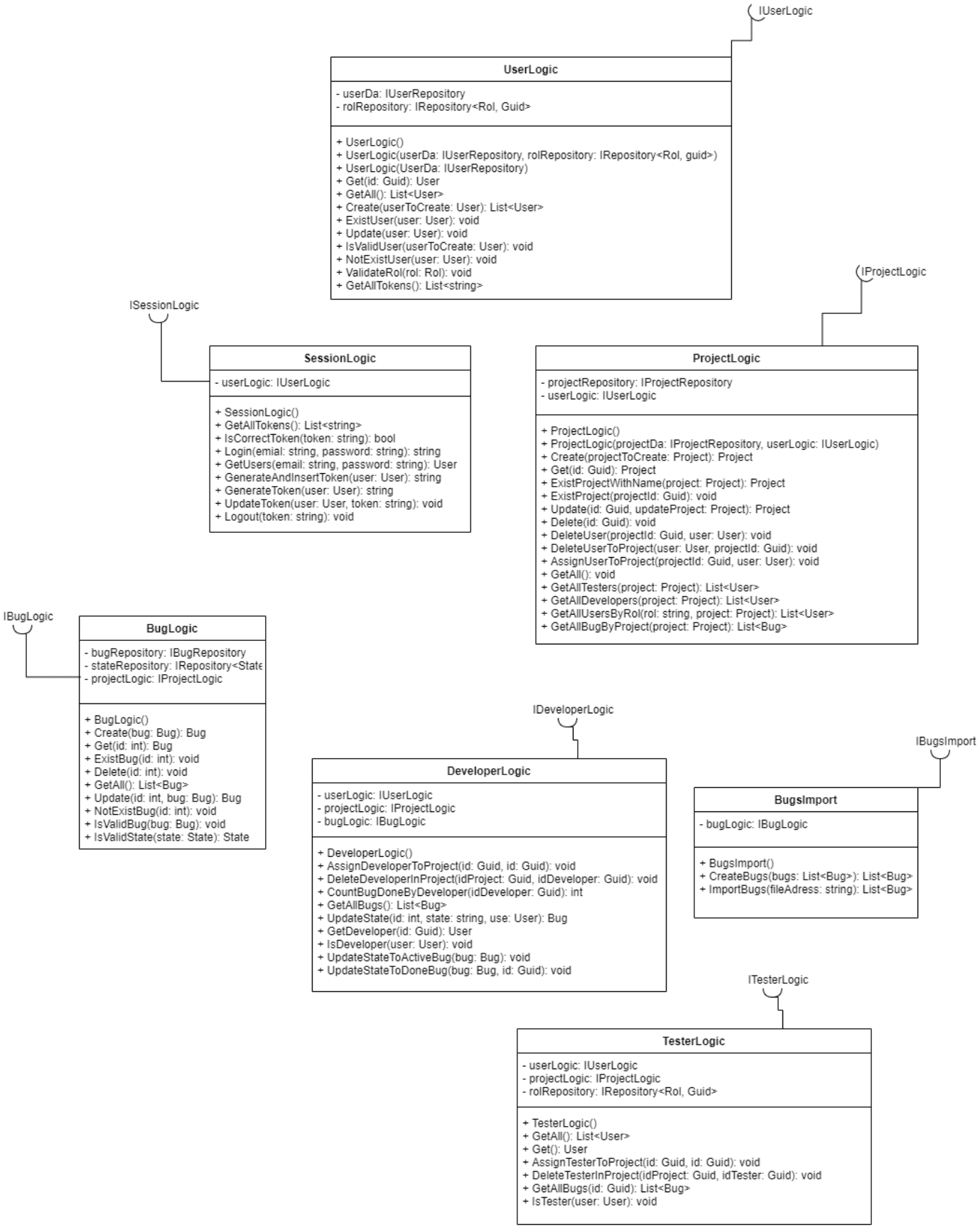

IUserLogic
UserLogic
- userDa: IUserRepository
- rolRepository: IRepository<Rol, Guid>
+ UserLogic()
+ UserLogic(userDa: IUserRepository, rolRepository: IRepository<Rol, guid>)
+ UserLogic(UserDa: IUserRepository)
+ Get(id: Guid): User
+ GetAll(): List<User>
+ Create(userToCreate: User): List<User>
+ ExistUser(user: User): void
+ Update(user: User): void
+ IsValidUser(userToCreate: User): void
+ NotExistUser(user: User): void
+ ValidateRol(rol: Rol): void
+ GetAllTokens(): List<string>
IProjectLogic
ISessionLogic
SessionLogic
- userLogic: IUserLogic
+ SessionLogic()
+ GetAllTokens(): List<string>
+ IsCorrectToken(token: string): bool
+ Login(emial: string, password: string): string
+ GetUsers(email: string, password: string): User
+ GenerateAndInsertToken(user: User): string
+ GenerateToken(user: User): string
+ UpdateToken(user: User, token: string): void
+ Logout(token: string): void
ProjectLogic
- projectRepository: IProjectRepository
- userLogic: IUserLogic
+ ProjectLogic()
+ ProjectLogic(projectDa: IProjectRepository, userLogic: IUserLogic)
+ Create(projectToCreate: Project): Project
+ Get(id: Guid): Project
+ ExistProjectWithName(project: Project): Project
+ ExistProject(projectId: Guid): void
+ Update(id: Guid, updateProject: Project): Project
+ Delete(id: Guid): void
+ DeleteUser(projectId: Guid, user: User): void
+ DeleteUserToProject(user: User, projectId: Guid): void
+ AssignUserToProject(projectId: Guid, user: User): void
+ GetAll(): void
+ GetAllTesters(project: Project): List<User>
+ GetAllDevelopers(project: Project): List<User>
+ GetAllUsersByRol(rol: string, project: Project): List<User>
+ GetAllBugByProject(project: Project): List<Bug>
IBugLogic
BugLogic
- bugRepository: IBugRepository
- stateRepository: IRepository<State
- projectLogic: IProjectLogic
+ BugLogic()
+ Create(bug: Bug): Bug
+ Get(id: int): Bug
+ ExistBug(id: int): void
+ Delete(id: int): void
+ GetAll(): List<Bug>
+ Update(id: int, bug: Bug): Bug
+ NotExistBug(id: int): void
+ IsValidBug(bug: Bug): void
+ IsValidState(state: State): State
IDeveloperLogic
DeveloperLogic
- userLogic: IUserLogic
- projectLogic: IProjectLogic
- bugLogic: IBugLogic
+ DeveloperLogic()
+ AssignDeveloperToProject(id: Guid, id: Guid): void
+ DeleteDeveloperInProject(idProject: Guid, idDeveloper: Guid): void
+ CountBugDoneByDeveloper(idDeveloper: Guid): int
+ GetAllBugs(): List<Bug>
+ UpdateState(id: int, state: string, use: User): Bug
+ GetDeveloper(id: Guid): User
+ IsDeveloper(user: User): void
+ UpdateStateToActiveBug(bug: Bug): void
+ UpdateStateToDoneBug(bug: Bug, id: Guid): void
IBugsImport
BugsImport
- bugLogic: IBugLogic
+ BugsImport()
+ CreateBugs(bugs: List<Bug>): List<Bug>
+ ImportBugs(fileAdress: string): List<Bug>
ITesterLogic
TesterLogic
- userLogic: IUserLogic
- projectLogic: IProjectLogic
- rolRepository: IRepository<Rol, Guid>
+ TesterLogic()
+ GetAll(): List<User>
+ Get(): User
+ AssignTesterToProject(id: Guid, id: Guid): void
+ DeleteTesterInProject(idProject: Guid, idTester: Guid): void
+ GetAllBugs(id: Guid): List<Bug>
+ IsTester(user: User): void

Diagrama de clases Data Access

Dicho paquete es el encargado de manejar el acceso a datos de cada entidad, ya que el mismo posee el repositorio genérico como también cada repositorio individual. En otras palabras es el encargado de acceder a los datos de cada entidad/tabla.
También contiene el contexto de nuestra base de datos y posee la implementación del paquete DataAccessInterface.

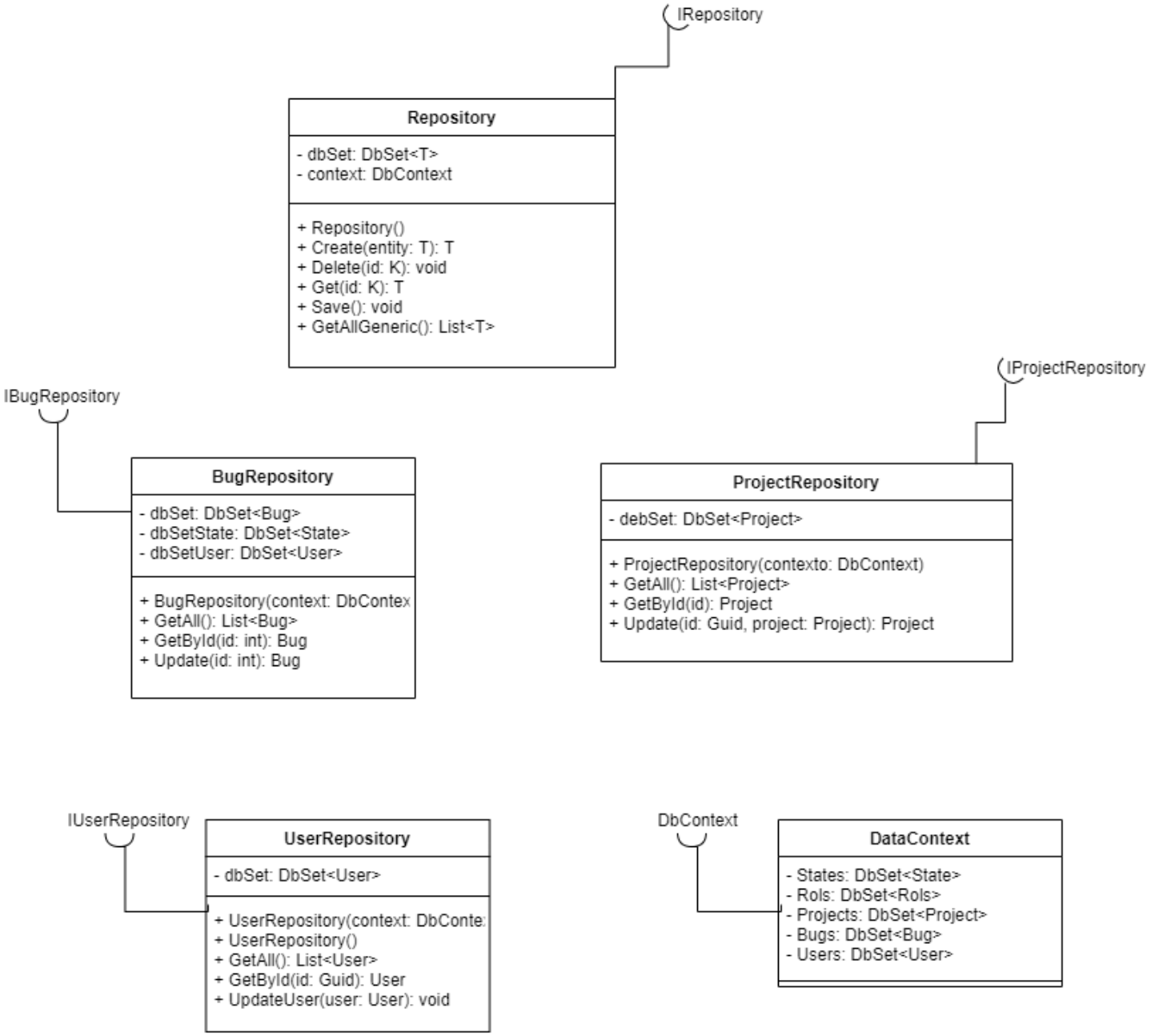


Diagrama de clases Domain

Este paquete posee las entidades que se manejan en todo el sistema. Estas mismas son persistidas en la base de datos. Algunas clases poseen algunas validaciones ya que son las encargadas de conocer por completo toda su información.

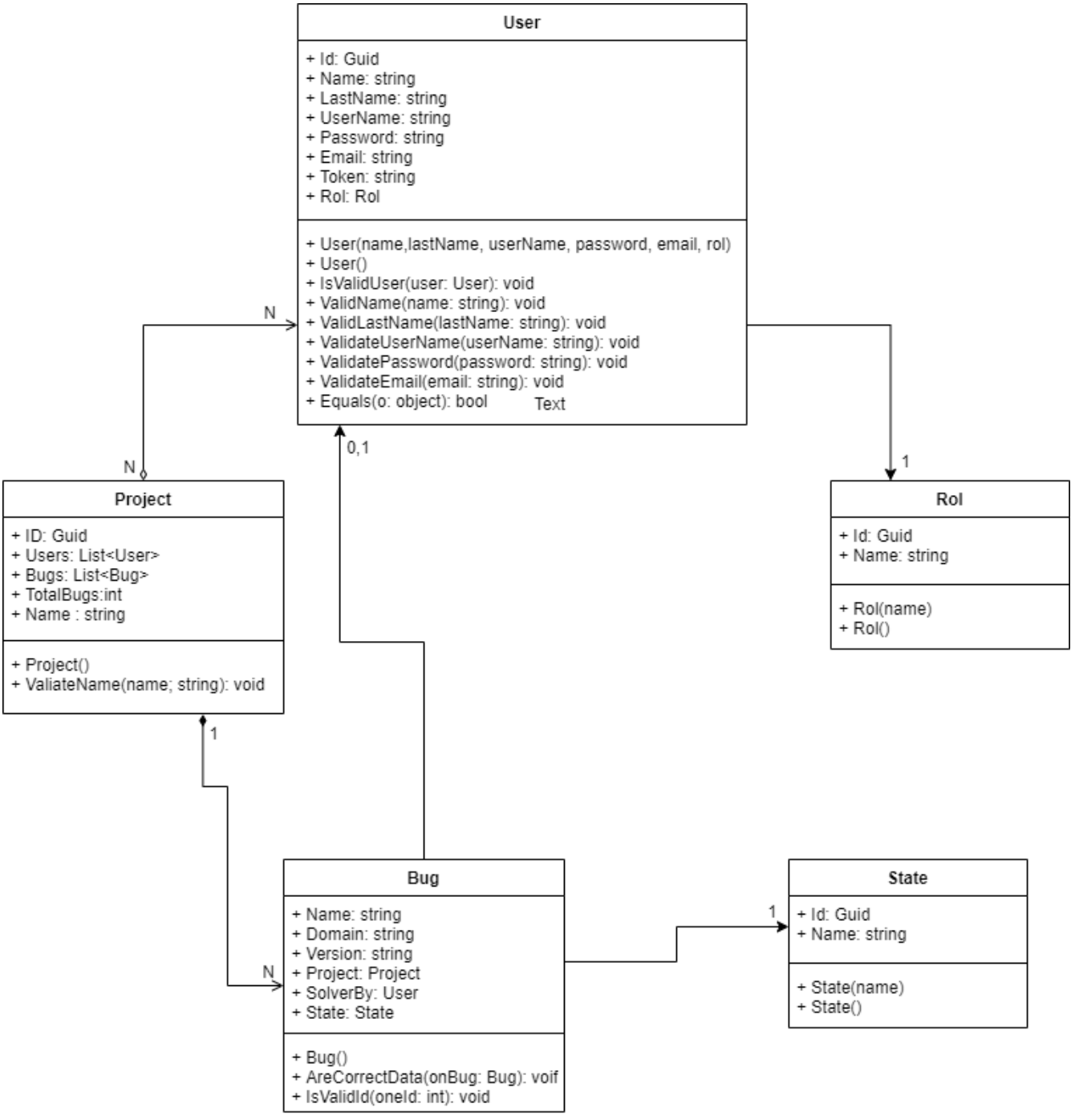
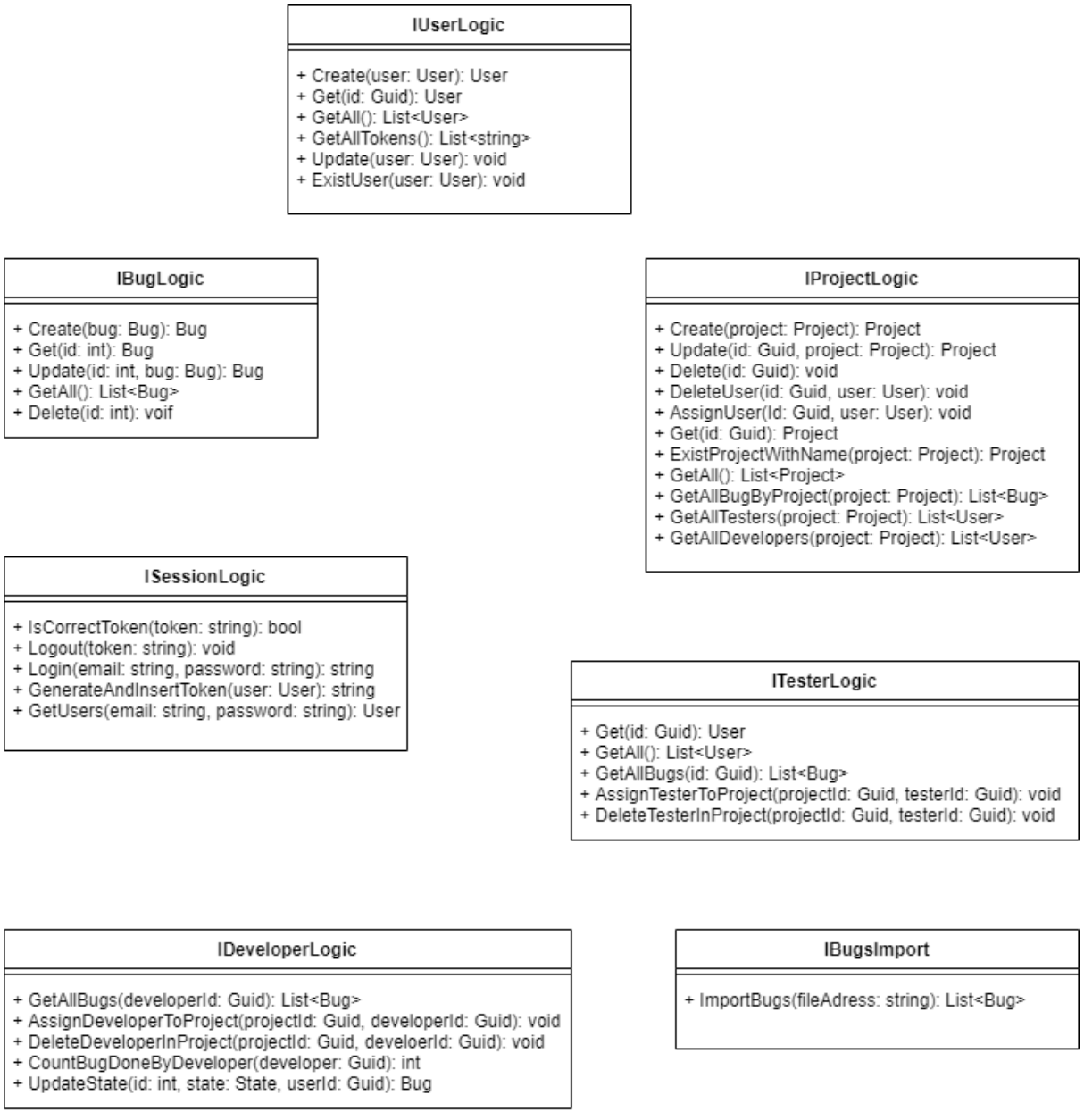


Diagrama de clases BusinessLogicInterface

Este paquete es el encargado de exponer los métodos de toda la lógica del sistema, para que otros paquetes puedan consumirlos. De esta manera estamos exponiendo un contrato y no implementación. Esto nos lleva a no acoplarse directamente en la lógica para que un cambio a futuro no impacte en los 3eros que la consuman.



Vista de Proceso

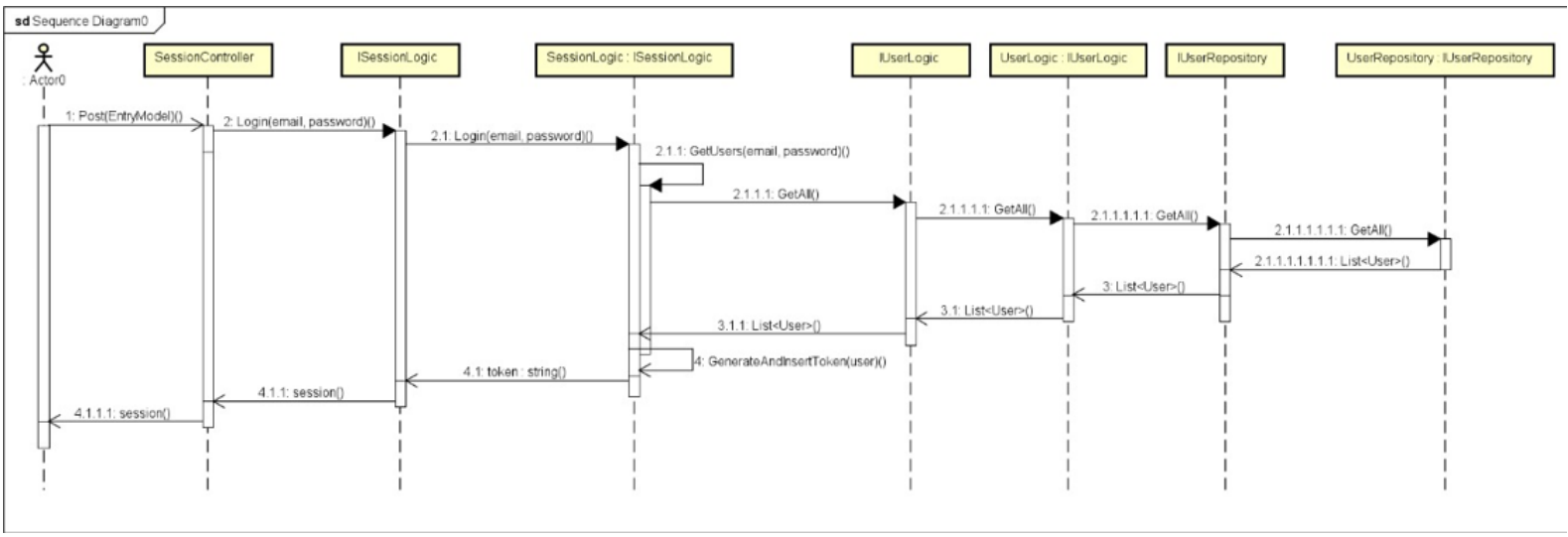
En esta sección trataremos de mostrar aspectos dinámicos del sistema, mostraremos algunas funcionalidades en tiempo de ejecución ya que dicha vista trata los aspectos dinámicos del sistema, explicando los procesos del sistema y cómo se comunican.
En otras palabras, la finalidad de esta sección es poder brindarle al usuario lector una forma rápida de entender el funcionamiento general de las determinadas funcionalidades del sistema.

Para este caso decidimos realizar los diagramas para las funcionalidades de obtención de un token de usuario(administrador) y la creación de un proyecto ya que ambas consideramos con son funcionalidades con suma importancia en la aplicación.

Obtención de token

Para llevar a cabo la creación del token lo primero que se debe verificar es que los datos que son enviados por el cliente (email y password) sean correctos, si dichos datos no son correctos el sistema brindará un mensaje de error notificando al usuario dicho problema.

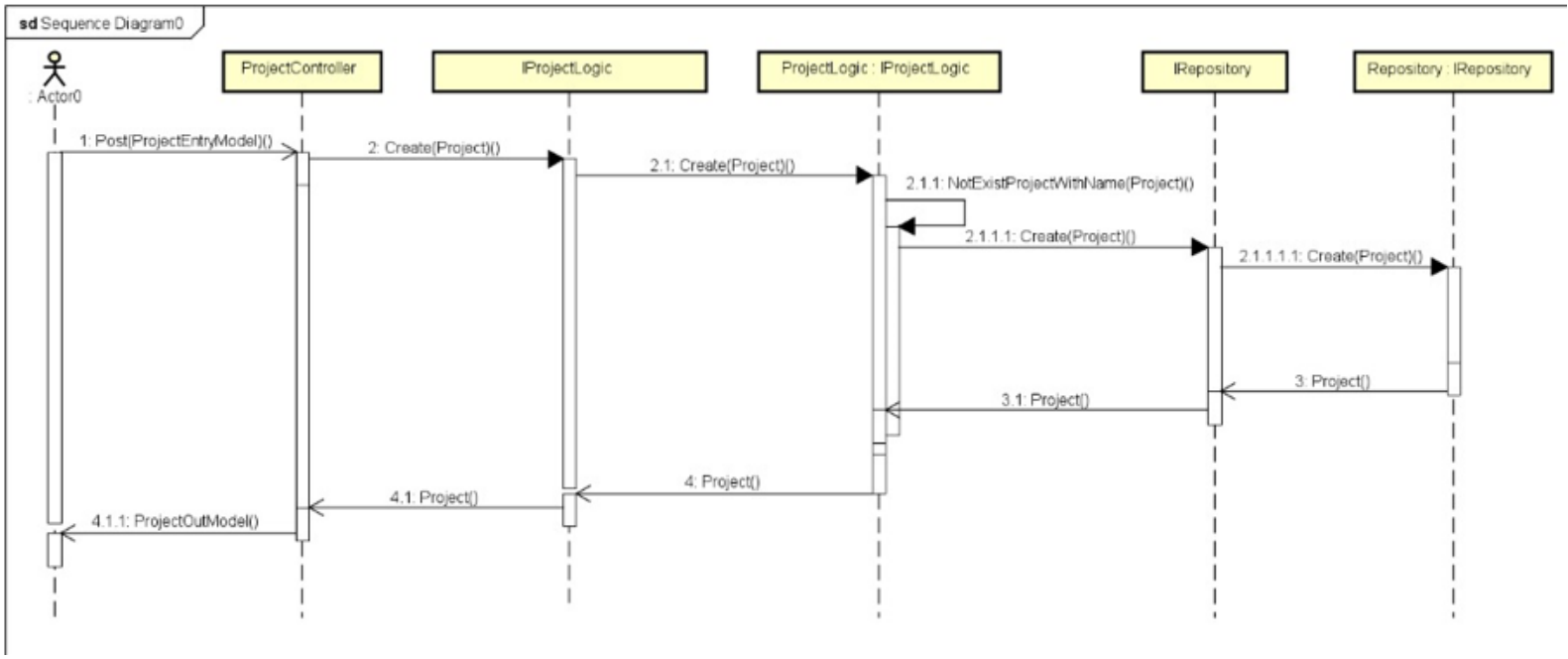
Luego de verificado dichos datos si son correctos, se llama a la lógica de session la cual se encarga de obtener el usuario en el sistema con dicho email y password, una vez obtenido el usuario, se genera el token para el mismo. Dicho token es generado con el Rol del usuario al principio seguido por un guid, la finalidad de esto fue lograr identificar de una forma sencilla cada Rol dentro del sistema.



Crear un proyecto en el sistema

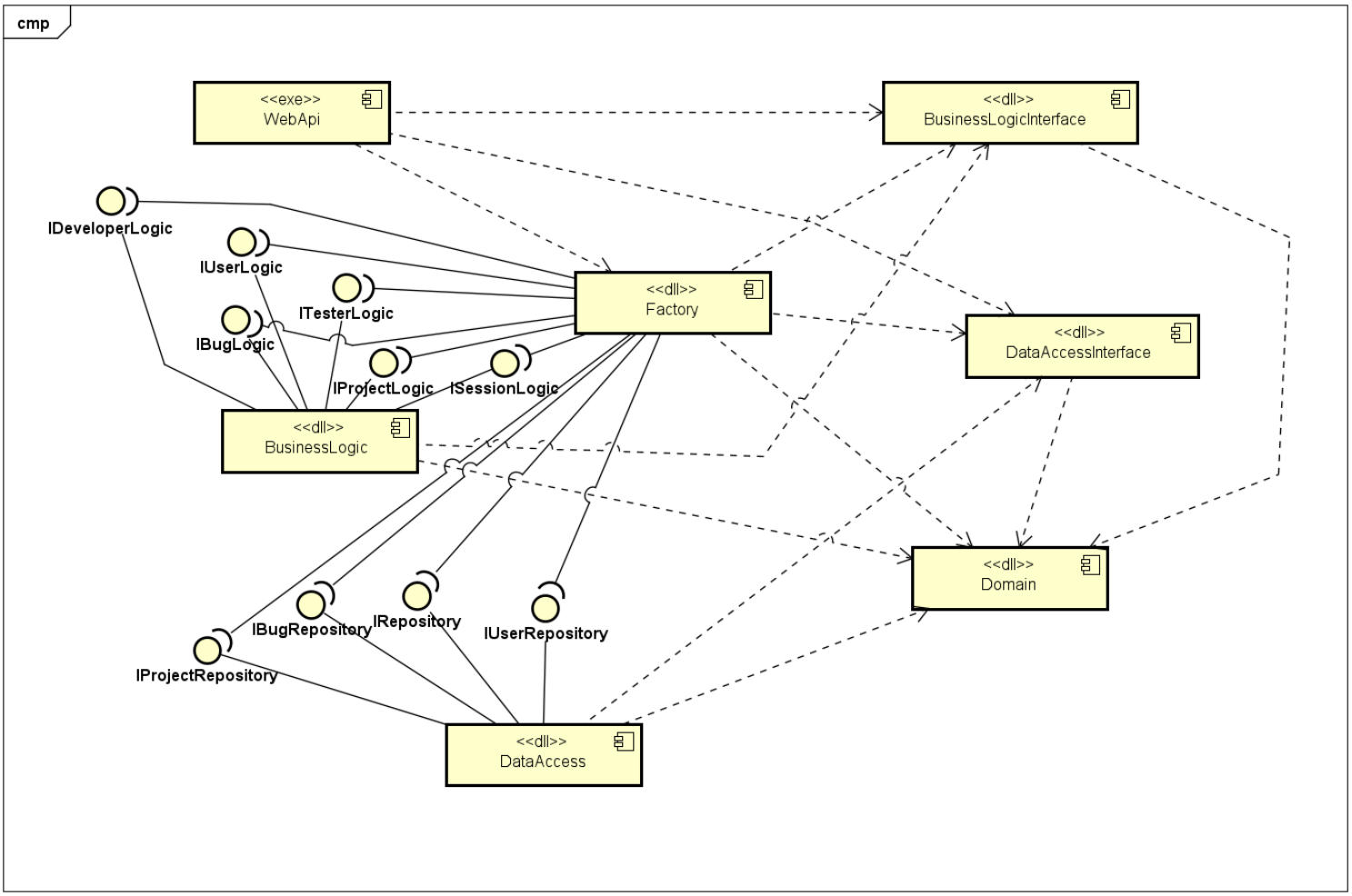
Para llevar a cabo esta funcionalidad lo que se hace es, primero el usuario debe ingresar los datos solicitados en el body del mismo, dichos datos ingresan el sistema como un modelo de entrada de proyecto.
Luego se llama a la lógica de negocio de proyecto y el mismo verifica que el proyecto a crear no exista en el sistema, validando mediante el nombre del mismo.
Una vez verificado que no exista dicho proyecto, se le crea una lista vacía de usuarios y de bugs, y se pasa el nuevo proyecto al repositorio para poder crearlo en la base de datos.

Para el diagrama se omite la autenticación del usuario.



Vista de Desarrollo

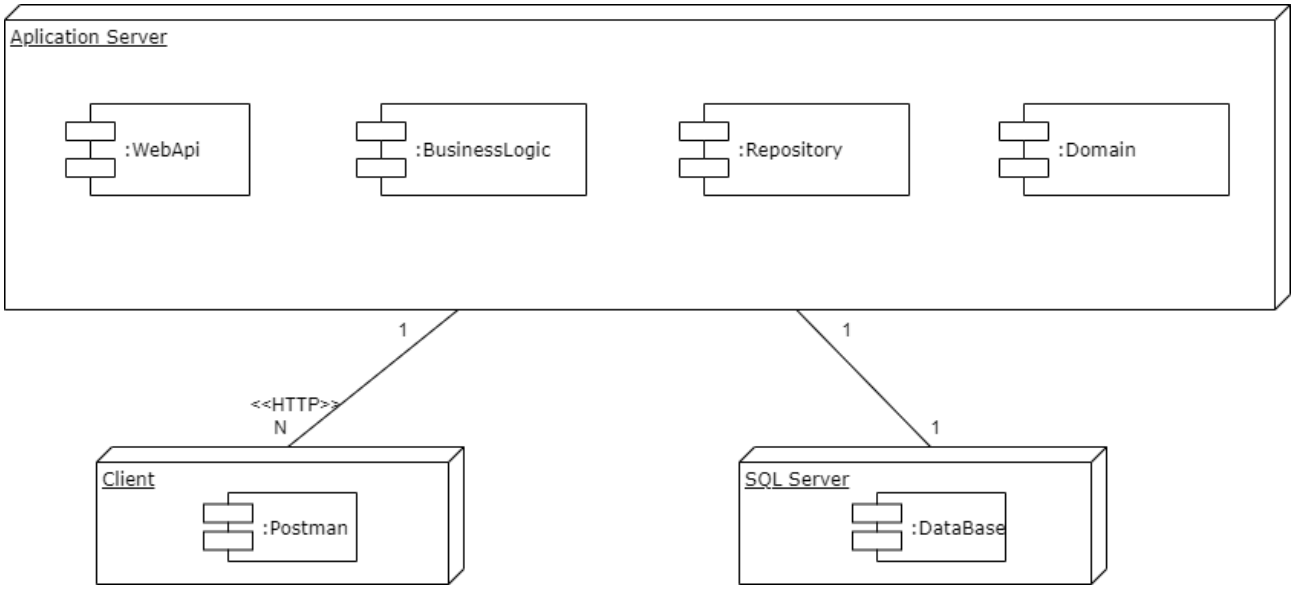
Esta vista está compuesta por el diagrama de componentes de nuestro sistema.
En el mismo se detallan los elementos o paquetes que interactúan en tiempo de ejecución.



Como se puede observar en el diagrama, BusinessLogic es el encargado de proveer las interfaces de la lógica de negocio que son requeridas por Factory, como también DataAccess es el encargado de proveer las interfaces de los repositorios que también son requeridas por Factory.

Vista de Despliegue

En esta última sección mostraremos los artefactos que utiliza nuestra aplicación en el ambiente de ejecución



Como se puede ver, nuestra aplicación servidor puede conectarse con múltiples clientes y los múltiples clientes se conectan con el único servidor, por otro lado nuestro servidor se conecta con una única Base de datos.

Modelo de base de datos