



Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2 - Obligatorio II

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

18 de Noviembre de 2021

Índice

Descripción General del trabajo/sistema

Alcance del sistema

Mejoras a tener en cuenta

Principales decisiones

Descripción y justificación de diseño BackEnd

Vista Lógica

Diagrama de clases Data Access

Diagrama de clases Domain

Diagrama de clases Business Logic Interface

Diagrama de clases Imports

Vista Proceso

Vista Desarrollo

Vista de Despliegue

Modelo de base de datos

Métricas

Abstracciones estables

Dependencias estables

Clausura Común

Reuso común

Conclusión

Extensibilidad solicitada - Imports

Resumen de las mejoras al diseño

Anexo

Justificación y diseño FrontEnd

Usabilidad

Especificación de la API - Cambios

Justificación y evidencia de TDD

Evidencia de commits

Análisis de cobertura de pruebas

Métricas - Cálculos

Especificación de la API - Completa

Descripción General del trabajo/sistema

En esta segunda instancia se desarrolló el FrontEnd del sistema previamente entregado. A su vez se implementaron nuevos requerimientos que mencionaremos a continuación.

- Manejo de tareas, se implementó una nueva entidad “Task” la cual se corresponde con un proyecto.
- Se manejan costos a nivel de usuarios con rol tester y desarrollador.
- Se maneja duración de bugs.
- Se maneja costo y duración de un proyecto, el cual es calculado dependiendo de los usuarios y bugs relacionados a dicho proyecto.
- Se maneja un nuevo mecanismo de importación de bugs el cual permite la importación de cualquier tipo de archivo a través de consumir una dll implementada por un tercero.
- Se implementó una aplicación cliente, la cual es encargada de importar bugs en formato JSON.

Por tanto, nuestro sistema es capaz de ayudar a las empresas de desarrollo de software a llevar un control sobre los bugs y tareas asociados a sus proyectos, como también los desarrolladores y testers involucrados.

Nuestro desarrollo BackEnd se llevó a cabo utilizando el framework de .Net Core 5. Por otro lado, nuestro FrontEnd fue desarrollado con angular utilizando la librería de angular-material.

Alcance del sistema

En esta primera entrega pudimos cumplir con todos los requerimientos funcionales y no funcionales especificados en la letra. Donde tampoco contamos con bugs conocidos, aunque no garantizamos la ausencia de los mismos.

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

Mejoras a tener en cuenta

- **IMPORTANTE** Nuestros endpoints que obtienen datos (por ejemplo, getBugs, getProject, etc) deben ser un único endpoint donde pasándole por parámetro el usuario logueado le traiga la información que corresponda. Dado que dependiendo el usuario logueado se le muestra los proyectos el cual está asociado así también con bugs y tareas.

Actualmente lo tenemos implementado en diferentes endpoints por lo desarrollado en la primera entrega, es decir, contamos con un endpoint para traer proyectos si soy tester, otro si soy desarrollador y otro para administrador. Esto complicó bastante al momento de mostrar los datos de proyectos, bugs y tareas dependiendo del usuario logueado ya que debemos preguntar por el rol del usuario logueado para poder realizar la request al endpoint correspondiente, además de no ser muy eficiente al momento de extendernos a otros roles dado que tendremos que estar agregando nuevos endpoints.

No pudimos corregir lo mencionado anteriormente por temas de tiempo y priorizaciones sobre los requerimientos y por tanto solucionamos dicho problema creando un controlador de usuario en nuestro FrontEnd el cual es llamado por los diferentes componentes y el mismo solo se encarga de preguntar por el rol del usuario logueado para realizar la request al endpoint correspondiente.

- Nuestros get a la base de datos de nuestros repositorios deberían solo incluir los datos de la entidad que queremos obtener y no realizar includes de otras tablas. Es decir, si estoy trayendo proyectos traer la información propia de los proyectos y no de bugs, tareas etc. Actualmente estamos trayendo toda esa información cuando obtenemos los proyectos esto nos trae problemas en temas de performance, además de que opaca la utilización de endpoints como serían obtener bugs de un proyecto, etc.

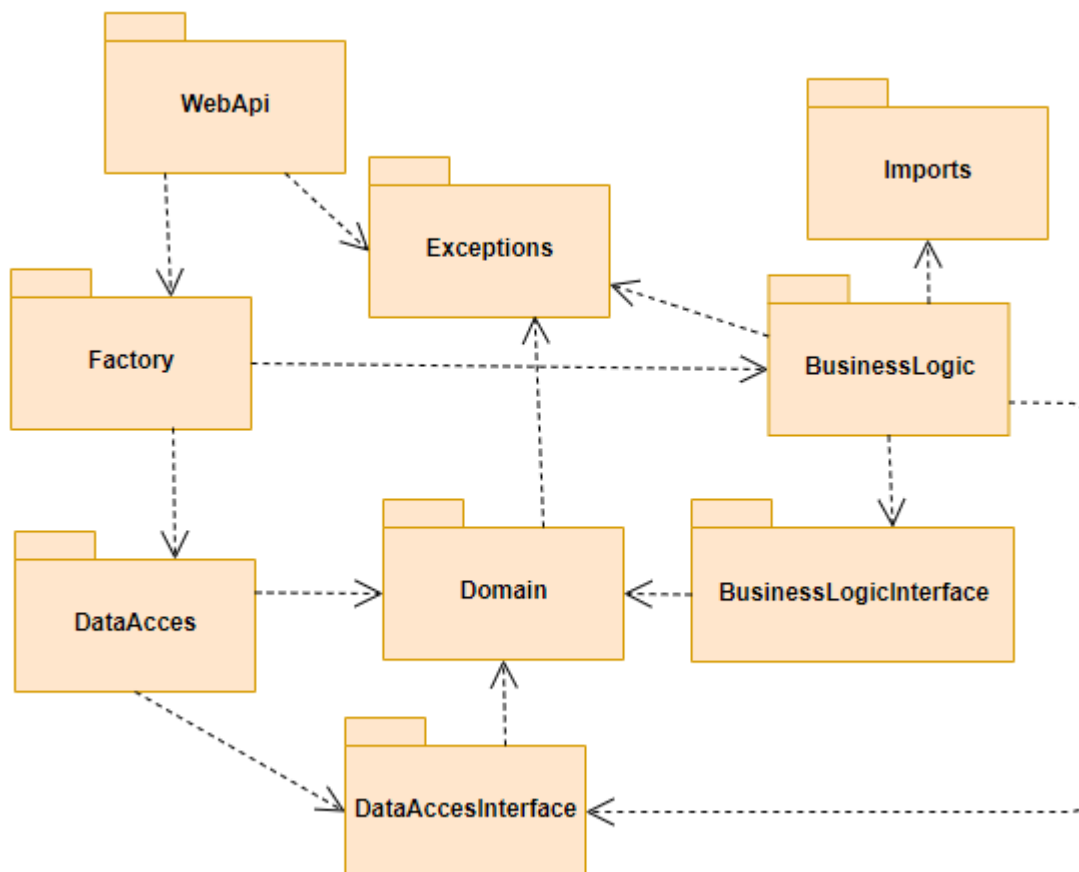
Principales decisiones

- Se quitaron los endpoints para buscar bugs por nombre y estado dado que los filtros fueron implementados del lado del FrontEnd para todas las entidades.
- No generamos reporte de proyecto ya que no vimos necesario manejar el total de bugs, el precio y la duración como información analítica del mismo por así decirlo. Estos datos los manejamos como atributos mismos del proyecto ya que creemos que un proyecto cuenta con su información de cuantos bugs tiene, su duración y su respectivo costo.
- Para realizar la importación mediante archivos de bugs se necesita ingresar el path completo de la ubicación del archivo a importar.
- Al entrar a editar un bug no se permite modificar el id del mismo ya que creemos que es un identificador único que no debe cambiar.
- Asumimos y creímos necesario que las tareas las pueden crear todos los roles del sistema.

Descripción y justificación de diseño BackEnd

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 mostrado anteriormente se observa 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.
Imports	Es el encargado de proveer una interfaz a nuestros proveedores para que puedan realizar importaciones de bugs.

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. También nos basamos en el bajo acoplamiento de GRASP.

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.

Por otro lado en nuestro diagrama se puede observar claramente que cumplimos con el Principio de Dependencias Acíclicas, es decir, el grafo de dependencias entre nuestros paquetes no forman ciclos.

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 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 IRepository (o IUserRepository, etc) para poder interactuar con la base de datos.

Siguiendo el principio Single Responsibility 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

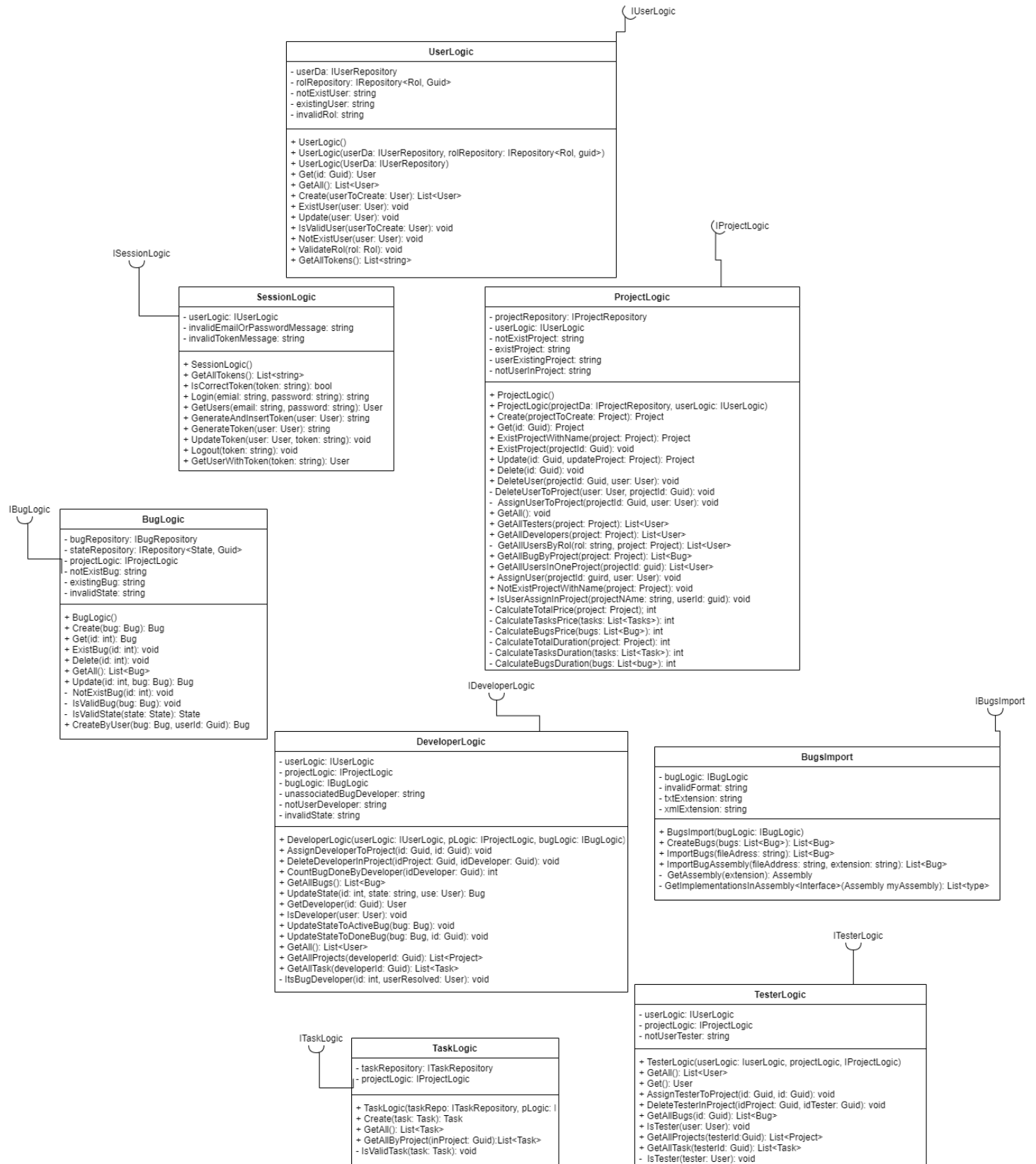


Diagrama de clases Data Access

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.

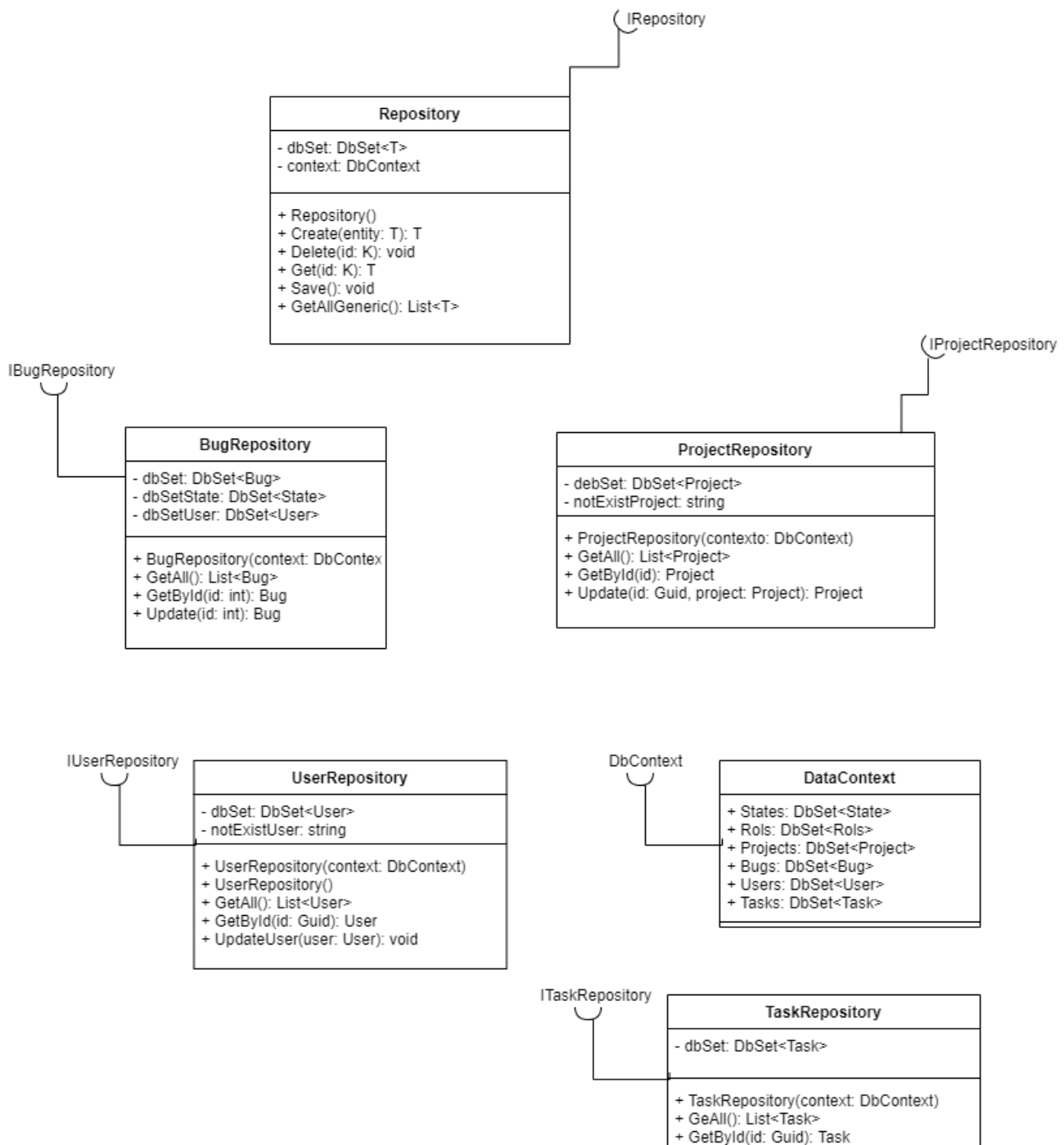


Diagrama de clases Domain

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 Business Logic Interface

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 terceros que la consuman.

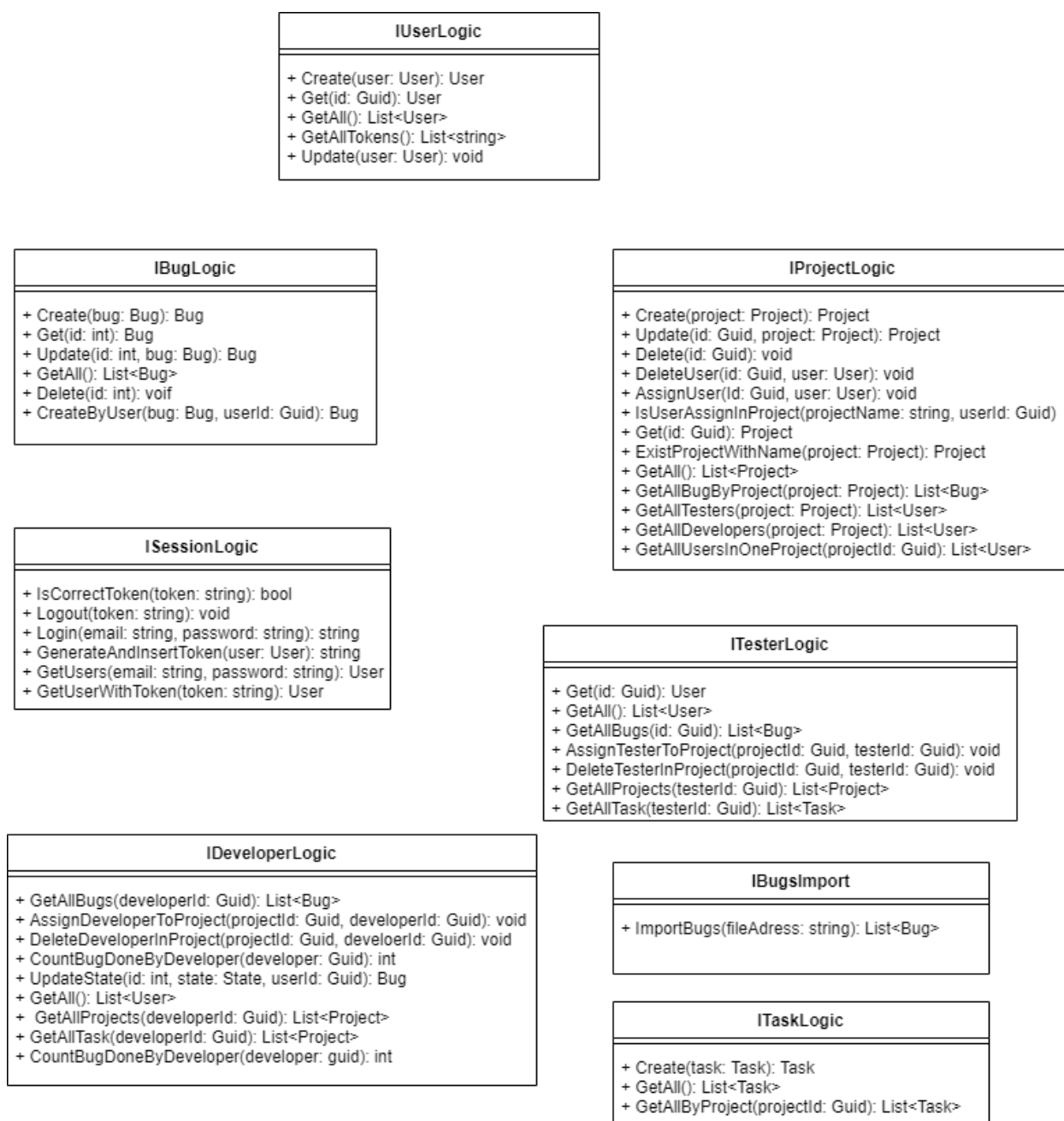
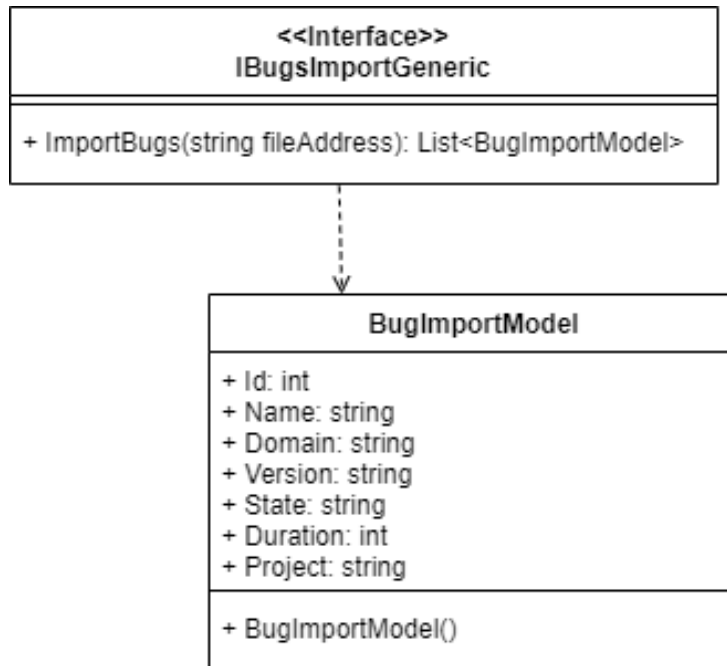


Diagrama de clases Imports

Este paquete es el encargado de exponer una interfaz y un modelo para poder realizar importaciones de bugs a través de distintos tipos de archivos.

Este paquete es consumido por terceros para que puedan realizar importaciones en el formato de archivo deseado por los clientes.



Vista 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.

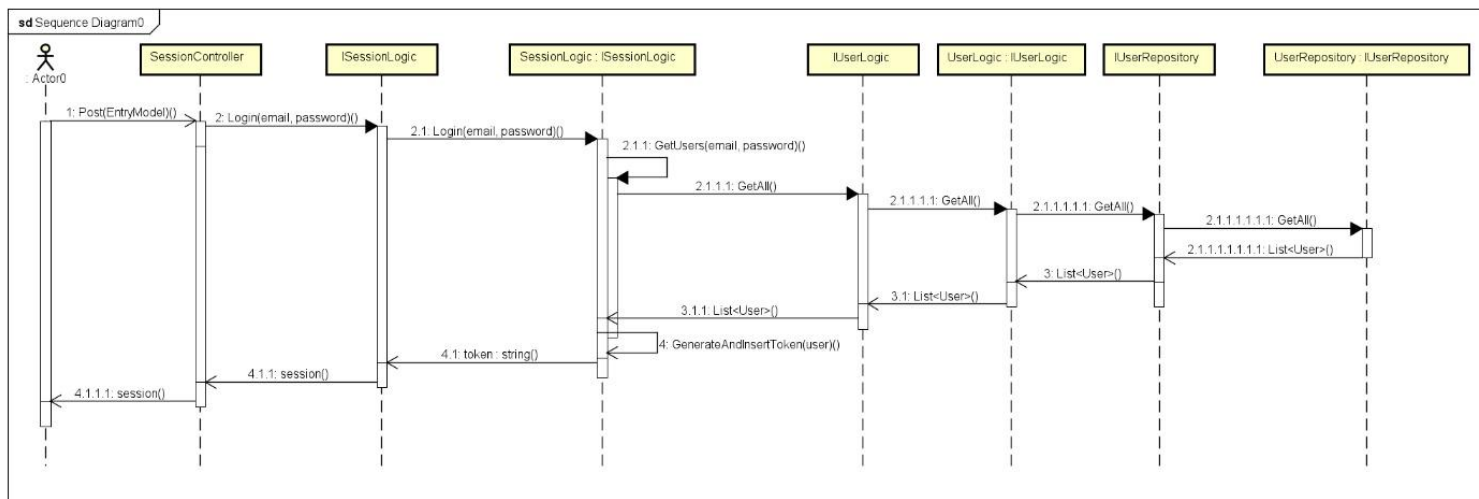
En otras palabras, la finalidad de esta sección es poder brindarle al usuario lector una forma rápida de entender el funcionamiento general de las determinadas funcionalidades del sistema.

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 que 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.



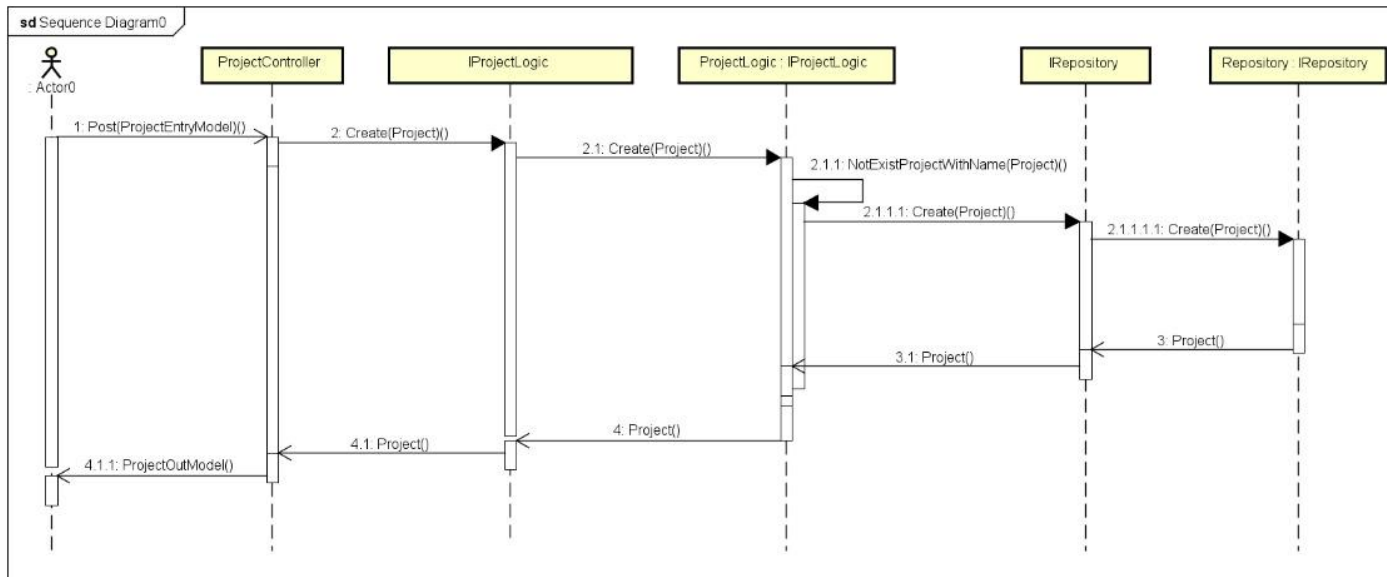
Crear un proyecto en el sistema

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.

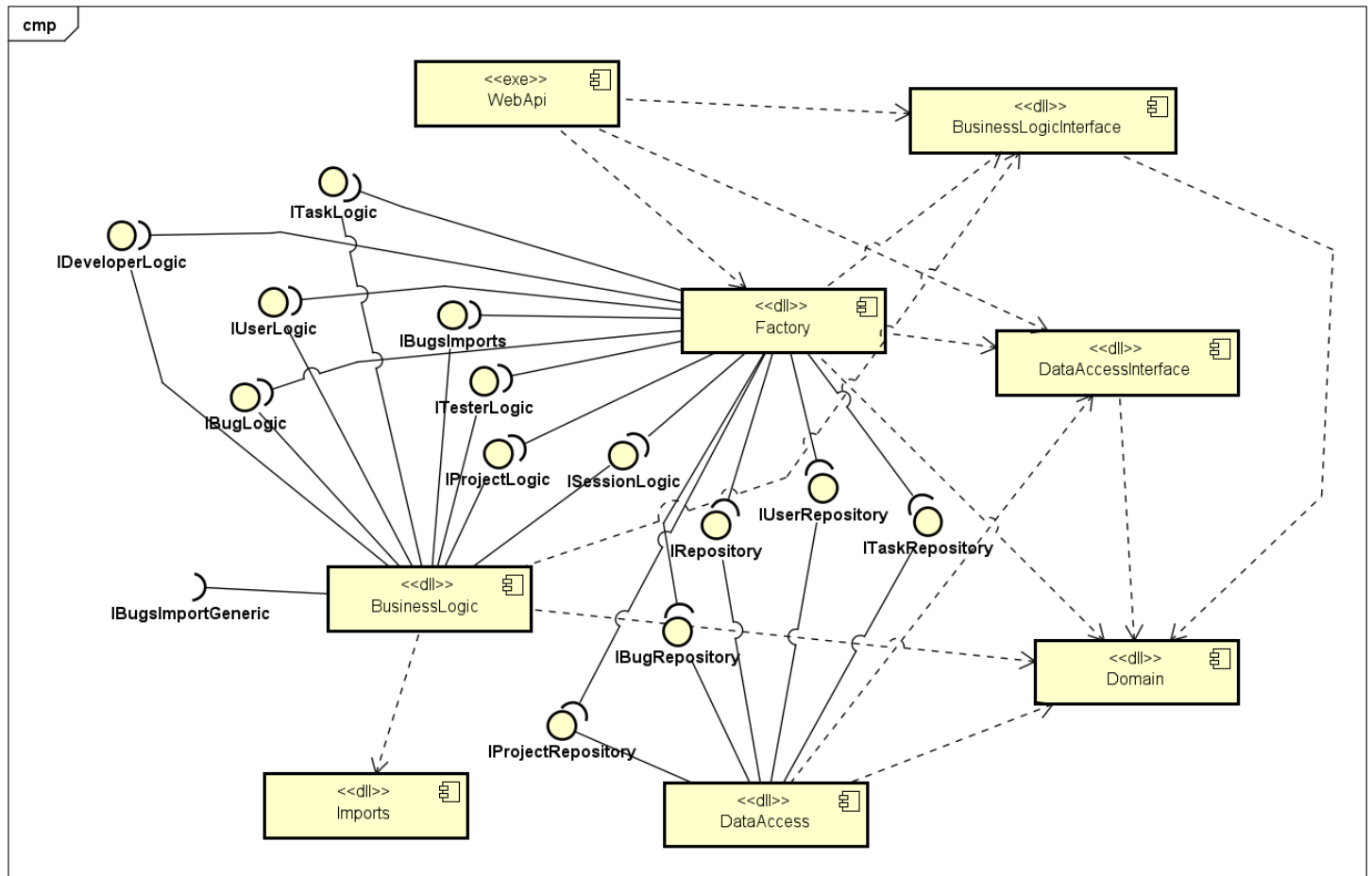
Una vez verificado que no exista dicho proyecto, se le crea una lista vacía de usuarios y de bugs, y se pasa el nuevo proyecto al repositorio para poder crearlo en la base de datos.

Para el diagrama se omite la autenticación del usuario.



Vista Desarrollo

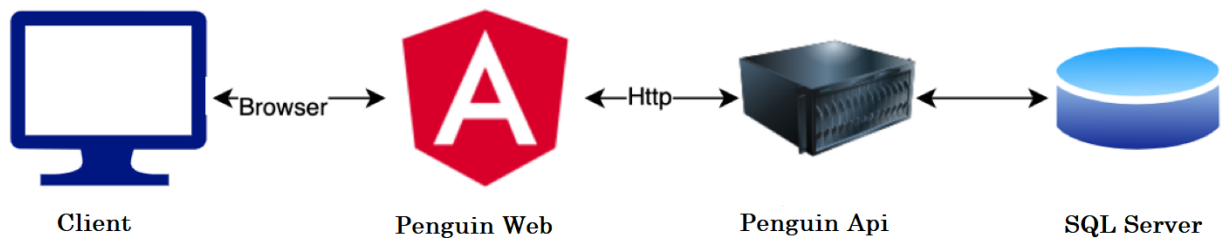
Esta vista está compuesta por el diagrama de componentes de nuestro sistema.
En el mismo se detallan los elementos o paquetes que interactúan en tiempo de ejecución.



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. Por otro lado BusinessLogic también se encarga de requerir la interfaz de IBugImportGeneric la cual la misma es provista por terceros externos a nuestro sistema, consumiendo sus implementaciones a través de los ensamblados cargados.

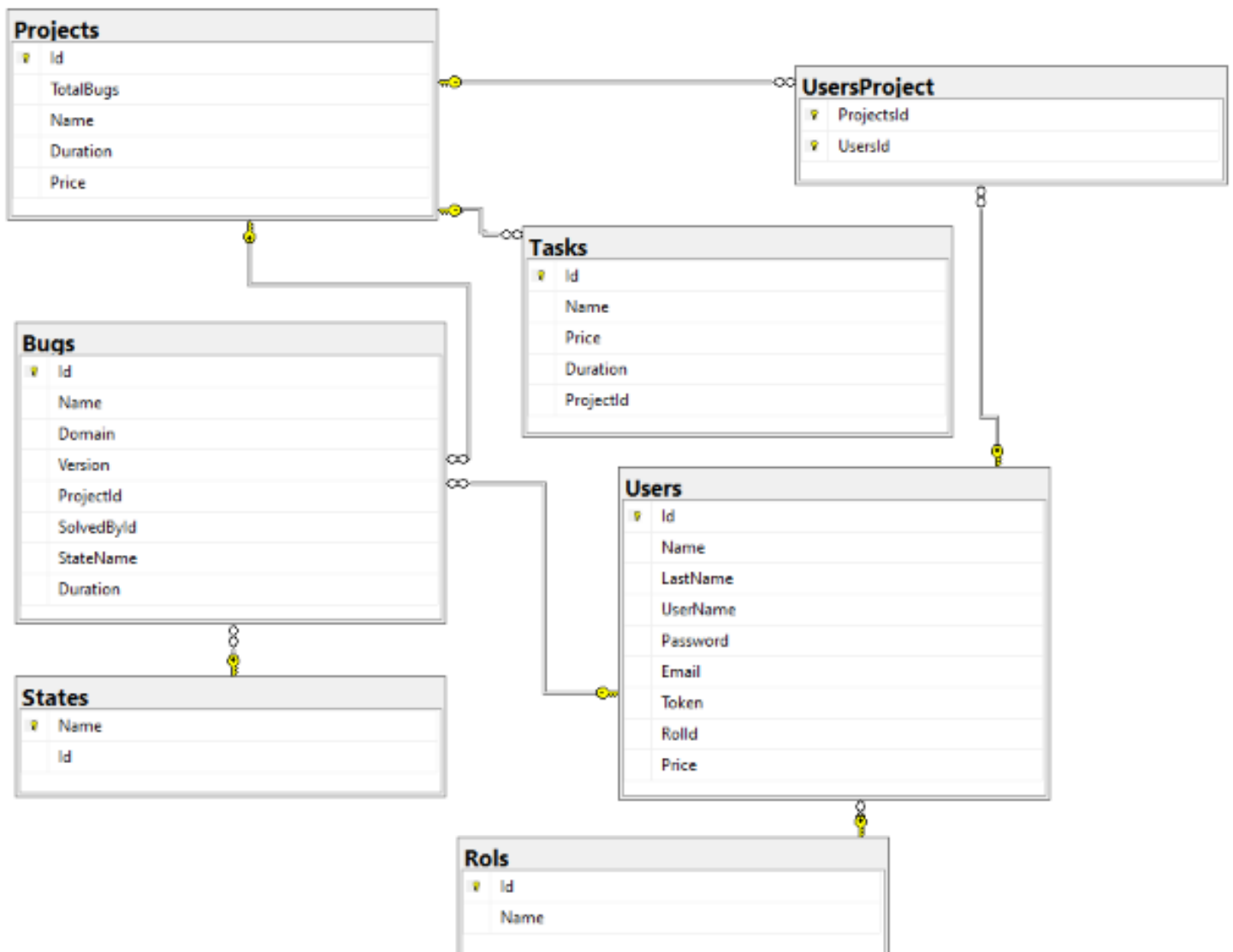
Vista de Despliegue

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



Como se puede ver en la imagen, el cliente se comunica con la Web, dicha Web se comunica con la Api y la Api con el SQL, luego de llegar al servidor, el mismo responde la petición por el usuario hasta ser mostrada en la Web.

Modelo de base de datos



Métricas

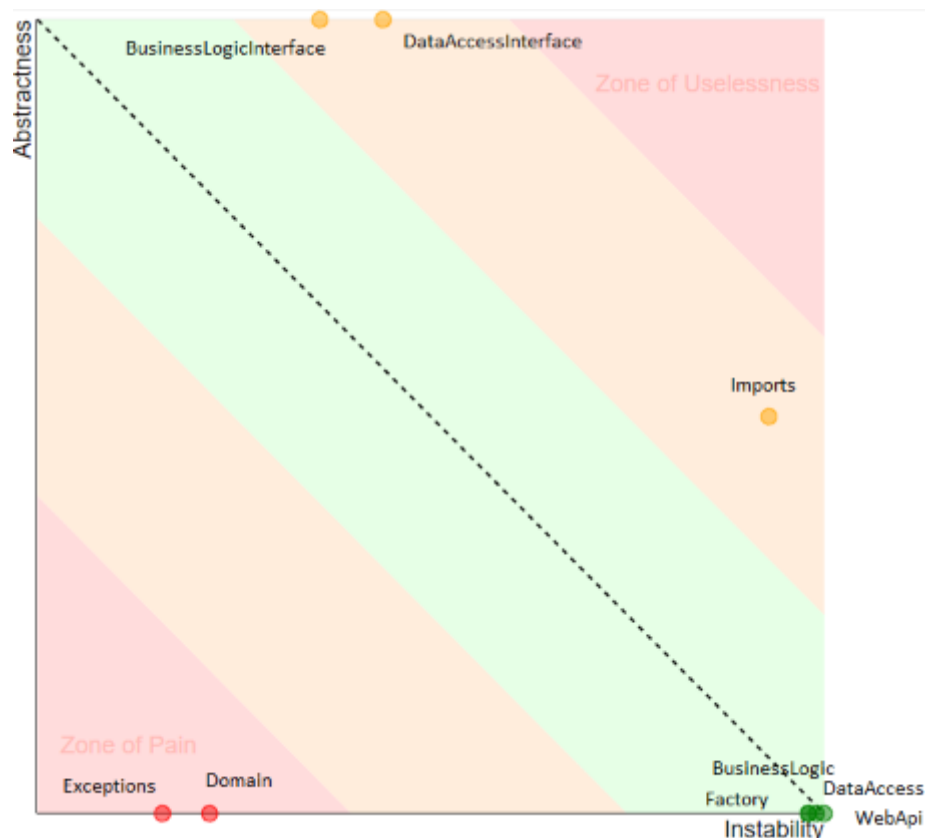
Las mismas son una medida cuantitativa que sirven para medir la calidad de nuestros sistemas. En esta sección hablaremos de métricas a nivel de paquetes las cuales son de Inestabilidad, Abstracción, Distancia' y cohesión relacional.

Para obtener dichas métricas utilizamos la herramienta NDepend, por tanto los cálculos realizados no fueron hechos a mano pero de igual manera recordamos cómo sería la realización de los mismos. De igual manera aclaramos cómo sería el cálculo de los mismos en la sección de "[Métricas - Cálculos](#)"

Paquete	(I)	(A)	(D')	(H)	Observaciones
WebApi	1	0	0	1.64	Con respecto a la inestabilidad obtuvimos un valor = 1, lo cual quiere decir que nuestro paquete está muy inestable ya que depende de otros paquetes. Con respecto a la Abstracción obtuvimos un valor = 0, lo cual indica que dicho paquete es concreto. Con respecto a la distancia obtuvimos un valor = 0, lo cual es positivo ya que no es un paquete estable y tampoco abstracto. Con respecto a la Cohesión, obtuvimos un valor 1,64 lo cual es superior a 1,5, lo que nos indica de que dicho paquete tiene una buena cohesión relacional.
BusinessLogicInterface	0,36	1	0,25	0,12	Con respecto a la inestabilidad obtuvimos un valor cercano a 0 (0,36) lo cual nos indica que dicho paquete es poco estable porque no depende de muchos otros paquetes. Con respecto a la Abstracción obtuvimos un valor = 1, lo que nos indica que dicho paquete es abstracto. Con respecto a la distancia obtuvimos una valor cercano a 0 (0,25) lo cual nos indica que el paquete es bastante concreto y estable. Con respecto a la Cohesión relacional obtuvimos una valor cercano a 0 (0,12) la cual no es aceptable pero es esperable a ser un paquete de interfaces.
BusinessLogic	0,98	0	0,01	0,3	Dicho paquete es inestable dado que depende mucho de otros paquetes y a su vez no es abstracto ya que el mismo está muy acoplado a las restricciones del negocio y a su dominio. Con respecto a la Cohesión relacional tenemos un valor muy bajo lo cual no es aceptable. De igual manera creemos que estos resultados son debido a que nuestras clases del paquete no se relacionan entre sí directamente sino que lo hacen a través de sus interfaces las cuales se encuentran en otro paquete.
Domain	0,22	0	0,55	1,67	- (I) Bastante estable porque tiene un valor cercano a 0, lo cual no depende de muchos paquetes. - (A) Es un paquete concreto ya que tiene un valor = 0. - (D) Tenemos un valor = 0,55 lo cual quiere decir que el paquete está en la zona de dolor y la zona de poca utilidad (Secuencia principal) pero esto lo vemos correcto al estar fuertemente

					<p>acoplado a las entidades del dominio.</p> <p>- (H) Tenemos un valor aceptable (rango 1,5 - 4,0).</p>
DataAccess	0,99	0	0,01	0,67	<p>- (I) Muy inestable porque tiene un valor casi = 1, lo cual indica que depende de muchos otros paquetes.</p> <p>- (A) Es un paquete concreto ya que tiene un valor = 0.</p> <p>- (D) Tenemos un valor = 0,01 lo cual es positivo.</p> <p>- (H) Tenemos un valor muy por debajo del 1,5 que es mínimo valor para que el paquete sea bueno. Esto es dado que nuestros repositorios y configuraciones con fluent api son independientes entre sí.</p>
DataAccessInterface	0,44	1	0,31	1	<p>- (I) Tenemos una valor = 0,44, lo cual quiere decir que el paquete está en el medio de estable e inestable.</p> <p>- (A) Es un paquete abstracto ya que tiene un valor = 1.</p> <p>- (D) Tenemos un valor = 0,31 lo cual quiere decir que el paquete está en el límite de la zona de dolor y la zona de poca utilidad (Secuencia principal) ya que está entre el medio de 0 y 1..</p> <p>- (H) Tenemos un valor muy por debajo del 1,5 que es mínimo valor para que el paquete sea bueno. Esto es esperado para paquetes, como es el caso, de interfaces.</p>
Factory	0,98	0	0,02	0,5	<p>- (I) Muy inestable porque tiene un valor casi = 1, lo cual indica que depende de muchos otros paquetes.</p> <p>- (A) Es un paquete concreto ya que tiene un valor = 0.</p> <p>- (D) Tenemos un valor = 0,02 lo cual quiere decir que el paquete está en la zona de dolor, quiere decir que es concreto y estable por lo tanto no es extensible.</p> <p>- (H) Tenemos un valor muy por debajo del 1,5 que es mínimo valor para que el paquete sea bueno. Esto es esperado a paquetes que realizan las inyecciones de dependencia.</p>
Exceptions	0,16	0	0,6	0,33	<p>Con los valores obtenidos somos concientes que nuestro paquete es:</p> <p>- (I) Bastante estable porque tiene un valor cercano a 0, lo cual no depende de muchos paquetes.</p> <p>- (A) Es un paquete concreto ya que tiene un valor = 0.</p> <p>- (D) Tenemos un valor = 0,6 lo cual quiere decir que el paquete está en el límite de la zona de dolor y la zona de poca utilidad (Secuencia principal) ya que está entre el medio de 0 y 1</p> <p>- (H) Tenemos un valor por debajo del 1,5 que es mínimo valor para que el paquete sea bueno. Pero se justifica dado que son paquetes de excepciones que sus clases no tienen relación alguna.</p>
Imports	0,88	0,5	0,27	1	<p>- (I) Inestable porque tiene cerca de 1, lo cual indica que depende de muchos otros paquetes.</p> <p>- (A) Es un paquete que está en el medio de ser abstracto y concreto ya que tienen un valor = 0,5.</p> <p>- (D) Tenemos un valor = 0,27 lo cual quiere decir que el paquete está en la zona de dolor, quiere decir que es concreto y estable por lo tanto no es extensible.</p> <p>- (H) Tenemos un valor por debajo del 1,5 que es mínimo valor para que el paquete sea bueno.</p>

Abstracciones estables



Este principio nos dice que nuestros paquetes deben ser lo más abstractos posibles si su estabilidad es alta. Si contamos con paquetes estables y concretos es peligroso dado que al ser concreto el paquete tiende a cambiar frecuentemente y cómo es estable otros paquetes están dependiendo de él por lo cual el cambio de nuestro paquete impactaran en los paquetes que lo necesitan.

Por otra parte si tengo un paquete inestable y abstracto no tiene sentido ya que al ser inestable nadie depende de él y al ser abstracto tiene muchas interfaces y clases abstractas que nadie utilizaría.

Los paquetes de nuestro sistema que se encuentran en la zona de dolor son solo los paquetes esperados, los paquetes de Exceptions y Domain, que son entidades y excepciones.

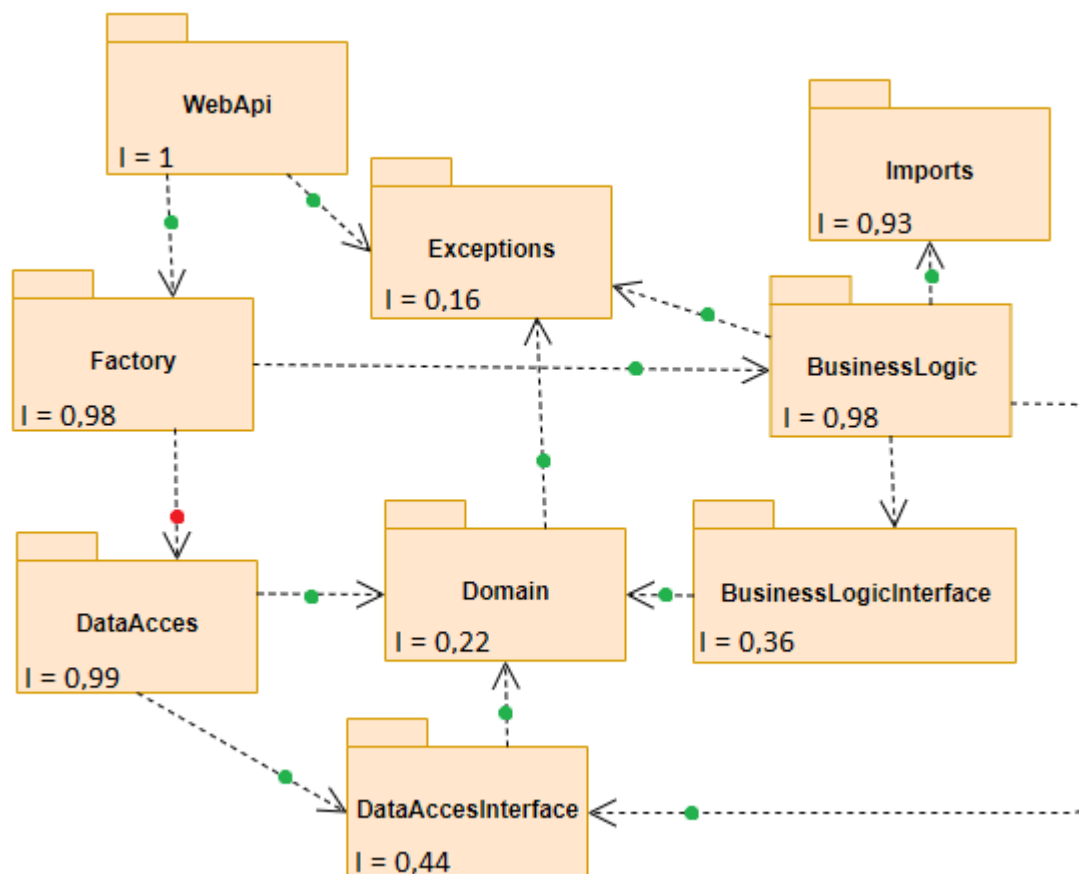
Las mismas son muy estables dado que no dependen de nadie y todos de él y en el caso de Domain no es abstracto dado que el paquete está altamente acoplado con el negocio.

Volviendo al paquete de Exceptions el mismo es muy abstracto y no como se muestra en la gráfica dado que solo contiene excepciones utilizadas en nuestro sistema y las mismas son bastantes genéricas.

También observando la gráfica nuestros paquetes más abstractos que son nuestras interfaces (BusinessLogicInterface y DataAccessInterface) se encuentran más cerca de la estabilidad que de la inestabilidad por tanto entendemos que las mismas cumplen con este principio. Al igual que los paquetes de Factory, BusinessLogic, DataAccess y WebApi los mismos son muy inestables pero son concretos.

Por último, nuestro paquete de Imports debería ser más abstracto y menos inestable dado que dicho paquete va a ser consumido por terceros. Deberíamos realizar un estudio más detallado de por qué nuestras métricas dieron estos resultados ya que creemos que nuestro Imports es uno de nuestros paquetes más estables y abstractos de la aplicación ya que él no depende de nadie y solo contiene una interfaz.

Dependencias estables



- Dependencia correcta
- Dependencia incorrecta

Nuestro sistema cumple con SDP a pesar de que tenemos una dependencia hacia un paquete menos estable como es el caso de **Factory** con **DataAcces**. El mismo creemos que es irrelevante dado que nuestro **Factory** solo depende de otros paquetes para realizar la inyección de dependencia por tanto si **DataAccess** cambia **Factory** no va a resultar afectado.

En conclusión nuestra aplicación cumple con el principio de dependencias estables ya que nuestros paquetes menos estables dependen de paquetes más estables.

Clausura Común

Dicho principio se basa en que las clases que cambian juntas deben pertenecer al mismo paquete, para así minimizar el impacto de cambio en la aplicación.

Nuestro sistema cumple con Clausura Común dado que nuestros paquetes cuentan con una única responsabilidad (“SRP a nivel de paquetes”). Aunque esto es contradictorio.

Nuestros paquetes tienen entidad por “acciones o áreas”, es decir, nuestro paquete BusinessLogic es el encargado de manejar toda la lógica/restricciones del negocio, nuestro DataAccess se encarga del acceso a la base de datos, y así con el resto.

Por tanto si queremos realizar cambios de restricciones de negocio las clases a cambiar afectan al mismo y único paquete, BusinessLogic.

En cambio sí quiero realizar cambios más bruscos, como por ejemplo, agregar un nuevo atributo en mi dominio a la entidad Bug realizando cálculos de datos con ese nuevo atributo y además que puede ser accedido a dicha información calculada desde un endpoint estaríamos cambiando más de un paquete, en este caso BusinessLogic, Domain y WebApi.

De igual manera concluimos que nuestro sistema cumple con clausura común.

Ya que el ejemplo brindado, que no lo cumple, estaría violando varios atributos de calidad de las métricas así como también patrones y principios de diseño.

Por tanto es más lo que dañamos que lo que aportamos para poder cumplir dicho principio.

Reuso común

Nuestra aplicación cumple con el principio de reuso común dado que cada paquete tiene su responsabilidad, desde acceder a nuestra base de datos, manejar los endpoints, restricciones de negocio, etc.

Por tanto si quiero realizar cambios de restricciones de negocio solo debería compilar dicho paquete modificado dado que no hay otro paquete que comparta dicha responsabilidad.

Aunque dependiendo del enfoque del cambio estaríamos cumpliendo o no con este principio. Dado que si quiero realizar cambios sobre una entidad del negocio, ejemplo Bugs, no tengo un único paquete que sea responsable de todas las acciones relacionadas al mismo como pueden ser el acceso a datos de bugs, los endpoints que reciben request a dicha entidad, restricciones de negocio de la misma, etc.

La idea de tener un paquete por entidad no fue considerada, ya que violaremos los atributos de calidad obtenidos de las métricas (I, A, H, D') sin mencionar a los patrones y otros principios de diseño.

Conclusión

Concluimos que nuestro sistema es sumamente mantenible dado que cumplimos con el principal principio que es Clausura Común y también permitimos el reuso ya que también cumplimos con el principio de Reuso Común. Así como también de cumplir con dependencias estables que es sumamente importante al momento de realizar cambios. También nuestro sistema evaluado en abstracciones estables el único paquete encontrado en la zona de peligro es el esperado, nuestro dominio.

En cuanto a nuestra d' no tuvimos valores cercanos al 1 (cerca de la zona de dolor) y casi todos estaban cerca del $d'=0$ a excepción de los paquetes particulares de domain y exception. Por tanto contamos con paquetes “concretos e inestables” y “abstractos y estables” que es lo que todos esperamos en nuestras aplicaciones.

Extensibilidad solicitada - Imports

Para esta segunda instancia se requirió poder importar bugs a través de cualquier formato de archivos.

Estas implementaciones para poder importar bugs en cualquier tipo de formato queda a cargo de nuestros proveedores. Los mismos serán encargados de realizar una aplicación que sea capaz de leer el archivo del formato deseado, dicha información levantada por la aplicación de nuestro proveedor deberá ser consumida por nuestra propia aplicación.

Para poder implementar una óptima solución del problema mencionado anteriormente utilizamos el patrón reflection.

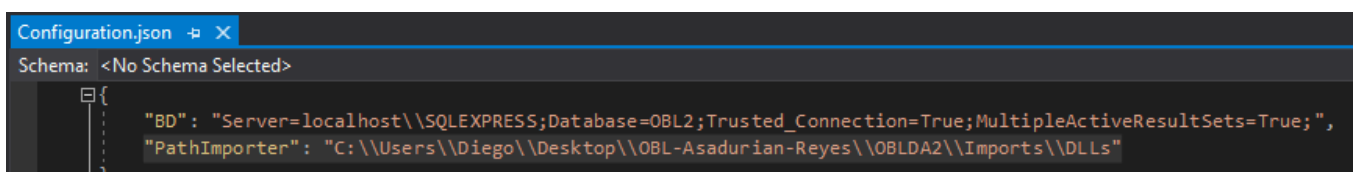
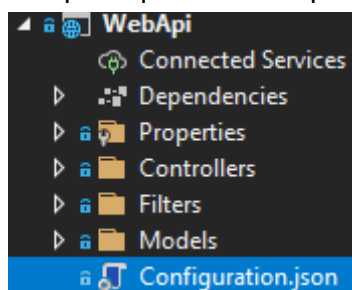
Nuestra implementación se basa en un nuevo paquete, el cual denominamos "Imports", en el cual contamos con una interfaz y un modelo (modelo de bug) el cual nuestros clientes van a consumir (a través de la generación de la dll de nuestro paquete).

Por tanto si un nuevo cliente desea importar bugs en un nuevo formato le brindaremos la dll de nuestro paquete Imports para que pueda utilizar el modelo de bug a dar de alta y además proveer una interfaz que deberá implementar.

Luego de que nuestro cliente implemente su aplicación deberá generar una dll de su aplicación para que nuestro sistema lo consuma a través del assembly. Por tanto nuestra aplicación podrá utilizar el código de nuestro proveedor y poder realizar los imports de bugs sin tener que modificar ni compilar nuestra aplicación, dado que todo esto sucede en tiempo de ejecución.

Notas importantes:

- Los archivos dll deben contener en su nombre el formato de archivo que está importando, es decir, si estamos importando bugs a través de formato CSV el nombre debe ser de la siguiente manera *CSV*.dll (es decir debe contener en su nombre, no importa donde, el nombre del formato que está importando).
- Le brindamos un modelo de bug al cliente para no brindarle información de la estructura/entidades de nuestra aplicación.
- Los dll de los proveedores deberán ser copiados en la ruta que configuremos en nuestro archivo de configuración para que nuestra aplicación pueda acceder..



Resumen de las mejoras al diseño

A partir de los cambios solicitados en la segunda entrega se detallarán cómo fueron implementados y como se llevaron a cabo.

1. **Tarea**, para poder llevar a cabo esta nueva funcionalidad lo que hicimos fue crear en el paquete dominio una nueva clase llamada Task, la cual contiene los atributos especificados por letra. Luego en la clase Project del dominio se le agregó un atributo tasks de tipo List<Task>, ya que cada proyecto tiene sus tareas.

Para seguir la misma lógica y cumplir con los patrones mencionados en la entrega uno, es que se implementó en el paquete BusinessLogic una nueva clase llamada TaskLogic, la cual es la encargada de manejar toda la lógica de la misma. También siguiendo la guía de desarrollo del OBL1, hicimos en el paquete BusinessLogicInterface, la clase ITaskLogic, la cual dicha interfaz permite que otras clases puedan instanciarla e utilizar sus métodos de forma correcta sin acoplarnos directamente a la lógica.

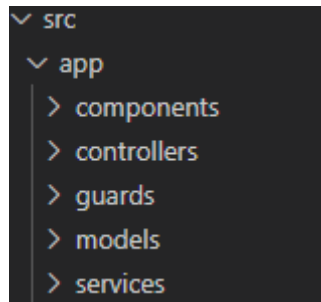
Por otro lado para continuar cumpliendo los principios de REST es que decidimos crear un nuevo controller TaskController, con sus correspondientes EndPoints los cuales permiten crear una nueva Task, obtener todas las Task y obtener todas las Task de un proyecto. Todos los métodos necesarios para este nuevo controller como para su lógica fueron implementados utilizando TDD.

2. **Valor hora**, para llevar a cabo este requerimiento se agregó en la clase User del Dominio un nuevo atributo llamado Price, el cual se va a encargar de guardar el valor costo/Hora del Tester como del Desarrollador.
3. **Extensibilidad importadores**, para llevar a cabo este requerimiento como comentamos al principio del documento se implementó utilizando reflection para poder lograr que las nuevas implementaciones puedan ser agregadas en tiempo de ejecución a la aplicación, sin necesidad de detener la ejecución de la misma. En caso de entrar más en detalle de la implementación revisar la sección [“Extensibilidad solicitada - Imports”](#).
4. **Incidentes (Bugs)**, para llevar a cabo este nuevo requerimiento lo que se hizo fue agregar un nuevo atributo a la clase Bugs del dominio, el cual se denominó “Duration” él mismo especifica la duración en horas que llevo solucionar dicho bug. Por otro lado en la lógica de ProjectLogic se implementó un método llamado “CalculateBugsDuration” él cual se encarga de especificar la duración total en horas de una cantidad determinada de bugs. Estos ajustes cumplieron con OCP dado que nunca tuvimos que realizar cambios en nuestro código sino que agregamos nuevo a nuestras clases correspondientes.

Anexo

Justificación y diseño FrontEnd

Nuestro diseño del FrontEnd está compuesto principalmente por componentes, controladores, guardas, modelos y servicios. Cada uno está separado por carpetas, donde cada carpeta tiene, por así decirlo, una única responsabilidad.



Por otro lado nuestra aplicación cuenta con dos módulos de rutas, uno de ellos es el principal “app-routing.module” el cual se encarga de navegar a “dashboard” y carga todos sus hijos, y también se encarga de navegar al “Login”.

El otro módulo de ruta que manejamos es el “dashboard-routing” el cual se encarga de navegar por todos sus componentes hijos.

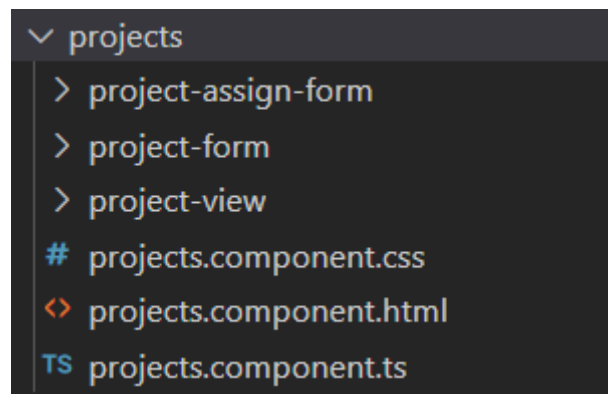
Components

El mismo está compuesto por 4 carpetas.

1. Nuestro dashboard la cual contiene un módulo para realizar routing sobre sus componentes y 5 carpetas de bugs, projects, start, tasks y users.
 - a. Bugs, contiene todos los componentes relacionados a las distintas acciones/pantallas de dicha entidad.
 - b. Projects, contiene todos los componentes relacionados a las distintas acciones/pantallas de dicha entidad.
 - c. Start, es a dónde se dirige nuestro sistema al haber ingresado a nuestro sistema. Prácticamente como un home que solo se ingresa una vez, es decir, únicamente luego de iniciado sesión.
 - d. Tasks, contiene todos los componentes relacionados a las distintas acciones/pantallas de dicha entidad.
 - e. Users, contiene todos los componentes relacionados a las distintas acciones/pantallas de dicha entidad.
2. Nuestro login el cual cuenta con dicho componente.
3. Nuestro navbar el cual cuenta con el componente de menú.
4. Nuestro shared que cuenta con una clase constants y un módulo shared.module el cual contiene todas las importaciones de angular material de los componentes que fueron utilizados en el sistema. Dicho modulo es importado en dashboard.module y app.module para poder consumir todos los componentes importados en el

SharedModule.

Algo importante a destacar es que cada “Funcionalidad” tiene su propio componente. Tratamos de dividir los componente como en dos secciones, aquellos que únicamente muestran datos como por ejemplo las tabla de Users, Projects, Bugs, Task, son llamados “nombreComponente”.component.ts, y aquellos componentes que brindan al usuario un formulario donde ingresar datos son llamados “nombreComponente”-form.component.ts. Aquellas funcionalidades como actualizar, decidimos utilizar el mismo componente del form, para evitar duplicar código, para ello utilizamos el componente de Material “Dialog”, el cual nos permite enviar data al componente y poder cargarla en el html y así poder actualizarla.



En esta imagen estamos viendo la carpeta del componente projects, la idea de esta imagen es mostrarles la nomenclatura de los nombres que se comentó anteriormente. Como se ve “projects.component” es donde se muestra toda la información de projects, en este caso se muestra una tabla, y como se puede ver los demás componentes son un formulario “project-form” donde el cliente crea su nuevo proyecto, y el otro “project-assign-form” donde el usuario asigna usuarios al proyecto.

Controllers

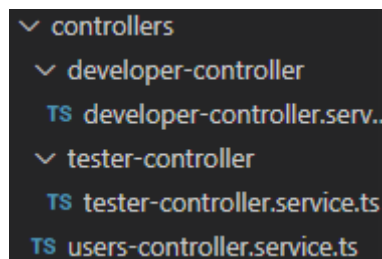
Tal como mencionamos en la sección de [“Mejoras a tener en cuenta”](#) dicha carpeta controllers, la cual contiene un controlador por rol de usuario, nace para solventar el problema de no tener endpoints con parámetros en las request de tipo get.

Dicha carpeta “Controllers” está compuesta por un servicio (users-controller.service.ts) que es utilizado por todos nuestros componentes que necesitan mostrar información dependiendo del usuario logueado.

Por ejemplo, si estoy en el componente de proyectos, el mismo debe mostrar los proyectos que están asignados al usuario logueado y dependiendo del rol del usuario llamar a determinado endpoint.

Por tanto nuestro servicio es el encargado de verificar el usuario logueado y su rol para realizar la request al endpoint correspondiente y devolver la información correcta al componente necesario.

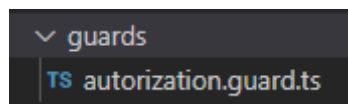
Nuestro “users-controller.service.ts” además se apoya de tester-controller.service.ts y de developer.controller.service.ts.



```
▼ controllers
  ▼ developer-controller
    TS developer-controller.serv.
  ▼ tester-controller
    TS tester-controller.service.ts
    TS users-controller.service.ts
```

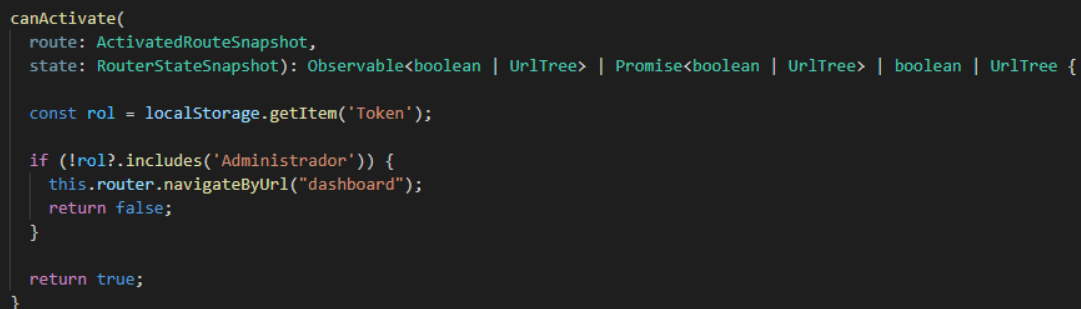
Guards

Dicha carpeta contiene las guardas, en este caso nosotros definimos una única guarda ya que consideramos que al tener únicamente 3 roles, y las diferencias de accesibilidad entre ambos son muy escasas nos pareció una buena práctica manejar una sola guarda.



```
▼ guards
  TS autorization.guard.ts
```

Dicha guarda lo que haces es verificar únicamente si el token del usuario logueado incluye o no la palabra administrador, esto quiere decir que si el que está logueado es un administrador, se le permitirá acceder a determinadas rutas que los demás roles no pueden.



```
canActivate(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

  const rol = localStorage.getItem('Token');

  if (!rol?.includes('Administrador')) {
    this.router.navigateByUrl("dashboard");
    return false;
  }

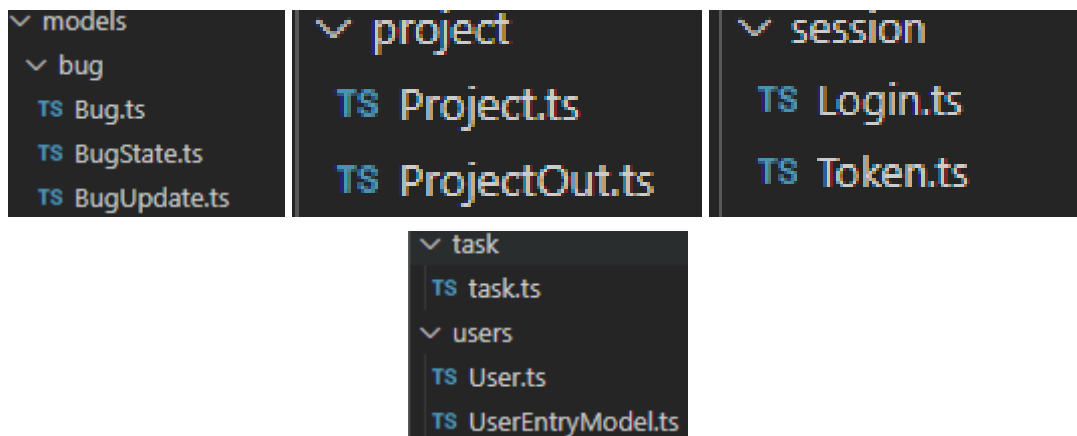
  return true;
}
```

Como se puede observar en la imagen siguiente hacemos uso de dicha guarda, lo que estamos haciendo ahí, es redireccionar al usuario a /users si el que está logueado es un administrador.

```
{
  path: 'users',
  canActivate: [AuthorizationGuard],
  component: UsersComponent
},
```

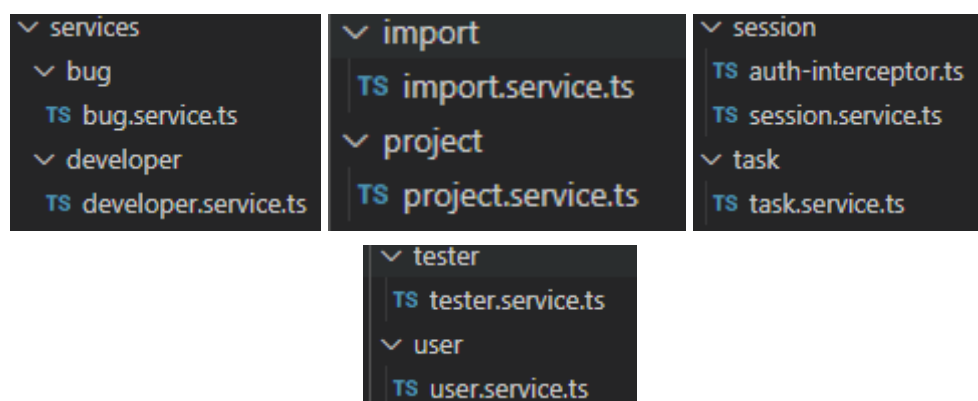
Models

Dicha carpeta contiene los modelos de bug, project, session, task y users que necesita nuestro FrontEnd para hacer request a nuestro backend y manejar la información de nuestros datos.



Services

Dicha carpeta contiene los servicios necesarios para realizar las requests a nuestros endpoints.



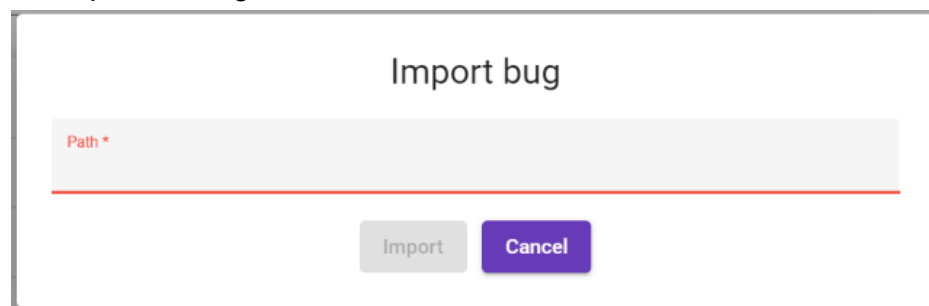
Usabilidad

Para llevar a cabo este sistema utilizamos la librería de angular material, ya que la misma nos brinda un sin fin de funcionalidades ya pre establecidas, como también brinda un diseño atractivo y fácil de usar para el usuario. Algo a destacar es que para usar la librería no se necesitan conocimientos sólidos en CSS, ya que los componentes de material los tienen pre establecidos, es por ello que también decidimos utilizar material.

Por otro lado nos basamos en las Heurísticas Nielsen, a continuación nombraremos algunas de las tantas respetadas.

1. Visibilidad del estado del sistema


La misma se basa en que el usuario debe recibir feedback cuando realiza una acción. Por ejemplo, si estamos cargando un formulario me marque los campos obligatorios que debe ingresar.

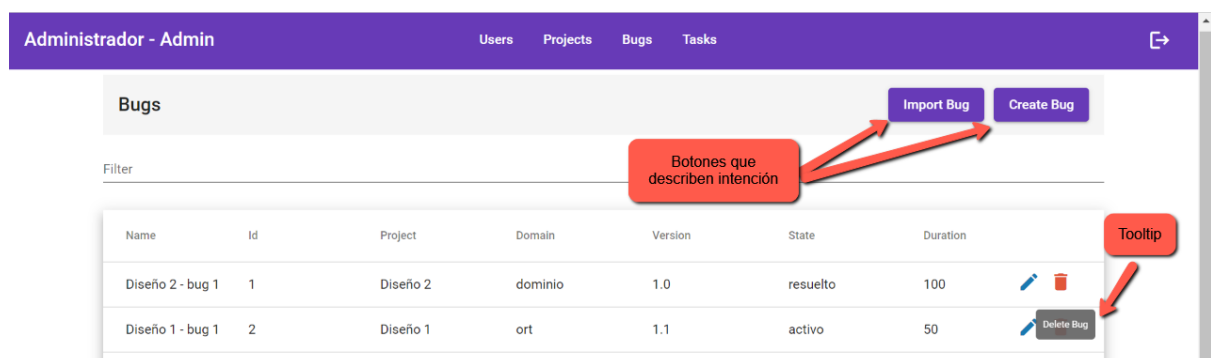






Por ejemplo en la captura anterior al no ingresar un campo requerido el mismo es marcado en rojo para resaltar al usuario que es un campo requerido, además de siempre contar con el carácter “*” informando que es un campo obligatorio.

2. Consistencia y estándares

Tratamos de seguir los máximos estándares que se utilizan en el diseño de UI, es

decir, si el usuario quiere eliminar información tenga asociado que el icono  con eliminar, así con modificar, etc. Por otro lado, tratamos de agregar botones con texto claros haciendo posible que el usuario al utilizarlos tenga clara su funcionalidad. En los casos donde se utilizaron iconos se agregaron tooltips para identificar la acción a realizar.



Name	Id	Project	Domain	Version	State	Duration	
Diseño 2 - bug 1	1	Diseño 2	dominio	1.0	resuelto	100	 
Diseño 1 - bug 1	2	Diseño 1	ort	1.1	activo	50	 

3. Reconocer en lugar de recordar

La misma recomendación minimizar la carga de memoria del usuario. Ejemplo, al contar con un menú hamburguesa estamos ocultando opciones del menú que podrían estar a la vista para el usuario. Esto provoca que el usuario deba recordar el icono del menú desplegable e interpretar que con dicho botón accedemos a las opciones de nuestro menú. Es un ejemplo simple pero en sistemas extremadamente grandes o complejos se debería manejar con cautela este tipo de cosas.

4. Diseño estético y minimalista

Se debe contar con pantalla que no revele información innecesaria. Por ejemplo si doy click en la opción crear bug solo contemos con las opciones necesarias para crearlo y no mezclar por ejemplo opciones como de importación, etc.

Especificación de la API - Cambios

En esta sección solo se especificarán los nuevos endpoints o cambios sobre endpoints ya existentes de nuestra aplicación con las respectivas justificaciones.

Por más especificación de nuestro modelado de nuestra api se podrá encontrar en la sección de [“Especificación de la API - Completa”](#) de nuestra primera entrega, dado que el mismo no cambia para esta segunda instancia, sin considerar los cambios que se especificarán a continuación.

Session:

Se implementó un nuevo endpoint para obtener el usuario logueado. El propósito del mismo es desde el front consumir el endpoint pasándole el token y obtener toda la información del usuario asociado al token (es decir, el usuario logueado actualmente).

Session	
POST	/penguin/sessions/login
POST	/penguin/sessions/logout
GET	/penguin/sessions/{userToken}

GET

/penguin/sessions/{userToken}

Parameters

Try it out

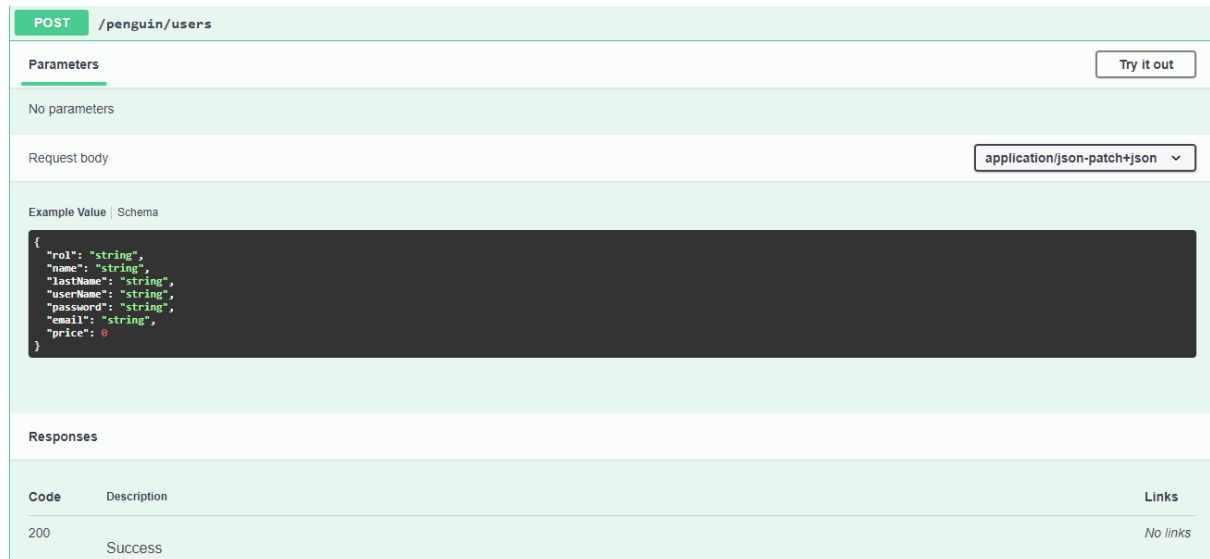
Name	Description
<div><div>userToken * required</div><div>string</div><div>(path)</div></div>	<div>userToken</div>

Responses

Code	Description	Links
200	Success	No links

Usuario:

Se modificó el modelo que recibe el método post de users. En este nuevo modelo se agregó el atributo “price”.



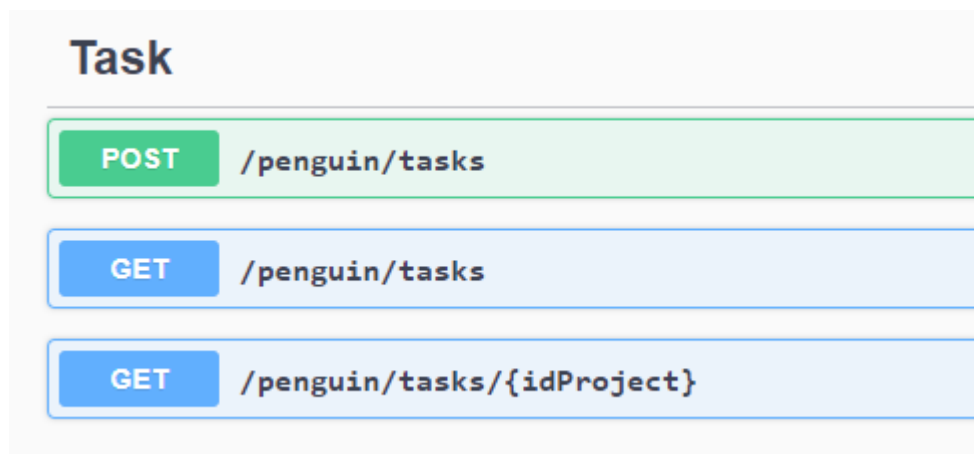
The image shows a Swagger UI interface for the **POST /penguin/users** endpoint. It includes a 'Parameters' section with 'No parameters', a 'Request body' section with a dropdown set to 'application/json-patch+json', and an 'Example Value' section showing a JSON object with fields: 'rol', 'name', 'lastName', 'userName', 'password', 'email', and 'price' (set to 0). The 'Responses' section shows a 200 status code with the description 'Success' and 'No links'.

```
{
  "rol": "string",
  "name": "string",
  "lastName": "string",
  "userName": "string",
  "password": "string",
  "email": "string",
  "price": 0
}
```

Code	Description	Links
200	Success	No links

Task:

Para esta nueva entrega se agregó un nuevo controlador con sus EndPoint específicos.



The image displays a list of endpoints under the heading 'Task'. It includes three entries: a POST endpoint for '/penguin/tasks', and two GET endpoints for '/penguin/tasks' and '/penguin/tasks/{idProject}'.

- POST** /penguin/tasks
- GET** /penguin/tasks
- GET** /penguin/tasks/{idProject}

Bug:

Se modificó el endpoint de método delete. Dado que en nuestra primera entrega no contemplabamos que un bug podía ser eliminado por un tester y dado que este tester debe estar asignado al proyecto de dicho bug a eliminar vimos necesario pasarle el el id del usuario que va a realizar la request para verificar que el mismo esté asociado al proyecto del bug.

Bug	
POST	/penguin/bugs
GET	/penguin/bugs
GET	/penguin/bugs/{bugId}
PUT	/penguin/bugs/{bugId}
DELETE	/penguin/bugs/{bugId}/byUser/{userId}

DELETE /penguin/bugs/{bugId}/byUser/{userId}		Try it out
Parameters		
Name	Description	
bugId * required integer(\$int32) (path)	<input type="text" value="bugId"/>	
userId * required string(\$uuid) (path)	<input type="text" value="userId"/>	
Responses		
Code	Description	Links
200	Success	No links

Developer:

Para esta segunda instancia fue necesario implementar un nuevo endpoint para obtener todas las tareas a la cual está asociado un desarrollador.

Developer

GET	/penguin/developers/{idDeveloper}/bugs
POST	/penguin/developers/{idDeveloper}/project/{idProject}
DELETE	/penguin/developers/{idDeveloper}/project/{idProject}
GET	/penguin/developers/{idDeveloper}/countBugs
PUT	/penguin/developers/{developerId}/bugState
GET	/penguin/developers
GET	/penguin/developers/{idDeveloper}/projects
GET	/penguin/developers/{idDeveloper}/tasks

GET /penguin/developers/{idDeveloper}/tasks

Parameters

Try it out

Name	Description
idDeveloper * required	
string(suuid)	idDeveloper
(path)	

Responses

Code	Description	Links
200	Success	No links

Tester:

Para esta segunda instancia fue necesario implementar un nuevo endpoint para obtener todas las tareas a la cual está asociado un tester.

Tester	
GET	/penguin/testers/{idTester}/bugs
GET	/penguin/testers/{idTester}/projects
POST	/penguin/testers/{idTester}/project/{idProject}
DELETE	/penguin/testers/{idTester}/project/{idProject}
GET	/penguin/testers
GET	/penguin/testers/{idTester}/tasks

GET

/penguin/testers/{idTester}/tasks

Parameters

Try it out

Name	Description
<div><div>idTester</div><div><div><div><div>required</div></div></div><div>string(\$uuid)</div><div>(path)</div></div></div>	<div>idTester</div>

Responses

Code	Description	Links
200	Success	No links

Justificación y evidencia de TDD

La metodología TDD se basa en crear nuestras pruebas y luego escribir el código de la funcionalidad para que las pruebas pasen. Luego de esto se realiza una etapa de refactor del código implementado.

Dicha metodología se realizó sólo sobre nuestro backend como fue especificado. Comenzamos nuestro TDD desde la lógica de negocios, luego pasando por el acceso a datos y por último por nuestra web api de las nuevas funcionalidades/requerimiento, las cuales fueron principalmente sobre la nueva entidad/lógica de Tareas y los cambios del dominio sobre bugs, proyectos, usuarios y distintos cálculos de costos y duración de proyecto.

Lo decidimos realizar de esta manera dado que debimos primero definir una estructura de negocio para cumplir con las distintas funcionalidades del negocio definiendo así una estructura de nuestro dominio y entidades.

Terminada la etapa de TDD de nuestra lógica continuamos con el TDD del acceso a datos ya que contábamos con un dominio y lógica estable.

Por último realizamos dicho proceso en nuestra WebApi ya que él mismo podía tener distintas flexibilidades a la hora de ser implementado debido a la escasez de lógica y estructura que la conforma comparado a nuestra lógica de negocio y acceso a datos.

Nota: TDD no fue realizado sobre reflection dado que para la implementación de la misma utilizamos la interface "IConfiguration" instanciada con la clase "ConfigurationBuilder" para cargar la ruta de donde se cargan nuestros archivos dll que iba a consumir nuestra aplicación. Esto nos trajo problemas a la hora de mockear la interface IConfiguration pudiendo no realizar pruebas sobre el módulo de importación a través de consumir dll de terceros.

Evidencia de commits

Implementación de clase TaskLogic, TDD

Primera fase de TDD. Implementación de clase TaskLogicTest (RED).
 condesahreyes committed on Oct 13
Segunda fase de TDD. Implementación de clase TaskLogic (GREEN)
 condesahreyes committed on Oct 13
Proceso de refactor
 condesahreyes committed on Oct 13


Implementación de clase TaskController, TDD

Primera fase de TDD. Implementación de clase TaskControllerTest (RED)
 condesahreyes committed on Oct 14
Segunda fase de TDD. Implementación de clase TaskController (GREEN).
 condesahreyes committed on Oct 14
Proceso de refactor
 condesahreyes committed on Oct 14

Implementación funcionalidad costo y duración de proyecto, TDD

Nuevos atributos en clases User, Project y Bug. Se agrego la migració...

...




condesahreyes committed 15 days ago

Commits on Oct 31, 2021

Primera fase de TDD. Implementación de metodos CalculateTotalDuration...


...



condesahreyes committed 15 days ago


Segunda fase de TDD. Implementación de metodos CalculateTotalDuration...

...



condesahreyes committed 15 days ago




Proceso de refactor





condesahreyes committed 15 days ago

Análisis de cobertura de pruebas



WebAPI

▲  webapi.dll	5	2.29%	213	97.71%
▷  OBLDA2.Controllers	5	9.62%	47	90.38%
▷  WebApi.Controllers	0	0.00%	166	100.00%




Exceptions

▲  exceptions.dll	0	0.00%	6	100.00%
▷  Exceptions	0	0.00%	6	100.00%






Dominio

▲  domain.dll	14	4.91%	271	95.09%
▷  Domain	14	4.91%	271	95.09%

DataAccess

▲  dataaccess.dll	62	20.33%	243	79.67%
▷  DataAccess	2	9.09%	20	90.91%
▷  DataAccess.Repositories	60	21.20%	223	78.80%

BussinessLogic

▲  businesslogic.dll	282	29.78%	665	70.22%
▷  BusinessLogic	127	25.30%	375	74.70%
▷  BusinessLogic.Imports	4	4.49%	85	95.51%
▷  BusinessLogic.UserRol	89	31.79%	191	68.21%
▷  Imports	62	81.58%	14	18.42%

Nuestra cobertura sobre la BussinessLogic fue baja dado lo comentado anteriormente de la no realización de TDD sobre la importación de bugs consumiendo extensiones de terceros por problemas de moquear la interface "IConfiguration".

Métricas - Cálculos

En la siguiente figura podemos visualizar la matriz de dependencia donde los cuadrados azules simbolizan dependencias entrantes y los verdes dependencias salientes.

		0 WebApi	1 Factory	2 DataAccess	3 BusinessLogic	4 DataAccessInterface	5 Imports	6 BusinessLogicInterface	7 Domain	8 Exceptions
0 WebApi	0		1					9	15	1
1 Factory	1	2		1	1	1		1	1	
2 DataAccess	2		6			5			11	2
3 BusinessLogic	3		8			4	1	8	10	9
4 DataAccessInterface	4		5	5	5				4	
5 Imports	5				2				1	
6 BusinessLogicInterface	6	8	8		8				8	
7 Domain	7	6	2	6	6	4	3	4		4
8 Exceptions	8	3		1	3				1	

Para calcular la **estabilidad** de cada paquete utilizamos la siguiente fórmula, $\frac{C_e}{C_a + C_e} = I$.
 Donde Ce es las dependencias salientes (cantidad de cuadraditos verdes del paquete x) y Ca son las dependencias entrantes (cantidad de cuadraditos azules de dicho paquete).
 La misma se calcula de la siguiente manera, si quiero calcular Ce de web api debo contar los cuadraditos verdes de la fila 0 y para calcular el Ca debo contar los cuadraditos azules de la fila 0.

Para el cálculo de la **abstracción** utilizamos la siguiente fórmula $A = N_a / N_c$.
 Donde Na son la cantidad de clases abstractas e interfaces en el paquete y Nc son la cantidad de clases concretas, abstractas e interfaces del paquete.

Para calcular la **distancia** utilizamos la siguiente fórmula $D' = |A + I - 1|$

Para el **cálculo de la cohesión** relacional utilizamos la siguiente fórmula $H = (R + 1) / N$.
 Donde el R es el número de relaciones entre clases internas al paquete y N el número de clases e interfaces dentro del paquete.

Especificación de la API - Completa

Se detalla la estructura, los manejos de modelos, inyección de dependencia, mecanismo de autenticación, manejo de excepciones entre otras.

Nuestra estructura utilizada es REST donde la misma debe de cumplir con los 6 principios principales. La misma está basada en una arquitectura HTTP a la hora de comunicarse con el cliente/servidor.

Estructura

- 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 Exception

Por otro lado utilizamos un filter de excepción (ExceptionHandler), el cual lo denominamos en la clase ApiController 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.

Mecanismo de autenticación

Para autenticar a los distintos usuarios se creó un filtro, el mismo se ejecuta antes de que la request lleguen a los controllers.

El funcionamiento de nuestra autenticación es el siguiente:

El usuario solicita loguearse mediante en request (POST) enviando en su body el email y la contraseña.

El sistema valida que la combinación de email y contraseña existan en la base de datos. Es decir exista el usuario y haya puesto correctamente la contraseña.

En caso de que alguno de los datos sean incorrectos se devuelve un código de estado 400 y un mensaje de lo sucedido, en este caso, email y/o contraseña invalido.

Por otro lado en caso de que el email exista y se haya puesto correctamente la contraseña el sistema genera un token, dicho token es generado por un identificador y a continuación un guid generado.

El identificador que generamos es el rol del usuario. Para dar un ejemplo un token de los 3 tipos de usuarios se visualizarán de la siguiente manera.

- Administrador-866e12c3-7656-43c3-bf19-01141ce8cb82
- Tester-7a24fd81-1698-4be0-a431-18dea61c70e6
- Desarrollador-58721e37-df41-45a6-9504-5bb2f33b9efd

De esta manera podemos identificar qué permisos tiene dicho token.

Luego de haberse logueado correctamente el sistema le devolverá el token generado y el cliente podrá y deberá utilizarlo en cada request a realizar.

Dicho Token debe ser colocado en el Header con la key "Authorization".

En caso de que el token no tenga permiso para realizar dicha operación se le devolverá un código de error 403 Forbidden.

En caso de no haber ingresado token o el usuario no está logueado en el sistema se devolverá 401 Unauthorized.

- Filters Authorization

Para la autenticación comentada previamente 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 como especificamos anteriormente (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 AllAuthorization =  
    Rol.administrator + "," + Rol.developer + "," + Rol.testers;  
  
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 "AllAuthorization" la cual tiene los identificadores de todos los roles. Pudiendo ser consumido en nuestro filtro de la siguiente manera.

```
[HttpGet]  
[AuthorizationFilter(Authorization.AllAuthorization)]  
1 reference  
public IActionResult GetAllBugs()  
{
```

Descripción de los códigos devueltos

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

Resource de la API

Sesión

Session

POST

/penguin/sessions/login

POST

/penguin/sessions/logout

GET

/penguin/sessions/{userToken}

Login

POST

/penguin/sessions/login

Try it out

Parameters

No parameters

Request body

application/json-patch+json

Example Value

Schema

```
{
  "email": "string",
  "password": "string"
}
```

Responses

Code	Description	Links
200	Success	No links

Cerrar Sesión

POST

/penguin/sessions/logout

Try it out

Parameters

No parameters

Request body

application/json-patch+json

Example Value

Schema

```
{
  "token": "string"
}
```

Responses

Code	Description	Links
200	Success	No links

Obtener usuario logueado por token

GET

/penguin/sessions/{userToken}

Try it out

Parameters

Name	Description
userToken * required string (path)	<input type="text" value="userToken"/>

Responses

Code	Description	Links
200	Success	No links

Usuario

Crear Usuario

POST

/penguin/users

Try it out

Parameters

No parameters

Request body

application/json-patch+json

Example Value | Schema

```
{
  "rol": "string",
  "name": "string",
  "lastName": "string",
  "userName": "string",
  "password": "string",
  "email": "string"
}
```

Responses

Code	Description	Links
200	Success	No links

Obtener Usuario

GET

/penguin/users/{userID}

Try it out

Parameters

Name	Description
userID * required string(\$uuid) (path)	<input type="text" value="userID"/>

Responses

Code	Description	Links
200	Success	No links

Obtener Usuarios

POST

/penguin/users

Parameters

Try it out

No parameters

Request body

application/json-patch+json

Example Value

Schema

```
{
  "rol": "string",
  "name": "string",
  "lastName": "string",
  "userName": "string",
  "password": "string",
  "email": "string",
  "price": 0
}
```

Responses

Code	Description	Links
200	Success	No links

Proyecto

POST

/penguin/projects

GET

/penguin/projects

GET

/penguin/projects/bugs

GET

/penguin/projects/{projectId}

GET

/penguin/projects/{projectId}/bugs

DELETE

/penguin/projects/{id}

PUT

/penguin/projects/{id}

Crear Proyecto

POST

/penguin/projects

Parameters

Try it out

No parameters

Request body

application/json-patch+json

Example Value

Schema

```
{
  "name": "string"
}
```

Responses

Code	Description	Links
200	Success	No links

Obtener proyectos

GET

/penguin/projects

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	Success	No links

Obtener los Bugs de todos los proyectos

GET

/penguin/projects/bugs

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	Success	No links

Obtener un Proyecto

GET /penguin/projects/{projectId}

Parameters

Try it out

Name	Description
projectId * required string(\$uuid) (path)	<input type="text" value="projectId"/>

Responses

Code	Description	Links
200	Success	No links

Obtener bugs de un proyecto

GET /penguin/projects/{projectId}/bugs

Parameters

Try it out

Name	Description
projectId * required string(\$uuid) (path)	<input type="text" value="projectId"/>

Responses

Code	Description	Links
200	Success	No links

Eliminar un proyecto

DELETE /penguin/projects/{id}

Parameters

Try it out

Name	Description
id * required string(\$uuid) (path)	<input type="text" value="id"/>

Responses

Code	Description	Links
200	Success	No links

Modificar Proyecto

PUT

/penguin/projects/{id}

Try it out

Parameters

Name	Description
id * required	
string(\$uuid)	id
(path)	

Request body

application/json-patch+json

Example Value

Schema

```
{
  "name": "string"
}
```

Responses

Code	Description	Links
200	Success	No links

Bug

Bug

POST

/penguin/bugs

GET

/penguin/bugs

GET

/penguin/bugs/{bugId}

PUT

/penguin/bugs/{bugId}

DELETE

/penguin/bugs/{bugId}/byUser/{userId}

Crear Bug

POST

/penguin/bugs

Try it out

Parameters

No parameters

Request body

application/json-patch+json

Example Value

Schema

```
{
  "project": "string",
  "id": 0,
  "name": "string",
  "domain": "string",
  "version": "string",
  "state": "string",
  "createdBy": "3fa85f64-5717-4562-b3fc-2c963f66af6"
}
```

Responses

Code	Description	Links
200	Success	No links

Obtener bugs

GET

/penguin/bugs

Try it out

Parameters

No parameters

Responses

Code	Description	Links
200	Success	No links

Obtener un bug

GET

/penguin/bugs/{bugId}

Try it out

Parameters

Name	Description
bugId * required integer(\$int32) (path)	<input type="text" value="bugId"/>

Request body

application/json-patch+json

Example Value

Schema

```
{
  "userId": "3fa85f64-5717-4562-b3fc-2c963f66afa6"
}
```

Responses

Code	Description	Links
200	Success	No links

Eliminar un bug

DELETE

/penguin/bugs/{bugId}/byUser/{userId}

Try it out

Parameters

Name	Description
bugId * required integer(\$int32) (path)	<input type="text" value="bugId"/>
userId * required string(\$uuid) (path)	<input type="text" value="userId"/>

Responses

Code	Description	Links
200	Success	No links

Modificar un bug

PUT /penguin/bugs/{bugId}

Parameters

Try it out

Name	Description
bugId <small>* required</small> integer(int32) (path)	<input type="text" value="bugId"/>

Request body

application/json-patch+json

Example Value | Schema

```
{
  "project": "string",
  "name": "string",
  "domain": "string",
  "version": "string",
  "state": "string",
  "userId": "3fa85f64-5717-4562-b3fc-2c963f66afae"
}
```

Responses

Code	Description	Links
200	Success	No links

Developer

Developer

GET

/penguin/developers/{idDeveloper}/bugs

POST

/penguin/developers/{idDeveloper}/project/{idProject}

DELETE

/penguin/developers/{idDeveloper}/project/{idProject}

GET

/penguin/developers/{idDeveloper}/countBugs

PUT

/penguin/developers/{developerId}/bugState

GET

/penguin/developers

GET

/penguin/developers/{idDeveloper}/projects

GET

/penguin/developers/{idDeveloper}/tasks

Obtener un bug asociado a un desarrollador

GET /penguin/developers/{idDeveloper}/bugs		
Parameters		Try it out
Name	Description	
idDeveloper * required string(\$uuid) (path)	<input type="text" value="idDeveloper"/>	
Responses		
Code	Description	Links
200	Success	No links

Agregar un desarrollador a un proyecto

POST /penguin/developers/{idDeveloper}/project/{idProject}		
Parameters		Try it out
Name	Description	
idProject * required string(\$uuid) (path)	<input type="text" value="idProject"/>	
idDeveloper * required string(\$uuid) (path)	<input type="text" value="idDeveloper"/>	
Responses		
Code	Description	Links
200	Success	No links

Eliminar un desarrollador de un proyecto

DELETE /penguin/developers/{idDeveloper}/project/{idProject}		
Parameters		Try it out
Name	Description	
idDeveloper * required string(\$uuid) (path)	<input type="text" value="idDeveloper"/>	
idProject * required string(\$uuid) (path)	<input type="text" value="idProject"/>	
Responses		
Code	Description	Links
200	Success	No links

Obtener cantidad de bugs resueltos por un desarrollador

GET

/penguin/developers/{idDeveloper}/countBugs

Try it out

Parameters

Name	Description
idDeveloper * required string(\$uuid) (path)	<input type="text" value="idDeveloper"/>

Responses

Code	Description	Links
200	Success	No links

Actualizar el estado de un bug con un desarrollador

PUT

/penguin/developers/{developerId}/bugState

Try it out

Parameters

Name	Description
developerId * required string(\$uuid) (path)	<input type="text" value="developerId"/>

Request body

application/json-patch+json

Example Value | Schema

```
{
  "state": "string",
  "bugId": 0
}
```

Responses

Code	Description	Links
200	Success	No links

Tester

Tester

GET

/penguin/testers/{idTester}/bugs

GET

/penguin/testers/{idTester}/projects

POST

/penguin/testers/{idTester}/project/{idProject}

DELETE

/penguin/testers/{idTester}/project/{idProject}

GET

/penguin/testers

GET

/penguin/testers/{idTester}/tasks

Obtener bugs asociados a un tester

GET /penguin/testers/{idTester}/bugs		
Parameters		Try it out
Name	Description	
idTester * required string(\$uuid) (path)	<input type="text" value="idTester"/>	
Responses		
Code	Description	Links
200	Success	No links

Agregar un tester a un proyecto

POST /penguin/testers/{idTester}/project/{idProject}		
Parameters		Try it out
Name	Description	
idProject * required string(\$uuid) (path)	<input type="text" value="idProject"/>	
idTester * required string(\$uuid) (path)	<input type="text" value="idTester"/>	
Responses		
Code	Description	Links
200	Success	No links

Eliminar un tester de un proyecto

DELETE /penguin/testers/{idTester}/project/{idProject}		
Parameters		Try it out
Name	Description	
idTester * required string(\$uuid) (path)	<input type="text" value="idTester"/>	
idProject * required string(\$uuid) (path)	<input type="text" value="idProject"/>	
Responses		
Code	Description	Links
200	Success	No links

Obtener tareas de un tester

GET

/penguin/testers/{idTester}/tasks

Try it out

Parameters

Name	Description
idTester <small>* required</small> string(\$uuid) (path)	<input type="text" value="idTester"/>

Responses

Code	Description	Links
200	Success	No links

Imports

POST

/penguin/import/bugs

Crear import de bugs

POST

/penguin/import/bugs

Try it out

Parameters

No parameters

Request body

application/json-patch+json

Example Value

Schema

```
{
  "fileAddress": "string"
}
```

Responses

Code	Description	Links
200	Success	No links