Universidad ORT Uruguay

Facultad de ingeniería

Diseño de aplicaciones 2

Obligatorio 2

https://github.com/ORT-DA2/SobralGoni

210361 - José Pablo Goñi

192247 - Juan Pablo Sobral

Descripción general del trabajo

La solución brindada es un sistema que maneja carpetas y archivos de usuarios, los cuales pueden tener amigos y compartir sus trabajos. Cada usuario se debe loguearse al sistema y ya dentro se le brindan funcionalidades para manejar sus archivos y carpetas, como para agregar amigos nuevos. En el caso de un usuario administrador el mismo tiene más privilegios que un usuario común, pudiendo borrar todos los archivos y carpetas de todos los usuario y viendo reportes que el sistema le brindará. Más adelante se explican las funcionalidades que tiene el sistema y cómo se implementaron.

Diagrama de descomposición de paquetes

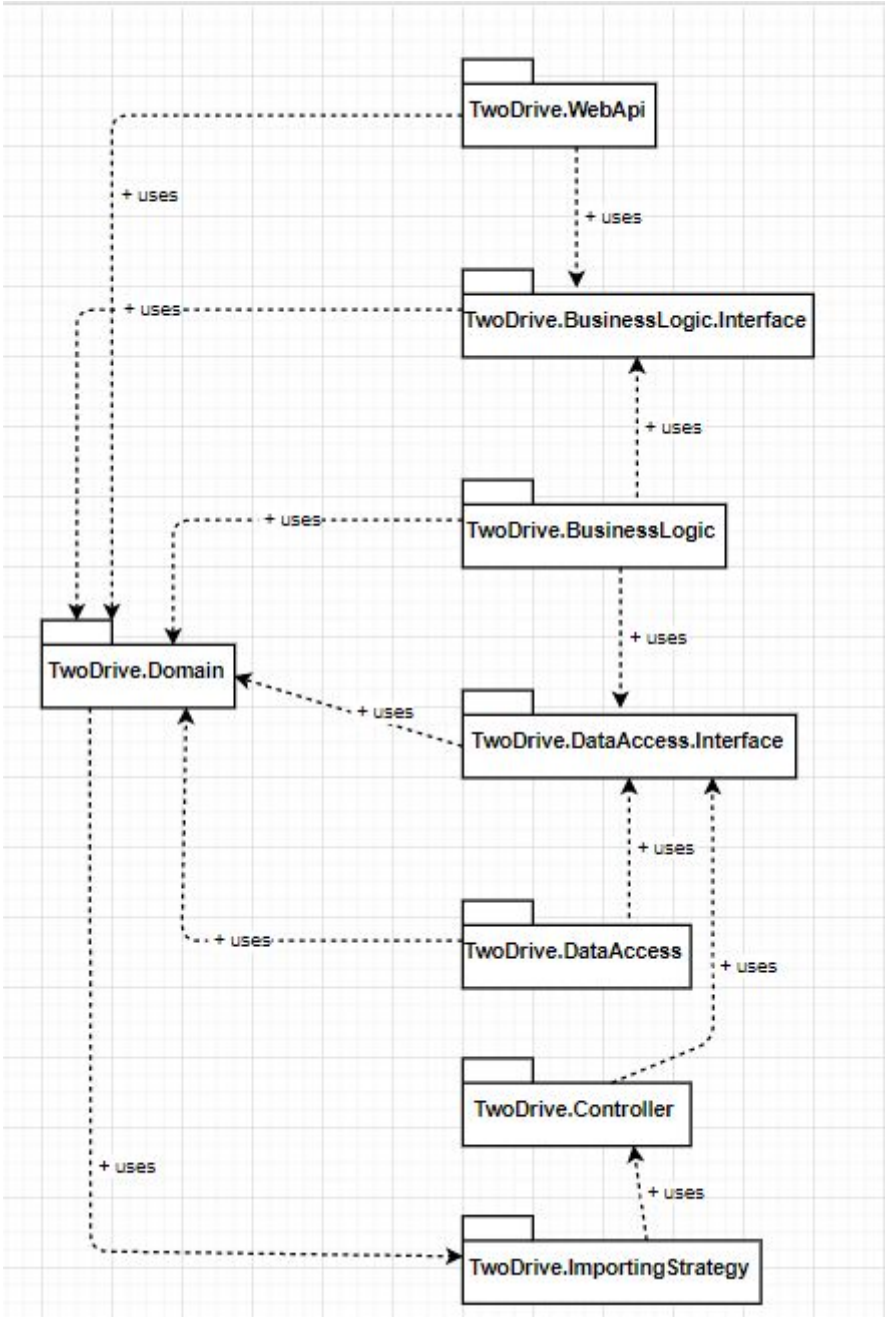
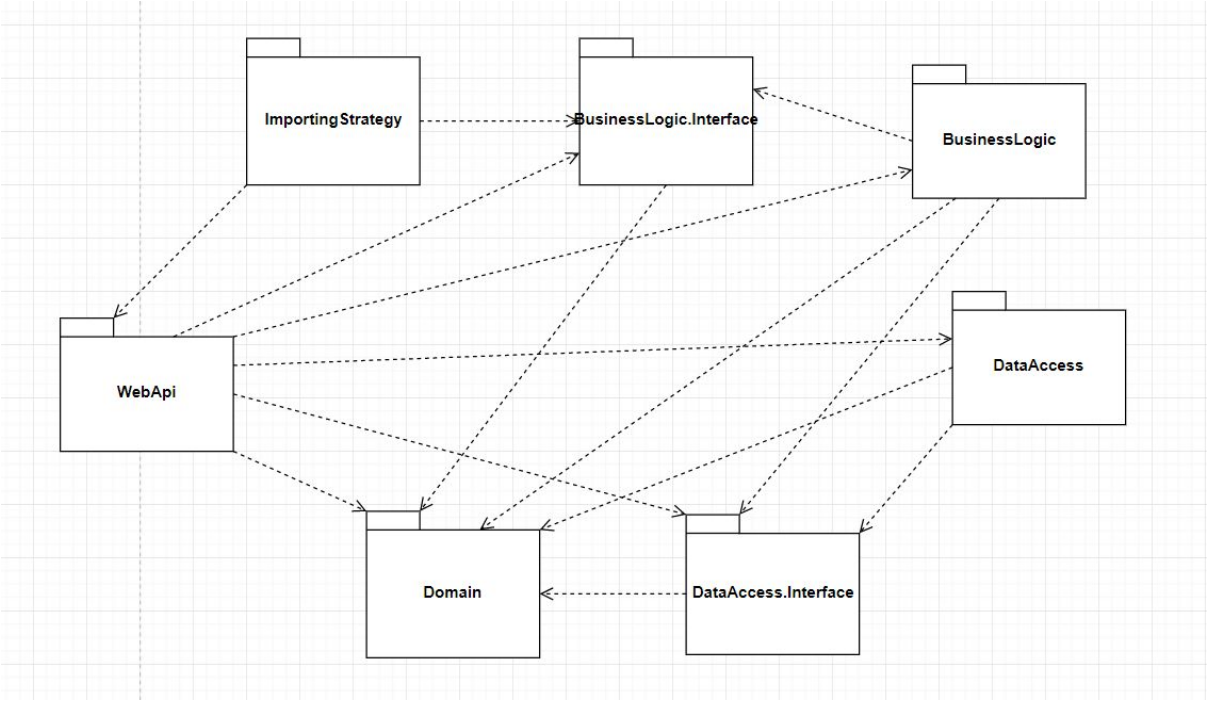


Diagrama general de paquetes

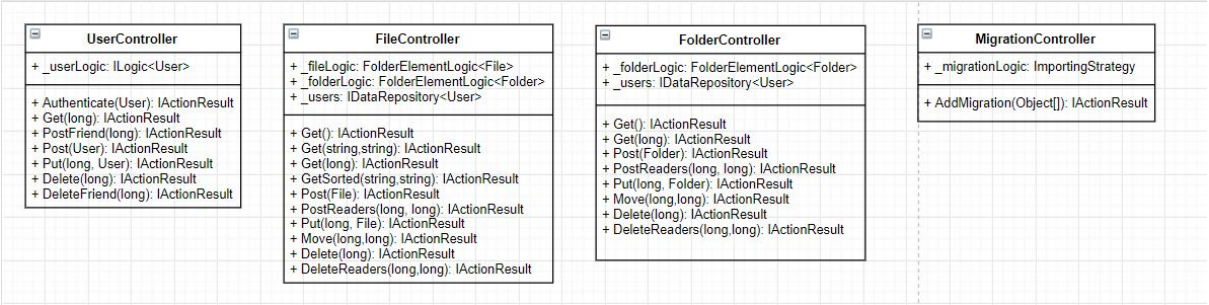


TwoDrive.WebApi

Descripción de responsabilidad

El paquete es el encargado de controlar las request que llegan de los clientes y de mandar a sus respectivos responsables para que estos ejecuten la lógica del sistema y le devuelvan algo para mandarle una respuesta al mismo. Se reparten en 4 diferentes controladores, los cuales cada uno por separado se encarga de resolver request de su propia entidad del sistema que controla, dependiendo el controlador que se disponga es el endpoint que se le asignó para ser llamado. En común todos tienen la url: http://localhost:57902/
y luego el recurso por el que se quiere preguntar. Lo controladores no tienen idea qué es lo que se hace en la lógica, solo tienen la responsabilidad de atender solicitudes y mandarsela al sistema para que este haga algo. De esta manera si cambian los requerimientos de la lógica no afecta este paquete.

Diagrama UML

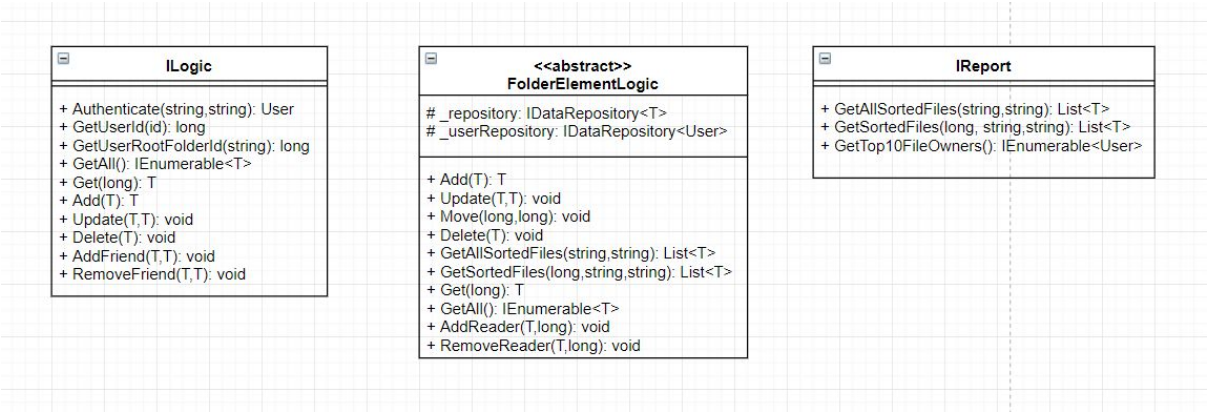


TwoDrive.BusinessLogic.Interface

Descripción de responsabilidad

Como se ha dicho anteriormente, quisimos separar componentes lo más que podamos con el fin de tener un menor acoplamiento posible, por eso mismo no quisimos que el paquete de web api se enterara de cómo se maneja la lógica ya que la web api es un componente de alto nivel y no tienen porque enterarse. Por ese mismo motivo se decidió crear un paquete de interfaces las cuales se utilizan para desacoplar nuestro sistema.

Diagrama UML



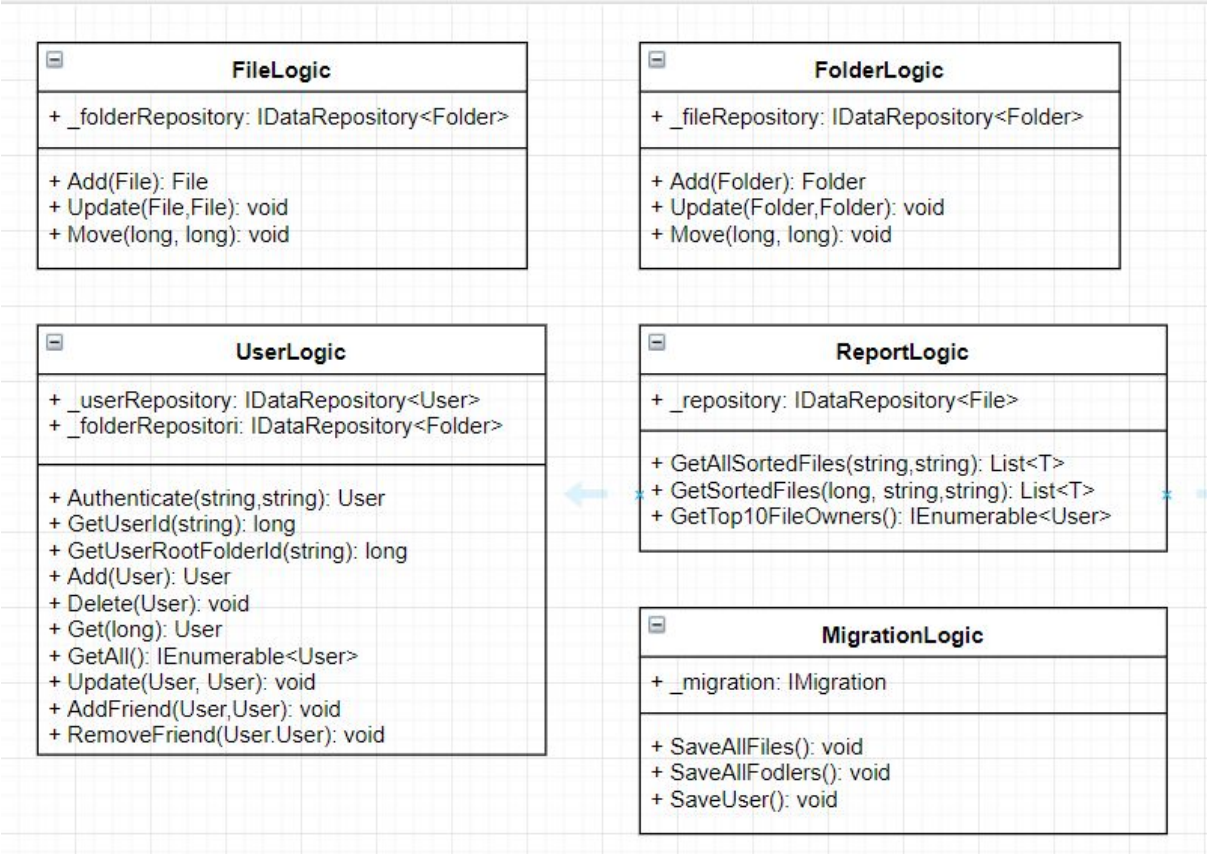
TwoDrive.BusinessLogic

Descripción de responsabilidad

Este paquete es el encargado de ejecutar toda la lógica del sistema, se quiso que cada clase que se creara sea lo más cohesiva posible y que no se repitiera código en ninguna de ellas, lo cual pudimos solucionar con una clase abstracta entre archivos y carpetas ya que nos dimos cuenta que estos tenían y se comportan más o menos parecidos, entonces derivamos los métodos que se comportan distintos a cada uno de ellas para que se les dé el correspondiente comportamiento y poniendo los comportamientos en común en la clase abstracta. Para esta última entrega se agrego la clase MigrationLogic que se encarga de

guardar todos los archivos, carpetas y el usuario que le pasa la interfaz que los usuarios deben implementar para importar un sistema completo al nuestro. Por más información sobre la nueva funcionalidad más abajo se explica.

Diagrama UML

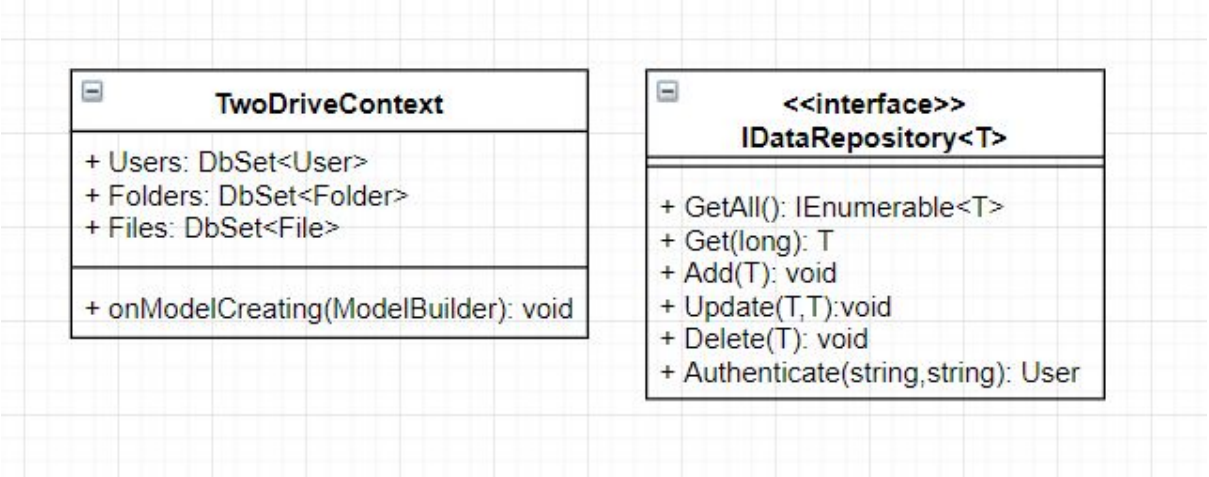


TwoDrive.DataAccess.Interface

Descripción de responsabilidad

Como explicamos anteriormente con el paquete de interfaz de la lógica del sistema, este se creo para un mismo fin, el cual la lógica del sistema no tiene ni debe conoces dónde es que estamos guardando todo. Ya que el dia de mañana debemos migrar la base de datos a otra y la lógica no se entera de nada gracias al bajo acoplamiento que tenemos.

Diagrama UML

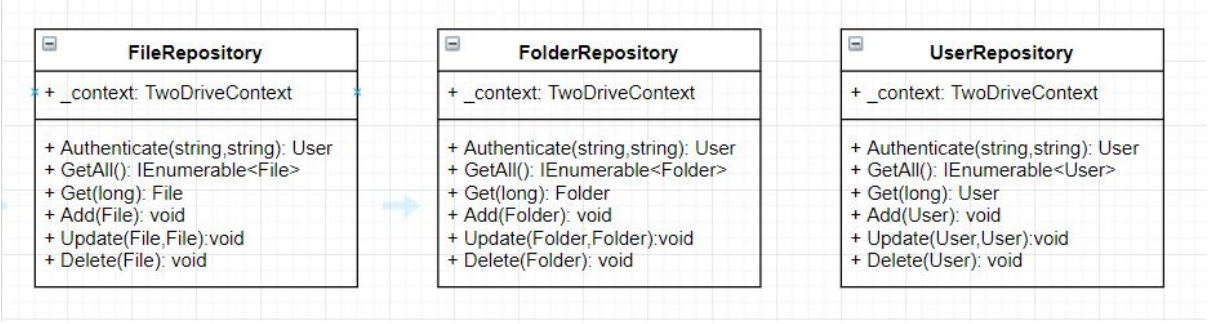


TwoDrive.DataAccess

Descripción de responsabilidad

En este paquete introducimos como guardamos todos los datos con nos llegan por la interfaz, en este caso la guardamos en la base de datos de sql, creando un contexto único para todos.

Diagrama UML



TwoDrive.ImportingStrategy

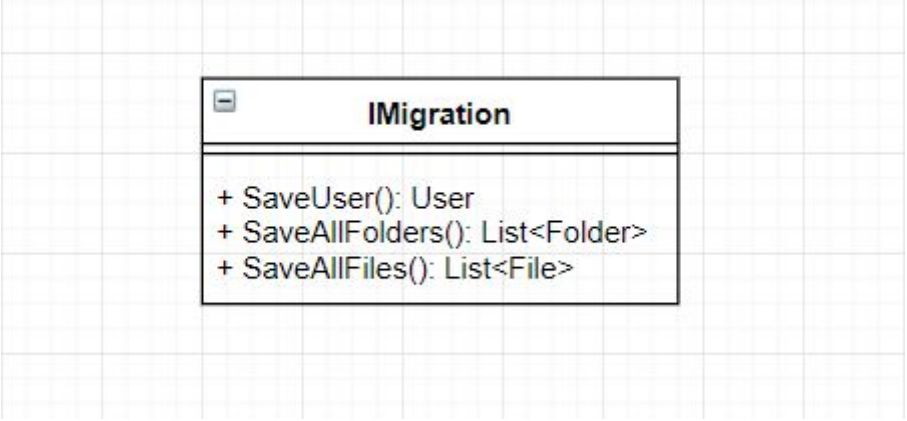
Descripción de responsabilidad

Este paquete fue creado para implementar la funcionalidad nueva, aca vamos a ver principalmente la interfaz que cada usuario que quiera importar algo a nuestro sistema debe implementar. La clase que se implemente se podrá poner aquí mismo, osea la dll de la misma, pudiendo el sistema en orden de ejecución tomar esa dll nueva y ejecutarla importando el archivo del tipo de dato que fue especificado. El usuario tercero que la implemente lo único que debe programar son estas 3 funciones que se les da:

- SaveUser() : El cual debe darme el usuario dueño este sistema que se va a importar.

- SaveAllFolders() : Una lista de todos las carpetas respetando los id para que el sistema se pueda fijar en las jerarquías que tiene.
- SaveAllFiles() : Una lista de todos los archivos, también se debe respetar los ids.

Diagrama UML

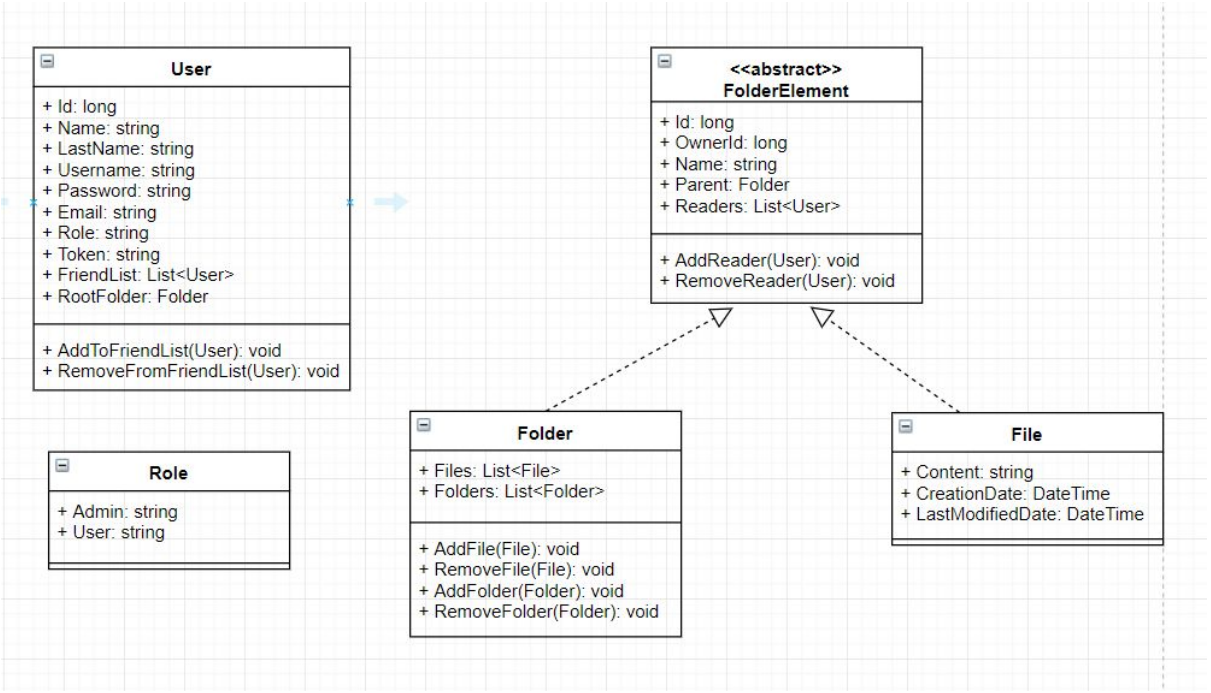


TwoDrive.Domain

Descripción de responsabilidad

Dominio del sistema, aquí se encuentran todas las entidades que maneja el sistema. Tenemos un elemento carpeta el cual es abstracta e implementa atributos en común que deben tener los archivos y las carpetas y por debajo con herencia los archivos y carpetas con los atributos complementarios que hacen la entidad en sí. Por otro lado usuario que como la letra del primer parcial decía que no iba a ser necesario agregar diferentes usuarios solo admin y usuario común se implementó de esta manera, dando enfasis en otras cosas que si se decia que podia cambiar.

Diagrama UML



Modelo tabla base de datos

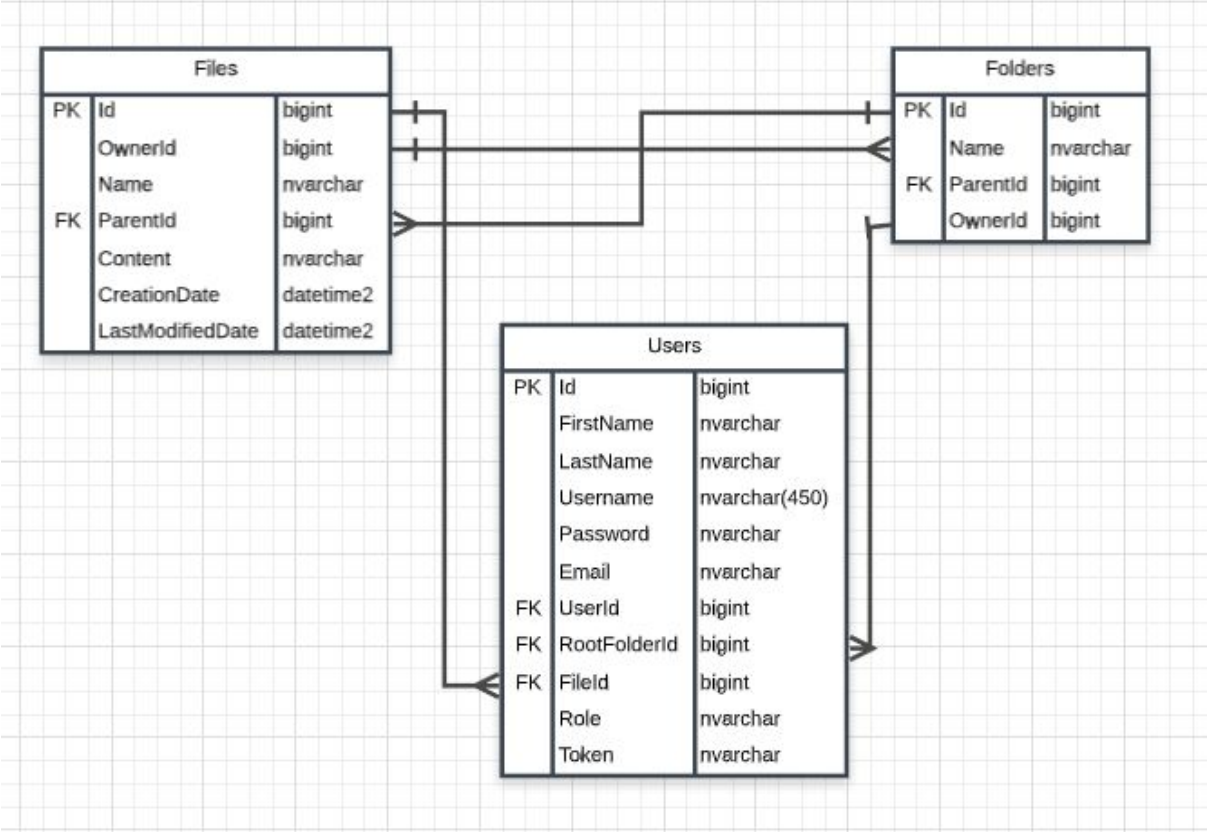
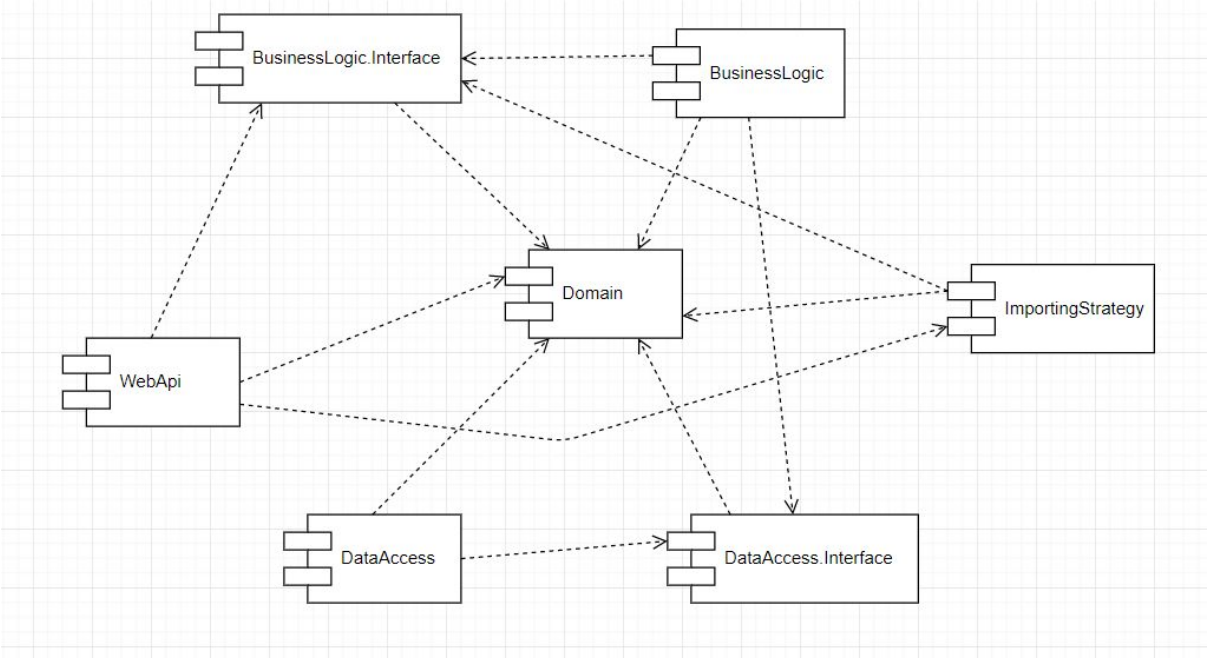


Diagrama de componentes



Justificación

Tenemos un componente nuevos desde el último sistema que se entregó, el cual se resolvió implementar por la funcionalidad nueva de importar un sistema, la web api utiliza el componente de importing strategy y este usa la lógica del nuestro sistema para guardar todo lo recibido. Este diseño lo explicaremos más adelante. Dando una justificación a los componentes podemos decir que el sistema está bien dividido y apartado cada uno de los componentes con interfaces apropiadas y estables las cuales no dejan que se vea las implementaciones, por un lado el servicio de web Api que se comunica solamente con la interfaz de businessLogic y esta se encarga de hacer todo el trabajo. Lo mismo pasa mas adentro del sistema con la base de datos, ya que la lógica no sabe en que se guarda sino que solo pide o manda a ejecutar alguna acción a la interfaz de la misma.

Métricas

Para la extracción de métricas sobre nuestra solución utilizamos la herramienta NDepend, como se verá reflejado en las imágenes que vamos a utilizar para ilustrar el estado de la misma.

Principio de dependencias estables

Según Robert Martin, el principio de dependencias estables plantea que nuestros paquetes deben depender de paquetes cuya métrica *I* sea menor que la del mismo.

Nuestro paquete **WebApi** tiene una métrica *I* de 1 y depende de varios otros paquetes en nuestra solución (**Domain, DataAccess, DataAccessInterface, BusinessLogic, BusinessLogicInterface**) pero por tener un valor tan alto de inestabilidad, cumple con la métrica del principio de dependencias estables.

Nuestro paquete **BusinessLogicInterface** tiene una métrica *I* de 0.59 y depende de los paquetes **Domain**(*I* = 0.35) y **DataAccessInterface**(*I* = 0.65). Rompe el principio de dependencias estables pero los valores no difieren tanto.

Nuestro paquete **BusinessLogic** tiene una métrica *I* de 0.89 y depende de los paquetes **Domain**(*I* = 0.35), **DataAccessInterface**(*I* = 0.65) y **BusinessLogicInterface**(*I* = 0.59). Cumple con la métrica del principio de dependencias estables.

Nuestro paquete **DataAccessInterface** tiene una métrica *I* de 0.65 y depende únicamente del paquete **Domain**(*I* = 0.35) por lo cual cumple con la métrica del principio de dependencias estables.

Nuestro paquete **DataAccess** tiene una métrica *I* de 0.93 y depende de los paquetes **Domain**(*I* = 0.35) y **DataAccessInterface**(*I* = 0.65). Cumple con la métrica del principio de dependencias estables.

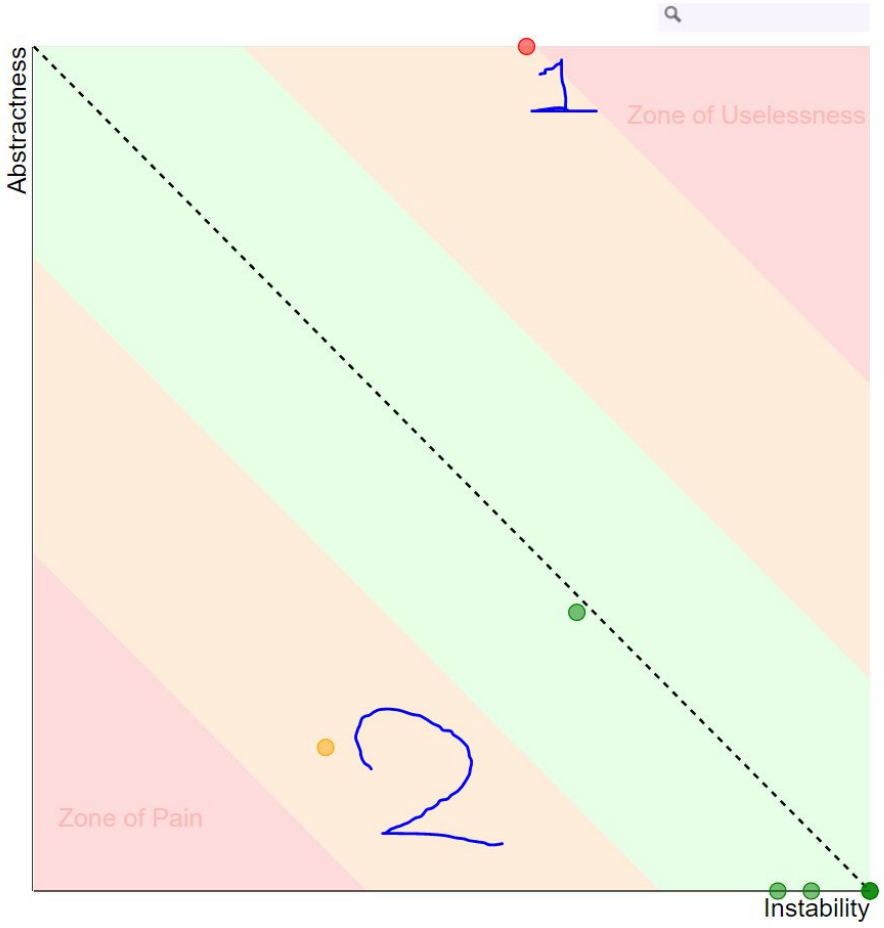
Nuestro paquete **ImportingStrategy** tiene una métrica *I* de 0.65 y depende únicamente del paquete **Domain**(*I* = 0.35) por lo cual cumple con la métrica del principio de dependencias estables.

En conclusión, de los seis paquetes que juegan un rol relevante en nuestra aplicación (no tenemos en cuenta el paquete de pruebas) cinco cumplen con la métrica que plantea el principio de dependencias estables (y el que no cumple la métrica lo hace por muy poco, de 3 paquetes de los que depende sólo uno tiene un *I* apenas mayor), por lo cual consideramos correcto afirmar que nuestra solución lo cumple.

Principio de abstracciones estables

Según Robert Martin, el principio de abstracciones estables plantea que nuestros paquetes más estables (o de los que más dependemos) deben de ser los más abstractos.

También nos permite graficar en función de las métricas *I* y *A* a nuestros paquetes, como se puede ver en la siguiente imágen:



Los paquetes que se encuentran dentro de la zona verde son los que cumplen el principio de inversión de dependencias (DIP). Los altamente inestables son BusinessLogic, DataAccess e ImportingStrategy. El paquete que se encuentra en el área verde pero más centrado es DataAccessInterface. El paquete señalado con el 1 es BusinessLogicInterface y está en una zona peligrosa dado que corre riesgo de dejar de ser útil (Muy abstracto y sin dependencias) aunque dado que pocos paquetes dependen del mismo por el diseño elegido, no nos preocupa demasiado. El paquete señalado con el 2 es Domain y corre riesgo de entrar en la zona de dolor (muy concreto y muchas dependencias) pero no creemos que haya forma de manejar un dominio dentro de un sistema de manera de no generar dependencias al mismo.

Al igual que con el principio anterior, consideramos que nuestra solución cumple con el principio de manera aceptable.

Principio de reuso común

Según Robert Martin, el principio de reuso común plantea que las clases que se reutilizan juntas deben agruparse juntas.

Esto es porque si yo dependo del paquete A, que tiene las clases B, C y D pero solo uso la clase C, un cambio en D me obligaría a actualizar mi versión del paquete lo cual no es deseable dado que el cambio que hubo no me afecta.

Según lo que interpretamos es una forma de mantener la cohesión dentro de un paquete, de manera similar a la que uno buscaría mantener la cohesión dentro de una clase.

Siguiendo esa visión, consideramos que se podría mejorar por ejemplo los paquetes DataAccess y DataAccessInterface dado que ambos trabajan como repositorios de las entidades que nuestra solución maneja. Por un lado, cuando se los modifica normalmente solo se toca una clase de las que agrupan. Por el otro, si yo separo por ejemplo DataAccess en cuatro paquetes (uno para cada repositorio) también tendría que separar DataAccessInterface en dos. Luego, esto afectaría la inestabilidad de todos los paquetes de la solución que dependan de estos (y son varios) aumentando su nivel de inestabilidad, lo cual tampoco es deseable.

Nos parece que llegamos a una descomposición en paquetes de manera razonable, todos tienen alta cohesión y no vemos los beneficios de separar aún más los ya existentes, dado que aumentaría muchísimo el acoplamiento entre los paquetes de la solución.

Principio de clausura común

Según Robert Martin, el principio de reuso común plantea que las clases que cambian juntas deben agruparse juntas.

En este caso creemos que cumplimos este principio mejor que el de Reuso Común, por los motivos planteados anteriormente.

Mecanismos utilizados para lograr la extensibilidad pedida

Importación de estructuras

Para la nueva funcionalidad se nos ocurrió crear un contrato, el cual cada desarrollador tercero a nosotros, podrá crear nuevas migraciones de diferentes archivos para nuestro sistema. Necesitábamos crear algo del lado de nuestro sistema, que conociera la lógica y todo lo necesario para guardar lo que por otro lado nos iba a llegar, no quisimos que cualquier desarrollador externos tenga que saber como guardar todas las cosas en nuestro sistema, esta fue la razón por la cual se implementa la interfaz IMigrationController, nombrada así por su propósito de controlar todas las migraciones. Esta interfaz será una mediadora entre la interfaz que el 3ero deberá desarrollar y nuestro sistema. La misma en tiempo de ejecución podrá leer cualquier archivo que se le haya implementado la interfaz que proponemos, a continuación explicaremos la misma:
Esta interfaz contará con 3 métodos y el desarrollador que la implemente **debe crear un constructor** el cual el usuario deberá especificar donde leer el archivo para el cual se implemente. El nombre de la clase que implemente esta interfaz debe ser llamada “(El tipo de archivo)Migration”, Ej: JsonMigration. Ya que luego se buscará así.
Los métodos que debe implementar son:

+ User GiveMeUser();

  Se debe parsear el usuario que se encuentra como lector en la carpeta raíz.
+ List<Folder> GiveMeFolders();

  Se debe parsear todas las carpetas y agregarlas a una lista de carpetas. Esta lista debe estar ordenada por jerarquía.
+ List<File> GiveMeFiles();

  Se debe parsear todos los archivos y agregarlos a una lista de archivos. Esta lista debe estar ordenada por jerarquía.

Se le obliga al usuario que el archivo que brinde cumpla con las siguientes condiciones:

+ El archivo debe contener todo el árbol completo con las carpetas y archivos respetando las jerarquías, los id que se les ponga se ignoran ya que se autogeneran nuevos al guardar todo en el sistema.
+ El usuario nuevo del sistema, debe estar como lector en la carpeta raíz, con todos sus datos.
+ El nombre de la carpeta raíz debe ser del estilo: nombre-rootFolder
+ Cuando se especifica el padre del archivo o carpeta se debe poner solo la id del mismo. (Cambiar esto en código.)

Por otro lado se creó un end point el cual se debe especificar el tipo de archivo que se desea importar y en el cuerpo de la request debe ir todos los parámetros los cuales se necesitan para leer el archivo en un array.

Nuevo reporte

Para el nuevo reporte lo que tuvimos que hacer fue utilizar el log que ya teníamos antes y ponerlo en la edición, borrado o modificación de carpetas y archivos pero por cada operación de esta, ir a todas las carpetas padres desde donde se ejecutó y crear un log para las mismas.

Patrones y principios de diseño utilizados

+ Inversión de dependencias(SOLID) , ya que apunta siempre a la lógica de negocios. Esto es porque ninguno de los módulos de alto nivel dependen de módulos de bajo nivel. Para ello se crearon abstracciones de cada nivel que generan esta separación.

+ Principio de segregación de interfaz(SOLID) , pudimos apreciar luego de armar todo que la lógica de archivos, la de carpetas y la de usuario tenian muchas cosas en comun ya que en todas teníamos ABM aunque por el lado de archivos y carpetas teníamos algunas otras funciones como mover de lugar. Dado esto no podíamos tener la misma interfaz para todo ya que usuarios no iba a implementar cosas de la interfaz declarada y ademas teniamos el problema de que la logica de archivo y carpeta tenían muchas cosas que se hacen igual, lo único que variaba era el tipo de dato que se manejaba, por eso implementamos una solución basada en estos hechos.

+ Indirection(GRASP) , creamos clases intermedias para desacoplar clientes de servicios. De esta forma creamos estabilidad en el sistema.

+ Creator(GRASP) , derivamos la creación de todo a la webApi, inyectando dependencias.

+ Protección de variaciones(GRASP) , se decidió que las dependencias fueran a nivel de abstracciones, como se verá en los siguientes diagramas. De esta manera se buscó que el sistema fuera mantenible y extensible siendo este robusto frente a cualquier posible modificación y resolviendo el menor impacto posible sobre el código fuente ya existente, tal como es definido en el principio de abierto cerrado. Al hacer que el alto nivel no dependa del bajo nivel, se logra que el sistema sea más resistente a cambios. Esto también se logra haciendo que las dependencias se encuentren a nivel de abstracciones ya que se crea un efecto de ocultamiento hacia los detalles cambiantes.

+ Polimorfismo(GRASP), el mismo se utiliza para la nueva funcionalidad que nos brindaron, ya que el algoritmo debe ser capaz en tiempo de compilación de usar diferentes dll que un tercero le brinde para poder importar un sistema.

+ Strategy(GRASP), también con la nueva funcionalidad se usó strategy ya que el sistema debe poder cambiar de comportamiento según el cliente lo prefiera.

Mejoras realizadas al diseño

Nuestro diseño no cambió entre el primer obligatorio y el segundo. A nivel paquetes, se mantuvieron los mismos, simplemente se agregó uno para una de las funcionalidades pedidas y a nivel de clases cambiaron muy pocas y muy poco.

Esto es porque consideramos que nuestro diseño inicial era muy bueno, separando de manera correcta en paquetes y clases, llevando a que no hubiese problema en acomodar los nuevos cambios pedidos por letra o que surgieron como necesidades.

Ejemplo 1: Necesidad surgida para agregar amigos desde UI
Cuando realizamos el primer obligatorio decidimos utilizar la librería *Authorization* de Microsoft, proveída por AspNetCore. Esto en conjunto con el uso de Roles en nuestro modelo de Usuario y la creación del modelo de Roles nos permitió un nivel de granularidad en los endpoints de nuestra API super interesante. Podíamos por ejemplo determinar dentro de un controller que todos los endpoints estuviesen protegidos, es decir, que no permitiesen acceso sin autorización. Yendo a nivel más bajo, podemos determinar para cada uno de los endpoints cuáles de los roles están autorizados a acceder y cuales no.

Haciendo la UI en Angular nos percatamos que cuando agregábamos amigos en el primer obligatorio lo hacíamos desde el conocimiento del atributo Id por ejemplo, al cual no íbamos a tener acceso como usuario cualquiera desde la interfaz gráfica. La conclusión lógica era crear otro endpoint para poder ver los usuarios en el sistema y poder agregarlos como amigos, pero como estábamos usando *Authorization*, fue tan sencillo como borrar el decorador que teníamos sobre la función asociada a esa funcionalidad que decía **[Authorize(Roles = Role.Admin)]** y quedó funcionando a la perfección.

Ejemplo 2: Manejo de reportes
En el primer obligatorio dejamos el manejo de reportes como un tema final y cuando lo fuimos a implementar no hubo que hacer ningún refactor ni modificar demasiado el código existente.

Lo primero que se hizo fue agregar la clase LogItem al dominio, definiendo el modelo a utilizar. Luego, usando nuestra interfaz genérica IDataRepository

```
using System.Collections.Generic;
using TwoDrive.Domain;

namespace TwoDrive.DataAccess.Interface
{
    58 references
    public interface IDataRepository<T>
    {
        26 references
        IEnumerable<T> GetAll();
        67 references
        T Get(long id);
        18 references
        void Add(T entity);
        30 references
        void Update(T dbEntity, T newEntity);
        11 references
        void Delete(T entity);
        3 references
        User Authenticate(string username, string password);
    }
}
```

pudimos implementar la lógica de negocios de los reportes sin necesidad de tener implementada de manera concreta la lógica de acceso a datos sobre LogItems.
Luego, agregamos su interfaz correspondiente al paquete BusinessLogicInterface, agregamos los endpoints, implementamos las clases que aún no habíamos necesitado pero íbamos a necesitar (implementaciones concretas de ReportLogic y LogItemRepository) y quedó andando, no solamente amoldándose a nuestro sistema sin requerir modificaciones en el código ya existente, si no que extendiéndo su funcionalidad de manera poco dolorosa.

Sobre el uso de interfaces
Dada la cantidad de trabajo que implicó este obligatorio, era vital que ambos integrantes pudiésemos colaborar de manera independiente pero conjunta, es decir, que ninguno quedase trabado esperando por el otro pero que estuviésemos de acuerdo en ciertas cosas.

No estábamos muy seguros sobre cómo íbamos a hacer funcionar el proceso, pero cuando estábamos en la etapa de diseño vimos la posibilidad de poner interfaces entre nuestros componentes de manera de cumplir el principio de inversión de dependencias y a la vez definir ciertas reglas que nos permitían trabajar en partes separadas sabiendo que cuando las uniésemos iban a comportarse de manera esperada.

De esa forma, definimos nuestra clase **IDataRepository** y las interfaces contenidas en el paquete **BusinessLogicInterface**. Esto permitió que uno de nosotros se encargase de implementar las clases del paquete DataAccess en concordancia con la interfaz IDataRepository para luego pasar a trabajar en la capa de WebApi directamente, mientras que el otro se dedicó a implementar las clases concretas de BusinessLogic en concordancia con las interfaces de BusinessLogicInterface, haciendo uso de repositorios genéricos como el definido en IDataRepository.

Esta decisión de diseño nos facilitó muchísimo la división de trabajo inicial, y a lo largo del proyecto demostró varias veces haber sido una buena decisión.

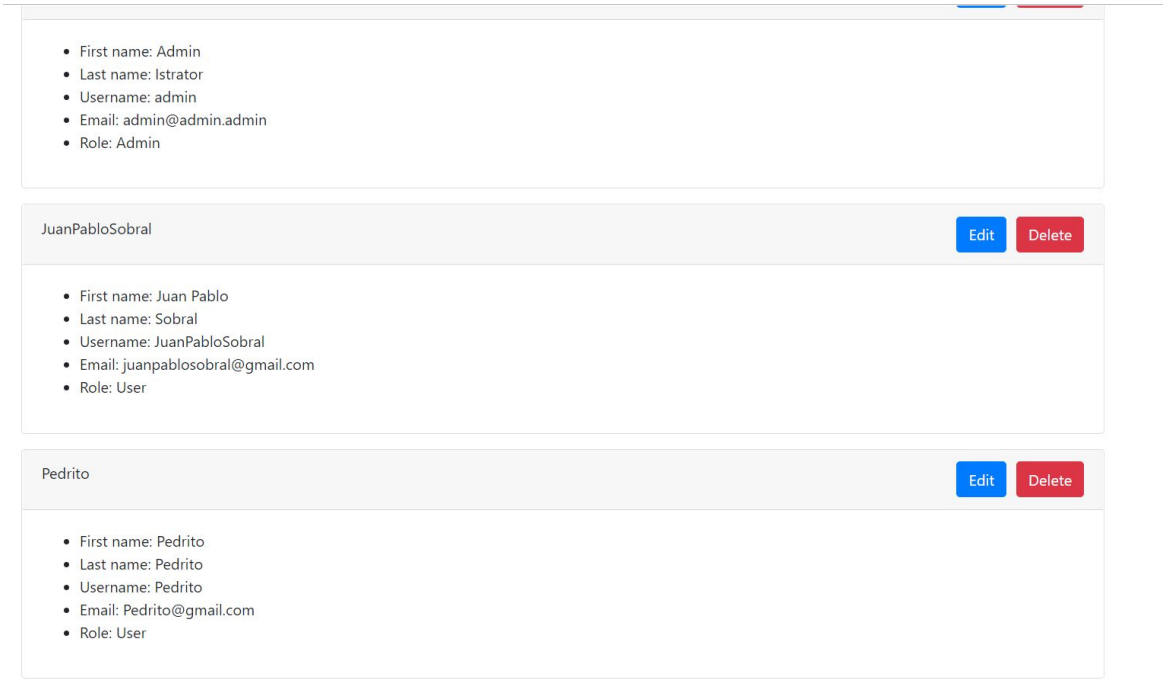
Anexo

Documentación API

Se mostrará un ejemplo de la nueva funcionalidad, también si se desea se puede ver todas las otras funcionalidades que se implementaron en la documentación anterior con imágenes y datos de prueba o en youtube con el video explicando todo. También brindamos la documentación de nuestra api en swagger en el siguiente link: https://app.swaggerhub.com/apis-docs/juanpaSobral/TwoDrive-APIGoniSobral/1

+ Probando la nueva funcionalidad

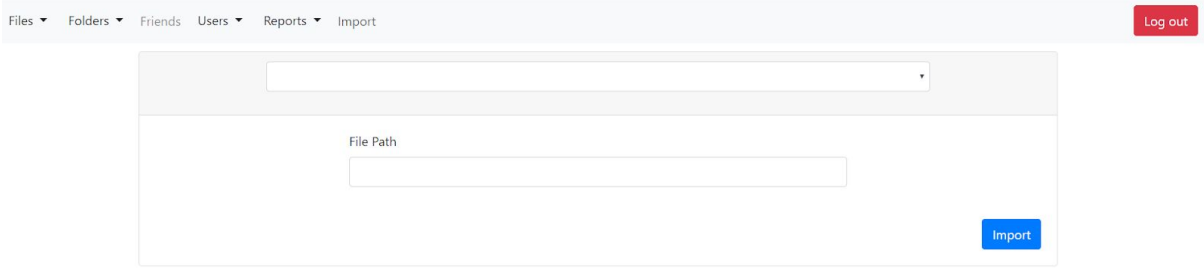
- Verificamos que el sistema no tiene al usuario ni carpetas con el archivo que vamos a importar.



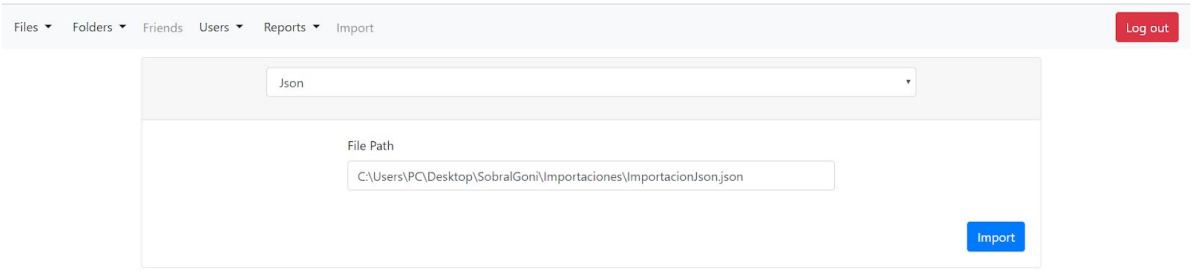
  Aparecen solo esos dos usuarios en el sistema.
- Vemos el archivo Json que importamos, con sus datos. (También existe el archivo Xml, estos se encuentran en la carpeta “importaciones”)

```
{
    "files": [
          {"content": "Este es un archivo que se encuentra en el root folder",
               "creationDate": "2019-10-30T09:23:15.5504079-03:00",
               "lastModifiedDate": "2019-10-30T09:23:15.5500328-03:00",
               "id": 1,
               "ownerId": 4,
               "name": "ArchivoEnRootFolder",
               "parent": {"id": "1"}
          }
         ],
    "folders":
         [
          {
        "files": [
                  {"content": "Este es un archivo que se encuentra en una subcarpeta de root folder",
                   "id": 2,
                   "ownerId": 4,
                   "name": "ArchivoEnFolder1",
                      "parent": {"id": "2"}
                  }
                 ],
          "folders": [],
          "id": 2,
          "ownerId": 4,
          "name": "Folder1",
          "parent": {"id": "1"}
         }
           ],
    "id": 1,
    "ownerId": 4,
    "name": "JosePabloGoni-rootFolder",
    "parent": null,
    "readers": [
        {
         "id": 4,
            "firstName": "José Pablo",
            "lastName": "Goñi Estefan",
            "username": "JosePabloGoni",
            "password": "Josepa",
            "email": "josepablogoni@gmail.com",
            "role": "User",
            "token": null,
            "friendList": [],
            "rootFolder": {
               "files": [],
               "folders": [],
               "id": 1,
               "ownerId": 4,
               "name": "JosePabloGoni-rootFolder",
               "parent": null,
               "readers": []
            }
        }
           ]
}
```

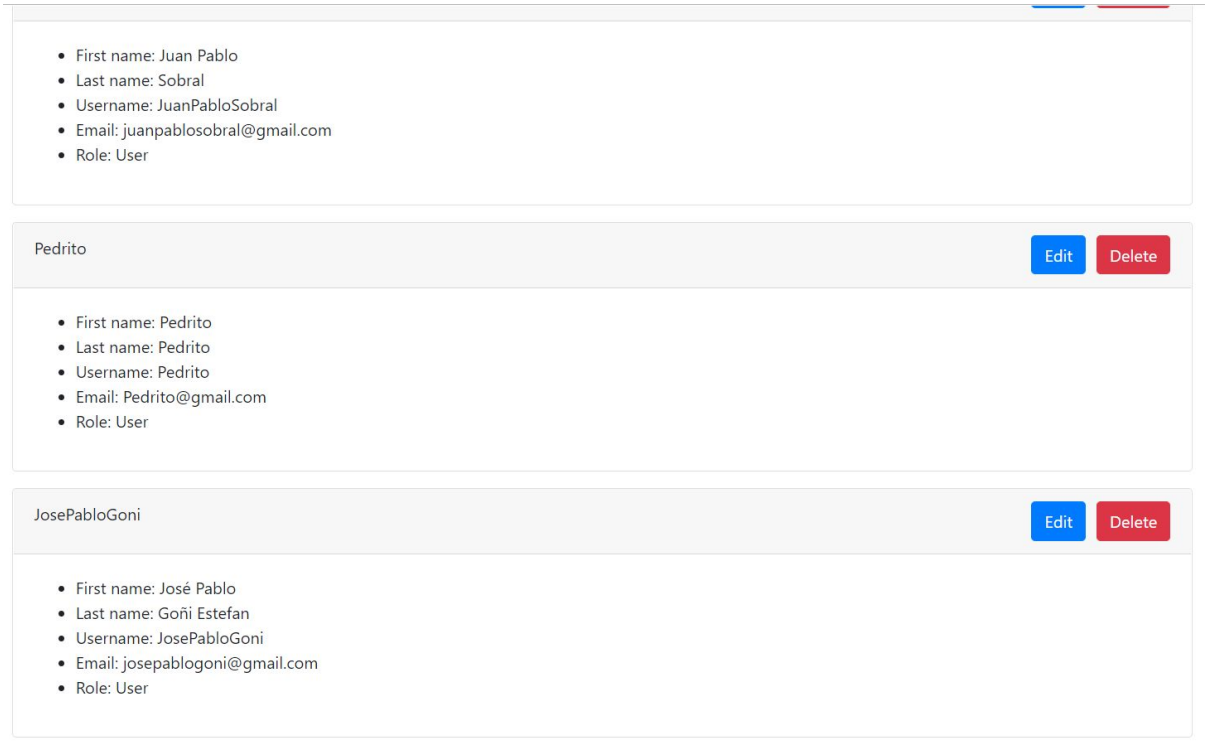

- Nos dirigimos a la página importación, siendo un administrador.



- Rellenamos los campos con el tipo de archivo y la dirección donde se encuentra.



- Verificamos que se importó correctamente.



Aparece el usuario importado en el sistema.

Informe de cobertura

Para las nuevas funcionalidades no se uso tdd por motivos de tiempo, lo cual llevó a que nuestra cobertura baje.

Jerarquía	No cubiertos (bloques)	No cubiertos (% de bloques)	Cubiertos (bloques)	Cubiertos (% de bloques)
PC_JOSEPABLOGOÑI 2019-10-08 ...	532	19.76%	2160	80.24%
twodrive.businesslogic.dll	309	44.85%	380	55.15%
TwoDrive.BusinessLogic	309	44.85%	380	55.15%
FileLogic	19	12.42%	134	87.58%
FolderLogic	55	25.94%	157	74.06%
ReportLogic	141	100.00%	0	0.00%
ReportLogic.<>c	37	100.00%	0	0.00%
ReportLogic.<>c__Dis...	3	100.00%	0	0.00%
ReportLogic.<>c__Dis...	5	100.00%	0	0.00%
ReportLogic.<>c__Dis...	11	100.00%	0	0.00%
ReportLogic.<>c__Dis...	11	100.00%	0	0.00%
UserLogic	27	24.77%	82	75.23%
UserLogic.<>c__Displa...	0	0.00%	3	100.00%
UserLogic.<>c__Displa...	0	0.00%	4	100.00%
twodrive.businesslogic.interfa...	29	38.16%	47	61.84%
TwoDrive.BusinessLogic.In...	29	38.16%	47	61.84%
FolderElementLogic.Us...	6	100.00%	0	0.00%
FolderElementLogic<T>	23	32.86%	47	67.14%
twodrive.dataaccess.dll	66	22.45%	228	77.55%
TwoDrive.DataAccess	66	22.45%	228	77.55%
FileRepository	2	2.86%	68	97.14%
FolderRepository	2	2.17%	90	97.83%
LogRepository	16	100.00%	0	0.00%
UserRepository	46	39.66%	70	60.34%
twodrive.dataaccess.interface....	29	93.55%	2	6.45%
TwoDrive.DataAccess.Inter...	29	93.55%	2	6.45%
TwoDriveContext	29	93.55%	2	6.45%
twodrive.domain.dll	6	8.11%	68	91.89%
TwoDrive.Domain	6	8.11%	68	91.89%
File	0	0.00%	6	100.00%
Folder	0	0.00%	16	100.00%
FolderElement	0	0.00%	16	100.00%
LogItem	4	40.00%	6	60.00%
User	2	7.69%	24	92.31%