Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2
Obligatorio 1:
Justificación del Diseño

Juliette Ruchel - 203942
Francisco Martinez - 233126

Docente: `Ignacio Valle`

Entregado como requisito de la materia Diseño de Aplicaciones 2

https://github.com/ORT-DA2/233126-203942.git

10 de octubre de 2019

Declaraciones de autoría

Nosotros, Juliette Ruchel y Francisco Martinez, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos Diseño de aplicaciones 2;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Resumen

El presente documento tiene el propósito de exponer nuestra justificación del diseño de nuestro sistema para el problema presentado.

Índice general

1. Descripción del sistema

1.1. Funcionamiento

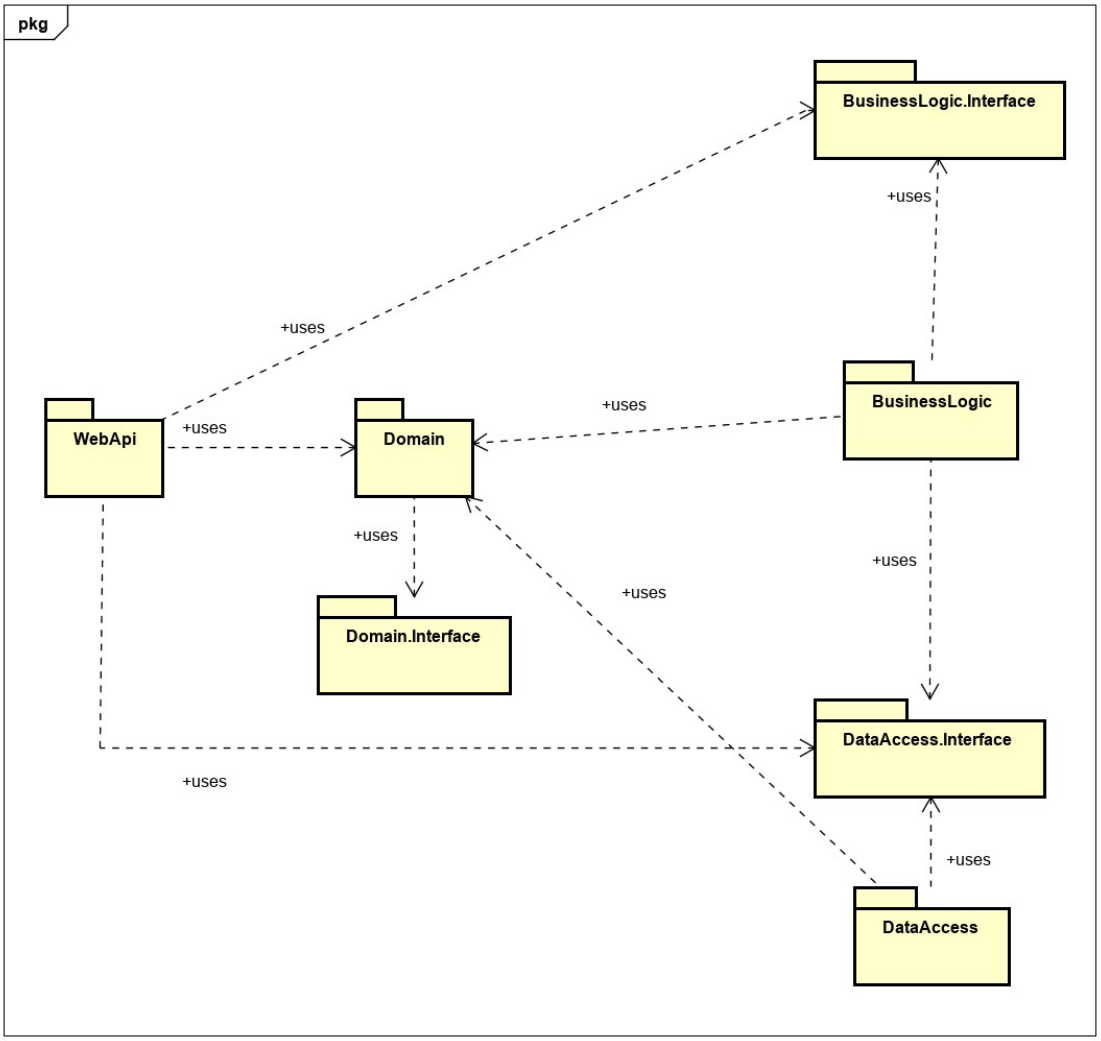
El funcionamiento del sistema se basa en un usuario que realice un log in el sistema y obtenga un token de autenticación. Luego segun su rol al ser creado podrá tener acceso a los diferentes end points y consumir sus datos. Es esencial el detalle de quien este loggeado porque muchos end points ejecutan su accion para quien este loggeado al menos que este especificado. Para obtener esta informacion lea el documento de especificación de la API, en la tabla esta detallado lso paramentros y respuestas de cada endpoint.

1.2. Bugs detectados:

- Permite loggearse de nuevo ya estando loggeado. Impacto: ocupa espacio en la base de datos.
- El Admin puede var carpetas y archivos de otros usuarios cuando solo debería poder hacerlo con archivos. Impacto: Si pasa algún error humano se podrían perder datos.
- Cuando se edita el usuario se borran los Custom Claims

2. Diagrama de paquetes

2.1. Paquetes principales

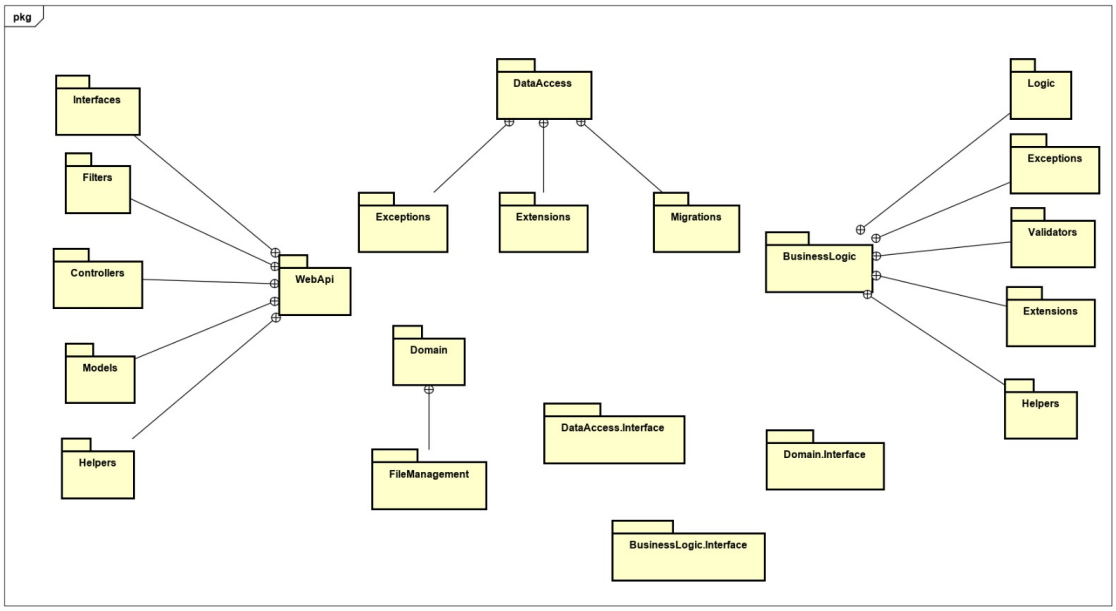


Nuestro diseño se baso en dividir por capas, cada un de estas representada por uno de los paquetes. Estos paquetes principales son: Dominio, BusinessLogic, DataAccess y Web Api. Luego existe un paquete de interfaces para los primeros 3 que nos permitió desacoplar cada paquete la implementacion del otro. Consecuentemente existe una mayor libertad de poder quitar un paquete y reemplazarlo con otro que implemente las interfaces.

Función de cada paquete:

- **Domain:** El paquete de dominio contiene todas las clases del problema a solucionar y que se utilizan para moldear el el sistema.
- **Domain.Interface:** Contiene las interfaces que implementan las clases del dominio. Actualmente es una sola: ISoftDelete.
- **BusinessLogic:** Contiene las clases que representan el problema logico del negocio, las operaciones y relaciones entre las clases. Tambien funciona como puente entre el acceso a datos y la web api.
- **BusinessLogic.Interface:** Contiene las interfaces que definien el comportamiento de la logica.
- **DataAccess:** Se encarga de acceder a la base de datos para guardar y proveerselos a las clases de logica.
- **DataAccess.Interface:** Determina el comportamiento de las clases encargadas de acceder a la base de datos
- **Web Api:** Es el paquete que expone los end points para que el sistema pueda ser consumido por diferentes clientes.

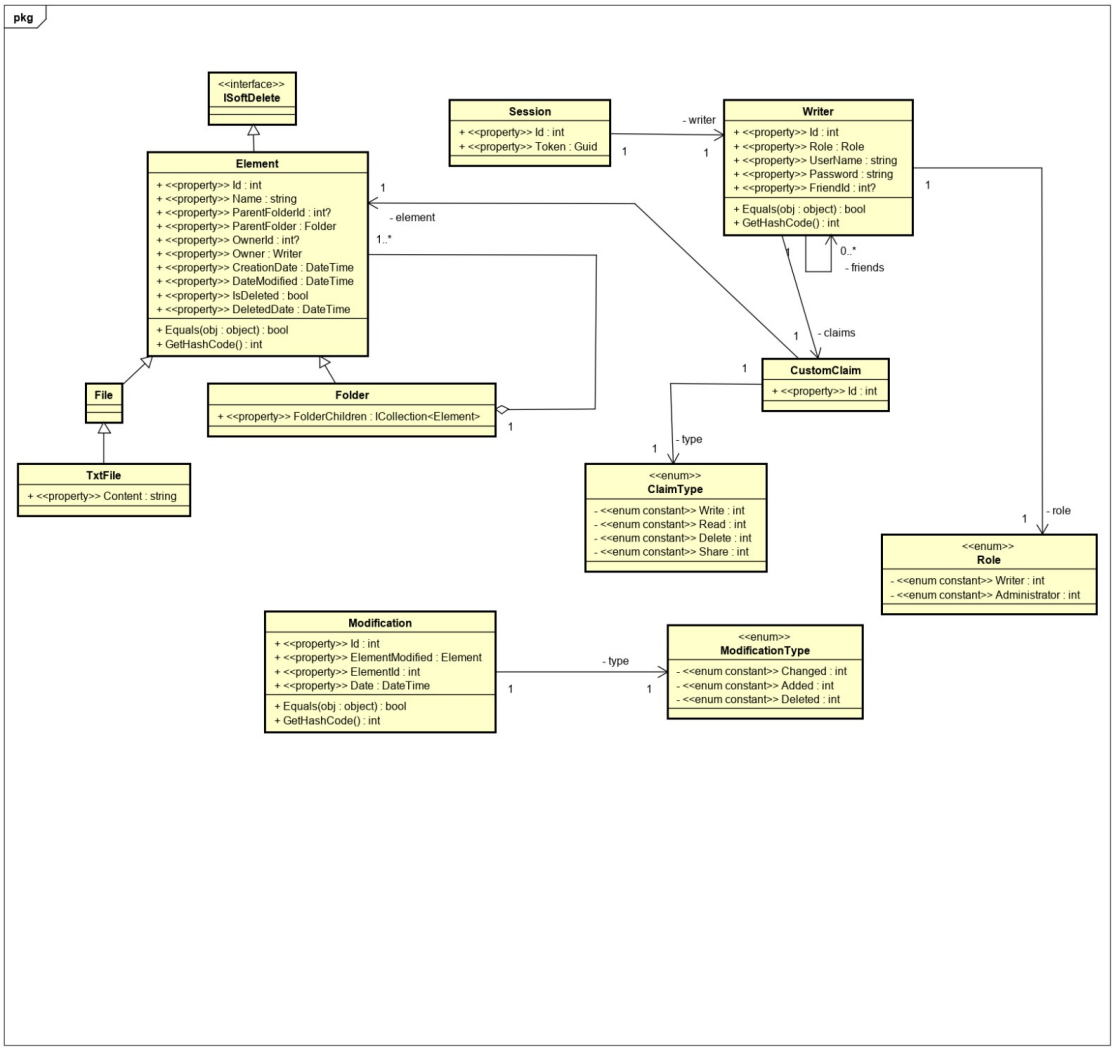
2.2. Paquetes anidaddos



Dentro de cada paquete ademas, dividimos las funcionalidades entre ellas para no tener un paquete con demasiadas responsabilidades y sea más facil reemplazar ciertas partes del sistema a futuro. Un ejemplo de esto es el paquete de validators, en el que se realizan todas las validaciones de los objetos que se persisten del sistema. Los paquetes internos tambien se conectan a través de interfaces que se encuentran en los paquetes de .Interface

3. Diagrama de clases

3.1. Dominio

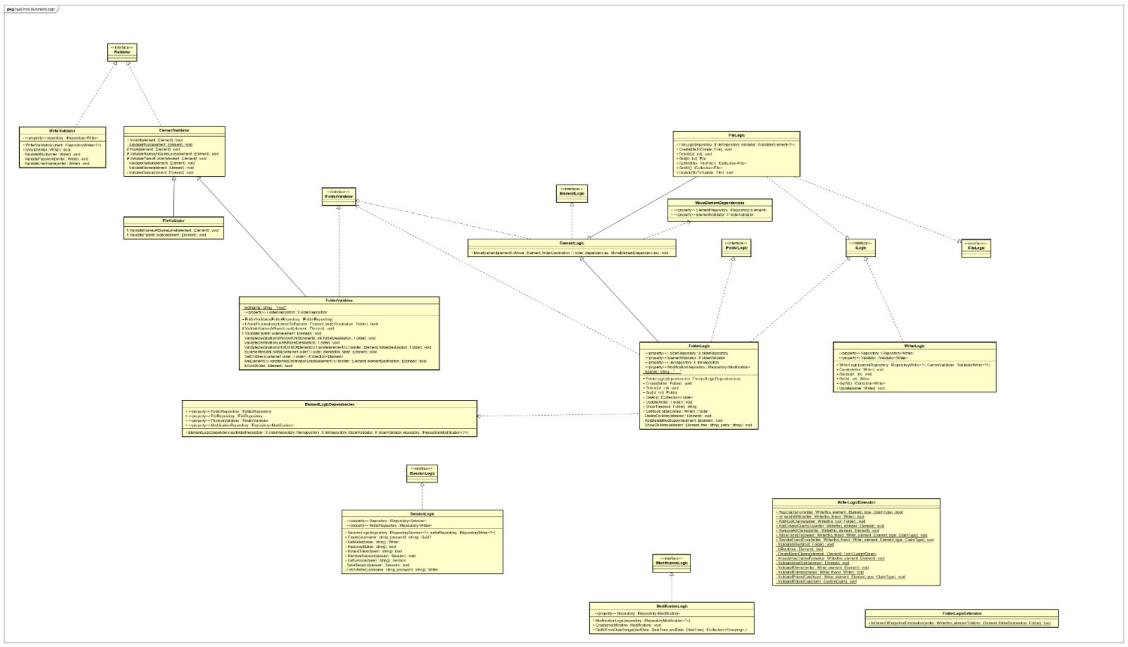


Existen dos grandes decisiones de diseño respecto al diseño del domino. La primera es la utilización de patron *Composite* para implementar la relación entre las carpetas y los archivos. Estos elementos presentan muchos atributos y comportamientos parecidos por lo que tenia sentido agruparlos en un clase abstracta. El patron composite nos permite llevar la abstracción a un nivel más y establecer la relación entre la carpeta y sus hijos elemento, de una manera simple de persistir.

Otra ventaja es que nos da la flexibilidad de agregar otro tipo de elemento que también pueda ser hijo de una carpeta. Existe una clase abstracta archivo, por que entendemos que hoy existen muchos tipos de archivos con comportamiento similiar al que luego se pueda decidir implementar en el sistema.

La segunda gran decisión fue decidir implementar los CustomClaim. El problema inicial era más simple de realizar y no implicaba darle a conocer al sistema el concepto de claim. Sin embargo, nos permitió dejar al sistema preparado para un incorporar formas básicas de seguridad y de autenticación que sean más extensibles que depender únicamente del rol de un usuario.

3.2. Logica



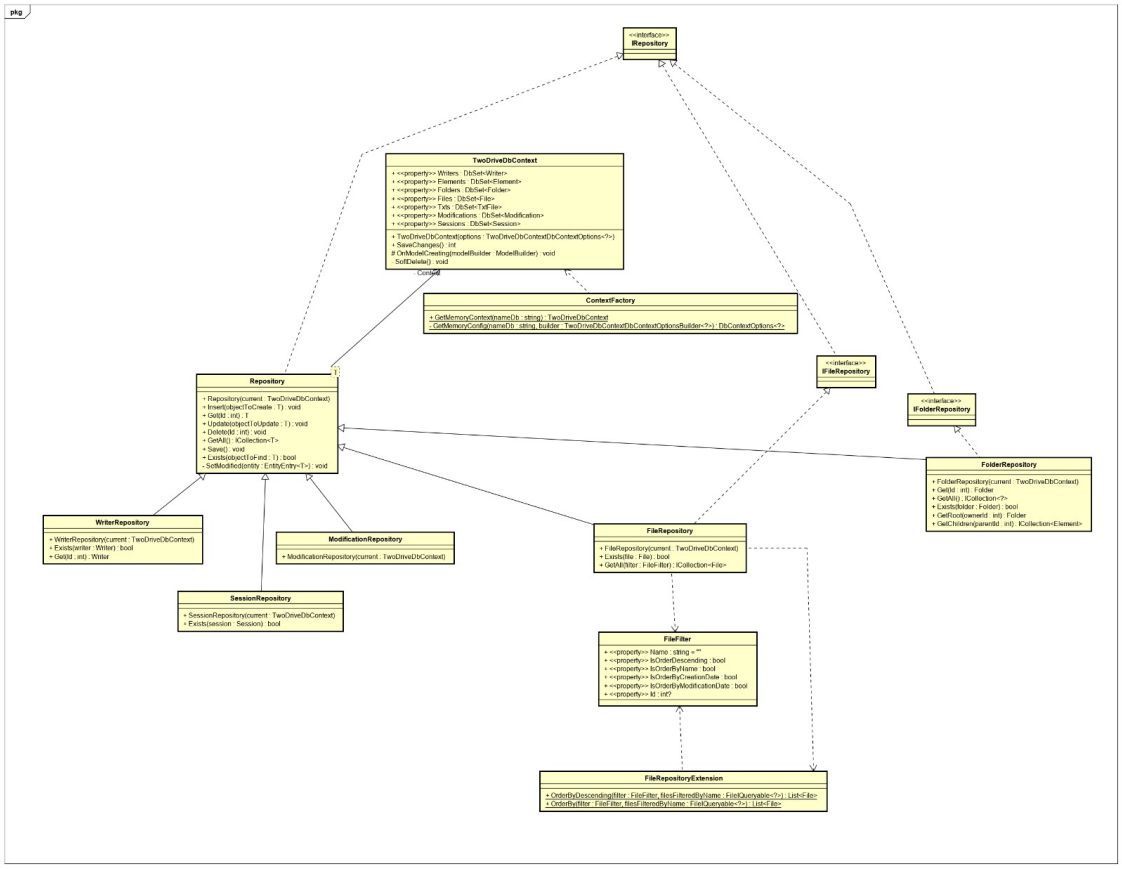
La logica tiene dos grandes aspectos. El primero, en el sub paquete *Logic*, se encuentran las clases que se encargan manejar los objetos, mandarlos al data access para persistirlos o para obtenerlos de ella y realizar sobre los mismos cuaquier operacion necesaria. En este paquete se implemento una interfaz base *ILogic<T>* que al ser generica permite que cualquier clase a cargo de una entidad del dominio presente el mismo comportamiento.

Un problema con esta implementacion, fue que al definir una clase de logica base para *Element* entonces metodos especificos para *Folder* o *File* eran inaaccesibles. Por que se implemento herencia de interfaces y se crearon dos interfaces hijas de *ILogic<Element>*

El otro aspecto es haber decidido implementar validadores propios como un paquete aparte. Este paquete tiene la sola responsabilidad de validar los objetos. En el caso de de los elementos, al tener casi los mismos comportamiento se implemento

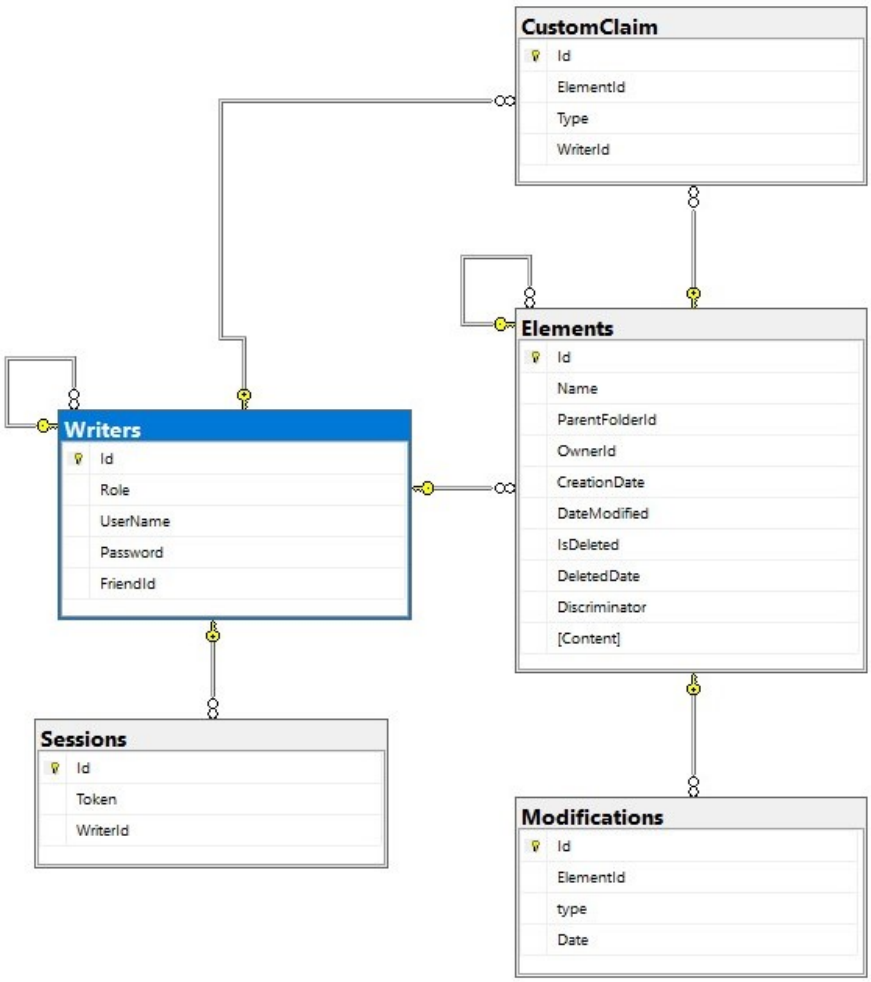
un patron template. Se definió el esqueleto del las validaciones necesarias para ambos objetos y luego se redefinieron los puntos en los que difieren en cada clase. Al igual que en el paquete de logica, se tuvieron que crear interfaces que hereden de *IValidator¡Element¿*

3.3. Data Access



En el Data Access tenemos un repositorio genérico del que heredan el resto de los repositorios. El objetivo es evitar repetir la misma implementación de esos métodos. El repositorio genérico implementa una interfaz que define el comportamiento general para todos los repositorios. El problema de este repositorio es traerse de la base de datos los atributo de navegación de cada clase, debido a que hay que pedirle a entity que los cargue. Para eso cada implementación redefine los métodos que trae los datos de la base. En el caso de los elements de nuevo se tuvo que implementar dos interfaces que hereden de la interfaz principal poder implementar métodos propios de los repositorios de Folder y File.

3.3.1. Entidades Base de Datos

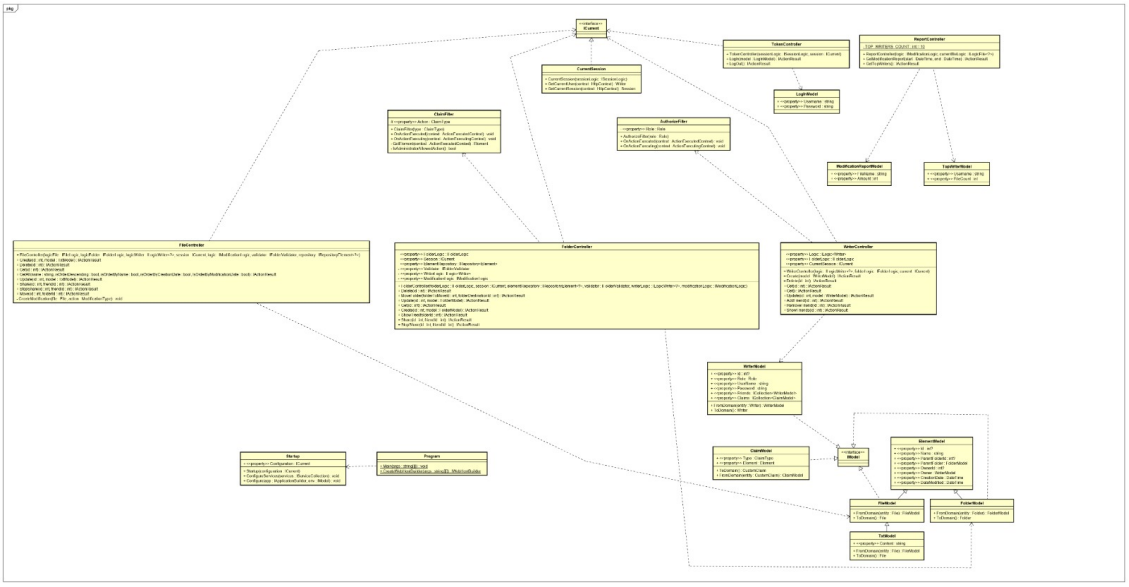


En este diagrama se puede apreciar, las relaciones de las entidades de la base de datos. En el caso de Writer y Folder que ambos poseen una lista de sus mismos elementos. Se le agrego en ambas clases dos atributos que le permiten a entity identificar el tipo de relación. En writer se agrego, "Friend" y "FriendId", mientras que en Element se agrego, "ParentFolder", "ParentFolderId".

Otro aspecto a destacar es que tenemos una unica clase de elementos que contiene tanto los archivos, como las carpetas. Los diferenciamos por su "Discriminator" que es un field agregado por entity framework que identifica el tipo de entidad que se esta guardando.

Pensado a futuro, si se fuera a agregar muchos tipos de archivos y muchos tipos de carpetas esta tabla crecería demasiado y tendría atributos que no son comunes a todos. Aunque entity cargue los datos según el tipo, tener tablas muy grandes puede llevar a problemas de performance. Sin embargo, como nuestra situación actual esta todavía muy lejos de esa situación, se decidio que se iba la iba a dejar por defecto.

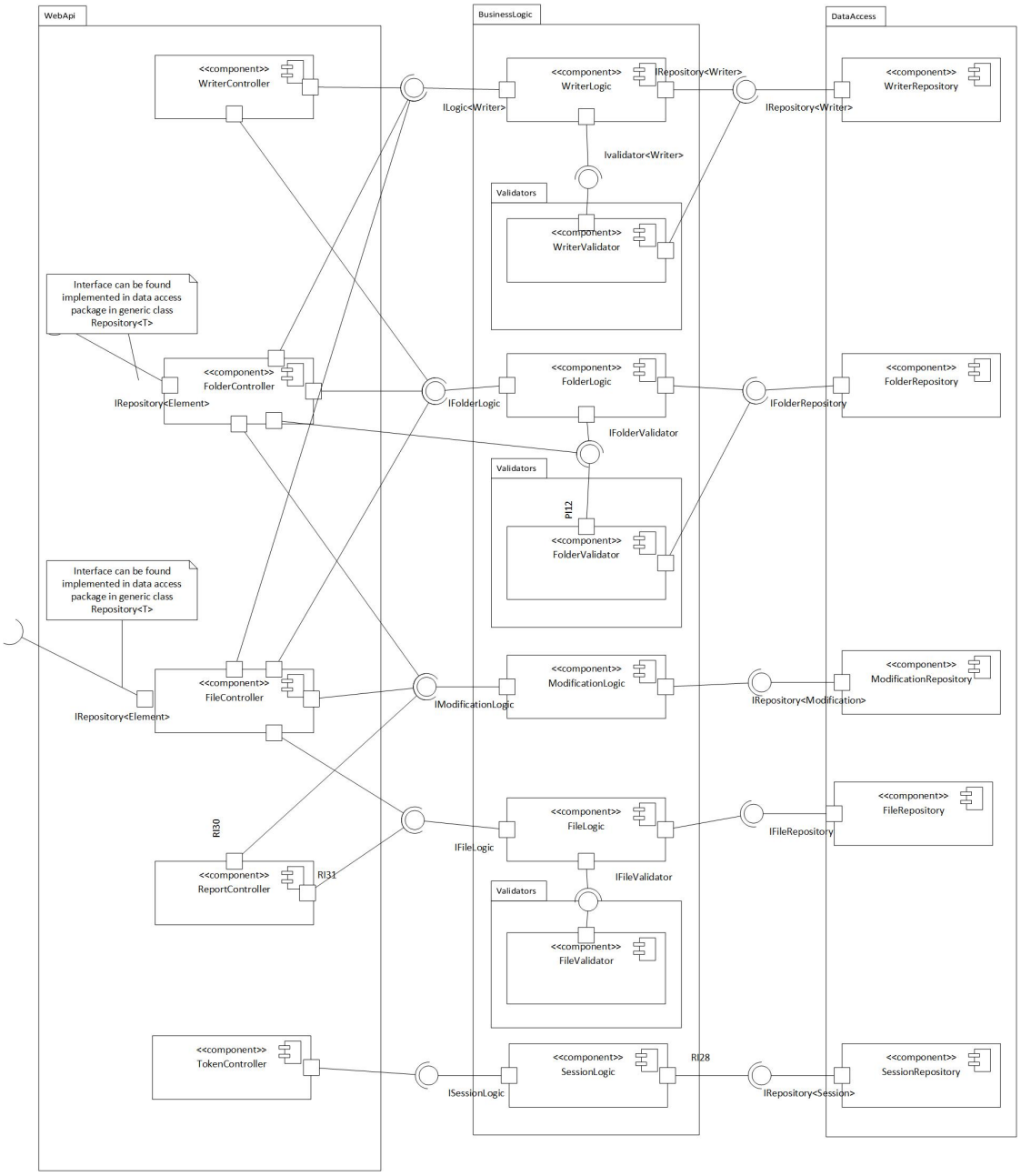
3.4. Web API



La Web Api se compone de 4 paquetes internos. El paquete de controllers tiene los 5 controllers del sistema. El paquete de filters se encarga de implementar los filtros de autorizacion que se usan en los diferentes filtros. Luego un paquete de interfaces, la razon por la que estas interfaces estan dentro del paquete y no por fuera como el resto son muy especificas de esta implementación de la web api. La interfaz *ICurrent* se utiliza como helper para mockear el httpcontext de las request. La de Model, podría extraerse a otras implementaciones pero no se considero necesario crear un paquete nuevo por una unica clase.

4. Componentes

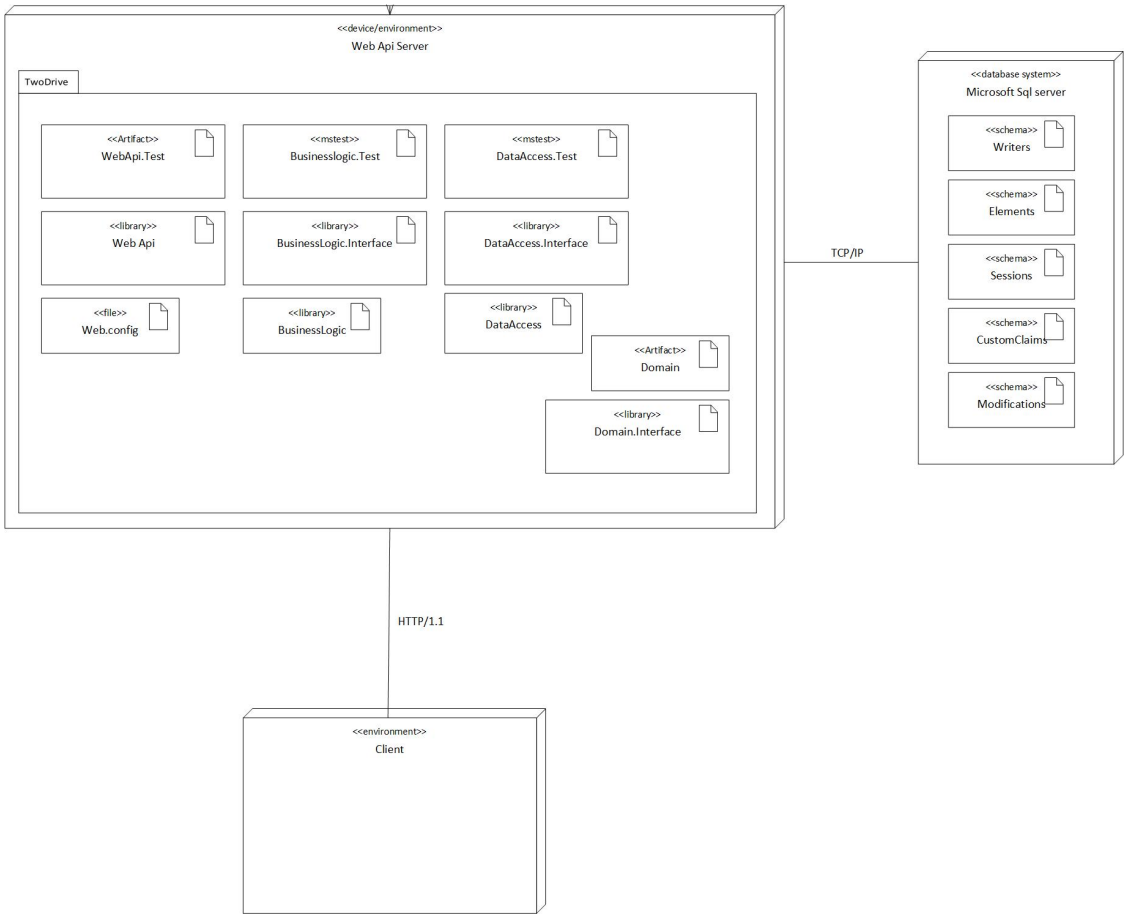
4.1. Diagrama de componentes



Este diagrama de componentes busca transmitir como los diferentes paquetes se

comunican entre si a través de las interfaces. Como mencionamos en la descripción de los paquetes, esto nos da la flexibilidad de sacar un paquete y reemplazarlo por otro, siendo le único impacto conectar las interfaces, pero todo el comportamiento detrás de ellas es totalmente desconocido por el nuevo paquete.

4.2. Diagrama de deploy

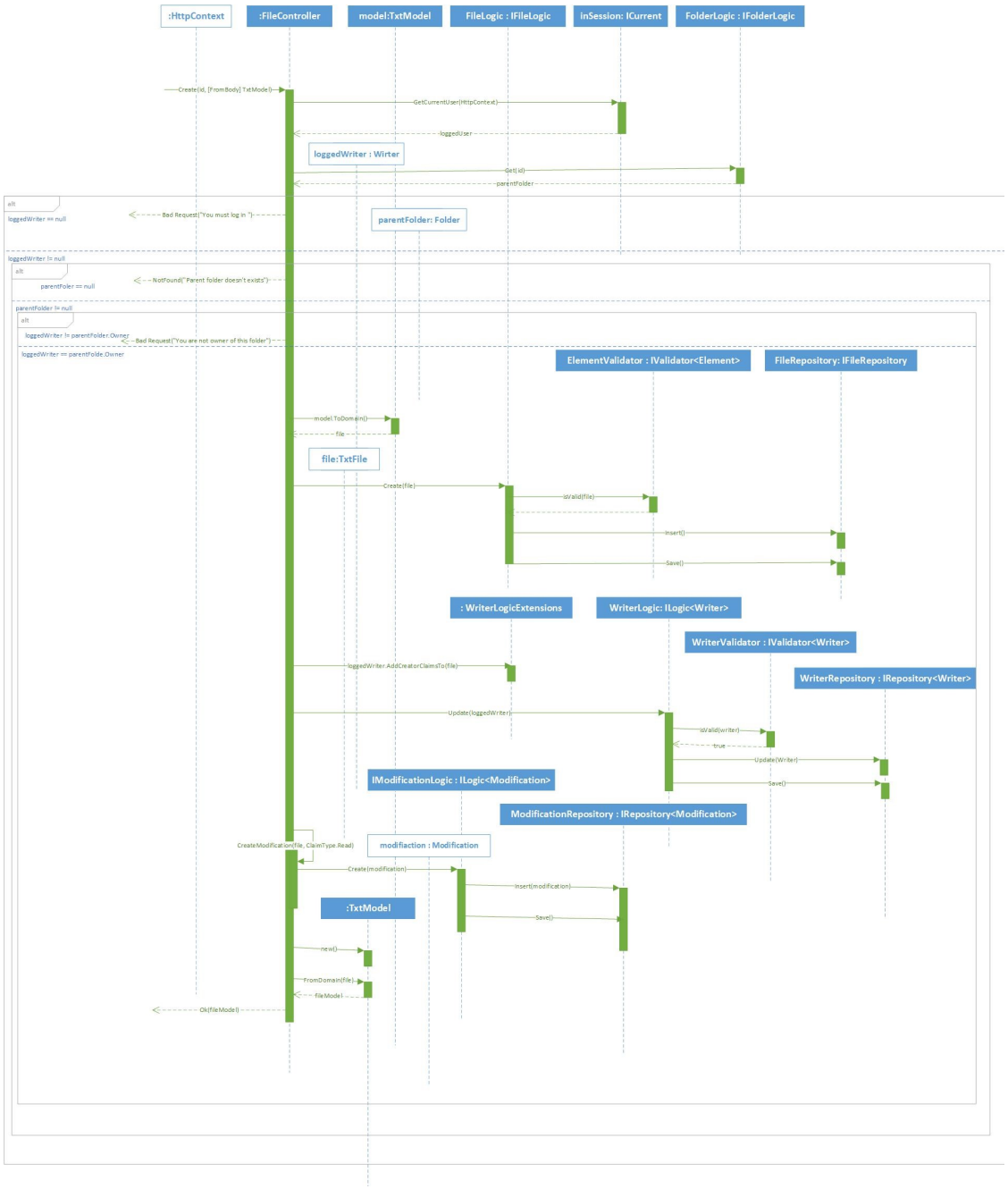


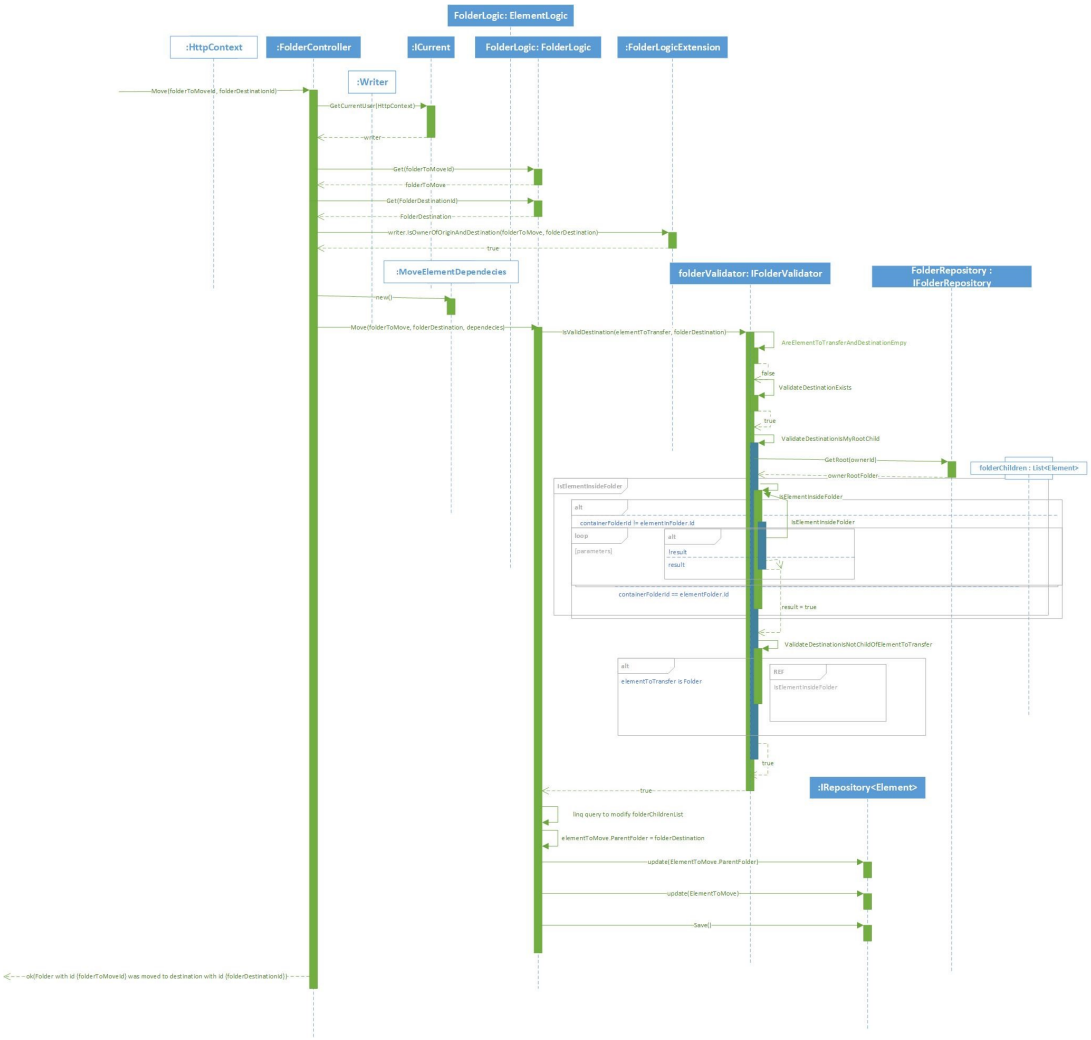
El paquete de deploy muestra todos los artefactos en el servidor donde se encuentra el sistema, los cuales son las dlls. Este servidor se comunica con el servidor de base de datos a través de protocolos TCP/IP. La base de datos tiene las tuplas con los datos necesarios para la correcta ejecución del sistema. Por ultimo, se encuentra el cliente, quien consume de nuestra api para obtener datos. En esta instacia el cliente esta representado por postman.

5. Interaccion entre clases

A continuación se presentan ejemplos del comportamiento del sistema a través de diferentes diagramas de interacción. El objetivo es que se visualize el flujo de las clases con las interfaces de por medio.

5.1. Sequencia




FolderLogic: ElementLogic
:HttpContext
:FolderController
:ICurrent
FolderLogic: FolderLogic
:FolderLogicExtension
:Writer
Move(folderToMoveId, folderDestinationId)
GetCurrentUser(HttpContext)
writer
Get(folderToMoveId)
folderToMove
Get(FolderDestinationId)
FolderDestination
writer.IsOwnerOfOriginAndDestination(folderToMove, folderDestination)
true
:MoveElementDependecies
folderValidator: IFolderValidator
FolderRepository : IFolderRepository
new()
Move(folderToMove, folderDestination, dependecies)
IsValidDestination(elementToTransfer, folderDestination)
AreElementToTransferAndDestinationEmpy
false
ValidateDestinationExists
true
ValidateDestinationIsMyRootChild
GetRoot(ownerId)
folderChildren : List<Element>
ownerRootFolder
IsElementInsideFolder
alt
containerFolderId != elementInFolder.Id
IsElementInsideFolder
loop
[parameters]
alt
!result
result
containerFolderId == elementFolder.Id
result = true
ValidateDestinationIsNotChildOfElementToTransfer
alt
elementToTransfer is Folder
REF
IsElementInsideFolder
true
true
:IRepository<Element>
linq query to modify folderChildrenList
elementToMove.ParentFolder = folderDestination
update(ElementToMove.ParentFolder)
update(ElementToMove)
Save()
ok(Folder with id {folderToMoveId} was moved to destination with id {folderDestinationId})

5.2. Colaboracion