

Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de aplicaciones 2

Obligatorio 2

Juan Pablo Poittevin(169766)

Joselen Cecilia(233552)

Entregado como requisito de la materia Diseño de
aplicaciones 2

Link al repositorio de GitHub

[https://github.com/ORT-DA2/Poittevin-169766-
-Cecilia-233552-](https://github.com/ORT-DA2/Poittevin-169766-Cecilia-233552)

17 de junio de 2021

Índice general

| | |
|--|-----------|
| 1. Descripción del diseño | 3 |
| 1.1. Suposiciones y decisiones de diseño | 3 |
| 1.2. Descripción general del diseño | 5 |
| 1.3. Diagrama general de paquetes | 6 |
| 1.4. Diagrama de componentes | 7 |
| 1.5. Asignación de responsabilidades | 8 |
| 1.5.1. Importer | 8 |
| 1.5.2. Importer.Json | 10 |
| 1.5.3. Importer.Xml | 10 |
| 1.5.4. BusinessLogic | 10 |
| 1.6. BusinessLogicInterface | 13 |
| 1.6.1. Domain | 14 |
| 1.6.2. DataAccess | 16 |
| 1.6.3. WebAPI | 18 |
| 1.7. Descripción del mecanismo de acceso a datos utilizado | 19 |
| 1.7.1. width= | 19 |
| 1.8. Descripción del manejo de excepciones | 20 |
| 1.9. Diagrama de secuencia Import | 20 |
| 1.10. Diagrama de secuencia LogIn | 21 |
| 1.11. Diagrama de secuencia bonificación paciente | 21 |
| 1.12. Justificación del diseño en base a principios y patrones | 21 |
| 1.12.1. Diseño general frontend | 24 |
| 1.13. Justificación del diseño en base a métricas | 25 |
| 1.14. Métrica de Estabilidad | 25 |
| 1.15. Métrica de la abstracción | 26 |
| 1.16. Métrica de distancia | 26 |
| 1.17. mecanismos utilizados para permitir la extensibilidad | 29 |
| 1.18. Resumen mejoras | 31 |
| 1.19. Cambios de la API | 32 |
| 1.20. Evidencia de TDD | 33 |
| 2. Anexo | 34 |
| 2.1. Anexo 1: Paquete BusinessLogic | 34 |
| 2.1.1. Services | 34 |
| 2.2. Anexo 2: Objetos DTO DataAccess | 35 |
| 2.3. Anexo 3: diagrama de secuencia manejo de excepciones | 38 |

| | | |
|--------|---|----|
| 2.4. | Anexo 4: diagrama de secuencia import | 39 |
| 2.5. | Anexo 5: diagrama de secuencia login | 40 |
| 2.6. | Anexo 6: diagrama de secuencia bonificación paciente | 41 |
| 2.7. | Anexo 7: Descripción de la API | 41 |
| 2.7.1. | Criterios REST | 41 |
| 2.7.2. | Descripcion endpoints | 41 |
| 2.7.3. | Descripción de códigos de éxito y error | 43 |
| 2.7.4. | Descripción proceso de Autenticación | 44 |
| 2.8. | Anexo 8: evidencia de TDD y Clean Code | 44 |
| 2.8.1. | Descripción de la estrategia de TDD seguida | 45 |
| 2.8.2. | Informe de cobertura para todas las pruebas desarrolladas | 45 |
| 2.8.3. | Evidencia de Clean Code | 49 |

1. Descripción del diseño

1.1. Suposiciones y decisiones de diseño

Para realizar el diseño del obligatorio nos basamos en la letra del mismo y en las preguntas que se fueron realizando en el foro, además hicimos algunos supuestos:

1-Al agregar una playlist, si se agrega con esta un Audio o se le asocia luego una Audio que en ambos casos o este en la base de datos, esta song "hereda" las categorías de la playlist.

2-El nombre de un Audio no puede ser vacío.

3-El nombre de una playlist no puede ser vacío.

4- La duración de un Audio se ingresa como un string; este string debe ser del formato dh o dm o ds, donde d seria la duración del mismo. Esta duración se maneja luego dentro del sistema como un double en segundos. Si la duración es menor a 60 minutos al mostrar un audio esta se muestra como dm (duración en minutos) en caso contrario si la duración es mayor a 60 minutos se muestra como dh (duración en horas). Para esto creamos el objeto AudioDto que tiene la propiedad duration de tipo string.

5-Al momento de listar los audios, si un audio se encuentra en la relación playlist audio, significa que este esta asociado a una playlist(no esta suelto) por lo cual no se muestra, si no que se muestra solo dentro de la playlist/s correspondiente/s.

6- Si al momento de ingresar una song o una playlist se ingresa una categoría que no esta en la base de datos significa que se intenta asociar a una categoría que no esta dentro de las fijadas ("Dormir", "Meditar", "Música" y "Cuerpo") entonces se lanza un mensaje de error indicando que la categoría es incorrecta, de igual manera funciona al momento de asociar una problematic si esta no esta dentro de las fijadas ("depresión", "estrés", "ansiedad", "autoestima", "enojo", "relaciones", "duelo", "otros")

7-No se pueden ingresar dos audios con la pareja nombre del audio - autor repetida, ya que en este caso se estaría registrando dos veces el mismo audio.

8- Se decidió agregar Problematics y Categories a la BD para de esa forma tener las opciones de filtros a nivel de BD.

9-Un paciente no puede tener dos consultas el mismo día, con el mismo psicólogo, ya que no se maneja la hora de la consulta, solo la fecha por letra.

10-En las consultas no se muestran las columnas que en la bdd están guardadas como null, para esto se ignora en el startup los valores null de la siguiente manera; `.options.SerializerSettings.NullValueHandling = NullValueHandling.Ignore;`

11-Para evitar el ciclo que se generaba con la relación; patient-meeting y meeting patient y psychologist-meeting y meeting psychologist, ignoramos en la clase startup las referencias que generan loop de la siguiente manera; `.options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore`

12- Para agendar una consulta, se busca semana a semana, hasta encontrar una fecha libre.

13- Para devolver el error correspondiente, con el mensaje, el código de error, el content y si es success, en el filterException, creamos el objeto ErrorDto.

14- Para recibir en el body el patient y la problematic, para agendar una nueva consulta, creamos el objeto ScheduleMeetingDto, que tiene un patient y una problematic.

15- La descripción de una playlist debe ser menor a 150 caracteres, en caso contrario se lanza una exception.

16-En caso de que halla dos psicólogos disponibles, se asigna el mas antiguo, entendemos por antigüedad la fecha en que este se registró en el sistema (created date).

17- En los métodos de las clases mappers tenemos que validar que los objetos devueltos por el firstOrDefault no sean nulos, es por esto que hacemos if objeto is null, ya que este método es ajeno a nosotros decidimos validar esa condición en lugar de tirar una excepción como recomienda clean code, de lo contrario puede generar errores.

18- Decidimos usar objetos DTO correspondientes a cada objeto del dominio y a las relaciones n a n para persistir en la base de datos, en los cuales indicamos cual es el ID de cada uno, junto con sus datos y las relaciones entre ellos.

20-Asumimos que para el correcto funcionamiento del sistema, quien lo vaya a utilizar deberá tener correctamente configurado el connectionString en el archivo AppSettings.json.

21- Al momento de importar contenido si dentro del archivo importador hay contenidos con errores, esos contenidos no se agregan pero si se agregan los demás.

22-Se decide dejar para el importador un `choose file` por mas que no sea tan extensible a nuevos parámetros porque es mas fácil que poner una ruta a mano, en caso de cambiar los parámetros se cambia este por un `input` y se sigue usando igual.

1.2. Descripción general del diseño

Para la solución del obligatorio se implementa una WEB API para el MSP (ministerio de salud publica) llamada BetterCalm que ayuda a que las personas aprendan a sobrellevar el estrés de una manera mas sana para que cada persona y sus seres queridos puedan desarrollar una mejor capacidad de adaptación y resiliencia.

El sistema tiene dos funcionalidades principales e independientes; estas dos funcionalidades son, reproductor y consulta con un psicólogo.

En la funcionalidad reproductor los usuarios pueden acceder a contenido reproducible el cual puede ser formato audio o video a traves de la pestaña content que despliega todos los contenidos y donde se pueden filtrar los mismos para facilitar la busqueda, un vez el usuario encuentre el contenido que busca puede clicar en este para acceder al detalle del mismo y en caso de ser vídeo poder visualizar este.

Estos contenidos están disponibles a través de diferentes playlists las cuales se listan en la pestaña playlist y al igual que los contents se pueden filtrar para facilitar la búsqueda, un vez el usuario encuentre la playlist que busca puede clicar en esta para acceder al detalle de la misma, y ver que contenidos tiene.

Dicho contenido es subido ya sea por diferentes administradores que respaldan el sistema o importado desde diversas fuentes de datos y formatos a través de la pestaña import, si un tercero desea puede importar desde otro formato el cual debe implementar ya que el sistema esta pensado para que esto sea posible.

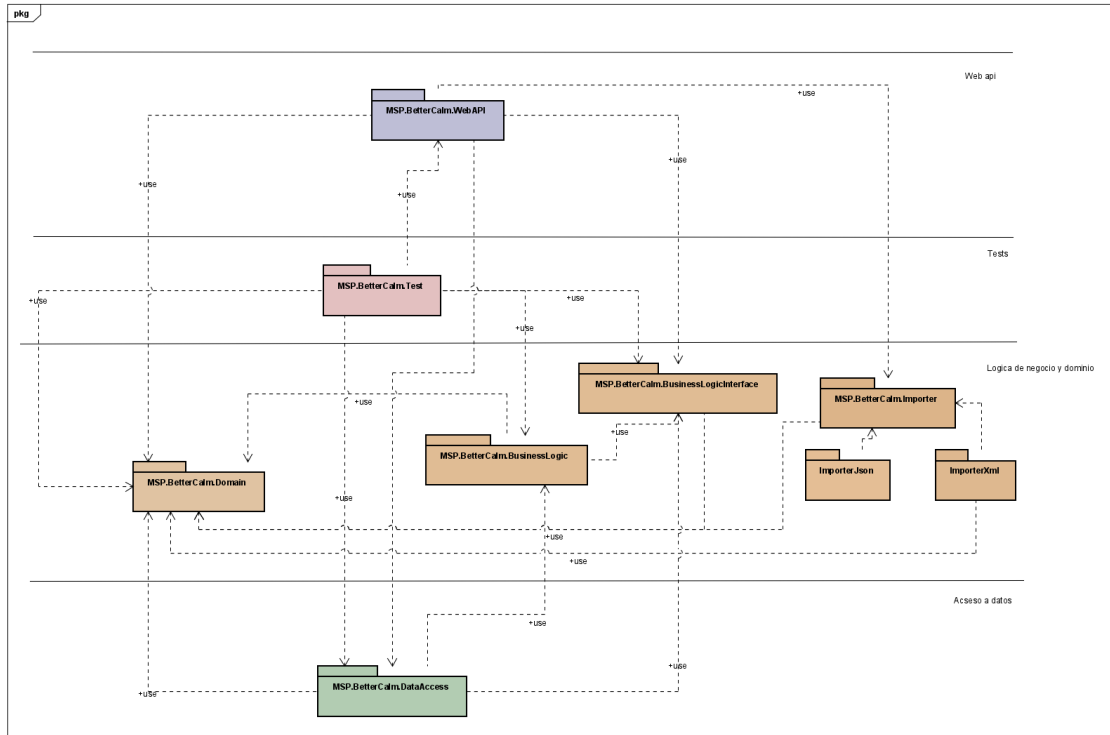
En la funcionalidad consultas con un psicólogo: los usuarios tienen la posibilidad de agendar consultas con un psicólogo de acuerdo a sus dolencias actuales.

Al acceder a esta funcionalidad los mismos deberán elegir cuál es su principal dolencia/problemática actual y posteriormente, en base a esa información, la aplicación lo “matchea” con el psicólogo más acorde de acuerdo a la información recolectada en el paso previo y la disponibilidad de psicólogos en esa semana.

Al finalizar la reserva los pacientes reciben el costo de la consulta el cual se calcula según la duración de la misma y la tarifa de cada psicólogo, además se pueden incluir descuentos si el paciente ya hizo mas de 5 consultas consecutivas.

1.3. Diagrama general de paquetes

Dentro de nuestro sistema tenemos los siguientes paquetes, con las distintas responsabilidades:



WebAPI

-Se encarga de definir los endpoint y procesar las request de los clientes. Dentro tenemos tres carpetas distintas una que contiene los controller, otra que contiene los filters y una llamada parser para los import.

Domain

-Contiene las entidades del proyecto, y una carpeta con las excepciones creadas por nosotros.

BusinessLogic

-En este paquete se encuentran los manager de cada repositorio, la implementación de los services y las excepciones, es el paquete que contiene la lógica.

BusinessLogicInterface -En este paquete se encuentran las interfaces de la lógica, los IServices, además del contrato del repositorio genérico IRepository.

DataAccess

-Dentro de este paquete se encuentran las clases que permiten el acceso a la base de datos, los dtos, mappers, contexto y las inyecciones de dependencia. También se encuentran en este paquete los repositories de cada entidad.

Test

-Este paquete tiene dentro una división en carpetas, una carpeta test por cada paquete antes mencionado, donde se encuentran los test respectivos a las las clases de esos paquetes.

Como vemos entonces en nuestro sistema se conservan los mismos paquetes que en la primer entrega y a estos se suman tres nuevos:

Importer En este paquete se encuentran los contratos para los importadores, la interface, los parámetros y el objeto importador, además de los objetos que se usan para importar el contenido.

ImporterJson En este paquete se encuentra la lógica de importación para los archivos de tipo json.

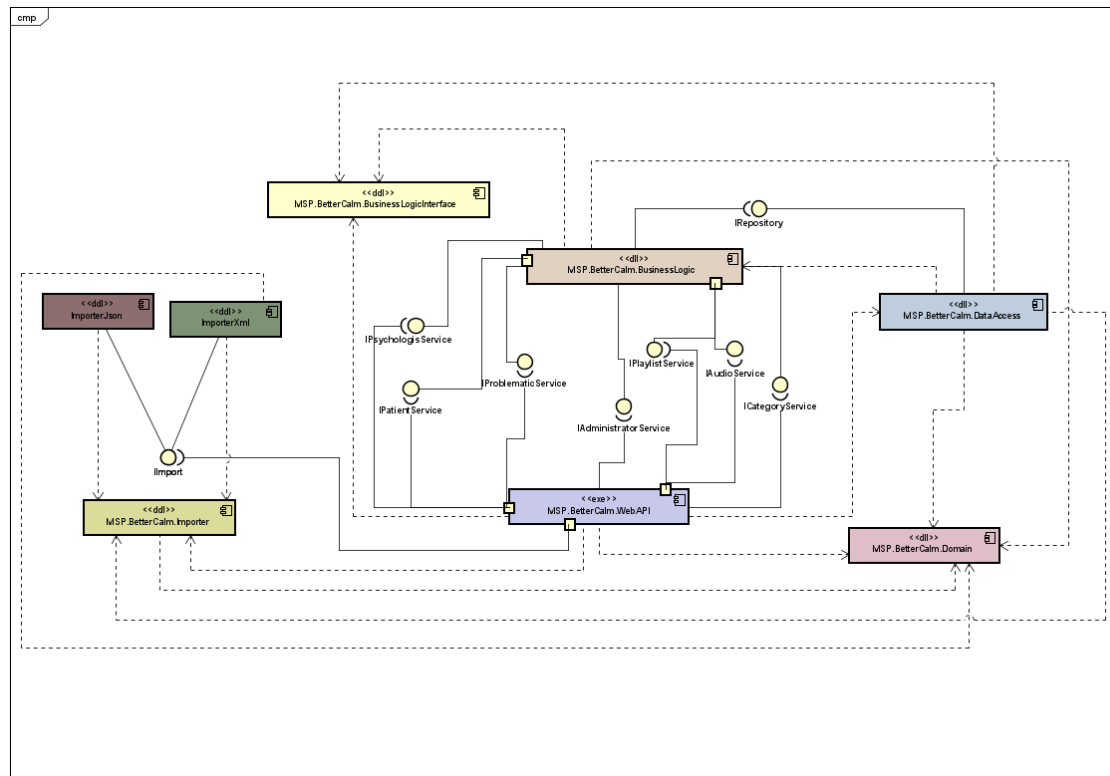
ImporterXml En este paquete se encuentra la lógica de importación para los archivos de tipo xml.

1.4. Diagrama de componentes

En este diagrama se muestra la interacción entre los componentes, la cual puede ser por; relación interfaz provista/requerida o por relación dependencia/uso.

Como vemos en el diagrama WebAPi conoce todas las implementaciones de los IService y de IImport dado que es quien resuelve las dependencias, usando inyección de dependencias.

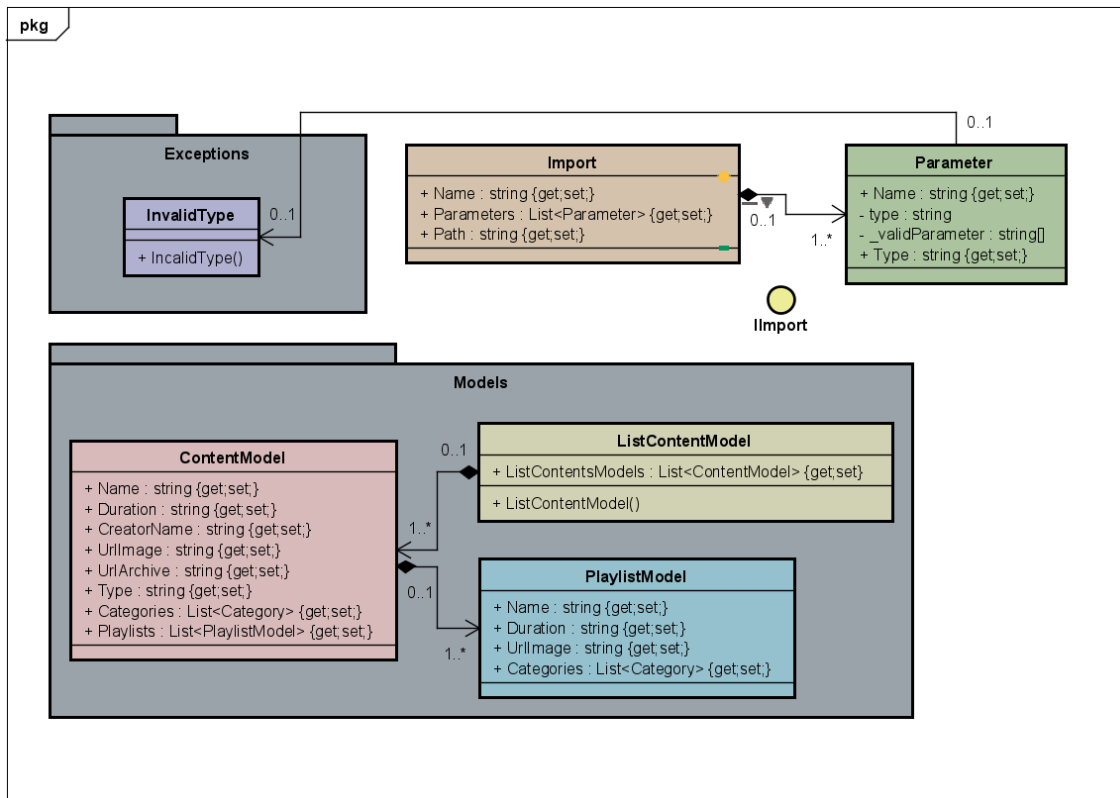
También vemos que los paquetes ImporterJson e ImporterXml implementan la clase IImport que se encuentra en el paquete Importer.



1.5. Asignación de responsabilidades

1.5.1. Importer

En este paquete se encuentra el contrato de lo que un importador de terceros debe cumplir para poder importar nuevo contenido para esto tenemos las clases:



Import Esta clase es el objeto que se recibe cuando se va a importar contienen una lista de parametros que se explica mas adelante que son, un name y el path donde se encuentra el archivo con el contenido

IImporter esta clase es la interface que contiene los métodos necesarios para crear un importador; obtener el nombre, los parámetros e importar el contenido.

Parameters Esta clase obliga a que todos los importadores tengan el formato de parámetros (tipo,nombre), el tipo se restringe a "string", "int", "date", "bool".

En este paquete también se encuentran en la carpeta Models, los objetos intermedios que se usan para pasar del contenido en el archivo de importación a objetos de dominio

ContentModel Esta clase es el objeto intermedio que se usa para pasar del resultado del archivo importador (ListContentModel) al objeto content del dominio por lo tanto tiene los mismo atributos.

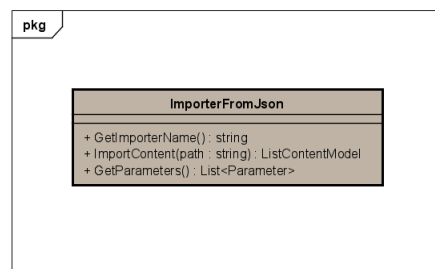
PlaylistModel Esta clase es el objeto intermedio que se usa para pasar del resultado del archivo importador (ListContentModel) al objeto del dominio playlist por lo tanto tiene los mismo atributos.

A diferencia del dominio en los models es ContentModel quien tiene lista de PlaylistModel.

ListContentModel Esta clase es el objeto que recibimos cuando se importa un contenido, agrupa las listas de los contenidos.

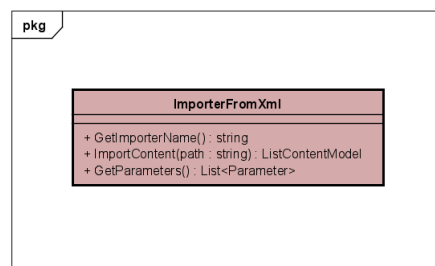
1.5.2. ImporterJson

En este paquete tenemos una única clase **ImporterJson** que se encarga de implementar la interface **IImporter** para el tipo json, esta clase define como nombre de importador Json y a los parámetro como un path de tipo string, luego en el método import content se deserealiza el archivo pasado en el path y se asigna al objeto ListContentModel.



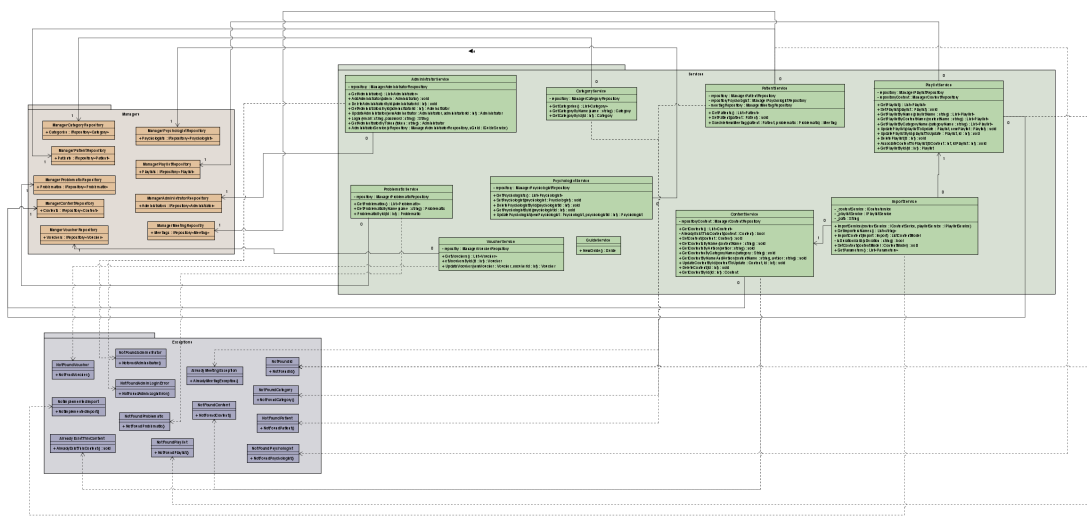
1.5.3. ImporterXml

En este paquete tenemos una única clase **ImporterXml** que se encarga de implementar la interface **IImporter** para el tipo Xml, esta clase define como nombre de importador Xml y a los parámetro como un path de tipo string, luego en el método import content se deserealiza el archivo pasado en el path y se asigna al objeto ListContentModel.

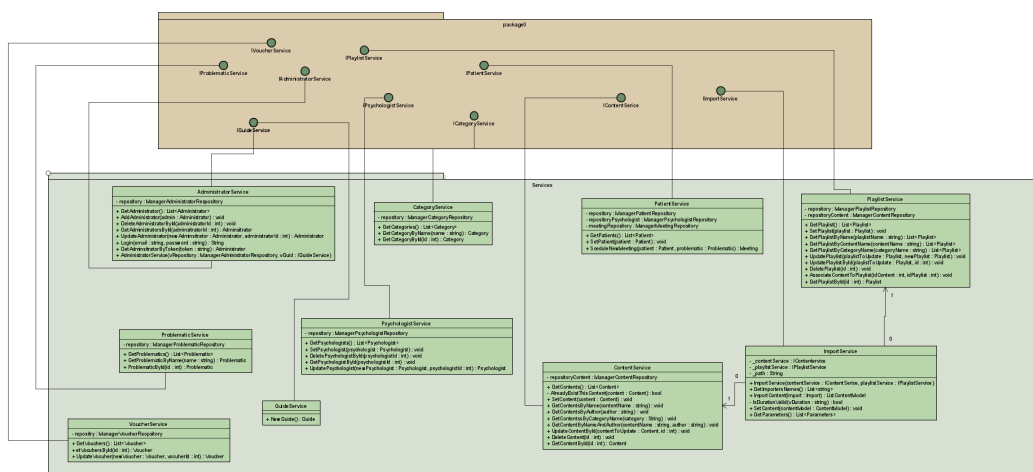


1.5.4. BusinessLogic

Dentro de BusinessLogic creamos tres carpetas para agrupar las clases, y así facilitar la búsqueda de estas:



- El **namespace Services** contiene las clases para implementar los Iservices del paquete BusinessLogicInterface, en el siguiente diagrama se muestra esta relación



Los services que no se mencionan se mantienen igual y con la misma funcionalidad que la entrega 1, en caso de dudas ver [Anexo 1: paquete BusinessLogic]

IContentService y ContentService: Se crea la clase IContentService que es la interfaz implementada por ContentService para poder hacer inyección de dependencias, estas clases se encargan de definir e implementar las funcionalidades que se necesitan desde ContentController; agregar, actualizar, eliminar y buscar las contents, usando la instancia ManagerContentRepository, para poder acceder a las contents ingresadas en el sistema. Esta clase es la intermediaria entre la API(ContentController) y el acceso a los datos (ManagerContentRepository).

IImportService y ImportService Se crea la clase IImportService que es la interfaz implementada por ImportService para poder hacer inyección de dependencias,

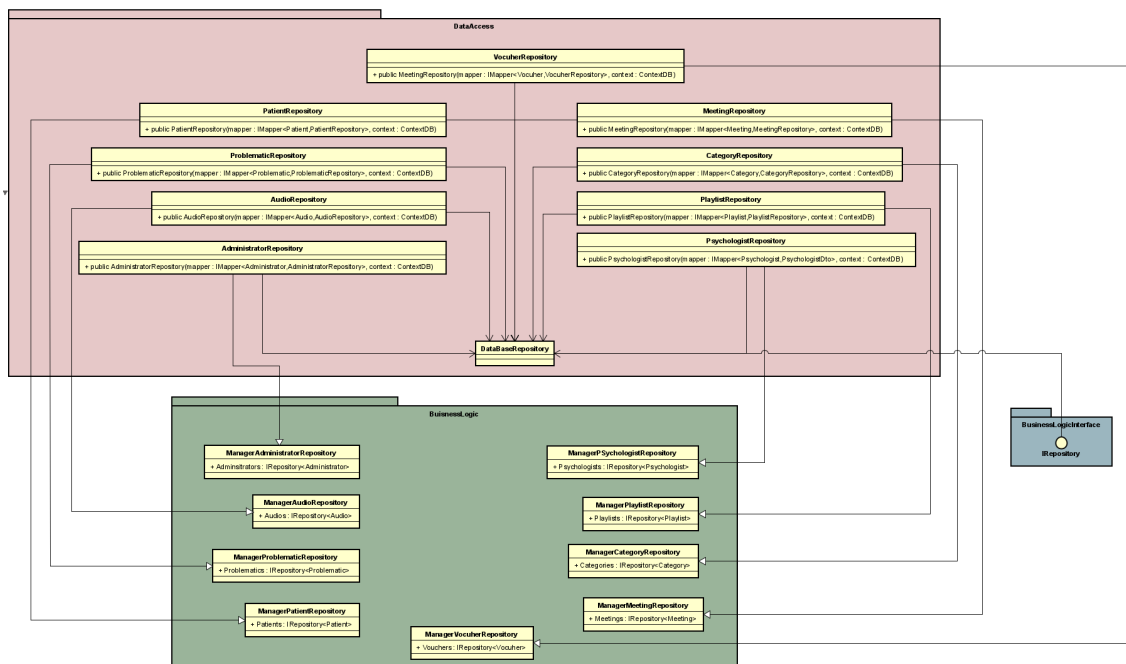
estas clases se encargan de definir e implementar las funcionalidades que se necesitan desde ImportController; obtener los parámetros, el nombre del importador e importar nuevo contenido, usando IContentService e IPlaylistService para guardar el nuevo contenido.

IVoucherService y VoucherService Se crea la clase IVoucherService que es la interfaz implementada por VoucherService para poder hacer inyección de dependencias, estas clases se encargan de definir e implementar las funcionalidades que se necesitan desde VoucherController; obtener los vouchers, filtrarlos por id y actualizarlos.

-El **namespace Managers** contiene los managerRepository de cada entidad:

Estas clases se crean para invertir la dependencia y que el dataAccess dependa de businessLogic, de esta forma es el businessLogic quien le marca al dataAccess que funcionalidades debe implementar. Es en esta clase donde se guarda el IRepository de las distintas entidades: content, Playlist, Category, Problematic, Psychologists, Patient, Administrator, Meeting.

En el siguiente diagrama se muestra la relación entre los repositories de DataAccess con el IRepository de BusinessLogicInterface y managers de BusinessLogic.



-Por ultimo tenemos dentro de este paquete el **namespace Exceptions** que contiene las clases con las excepciones creadas por nosotros.

NotFoundAdministrator, NotFoundcontent, NotFoundCategory, NotFoundId, NotFoundPlaylist, NotFoundProblematic, NotFoundPsychologist, NotFoundVoucher Estas clases NotFound[Object], heredan de la clase exception, y se

crea para lanzar esta excepción cuando no se encuentra un objeto, se usa en los find, get, delete y update.

NotFoundId Esta clase NotFoundId, hereda de la clase exception, y se crea para lanzar esta excepción cuando se hace la búsqueda por un Id que no se encuentra actualmente en la base de datos.

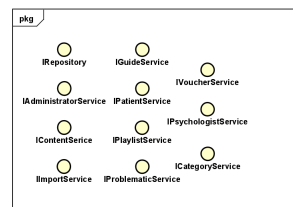
InvalidCategory, InvalidProlematic Estas clases InvalidCategory y InvalidProlematic, heredan de la clase exception, y se crean para lanzar esta excepción cuando se intenta asociar una problematic o una category que no son de las fijas, que se encuentran en la base de datos.

AlreadyExistThiscontent, AlreadyMeetingException Estas clases AlreadyExistThiscontent, AlreadyMeetingException, heredan de la clase exception, y se crean para lanzar esta excepción cuando se intenta agregar un objeto que ya existe en la base de datos.

InvalidContentType Esta clase InvalidContentType, hereda de la clase exception, y se crea para lanzar esta excepción cuando se agrega un content con un tipo que no es audio o video.

1.6. BusinessLogicInterface

En este paquete se encuentran como mencionamos anteriormente las interfaces que se implementan en BusinessLogic



IRepository Esta clase se encarga de mostrar la firma, de los métodos básico para manejar un Repository, sin importar de que tipo sea la entidad, es decir es el contrato de este, es una clase interfaz generic, que es implementada por DataBaseRepository la cual se encuentra en el paquete DataAcces.

Como vemos en el siguiente diagrama tenemos un IRepository de cada entidad del dominio.

los datos de Category; id y name, ver si dos categories son iguales y comparar los nombres de dos categories.

Patient: esta clase hereda de la clase User y se encarga de: crear los objetos de tipo Patient, manejar los datos del Patient; id, cellphone ,Birthday y meetings, y ver si dos patients son iguales.

Playlist: esta clase se encarga de: crear los objetos de tipo Playlist, manejar los datos de playlist; id y name, urlImage, description, categories y contents, ver si dos Playlists son iguales, comparar los nombres de dos Playlists y comparar los nombres de las content con el nombre de una content particular.

Problematic: esta clase se encarga de: crear los objetos de tipo Problematic, manejar los datos del Problematic; id y name, ver si dos problematics son iguales, y comparar los nombres de dos Problematics.

Psychologist: esta clase hereda de la clase User y se encarga de: crear los objetos de tipo psychologist, manejar los datos del Psychologists; id, Address,worksOnline, Problematics, Meetings, Creation date, Rate, ver si dos Psychologist son iguales, y obtener el siguiente dia libre para agendar una meeting.

Content: esta clase se encarga de: crear los objetos de tipo Content, manejar los datos del Content; id, name, urlImage, duration, creatorName, categories y urlArchive, associatedToPlaylist y type que indica si el contenido es de tipo content o video, ver si dos contents son iguales y comparar los nombres de dos Contents.

User: esta clase se encarga de: crear los objetos de tipo User, y manejar los datos de User; name y lastName.

Meeting: esta clase se encarga de: crear los objetos de tipo Meeting, manejar los datos de la meeting; id, psychologist, patient, datetime y address, Times, y ver si dos meetings son iguales.

Voucher: esta clase se encarga de: manejar las bonificaciones de los pacientes, y manejar los datos de los bonos; id, Patient, status, discount, MeetingsAmount.

Rates, Times, Discounts, Status:

Estos cuatros son enums utilizados para la parte de consultas con el psicologo;

-Times: Short,Medium, Long

-Rates: Cheap = 500, Medium = 750, Expensive = 1000, Delux = 2000

-Status: Approved, Rejected, Used, Pending, NotReady

-Discounts: Low = 15, Medium = 25, Large = 50

-Por ultimo tenemos dentro de este paquete el **namespace Exceptions** que contiene las clases con las excepciones creadas por nosotros.

InvalidDescriptionLength: hereda de la clase exception, y se crea para lanzar esta excepción cuando el largo de la descripción es mayor a 150.

InvalidNameLength hereda de la clase exception, y se crea para lanzar esta excepción cuando el largo del name es vacío.

InvalidDurationFormat: hereda de la clase exception, y se crea para lanzar esta excepción cuando la duración de un contenido no esta en el formato correcto numero seguido de s,d,m.

InvalidContentType: hereda de la clase exception, y se crea para lanzar esta excepción cuando el tipo de un contenido no es audio o video.

InvalidAmountOfProblematicsError: hereda de la clase exception, y se crea para lanzar esta excepción cuando la cantidad de problemáticas de un psicólogo es menor a tres.

InvalidMeetingDuration: hereda de la clase exception, y se crea para lanzar esta excepción cuando la duración de una meeting es incorrecta (2, 1 o 1.5).

InvalidUrl: hereda de la clase exception, y se crea para lanzar esta excepción cuando la url del archivo o de la imagen de un contenido o una playlist no son correctos.

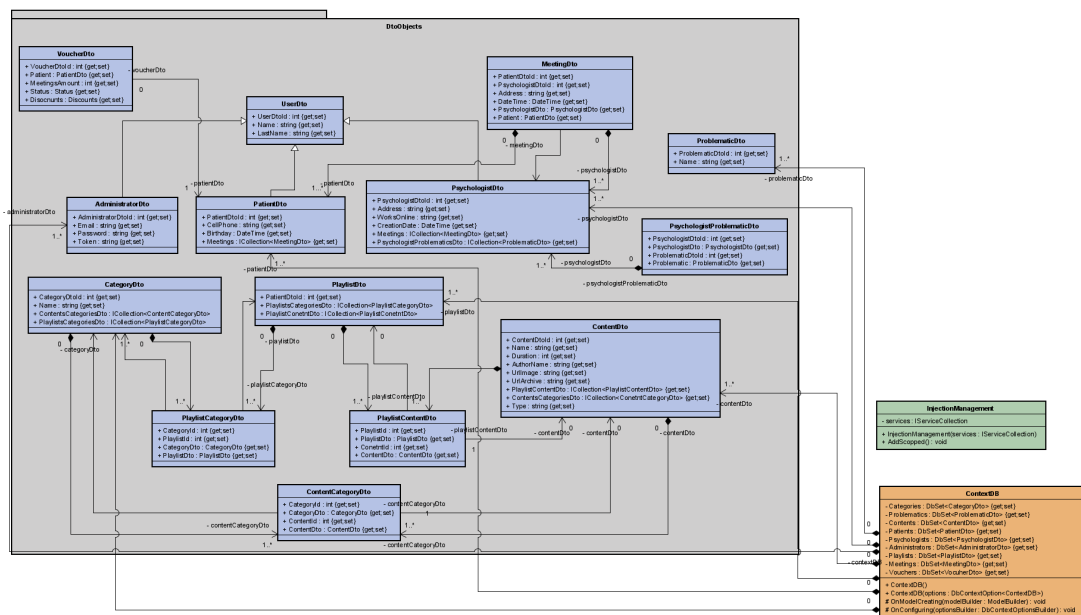
VoucherAlreadyClosed: hereda de la clase exception, y se crea para lanzar esta excepción cuando se esta esperando por la aprobación del admin para usar la bonificación o esta bonificación ya se uso.

1.6.2. DataAccess

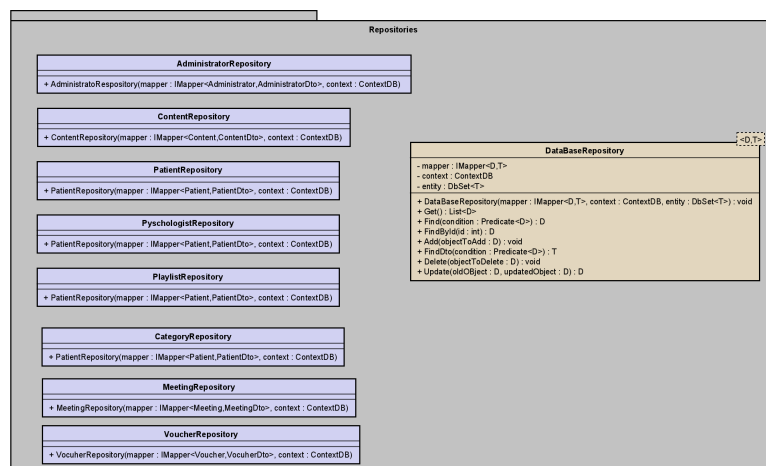
En el paquete DataAcces creamos cuatro carpetas que agrupan las clases para facilitar la búsqueda de estas, ademas se encuentran las clases contextDB e Injection-Manager

DtoObjects contiene las clases de los objetos de tipo Dto, respectivo a cada objeto del dominio:

Estas clases se mantiene similares a la entrega 1 con algunos cambios, para ver el detalles de estos ir a [Anexo 2: objetos DTO DataAccess]



Repositories contiene los repositories de cada object y la clase DataBaseRepository.



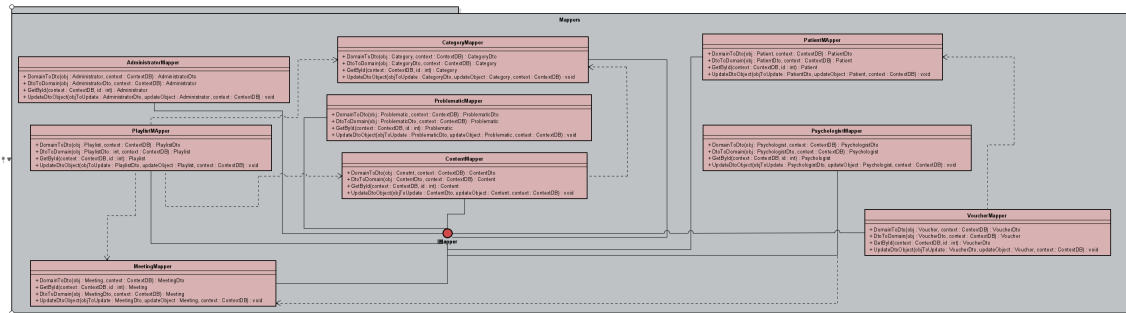
[Object]Repository: como se muestra en el diagrama hay una de estas por cada objeto, estas clases heredan de la clase Manager[Object]Repository correspondiente, y se encargan de inicializar el IRepository de ese objeto.

DataBaseRepository: esta clase se encarga de implementar la clase Irepository, por lo tanto al igual que esta es una clase generic, se implementa para el almacenamiento en base de dato, esta clase usa dos tipos genéricos donde uno de ellos representa los objetos de dominio y el otro los objetos de la base de datos, los cuales son mapeados con los métodos mappers que mencionaremos.

Mappers contiene las clases mapper de cada objeto del dominio; estas clases heredan de la clase IMapper que también se encuentra en este paquete y se encarga de

definir la funcionalidad que debería tener una clase mapper.

Los mapper se encargan de pasar los objetos del dominio a objetos de la base de datos y viceversa, dentro de estas clases también tenemos el método update que se encargan de actualizar los objetos si ya están registrados en la base de datos y el metodo GetById, que dado un id me retorna el objeto de dominio correspondiente al objeto en la base de datos con ese Id.



-Además de estas clases mencionadas tenemos en el paquete DataAccess las clases:

ContextDB: esta clase hereda de la clase DbContext la cual permite consultar, crear, editar y eliminar registros en una base de datos. En esta clase se encuentran almacenados en DbSets los objetos Dto antes mencionados, respectivos a los objetos del dominio. En esta clase también se encuentra en el método OnModelCreating, la configuración de las relaciones n-n; de playlist con category, de playlist con Content y de Content con category. Y la configuración de la conexión a la base de datos.

InjectionManagement: en esta clase se hace con el metodo AddScoped la inyección de dependencias de los services, mappers y de los manager repositories. También se hace la inyección, con el metodo AddDbContext, del ContextDb.

1.6.3. WebAPI

Dentro de WebAPI creamos cuatro carpetas para agrupar las clases:

Controllers dentro de esta carpeta se encuentran los controllers con los endpoints respectivos a cada objeto del dominio, Los cuales se explican en la sección de descripción de la API

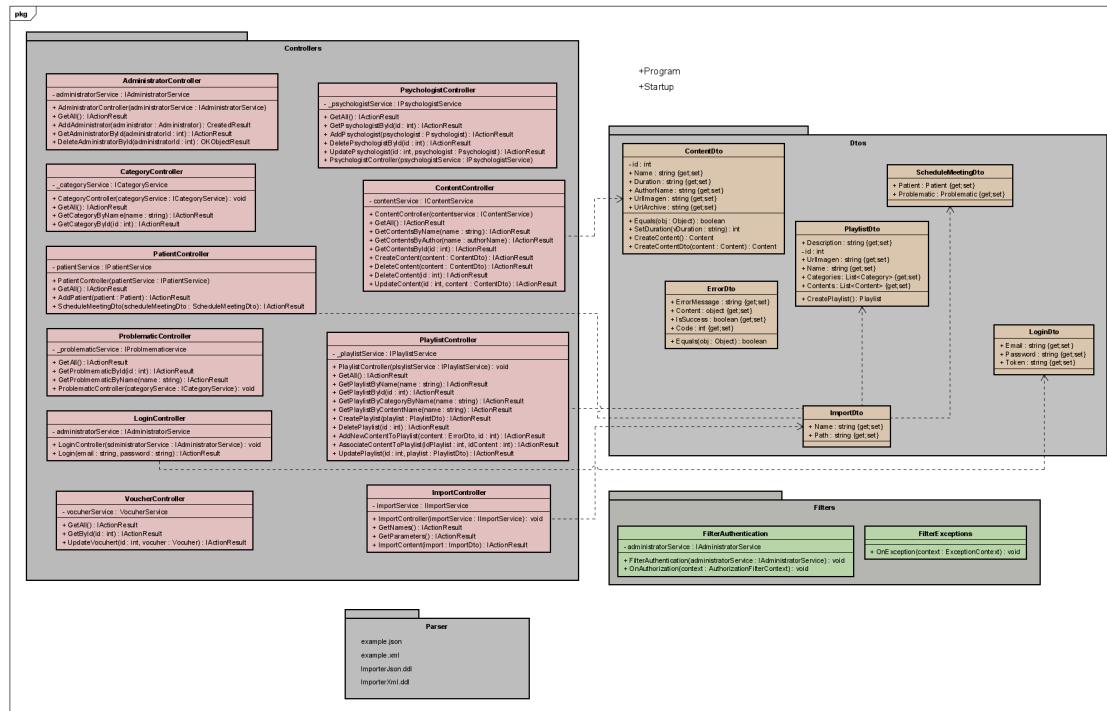
Filters dentro de esta carpeta se encuentran las clases AuthenticationFilter y el ExceptionFilter

La clase **AuthenticationFilter** se encarga de verificar que para ciertas funcionalidades siempre halla un adminitrador logueado, en caso contrario lanza una excepcion.

La clase **ExceptionFilter** se encarga de capturar las excepciones y devolver un objeto de error acorde a esta, con el mensaje y el código correspondiente al error

Dtos dentro de esta carpeta están los objetos dtos para recibir en la web api y luego mapearlos a los objetos del dominio.

Parser dentro de esta carpeta estan las ddl de los importadores de terceros; json y xml, y dos archivos de ejemplo para importar contenido desde json y dede xml



1.7. Descripción del mecanismo de acceso a datos utilizado

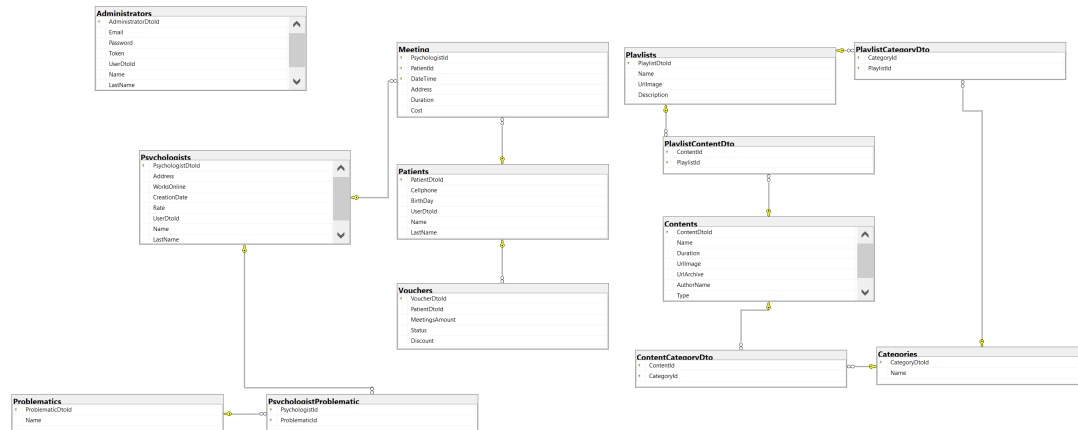
La persistencia de datos se realizo utilizando NET core, y como mencionamos anteriormente, para poder trabajar con la base de datos creamos objetos Dto, los cuales poseen la información que se deberá guardar en las tablas de la base de datos, tenemos una clase Dto por cada entidad de dominio, mas tres de la relación n a n entre playlist, category y song.

Estos objetos se mapean,, con los métodos definidos en las clases mappers, de Dto(Base de datos) a dominio y viceversa.

1.7.1. Modelado de la base de datos

Dentro de la base de datos se encuentran distintas tablas, correspondientes a los objetos del sistema, en el siguiente diagrama se muestra estas tablas y que columnas contienen, además se muestra la PK y FK de cada una.

Como se ve en el diagrama además de tablas para los objetos del dominio; CategoryDto, ProblematicDto, SongDto, PlaylistDto, PsychologistDto, PatientDto y AdministratorDto. También se crean tablas para la relación N a N entre playlist y song, playlist y category, song y category; PlaylistCategoryDto, PlaylistSongDto y SongCategoryDto.



1.8. Descripción del manejo de excepciones

Para el manejo de errores creamos nuestras excepciones las cuales fueron descriptas anteriormente en cada paquete.

Estas excepciones son lanzadas al encontrar un error, ya sea al setear una propiedad de un objeto o usar alguna de las funciones del DataBaseRepository, y controladas en la lógica (services) con el uso de try catch.

Luego estas excepciones son lanzadas a la API para ser controladas con la clase filterEceptions donde se capturan estos errores y se muestran, usando el objeto ErrorDto, el código de estado, si paso el error o no y el mensaje que corresponda para informar al usuario el error.

En el siguiente diagrama de secuencia se muestra el flujo de manejo de excepciones con un caso particular al agregar un audio con nombre de largo incorrecto o un audio repetido.

Ver diagrama de secuencia en [Anexo 3]

1.9. Diagrama de secuencia Import

En el siguiente diagrama se muestra la funcionalidad Import, desde el frontend como se recibe la request en el backend y como funciona internamente esta funcionalidad. Se omite el proceso que hacen SetContent en contentService y SetPlaylist en playlistService para simplificar el diagrama que ya es bastante extenso, ya que estas funcionalidades ya son conocidas desde la entrega anterior.

Ver diagrama de secuencia [Anexo 4]

1.10. Diagrama de secuencia LogIn

En el siguiente diagrama se muestra la funcionalidad LogIn, el acceso a datos con la utilización de los Mappers, y el manejo de error(no se muestra como respuesta ya que astah no nos permitía hacer esto, si no que queda representada como una llamada)

Ver diagrama de secuencia en [Anexo 5]

1.11. Diagrama de secuencia bonificación paciente

En el siguiente diagrama se muestra la funcionalidad voucher, específicamente desde el frontend la request put al backend y como funciona este internamente en el actualizar voucher.

Ver diagrama de secuencia en [Anexo 6]

1.12. Justificación del diseño en base a principios y patrones

El paquete BusinessLogic no depende del paquete dataAccess sino que DataAccess depende de una abstracción de BusinessLogic, lo que cumple con el **principio de inversión de dependencias**.

Las interfaces creadas en el sistema IImport, IRepository e IMapper se pensaron cumpliendo con el **principio de Segregación de la Interfaz**, es decir que las clases que implementan estas no estén obligados a implementar métodos que no usan, para esto se intento que estas clases fueran muy específicas en cuanto a que métodos poner como parte del contrato.

Utilizamos el **patrón inyección de dependencia** para desacoplar nuestra capa de negocio, BusinessLogic, de nuestro acceso a datos, DataAccess, mediante la utilización del IRepository, y de los repositories con los managers repositories.

Con la separación de la API de la lógica mediante los services logramos que el repositorio se encargue únicamente del almacenamiento de datos, lo que cumple con el **principio Single responsibility**, además cumple con el patrón controlador ya que resuelve el problema de manejar los eventos de entrada y salida del sistema.

En lo que respecta a la WebApi la misma mantiene una dependencia hacia BusinessLogicInterface, específicamente sobre los IServices, de manera que se acopla a una interfaz y no a una implementación. Además conoce la implementación de todos ya que es quien resuelve las dependencias utilizando **inyección de dependencias**.

Se aplica el **principio Open/Closed** el cual menciona que las clases deberían ser

abiertas a la extensión, pero cerradas a la modificación, sobre la clase IRepository generic, ya que se puede extender fácilmente a otra forma de almacenamiento que no sea en base de datos y se pueden almacenar nuevos tipos y la clase depende de una abstracción se puede extender sin modificar el código.

También se aplica el **principio OCP** sobre la nueva funcionalidad, específicamente sobre la interfaz IImport que permite fácilmente agregar nuevos tipos de importadores, pero cambiar el contrato de esta llevaría a un cambio muy grande en varios puntos del sistema, por lo que podemos decir que es cerrada a los cambios.

Se aplica el **principio LSP (Liskov Substitution Principle)** en el uso de los mappers en la clase DataBaseRepository. Este principio menciona que métodos que usan referencias a objetos de clases base, deben poder usar objetos de clases derivadas sin saberlo. Los que llaman a los métodos de DataBaseRepository no tienen que saber a que mapper se llama.

También se aplica este **principio LSP** en el ImportService cuando se decide al momento de importar a que importador llamar.

Se aplica durante el desarrollo de todo el código el **Single Responsibility Principle**, donde una clase debería tener una única razón por la cual cambiar.

Podemos ver el **patrón experto en los services**, ya que estos delegan la responsabilidad de agregar, borrar, actualizar o encontrar al correspondiente managerRepository.

Se aplica durante el desarrollo de todo el sistema el **Patrón alta cohesión**, se asignan responsabilidades de manera que la cohesión se mantenga alta, asignar responsabilidades que estén altamente relacionadas a una misma clase. Podemos ver esto claramente en los services, cada uno se encarga de las operaciones(set, get, delete, update) sobre un tipo de objeto. Lo mismo con los controllers de la API, creamos uno por objeto para mantener la alta cohesión.

Se puede ver en la implementación de nuestra API el uso del **patrón fachada**, lo que hace que el cliente que va a usarla se desacople de la implementación.

Los test fueron separados en carpetas diferenciados para cumplir **Single Responsibility Principle**.

Para el manejo de excepciones creamos la clase FilterExceptions, para juntar en una única clase la responsabilidad del manejo de errores para cumplir **Single Responsibility Principle**.

Utilizamos también **inyección de dependencias** sobre el filtro de autenticación, con el IAdministratorService.

Para la nueva funcionalidad donde se desea poder importar desde cualquier tipo de archivo nuevo contenido y playlist, se utilizó **reflection**, este nuevo concepto

consiste en crear una interfaz base, en este caso es la interfaz IImport para que terceros puedan implementarla.

El código utilizado para la importación es extensible y esta preparado para recibir cualquier tipo de importación y saber que dll utilizar para lograrla. Para esto, el IImport es una interface la cual no se le asigna ninguna implementación sino que esta implementación se decide dependiendo del tipo de importación se está utilizando en el momento.

Los services implementados en la UI siguen el **patrón singleton** ya que usan una única instancia de la misma clase.

Con el uso de la clase InjectionManager asignamos la responsabilidad de las dependencias a una sola clase cumpliendo así **SRP**.

Para la división de los paquetes cumplimos con el principio **ADP (Acyclic Dependencies Principle)** para evitar así la dependencias circulares que causan que un cambio afecte a todos los paquetes relacionados en este ciclo. Esto se aplico entre la relación de dataAccess, con BusinessLogic y BusinessLogicInterface aplicando el **principio de inversión de dependencias** ya que dataAccess depende de una abstracción de BuisnessLogic y el principio de segregación de la interface.

Al momento de generar las dependencias se hicieron cambios para cumplir con el **patron SDP** (mas adelante se da evidencia de esto y se muestra en que paquetes se cumple) y que la **dependencia entre los paquetes vaya en dirección de la estabilidad**.

Se creo una abstracción del paquete BusinessLogic creando el paquete BusinessLogicInterface, donde se guardan todas las interfaces de este, ya que es uno de los paquetes de los que mas se dependía y por lo tanto para cumplir con **SAP (principio de abstracciones estables)** también debería ser el mas abstracto. Esto no se cumplió con el paquete Domain ya que muchos dependen de este pero en cambio, es un paquete totalmente concreto, esto se considera igual correcto porque domain es un paquete que no debería sufrir fuertes cambios, ya que contienen las entidades del dominio.

En cuanto a la cohesión de los paquetes se siguió el principio **CCP (principio de clausura común)** agrupando las clases que son candidatas a cambiar por el mismo tipo de cambio.

Se agrego a la clase contenido un atributo tipo para poder soportar nuevo contenido (audio o vídeo).

Usamos el **patrón proxy** al momento de restringir el uso de ciertas funcionalidades a los administradores, limitando el acceso con una clase filterAuthentication que actúa como Proxy.

Los mappers usados en nuestro sistema para pasar objetos del dominio a base de datos o viceversa funcionan como **adapters**, ya que convierten el objeto en el tipo que se necesita. Esto se da al momento de hacer operaciones sobre el IRepository genérico si necesito pasar de un objeto dominio a un objeto DTO, uso los mappers adaptando al tipo que mi interface necesita. También cumplen con el **patrón adapter** los services implementados en el backend

1.12.1. Diseño general frontend

Usamos el **patrón observer** cuando vinculamos la lógica con la web, para “suscribirnos” al resultado de una llamada HTTP que es asíncrona y de la cual nos interesa saber su resultado.

En cuanto al diseño de la parte de UI para que esta cumpliera con los **principios de diseño web** usamos como base la librería angular material. Algunos de los principios que cumplimos fueron:

- Se enfatiza en la funcionalidad y proporciona puntos de interés para el usuario.
- Los cambios en la interfaz derivan a partir de las acciones del usuario.
- El diseño es atrevido, jerárquico, da significado y enfoca.
- Se usan símbolos familiares para mostrar determinadas funcionalidades y facilitar el entendimiento.

Dentro del proyecto frontend tenemos separados en directorios para facilitar la búsqueda y el entendimiento :

Models dentro de models se encuentran los objetos.ts correspondientes a cada entidad del dominio en el backend, con los mismos atributos, para manejar estas entidades en el frontend.

Services dentro de services tenemos por cada entidad los services que se encargan de mandar las request del frontend al backend mediante un http client. Sobre estos services se hace **inyección de dependencias** en todos los components para luego usarlos en los métodos de estos. Sobre estos services se aplicó también el **principio de responsabilidad única**

Además tenemos por cada entidad un directorio con los **components** correspondientes a cada operación de esta, y un component para el detalle de esa entidad.

También tenemos dentro un **module** por entidad dentro de este se indica el path de cada component para hacer el ruteo.

Tenemos un **componente base** con un menú y una barra superior, este se encarga de redireccionar a la ruta correspondiente según donde el usuario haga click

1.13. Justificación del diseño en base a métricas

Para el análisis de las métricas se hacen cálculos manuales que son aproximados a partir de la siguiente matriz y cálculos con la herramienta ndepend

| Package | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------------------------------------|---|---|---|---|---|---|---|---|
| ImporterXml | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ImporterJson | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MSP.BetterCalm.WebAPI | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MSP.BetterCalm.DataAccess | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MSP.BetterCalm.BusinessLogic | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MSP.BetterCalm.BusinessLogicInterface | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MSP.BetterCalm.Importer | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MSP.BetterCalm.Domain | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

1.14. Métrica de Estabilidad

En primer lugar analizamos la métrica de estabilidad para determinar qué tan complicando es cambiar un paquete, mediante la cantidad de paquetes que dependen de él y la cantidad de paquetes de los cuales él depende.

$$I = Ce / (Ce + Ca)$$

Tendremos en cuenta que:

Valor cercano a 0 : máxima estabilidad porque no se depende de otros paquetes.

Valor cercano a 1 : máxima inestabilidad porque depende de otros paquetes.

- $I_{Domain} = 0/0+6 = 0$ estable
- $I_{DataAccess} = 3/3+1 = 0,7$ inestable
- $I_{BusinessLogic} = 2/2+1 = 0,7$ inestable
- $I_{BusinessLogicInterface} = 1/1+3 = 0,25$ estable
- $I_{Importer} = 1/1+3 = 0,25$ estable
- $I_{ImportJson} = 1/1+0 = 1$ inestable
- $I_{ImportXml} = 2/2+0 = 1$ inestable
- $I_{WebAPI} = 4/4+0 = 1$ inestable

Las estabilidad la relacionamos con el **principio de dependencias estables**. Entonces podemos decir que nuestro diseño cumple con SDP, muchos paquetes inestables dependen de uno estable.

Domain que es un paquete estable no depende de nadie, sino que los paquetes ImporterXml, Importer, WebAPI, DataAccess inestables dependen de el que es mas estable.

BusinessLogicInterface es un paquete estable depende solo de domain que es mas estable que el.

WebAPI, DataAccess paquetes inestables dependen de BusinessLogicInterface que es mas estable que ellos.

BusinessLogic es un paquete inestable depende de domain y de BusinessLogicInterface que son mas estable que el.

DataAccess paquete inestable depende de BusinessLogic que es mas estable que

ellos.

Importer también es un paquete estable depende solo de domain que es mas estable que el.

ImporterJson, **ImporterXml**, **DataAccess**, **WebAPI** paquetes inestables dependen de Importer que es mas estable que ellos.

Por lo tanto nuestro diseño de paquetes es mantenible, fácil de cambiar.

1.15. Métrica de la abstracción

En segundo lugar analizamos la métrica de abstracciones para ver que tan abstracto es un paquete respecto a la medida de interfaces o clases abstractas que posee.

$$A = N_a / N_c$$

Tendremos en cuenta que:

Valor cercano a 0 : indica que estamos frente a un paquete muy concreto.

Valor cercano a 1 : indica que estamos frente a un paquete muy abstracto.

- A Domain = 0 = concreto
- A DataAccess = $1/36 = 0,03$ concreto
- A BusinessLogic = 0 = concreto
- A BusinessLogicInterface = 1 = abstracto
- A Importer = $1/7 = 0,14$ concreto
- A ImportJson = 0 = concreto
- A ImportXml = 0 = concreto
- A WebAPI = 0 = concreto

Como vemos entonces se cumple con el **principio de abstracciones estables**: los paquetes más estables deberían ser los paquetes más abstractos o los que contienen más abstracciones en todos los paquete si comparamos con los resultados obtenidos en la primer métrica vemos que los paquetes inestables son todos concretos, y el estable BusinessLogicInterface es abstracto.

A excepción del paquete **domain** que es un paquete que no cambia ya que tiene los objetos del Domain e Importer que no cambia porque tiene el contrato de importación.

1.16. Métrica de distancia

Analizamos ahora la métrica de distancia que nos indica qué tan lejos está un paquete de la secuencia principal

$$D' = A + I - 1$$

Tendremos en cuenta que:

Valor cercano a 0 : indica que el paquete es concreto y estable (responsable). Estos paquetes no son buenos porque cambian mucho y muchos dependen de dll, por ende tienen un impacto muy alto.

Valor cercano a 1 : indica que los paquetes son muy abstractos y muy inestables. Estos paquetes son extensibles pero tienen pocos paquetes que dependan de él, por ende es “inútil”.

- D' Domain = 1
- D' DataAccess = 0,03
- D' BusinessLogic = 0,3
- D' BusinessLogicInterface = 0,25
- D' Importer = 0,41
- D' ImportJson = 0
- D' ImportXml = 0
- D' WebAPI = 0

ImportXml, ImportJson, WebApi, tienen una distancia de métrica de 0 indica que tienen una arquitectura robusta

Los paquetes **DataAccess, BusinessLogic, BusinessLogicInterface, Importer** están casi en 0. Por lo que en estos paquetes la métrica nos muestra buenos resultados.

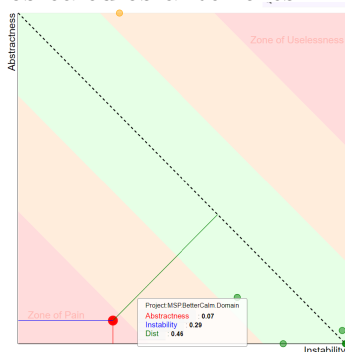
La excepción a lo anterior es el paquete **Domain** que como mencionamos anteriormente es un paquete estable y concreto, por lo que no debería cambiar.

Corroboración del análisis anterior:

Resultados obtenidos con NDEPEND:

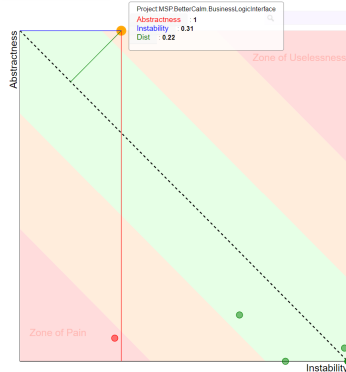
Los resultados obtenidos con esta herramienta se mantienen, en la mayoría de los casos con las mismas relaciones que los calculados a pesar de haber algunas diferencias en los valores.

Estas diferencias se deben a que la herramienta ndepend cuenta las dependencias a terceros es por esto que para el análisis nos basamos en los cálculos manuales y luego verificamos con los resultados de la herramienta. Para ver el análisis en detalle ir a los cálculos anteriores.

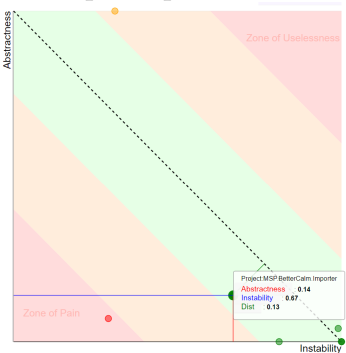


En este paquete Domain que los números son un tanto distintos a los calculados pero preferimos basarnos en los cálculos manuales para el análisis, de todas formas

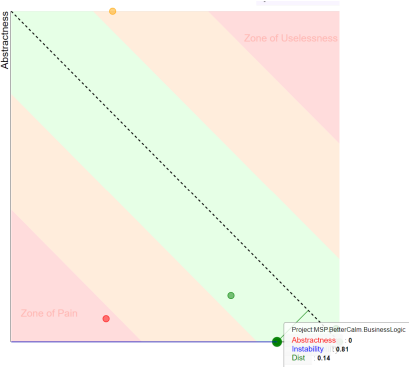
el numero que se encuentra mas lejos del calculado es la inestabilidad y esto se debe a como ya dijimos a las dependencias terceras que no fueron tomadas en cuenta en los cálculos manuales.



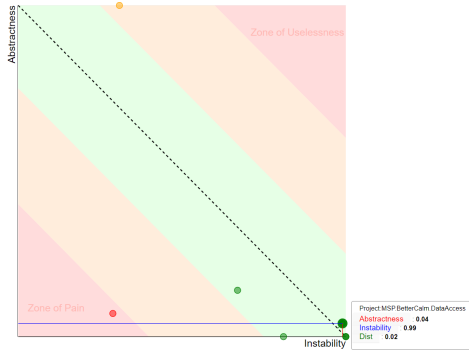
Los resultados de BusinessLogicInterface dieron con la misma relación y casi iguales a los calculados manualmente por esto se mantiene el análisis hecho anteriormente donde decimos que este paquete cumple con el principio de dependencias estables y con el principio de abstracciones estables.



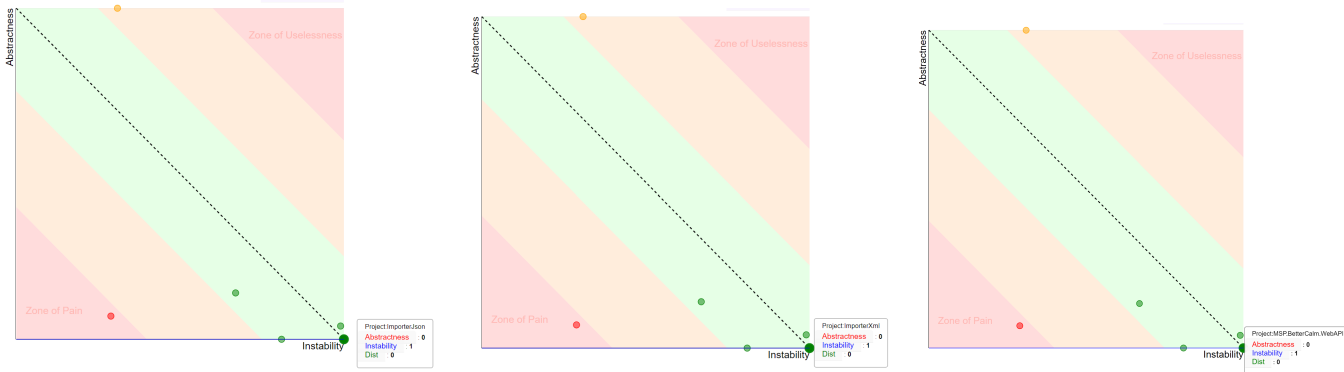
En este paquete Importer vemos que los números son un tanto distintos a los calculados pero preferimos basarnos en los cálculos manuales para el análisis, de todas formas el numero que se encuentra mas lejos del calculado es la inestabilidad y esto se debe a como ya dijimos a las dependencias terceras que no fueron tomadas en cuenta en los cálculos manuales.



Los resultados de BusinessLogic dieron con la misma relación y casi iguales a los calculados manualmente por esto se mantiene el análisis hecho anteriormente donde decimos que este paquete cumple con el principio de dependencias estables y con el principio de abstracciones estables.



En este paquete DataAccess vemos que los números son un tanto distintos a los calculados pero preferimos basarnos en los cálculos manuales para el análisis, de todas formas el número que se encuentra mas lejos del calculado es la inestabilidad y esto se debe a como ya dijimos a las dependencias terceras que no fueron tomadas en cuenta en los cálculos manuales.



En estos últimos tres paquetes ImporterJson, ImporterXml y WebAPI mantienen los números exactamente iguales a los calculados anteriormente, por lo cual se mantienen el análisis anterior donde mencionamos que estos paquete cumplen con el principio de dependencias estables y que también cumplen con el principio de abstracciones estables.

1.17. mecanismos utilizados para permitir la extensibilidad

Para permitir que terceros puedan implementar la importación desde cualquier tipo de archivo se va disponer de documentación para explicar esta. Nuestro sistema es capaz de leer cualquier tipo de archivo desde una ddl y convertirlo en contenido (audio o video) para guardarlo, se asume para esto que los archivos vienen con un formato correcto, y contienen al menos la lista vacía de contenidos. Para implementar esto utilizamos el concepto de reflection. En caso de haber un error con un contenido este en específico no se importara.

Para poder implementar esta extensibilidad disponemos de ciertas clases que es necesario conocer para entender como se comporta esta funcionalidad.

En primer lugar vamos a hablar de la interface **IImporter**: esta clase define el contra-

to para todos los importadores, es decir obliga a que cualquiera que quiera importar contenido mediante esta deba implementar ciertos métodos para poder hacerlo, estos son;

GetImporterName devuelve el nombre del tipo de importador para el cual se va a implementar el parseo (por ejemplo Json, Xml).

GetParameters devuelve la lista de parámetros que necesita ese importador, estos parámetros están compuestos por un nombre y un tipo, el cual esta restringido a ser string, int, date o bool (por ejemplo un posible parámetro es path de tipo string)

ImportContent se encarga, como dice su nombre, de importar el contenido para esto recibe el path del archivo a parsear y devuelve luego ese contenido como un objeto de tipo **ListContentModel** que es una lista del modelo **ContentModel**, el cual luego va ser utilizado por el sistema para guardar el nuevo contenido .

Siguiendo las implementaciones de las clases anteriores como se menciona durante esta documentación, el sistema sera capaz de leer la ddl y convertir el archivo que se encuentra en el path indicado en contenido para nuestro sistema.

Esto se hace en la clase **ImportService** la cual es la encargada, como mencionamos anteriormente, de implementar la interface **IImportService** que contiene los métodos que se llaman como también mostramos anteriormente desde el **ImportController** estos son;

GetImporterNames devuelve el nombre de los tipos de importadores disponibles.

GetParameters devuelve los parámetros del importador.

Por ultimo **ImportContent** que a partir de un objeto **ImportDto**, se encarga de ver a que importador instanciar y luego con el objeto **ListContentModel** devuelto por este lo convierte en objetos de dominio y los guarda en el sistema.

Mediante requests al endpoint api/import los usuario podrán así importar nuevo contenido llamando a la clase **IImportService**.

Todas las clases que se mencionan en esta sección están explicadas anteriormente con sumo detalle en su correspondiente paquete.

Resumiendo la idea de lo que debería hacer un tercero para poder importar nuevo contenido, debería implementar la interface **IImport** siguiendo el contrato de esta tal cual se explico mas arriba, para el contenido deseado, luego generar la .ddl y agregar este a la carpeta parser que se encuentra en el paquete webAPI y así el sistema pueda soportar el nuevo tipo.

Luego debe elegir el archivo que desea importar desde la web api y ya tendrá su nuevo contenido. El archivo que contenga el nuevo contenido debe estar correctamente cargado, contener la estructura como la explicada con los atributos necesarios, en caso de que alguno de los contenidos tenga errores ese contenido no se importa y se sigue importando los demas.

Esto genera una gran extensibilidad ya que podremos importar desde cualquier tipo de archivo siempre y cuando se tenga la implementación del mismo, y en caso contrario se puede implementar para luego utilizarla

1.18. Resumen mejoras

En esta sección se hace un resumen de las mejoras y cambios que se realizaron sobre el diseño del sistema;

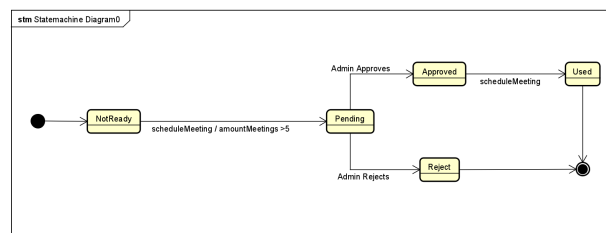
Como vimos a lo largo del documento, no se realizaron grandes cambios en el diseño, estructura del sistema ya que este había sido pensado desde un principio para adaptarse a los cambios;

El primer cambio que se realizo fue cambiar el objeto audio por contenido y se agrego a este un atributo type que puede ser de tipo audio o vídeo, soportando así nuevo contenido reproducible, lo que hace que el sistema mas interesante para los usuarios.

Se realizo este cambio ya que ambos contenidos tenían los mismos campos y para no repetir el mismo código se tomo la decisión de diseño de agregar un campo type donde a la hora de agregar un nuevo contenido el usuario debe elegir si el contenido es un audio o un video y luego completar todos los campos igual para ambos tipos.

Otro de los cambios que se realizo en base a los pedidos del cliente fue agregar una funcionalidad importar que hizo al sistema mucho mas extensible y completo, básicamente lo que se permite ahora es poder importar nuevo contenido y playlist a partir de cualquier tipo de archivo, a partir de la implementación de lo indicado en la sección de importación, actualmente se puede importar contenido desde una archivo json o desde un archivo xml.

Otro de los cambios que se realizo fue permitir guardar el costo de las consultas con el psicólogo, en el sistema a partir de ciertos atributos como lo son la duración de la consulta (en hs), tarifa por hora del psicólogo en cuestión (en pesos) y la posibilidad de aplicar bonificaciones. Esto ultimo depende de cuantas consultas ha realizado ya el paciente, si son mas de 5 se puede si el administrador lo permite aplicar un porcentaje de descuento. En caso de permitirlo en la siguiente consulta se aplico esa bonificación obtenida. El proceso de permitir es el siguiente:



Y el cambio mas grande que se realizo fue agregar el frontend utilizando angular

lo que permite al usuario interactuar con el sistema de una forma mas atractiva y fácil, este frontend manda las request al backend que ya teníamos de la primer entrega.

Además de estos cuatro grandes cambios a nivel de funcionalidad también realizamos algunos cambios a nivel de arquitectura de paquetes y de calidad de código que se fueron mostrando a lo largo de toda la documentación.

1.19. Cambios de la API

En caso de querer ver la documentación completa de la API ver [**Anexo 7: Descripción de la API**]

En esta sección se describen los nuevos endpoint y aquellos que sufrieron cambios.

Import

Representa un import, que esta compuesto por name y path, dentro de este endpoints tenemos la funcionalidad de obtener los parámetros y/o el nombre del importador, además de poder importar nuevo contenido reproducible.

- **Endpoint base:** DOMAIN/api/Import
- **Verbos HTTP:** GET - POST
- **Headers:** Authorization (solo en el caso de POST, DELETE y PUT, así validamos que sea administrador)

Login

Representa un login, que esta compuesto por name, password y token, dentro de este endpoints tenemos la funcionalidad de acceder como administrador.

- **Endpoint base:** DOMAIN/api/Login
- **Verbos HTTP:** POST

Voucher

Representa un Voucher, que esta compuesto por Patient, meetingsAmount, status y discount, dentro de este endpoints tenemos la funcionalidad de obtener las bonificaciones, filtrarlas por id y editarlas .

- **Endpoint base:** DOMAIN/api/Voucher
- **Verbos HTTP:** GET - POST
- **Headers:** Authorization (en ambos casos GET y POST así validamos que sea administrador)

Content

Representa un Content reproducible, que esta compuesto por name, duration, AuthorName, UrlImage, UrlArchive y type que puede ser audio o video.

Debemos aclarar que la duration es un string, el cual sera compuesto por un double y una "h", una "m", o una "s", de esta forma indicamos si el double esta en horas (h), minutos (m), segundos (s), ejemplo: "12h" seria un Content de 12 horas, "30s" seria un Content de 30 segundos, y "5m" seria un Content de 5 minutos, si se desea también se puede enviar por "30.4m" lo cual seria 30 minutos con 40 % de un minuto, es decir 24 minutos, esto mismo aplica con horas, en el caso de segundos se espera que sea un entero.

Cuando se obtiene un Content (HTTP GET), siempre se convierte este numero en un double que representa las horas.

- **Endpoint base:** DOMAIN/api/Content
- **Verbos HTTP:** GET - POST - DELETE - PUT
- **Headers:** Authorization (solo en el caso de POST, DELETE y PUT, asi validamos que sea administrador)








1.20. Evidencia de TDD

En este punto mostraremos evidencia de que utilizamos TDD para el desarrollo de la aplicación.

La evidencia de TDD se muestra mediante la creación de las clases de los tests de forma simultanea a la creación de las clases que estos prueban. Esto se puede ver en el repositorio.

De estas clases luego se realiza el refactorio y en caso de no haber cumplido cien por cien con tdd se agregan luego los test que faltan para cubrir todo el codigo con pruebas.

Se muestra en la siguiente imagen el porcentaje de cobertura general de cada paquete, para ver el detalle de TDD en cada paquete ver [anexo 8: evidencia de TDD y Clean Code]

| | | |
|--|-----|----------|
| ▼  Total | 97% | 204/5845 |
| >  MSP.BetterCalm.WebAPI | 99% | 6/405 |
| >  MSP.BetterCalm.DataAccess | 98% | 19/885 |
| >  MSP.BetterCalm.Test | 97% | 119/3437 |
| >  MSP.BetterCalm.Domain | 95% | 18/396 |
| >  MSP.BetterCalm.Importer | 95% | 3/61 |
| >  MSP.BetterCalm.BusinessLogic | 94% | 39/661 |

2. Anexo

2.1. Anexo 1: Paquete BusinessLogic

2.1.1. Services

ne **IProblematicService** y **ProblematicService**:

Se crea la clase **IProblematicService** que es la interfaz implementada por **ProblematicService** para poder hacer inyección de dependencias, estas clases se encargan de definir e implementar las funcionalidades que se necesitan desde **ProblematicController**; buscar las problematics, usando la instancia **ManagerProblematicRepository**, para poder acceder a las problematics ingresadas en el sistema. Esta clase es la intermediaria entre la API(**ProblematicController**) y el acceso a los datos (**ManagerProblematicRepository**).

IPlaylistService y **PlaylistService**

Se crea la clase **IPlaylistService** que es la interfaz implementada por **PlaylistService** para poder hacer inyección de dependencias, estas clases se encargan de definir e implementar las funcionalidades que se necesitan desde **PlaylistController**; crear, agregar un content, actualizar, eliminar y buscar las playlists, usando la instancia **ManagerPlaylistRepository**, para poder acceder a las playlists ingresadas en el sistema. Esta clase es la intermediaria entre la API(**PlaylistController**) y el acceso a los datos (**ManagerPlaylistRepository**).

ICategoryService y **CategoryService**

Se crea la clase **ICategoryService** que es la interfaz implementada por **CategoryService** para poder hacer inyección de dependencias, estas clases se encargan de definir e implementar las funcionalidades que se necesitan desde **CategoryController**; buscar las categories, usando la instancia **ManagerCategoryRepository**, para poder acceder a las categories ingresadas en el sistema. Esta clase es la intermediaria entre la API(**CategoryController**) y el acceso a los datos (**ManagerCategoryRepository**).

IPsychologistService y **PsychologistService**

Se crea la clase **IPsychologistService** que es la interfaz implementada por **Psycholo-**

gistService para poder hacer inyección de dependencias, estas clases se encargan de definir e implementar las funcionalidades que se necesitan desde PsychologistController; agregar, actualizar, eliminar y buscar los psychologists, usando la instancia ManagerPsychologistRepository, para poder acceder a los psychologists ingresados en el sistema. Esta clase es la intermediaria entre la API(PsychologistController) y el acceso a los datos (ManagerPsychologistRepository).

IPatientService y PatientService

Se crea la clase IPatientService que es la interfaz implementada por PatientService para poder hacer inyección de dependencias, estas clases se encargan de definir e implementar las funcionalidades que se necesitan desde PatientController; agregar y buscar los patients, y agendar una nueva consulta, usando la instancia ManagerPatientRepository, para poder acceder a los patients ingresados en el sistema. Esta clase es la intermediaria entre la API(PatientController) y el acceso a los datos (ManagerPsychologistRepository).

IAdministratosService y AdministratorService

Se crea la clase IAdministratosService que es la interfaz implementada por AdministratorService para poder hacer inyección de dependencias, estas clases se encargan de definir e implementar las funcionalidades que se necesitan desde AdministratorController; agregar, actualizar, eliminar y buscar los Administrators, usando la instancia ManagerAdministratorRepository, para poder acceder a los Administrators ingresados en el sistema. Esta clase es la intermediaria entre la API(AdministratorController) y el acceso a los datos (ManagerAdministratorRepository).

IGuideService y GuideService

Se crea la clase IGuideService que es la interfaz implementada por GuideService para poder hacer inyección de dependencias, esta clase es un wrapper de GUID para hacer inyección de dependencia en los servicios que usan GUIDS, y así poder en los mocks crear nuestro propio GUID y comparar el valor.

2.2. Anexo 2: Objetos DTO DataAccess

AdministratorDto: Esta clase es creada para la persistencia del objeto Administrator en la base de datos, hereda de la clase UserDto y tiene la misma información que el objeto mas una propertie AdministratorDtoId.

CategoryDto: Esta clase es creada para la persistencia del objeto Category en la base de datos y tiene la misma información que el objeto Category mas una propertie CategoryDtoID. En esta clase también se indica la relación n-n con playlist y Content, esto se hace teniendo en esta clase una lista de PlaylistCategoryDto y de ContentCategoryDto

PlaylistCategoryDto: Esta clase es la encargada de crear en la base de datos una tabla intermedia para la relación n-n de playlist con category, tiene una categoryDto y su Id, y una playlistDto y su Id.

PlaylistContentDto: Esta clase es la encargada de crear en la base de datos una tabla intermedia para la relación n-n de playlist con Content, tiene una ContentDto y su Id, y una PlaylistDto y su Id.

PlaylistDto: Esta clase es creada para la persistencia del objeto Playlist en la base de datos y tiene la misma información que el objeto Playlist mas una propiedad PlaylistDtoID. En esta clase también se indica la relación n-n con category y Content, esto se hace teniendo en esta clase una lista de PlaylistCategoryDto y de PlaylistContentDto

ContentCategoryDto: Esta clase es la encargada de crear en la base de datos una tabla intermedia para la relación n-n de category con Content, tiene un ContentDto y su Id, y una CategoryDto y su Id.

ContentDto: Esta clase es creada para la persistencia del objeto Content en la base de datos y tiene la misma información que el objeto Content mas una propiedad ContentDtoID. En esta clase también se indica la relación n-n con category y con playlist, esto se hace teniendo en esta clase una lista de ContentCategoryDto y de PlaylistContentDto

MeetingDto: Esta clase es la encargada de crear en la base de datos una tabla intermedia para la relación n-n de Patient con Psychologist, tiene un PsychologistDto y su Id, y un PatientDto y su Id, además tiene la fecha de la meeting y el Address.

PatientDto: Esta clase es creada para la persistencia del objeto Patient en la base de datos, hereda de la clase UserDto y tiene la misma información que el objeto mas una propiedad PatientDtoId. En esta clase también se indica la relación n-n con Psychologist, esto se hace teniendo en esta clase una lista de MeetingDto

PlaylistContentDto: Esta clase es la encargada de crear en la base de datos una tabla intermedia para la relación n-n de Psychologist con Problematic, tiene una ContentDto y su Id, y una ProblematicDto y su Id.

ProblematicDto: Esta clase es creada para la persistencia del objeto Problematic en la base de datos, tiene la misma información que el objeto mas una propiedad ProblematicDtoId. En esta clase también se indica la relación n-n con Psychologist, esto se hace teniendo en esta clase una lista de PsychologistProblematicDto

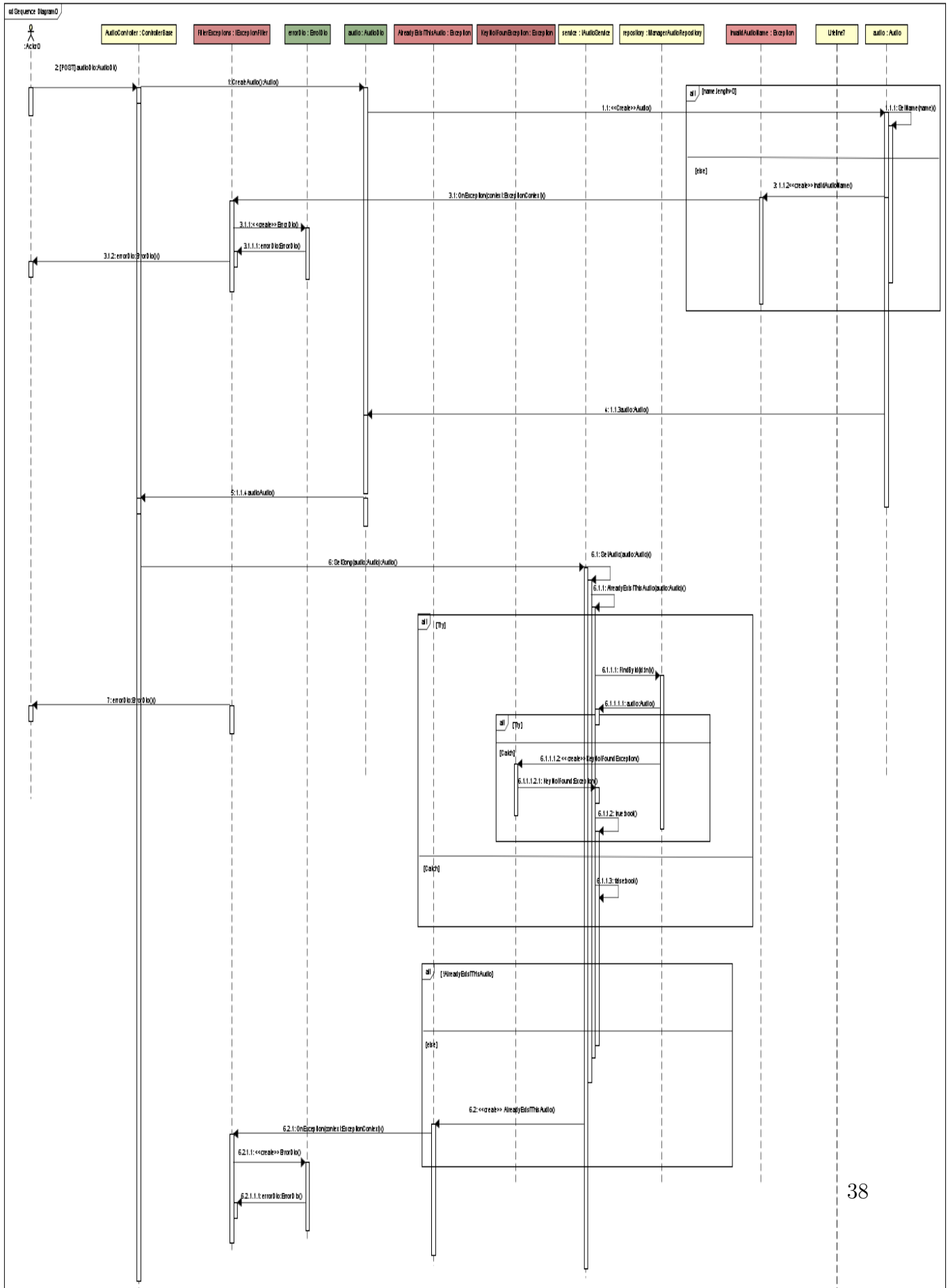
PsychologistDto: Esta clase es creada para la persistencia del objeto Psychologist en la base de datos, hereda de la clase UserDto y tiene la misma información que el objeto mas una propiedad PsychologistDtoId. En esta clase también se indica la

relación n-n con ProblematicDto y con MeetingDto, esto se hace teniendo en esta clase una lista de PsychologistProblematicDto, y de MeetingDto

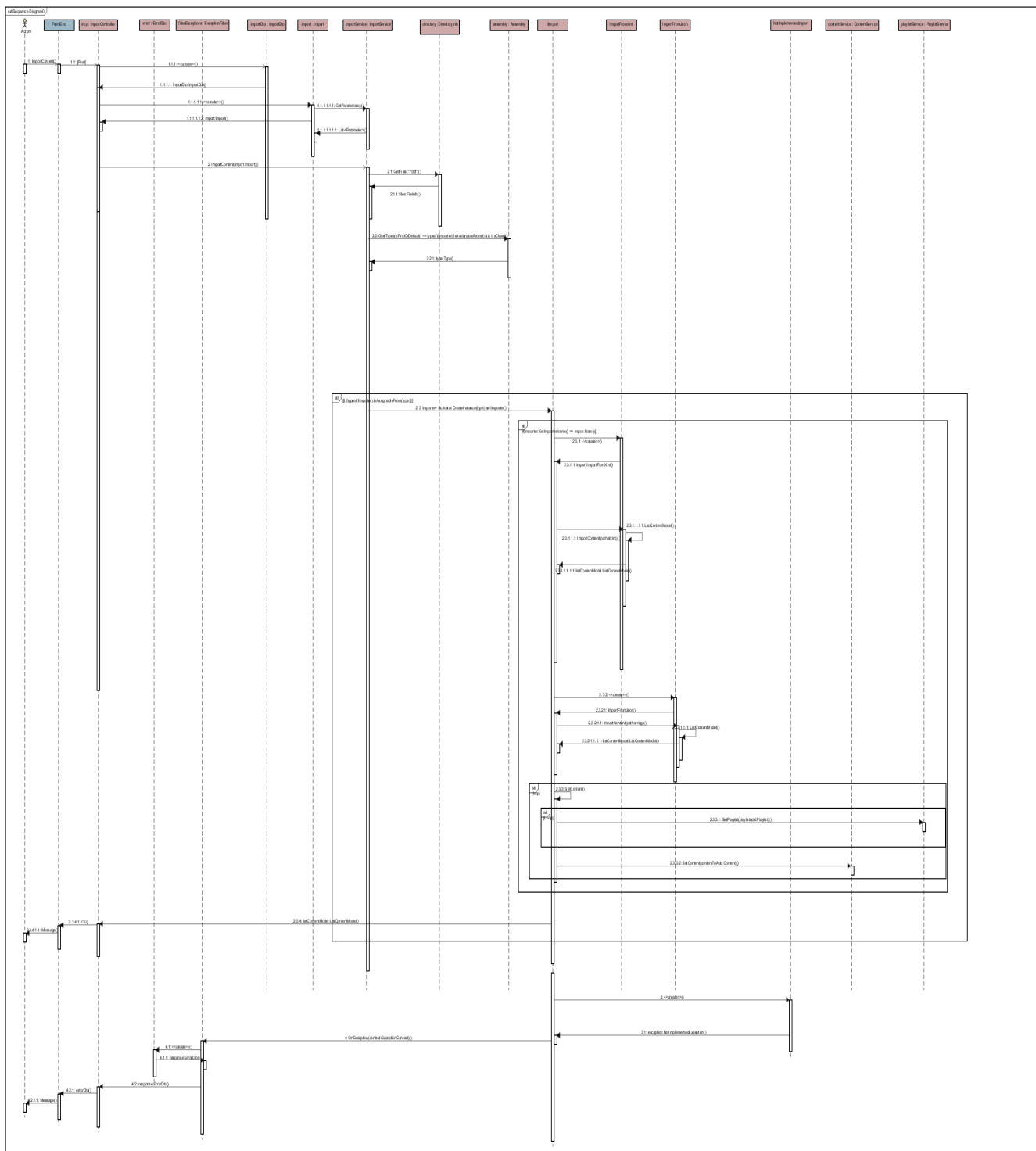
VocuherDto: Esta clase es creada para la persistencia del objeto Voucher en la base de datos, tiene la misma información que el objeto mas una propertie VoucherDtoId. En esta clase también se indica la relación 1-n con patient.

UserDto: Esta clase es para la persistencia del objeto User en la base de datos, y tiene la misma información que el objeto mas una propertie UserDtoId.

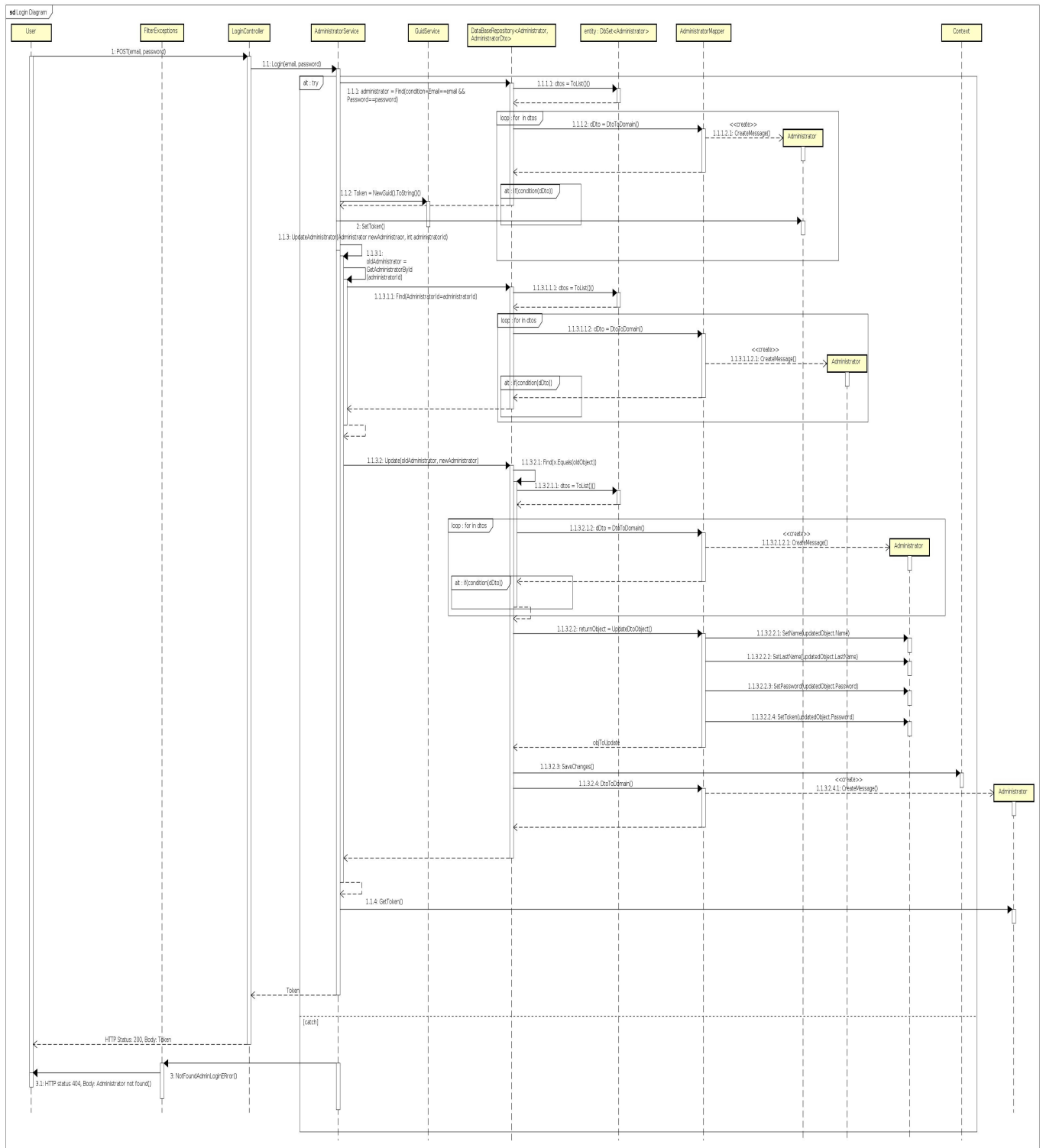
2.3. Anexo 3: diagrama de secuencia manejo de excepciones



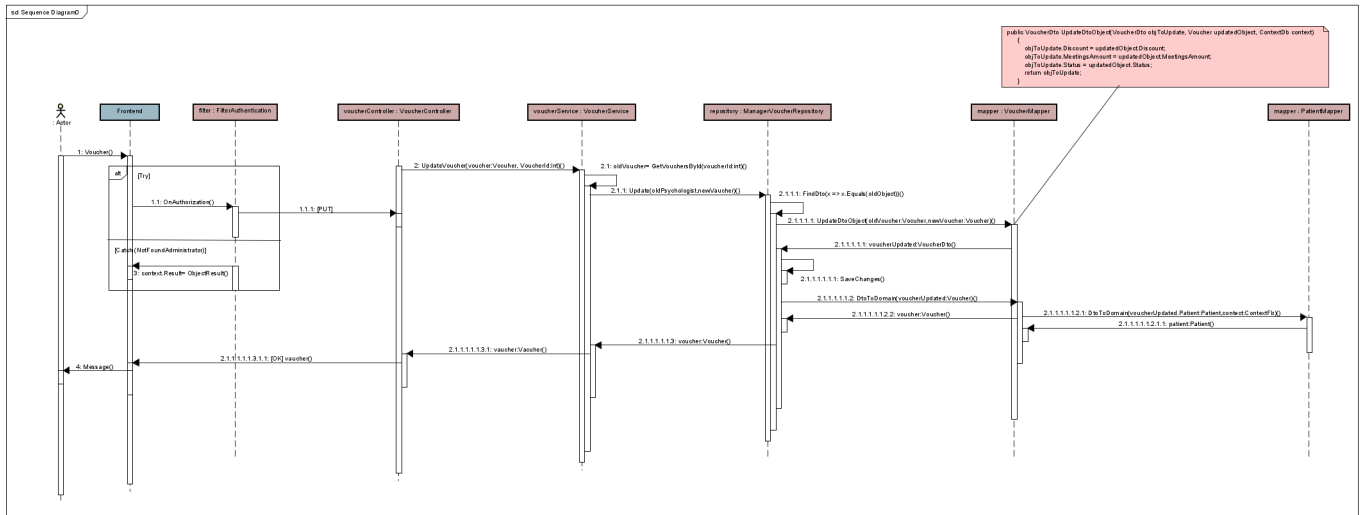
2.4. Anexo 4: diagrama de secuencia import



2.5. Anexo 5: diagrama de secuencia login



2.6. Anexo 6: diagrama de secuencia bonificación paciente



2.7. Anexo 7: Descripción de la API

2.7.1. Criterios REST

Nuestra API se basa en REST, por lo cual, hemos definido un conjunto de resources (Recursos) en los cuales se basan nuestros endpoints.

Entendemos que para los requerimientos del sistema, con estos Resources es suficiente, algunas funcionalidades como por ejemplo agendar una meeting con un psicologo, están fuertemente relacionadas a los resources ya mencionados y por esto se manejan como endpoints especiales dentro de estos, un ejemplo es el ya mencionado agendar una meeting, el cual es parte del patient (POST /patient/schedule)

En las siguientes secciones se detallan los resources que existen en el sistema y el endpoint base de cada uno, para ver un detalle de todos los endpoints ir al Anexo.

2.7.2. Description endpoints

Playlist

Representa una playlist, la cual es en si un conjunto de Contents, un conjunto de Categorías a las cuales pertenece la playlist, una descripción, un nombre y una URL a la imagen que representa la playlist.

- **Endpoint base:** DOMAIN/api/playlist
- **Verbos HTTP:** GET - POST - DELETE - PUT
- **Headers:** Authorization (solo en el caso de POST, DELETE y PUT, así validamos que sea administrador)

Category

Este recurso representa una categoria, tanto de una playlist o de un Content, es importante saber que no se puede crear una categoria, las mismas vienen predefinidas en el sistema, en el caso de los recursos de Playlist y Content que tienen un conjunto de Categories entre sus datos, al enviar una Category, esta se asocia a la Playlist o Content, pero no se crea, en caso de querer asociar una categoria que no existe en el sistema, se retornara un error. En el endpoint de category solo vamos a poder hacer la operacoin GET para obtener una categoria o un GET al /ID para obtener una especifica

- **Endpoint base:** DOMAIN/api/category
- **Verbos HTTP:** GET
- **Headers:** No Aplica

Problematic

Similar a las categorias, las problematic no se pueden crear, borrar o editar, simplemente estan predefinidas en el sistema y se asocian con otros recursos, los recursos con los cuales se pueden asociar lo veremos a continuacion. Similar a lo que sucede con category, solo tendremos los endpoints de GET

- **Endpoint base:** DOMAIN/api/problematic
- **Verbos HTTP:** GET
- **Headers:** No Aplica

Psychologist

Representa a un Psicólogo, los cuales tienen un Name, LastName, Address, WorksOnline, una lista de problematics, y una lista de meetings. Las problematics como ya se explico previamente, no pueden crearse al utilizar los endpoint de Psychologist, en las meeting pasa lo mismo, ya que para crear una meeting.

- **Endpoint base:** DOMAIN/api/psychologist
- **Verbos HTTP:** GET - POST - DELETE - PUT
- **Headers:** Authorization (solo en el caso de POST, DELETE y PUT, asi validamos que sea administrador)

Patient

Representa a un Paciente, los cuales tienen un Name, LastName, Cellphone, Birthday y una colección de meetings. El Birtday es un DateTime el cual debe enviarse utilizando el ISO 8601 https://en.wikipedia.org/wiki/ISO_8601

- **Endpoint base:** DOMAIN/api/patient

- **Verbos HTTP:** GET - POST - DELETE - PUT
- **Headers:** No Aplica

Administrator

El Administrator tiene Name, LastName, Email y Password, los mismos son los unicos que pueden hacer login en el sistema, asi mismo son los unicos que tienen acceso a algunas funcionalidades, como son el agregar Content (POST /Content), agregar Psychologist (POST /Psychologist), agregar otros Administradores y editar (PUT/PATCH) los resources de Content, Psychologist y Administrador.

- **Endpoint base:** DOMAIN/api/administrator
- **Verbos HTTP:** GET - POST - DELETE - PUT
- **Headers:** Authorization (en todos los casos, asi validamos que sea administrador)

2.7.3. Descripción de códigos de éxito y error

Pasamos a detallar como se manejan las distintas respuestas HTTP en nuestra API. Esto se hizo siguiendo las buenas practicas detalladas en libro *“API desing the missing link”*

Handling 2XX

- **200:** expresa que ”todo salio bien”, normalmente un GET satisfactorio es un ejemplo de esta respuesta
- **201:** expresa que ”el objeto se creo correctamente”, normalmente se usa durante en los metodos POST

Handling 4XX

- **401:** expresa que ”no estas autorizado”, lo cual significa que quisiste utilizar un endpoint el cual no tenes permisos basado en tu autentificacion, tambien se utiliza cuando hay un error de login.
- **404:** expresa que ”no se encontró el objeto o endpoint”, este error puede ocurrir en un GET, PUT, DELETE o PATCH
- **409:** expresa que ”no se pudo procesar debido a un conflicto”, un ejemplo de esto seria cuando se desea agregar una playlist con una categoria que no existe, al no poder crearse la categoria, se considera aun conflicto
- **422:** expresa que ”la entidad no se puede procesar”, esto son errores como un string superando el limite de caracteres, o un entero negativo donde no se permite para mencionar algunos ejemplos.

Handling 5XX

Se decidió controlar en un filtro todas las Excepciones y devolver siempre 500 y un mensaje genérico, de esta forma nos aseguramos no dejar posibles huecos de seguridad en los cuales se pueda obtener información sensible a través de una vulnerabilidad en el sistema. Idealmente esto se registraría en un log para poder ser analizado, pero escapa del alcance del proyecto

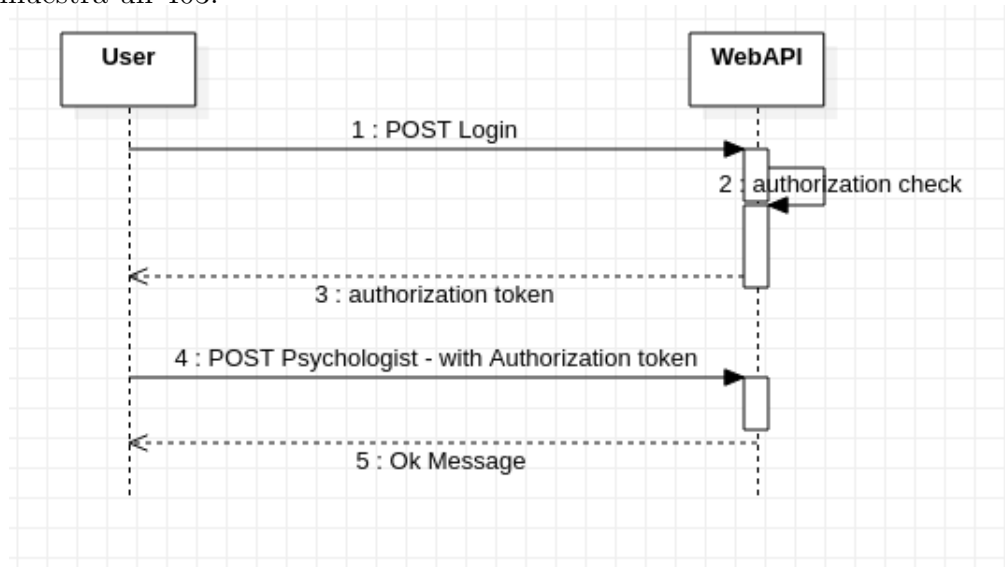
- **500:** Todo error no controlado por el sistema sera un 500.

2.7.4. Descripción proceso de Autenticación

Para el proceso de authentication se creo un endpoint específico, este es: **/api/-login** El proceso consiste en enviar mediante HTTP POST una request de login, la cual en caso exitoso respondera un 200 con un token. Dicho token debera ser enviado en la header Authorization, siendo esta la forma en la cual el sistema reconocera si el administrador esta o no logeado.

Para simplificar el uso de la aplicacion y segun lo dado en clase, el token no tendra expiracion, en un caso ideal deberia darse un token de refresh y una expiracion, de esta forma evitamos el tener un token que valido de forma infinita”.

En caso de no poder autenticarse (datos incorrectos) se devolvera un error 401, en caso de poder autenticarlo pero no tener permiso para realizar la accion deseada, se muestra un 403.



2.8. Anexo 8: evidencia de TDD y Clean Code

El código de la aplicación fue desarrollado en su totalidad usando TDD y siguiendo las buenas practicas establecidas por Clean Code.

2.8.1. Descripción de la estrategia de TDD seguida

Como se menciono anteriormente el obligatorio fue desarrollado utilizando TDD(Test-Driven Development), para esto se siguió el proceso de escribir primero los test y luego la implementación para que estos pasen, siguiendo los tres pasos o reglas:

Nunca escribir código si no hay una prueba unitaria.

Escribir el código mínimo y suficiente para que la prueba sea fallida.

Escribir sólo el código mínimo y suficiente para que la prueba pase.

Luego se realizo el refactor donde no solo se mejoro el código que se había escrito en las etapas anteriores, si no que también se arreglaron errores que se fueron encontrando en los test.

También en esta etapa se verifico que los test cumplieran con el principio FIRST:

F- FAST: Las pruebas se ejecutan rápido.

I- INDEPENDENT: La prueba no dependen del resultado de otra prueba.

R- REPEATABLE: Las pruebas son repetibles en cualquier ambiente.

S- SELF-VALIDATING: Las pruebas tienen una salida booleana. Ya sea que pasan o no.

T- TIMELY: Las pruebas son escritas antes que el código de producción que hace que pase dicha prueba.

Mas al final, cuando ya estábamos terminando el código del obligatorio, creamos ramas refactor para; agregar validaciones que habían faltado, mejorar las excepciones propias, borrar métodos que no se usaban, cambiar nombre(de métodos, variables y clases), separar métodos en partes mas pequeñas, mejorar el código de cada uno de estos para que sea lo mas entendible posible y eliminar codigo duplicado.

En esta etapa también se reviso la cobertura del código y en el caso donde se encontraron partes de código donde no se había aplicado cien por cien TDD, se agregaron los test que faltaban para mantener una alta cobertura.

2.8.2. Informe de cobertura para todas las pruebas desarrolladas

En este punto mostraremos evidencia de que utilizamos TDD para el desarrollo de la aplicación.

Evidencia de tdd en el paquete Domain

| | | |
|---|------|--------|
| ▼ <input type="checkbox"/> MSP.BetterCalm.Domain | 95% | 18/396 |
| ▼ <input checked="" type="checkbox"/> MSP.BetterCalm | 95% | 18/396 |
| ▼ <input checked="" type="checkbox"/> BusinessLogic.Exception | 100% | 0/15 |
| > <input checked="" type="checkbox"/> InvalidAmountOfPro | 100% | 0/3 |
| > <input checked="" type="checkbox"/> InvalidDurationForm | 100% | 0/3 |
| > <input checked="" type="checkbox"/> InvalidNameLength | 100% | 0/3 |
| > <input checked="" type="checkbox"/> InvalidUrl | 100% | 0/3 |
| > <input checked="" type="checkbox"/> VoucherAlreadyClosi | 100% | 0/3 |
| ▼ <input checked="" type="checkbox"/> Domain | 95% | 18/381 |
| > <input checked="" type="checkbox"/> Administrator | 100% | 0/23 |
| > <input checked="" type="checkbox"/> Category | 100% | 0/14 |
| > <input checked="" type="checkbox"/> Content | 100% | 0/77 |
| > <input checked="" type="checkbox"/> Patient | 100% | 0/27 |
| > <input checked="" type="checkbox"/> Playlist | 100% | 0/64 |
| > <input checked="" type="checkbox"/> User | 100% | 0/4 |
| > <input checked="" type="checkbox"/> Exceptions | 100% | 0/9 |
| > <input checked="" type="checkbox"/> Problematic | 94% | 1/16 |
| > <input checked="" type="checkbox"/> Psychologist | 93% | 4/60 |
| > <input checked="" type="checkbox"/> Meeting | 86% | 7/50 |
| > <input checked="" type="checkbox"/> Voucher | 84% | 6/37 |

Evidencia de tdd en el paquete WebAPI

| | | |
|---|------|-------|
| ▼ <input type="checkbox"/> MSP.BetterCalm.WebAPI | 99% | 6/405 |
| ▼ <input checked="" type="checkbox"/> MSP.BetterCalm.WebAPI | 99% | 6/405 |
| ▼ <input checked="" type="checkbox"/> Filters | 100% | 0/56 |
| > <input checked="" type="checkbox"/> FilterAuthentication | 100% | 0/22 |
| > <input checked="" type="checkbox"/> FilterExceptions | 100% | 0/34 |
| ▼ <input checked="" type="checkbox"/> Dtos | 99% | 1/109 |
| > <input checked="" type="checkbox"/> ErrorDto | 100% | 0/15 |
| > <input checked="" type="checkbox"/> ImportDto | 100% | 0/4 |
| > <input checked="" type="checkbox"/> LoginDto | 100% | 0/6 |
| > <input checked="" type="checkbox"/> PlaylistDto | 100% | 0/16 |
| > <input checked="" type="checkbox"/> ScheduleMeetingDtc | 100% | 0/6 |
| > <input checked="" type="checkbox"/> ContentDto | 98% | 1/62 |
| ▼ <input checked="" type="checkbox"/> Controllers | 98% | 5/240 |
| > <input checked="" type="checkbox"/> AdministratorContro | 100% | 0/24 |
| > <input checked="" type="checkbox"/> CategoryController | 100% | 0/16 |
| > <input checked="" type="checkbox"/> ContentController | 100% | 0/46 |
| > <input checked="" type="checkbox"/> LoginController | 100% | 0/8 |
| > <input checked="" type="checkbox"/> PatientController | 100% | 0/28 |
| > <input checked="" type="checkbox"/> PlaylistController | 100% | 0/45 |
| > <input checked="" type="checkbox"/> ProblematicControlle | 100% | 0/16 |
| > <input checked="" type="checkbox"/> PsychologistControll | 100% | 0/24 |
| > <input checked="" type="checkbox"/> VoucherController | 100% | 0/16 |

Evidencia de tdd en el paquete BuisnessLogic










| | | |
|--|------|--------|
| ▼ <input type="checkbox"/> MSP.BetterCalm.BusinessLogic | 94% | 39/661 |
| ▼ <input checked="" type="checkbox"/> MSP.BetterCalm.BusinessLog | 94% | 39/661 |
| ▼ <input checked="" type="checkbox"/> Services | 94% | 36/616 |
| > <input checked="" type="checkbox"/> ContentService | 100% | 0/96 |
| > <input checked="" type="checkbox"/> GuidService | 100% | 0/3 |
| > <input checked="" type="checkbox"/> PlaylistService | 100% | 0/145 |
| > <input checked="" type="checkbox"/> PsychologistService | 96% | 1/26 |
| > <input checked="" type="checkbox"/> VoucherService | 96% | 1/24 |
| > <input checked="" type="checkbox"/> CategoryService | 95% | 1/22 |
| > <input checked="" type="checkbox"/> ProblematicService | 95% | 1/22 |
| > <input checked="" type="checkbox"/> AdministratorService | 93% | 3/46 |
| > <input checked="" type="checkbox"/> PatientService | 92% | 8/104 |
| > <input checked="" type="checkbox"/> ImportService | 84% | 21/128 |
| > <input checked="" type="checkbox"/> Exceptions | 93% | 3/45 |

Evidencia de tdd en el paquete DataAccess

| | | |
|------------------------------|------|--------|
| ▼ MSP.BetterCalm.DataAccess | 98% | 19/885 |
| ▼ MSP.BetterCalm.DataAccess | 98% | 19/885 |
| ▼ ContextDb | 100% | 0/43 |
| > Categories | 100% | 0/2 |
| > Problematics | 100% | 0/2 |
| > Contents | 100% | 0/2 |
| > Patients | 100% | 0/2 |
| > Psychologists | 100% | 0/2 |
| > Administrators | 100% | 0/2 |
| > Playlists | 100% | 0/2 |
| > PsychologistProblem | 100% | 0/2 |
| > Meeting | 100% | 0/2 |
| > Vouchers | 100% | 0/2 |
| ContextDb() | 100% | 0/3 |
| ContextDb(DbConte | 100% | 0/3 |
| OnModelCreating(M | 100% | 0/17 |

| | | |
|-------------------------|------|-------|
| ▼ Repositories | 100% | 0/110 |
| > AdministratorReposi | 100% | 0/4 |
| > CategoryRepository | 100% | 0/4 |
| > ContentRepository | 100% | 0/4 |
| > DataBaseRepository | 100% | 0/74 |
| > MeetingRepository | 100% | 0/4 |
| > PatientRepository | 100% | 0/4 |
| > PlaylistRepository | 100% | 0/4 |
| > ProblematicReposito | 100% | 0/4 |
| > PsychologistReposito | 100% | 0/4 |
| > VoucherRepository | 100% | 0/4 |

| | | |
|------------------------|------|-------|
| ▼ DtoObjects | 100% | 0/138 |
| > AdministratorDto | 100% | 0/8 |
| > CategoryDto | 100% | 0/8 |
| > ContentCategoryDtc | 100% | 0/8 |
| > ContentDto | 100% | 0/18 |
| > MeetingDto | 100% | 0/16 |
| > PatientDto | 100% | 0/8 |
| > PlaylistCategoryDto | 100% | 0/8 |
| > PlaylistContentDto | 100% | 0/8 |
| > PlaylistDto | 100% | 0/12 |
| > ProblematicDto | 100% | 0/6 |
| > PsychologistDto | 100% | 0/14 |
| > PsychologistProblem | 100% | 0/8 |
| > UserDto | 100% | 0/6 |
| > VoucherDto | 100% | 0/10 |

| ▼ {} Mappers | 97% | 19/594 |
|---|------|--------|
| >  AdministratorMapper | 100% | 0/18 |
| >  CategoryMapper | 100% | 0/17 |
| >  PatientMapper | 100% | 0/43 |
| >  ProblematicMapper | 100% | 0/17 |
| >  PlaylistMapper | 99% | 2/216 |
| >  ContentMapper | 98% | 3/132 |
| >  MeetingMapper | 96% | 2/49 |
| >  PsychologistMapper | 93% | 6/84 |
| >  VoucherMapper | 67% | 6/18 |

2.8.3. Evidencia de Clean Code

En este punto mostraremos evidencia de que utilizamos clean code para el desarrollo de la aplicación.


Nombres con sentido Todos los nombres tanto de variables, métodos, clases y paquetes son descriptivos de lo que hacen o para que sirven. Los métodos llevan nombres que utilizan verbos y empiezan con mayúsculas. Los que retornan un booleano llevan la palabra *Is* delante, lo que son de acceso a datos llevan la palabra *Get* delante y los que son para el registro de datos la palabra *Set*.

 10+4 usages

```

public List<Playlist> GetPlaylist()
{
    return repository.Playlists.Get();
}

```

 11+4 usages

```

public void SetPlaylist(Playlist playlist)
{
    repository.Playlists.Add(playlist);
}

```

Las clases llevan nombres que no son verbos y empiezan con mayúsculas.

Variables

Los nombres de todas las variables van en minúsculas y se declaran lo mas cerca posible de uso

```

Playlist playlistToUpdate = repository.Playlists.FindById(id);
Playlist playlist = CreateNewPlaylist(id, newPlaylist);
repository.Playlists.Update(playlistToUpdate);

```

Los nombres de las properties van en mayúsculas y son publicas

```

25 usages
public int Id { get; set; }

66 usages
public string Name { get; set; }

```

Métodos

Los métodos son reducidos no tiene mas de 30 lineas y se encargan de una sola cosa. Además estos métodos no reciben mas de 2 o a lo sumo 3 parámetros.

```

public List<Playlist> GetPlaylistByName(string playlistName)
{
    List<Playlist> playlists = new List<Playlist>();
    foreach (var playlist in repository.Playlists.Get())
    {
        if (playlist.IsSamePlaylistName(playlistName))
            playlists.Add(playlist);
    }

    if (playlists.Count == 0)
        throw new NotFoundPlaylist();
    return playlists;
}

```

Las clases mapper son excepciones a este punto ya que en los métodos de estas clases es donde se concentra toda la dificultad, es por esto que estos métodos son mas extensos, algunos superan las 30 lineas, y reciben hasta 4 parámetros.

```

private static List<PlaylistAudioDto> UpdatePlaylistAudios(PlaylistDto objToUpdate, Playlist updatedObject, ContextDB context,
    AutoMapper audioMapper)
{
    List<PlaylistAudioDto> diffListOldValuesAudio = objToUpdate.PlaylistAudiosDto.Where(x => x.PlaylistAudioId ==
        updatedObject.Audios.Contains(x => audioMapper.DtoToDomain(x.AudioDto, context))).ToList();

    List<Audio> diffListNewValuesAudio = new List<Audio>();
    if (updatedObject.Audios != null)
    {
        foreach (var audio in updatedObject.Audios)
        {
            Audio audioToAdd = audioMapper.DtoToDomain(x => context.Audios.First(x => x.AudioId == audio.Id), context);
            PlaylistAudioDto playlistAudioDto = new PlaylistAudioDto()
            {
                AudioId = audio.Id, PlaylistId = objToUpdate.PlaylistId,
                AudioDto = audioMapper.DomainToDto(audio, context), PlaylistDto = objToUpdate
            };
            bool contain = false;
            if (objToUpdate.PlaylistAudiosDto != null)
            {
                foreach (var playlistAudio in objToUpdate.PlaylistAudiosDto)
                {
                    if (playlistAudio.AudioId == playlistAudioDto.AudioId &&
                        playlistAudio.PlaylistId == playlistAudioDto.PlaylistId)
                        contain = true;
                }
            }
            if (!contain)
                diffListNewValuesAudio.Add(audioToAdd);
        }
    }

    diffListOldValuesAudio.AddRange(diffListNewValuesAudio.Select(x => new PlaylistAudioDto() {
        AudioDto = audioMapper.DomainToDto(x, context), AudioId = x.Id,
        PlaylistId = objToUpdate.PlaylistId, PlaylistDto = objToUpdate}));
}

```

Para los casos en que las funciones deberían retornar null o que ocurre un error se

lanzan excepciones, que luego se controlan con bloques try catch y en la api con en el filter.

Los bloques se encuentran indentados.

Clases

Las clases no tienen más de 150 o a lo sumo 200 líneas de código, a excepción de la clase `playlistMapper` que tiene casi 300 líneas.

Los métodos que se llaman a otros se encuentran declarados debajo o encima de estos, para así facilitar la lectura.

Entre método y método se dejan espacio.

Ley de demeter

Se cumple con la ley de demeter, para esto se crean las clases `services` para evitar las llamadas de tipo `repository.Categories.Add`, en contrario se hace la llamada `categoryService.SetCategory` y en el service se hace el `add`.

```
[HttpPut("{id}")]
2 usages
public IActionResult UpdateAudio([FromRoute] int id, [FromBody] AudioDto audioUpdated)
{
    _audioService.UpdateAudioById(id, audioUpdated.CreateAudio());
    return Ok("Audio Updated");
}
```

Para esto tenemos en los controllers las instancias a los services. Cumpliendo así con lo que indica `clean code` que solo se debe invocar funciones de; si mismo, variables locales, un argumento o una variable de instancia.

Manejo de errores

No se devuelve `null`, en su lugar se devuelve una excepción.

```
23+16 usages
public D Find(Predicate<D> condition)
{
    List<T> dtos = entity.ToList();
    foreach (var dto in dtos)
    {
        var dDto = mapper.DtoToDomain(dto, context);
        var condResult = condition(dDto);
        if (condResult)
            return dDto;
    }

    throw new KeyNotFoundException();
}
```