Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de aplicaciones 2
Obligatorio 1:
Descripción del diseño

Juan Pablo Poittevin(169766)
Joselen Cecilia(233552)

Entregado como requisito de la materia Diseño de aplicaciones 2

6 de mayo de 2021

Link al repositorio de GitHub
https://github.com/ORT-DA2/Poittevin-169766-
-Cecilia-233552-

Índice general

1. Descripción del diseño **2**
1.1. Suposiciones y decisiones de diseño . . . 2
1.2. Descripción general del diseño . . . 3
1.3. Diagrama general de paquetes . . . 4
1.4. Diagrama de componentes . . . 5
1.5. Asignación de responsabilidades . . . 6
1.5.1. BusinessLogic . . . 6
1.5.2. Domain . . . 10
1.5.3. DataAccess . . . 12
1.5.4. WebAPI . . . 15
1.5.5. Test . . . 15
1.6. Mecanismos de inyección de dependencias, fábricas, patrones y principios de diseño 16
1.7. Descripción del mecanismo de acceso a datos utilizado . . . 18
1.7.1. Modelado de la base de datos . . . 18
1.8. Descripción del manejo de excepciones . . . 18
1.9. Diagrama de secuencia LogIn . . . 20

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 no este en la base de datos, esta song "hereda"las categorías de la playlist. En caso contrario si se asocia un audio que ya existe en la base de datos, si este tiene categorías en común con la playlist ya no se encuentra mas "suelta", de lo contrario si no coincida en ninunga categoría aparece en ambos lugares suelta y en 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 propertie 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-Se puede agregar dos canciones con el mismo nombre.

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 psychologis-meeting y meeting psychologis, 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 fillterException, 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- Se valida que las URl que aparecen en el sistema como atributos, en caso de no ser vacías, tengan un formato correcto, esto se hace definiendo una regex.

22- Se permite que el Administrador al actualizar, envié un nuevo token o no envie nada (eliminándolo), vemos de esta forma que es posible "desloguear.ª un administrador, cosa que podemos desear hacer, al mismo tiempo el administrador delogueado solo necesita volver a hacer login para tener un nuevo token

1.2. Descripción general del diseño

Para la solución del obligatorio se implementa un API REST para el MSP(ministerio de salud publica) 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 resilencia.

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

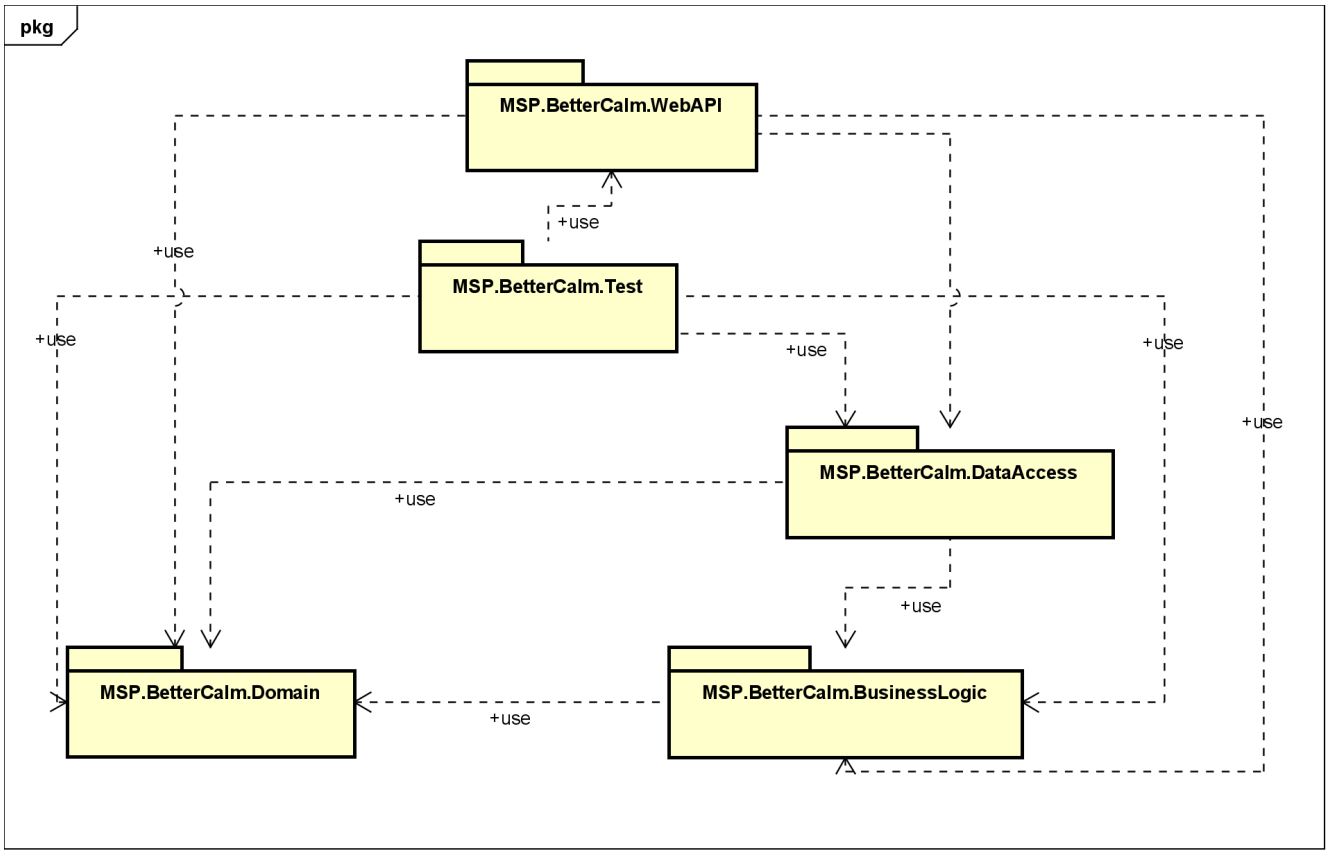
En la funcionalidad reproductor los usuarios pueden acceder a diversos tipos de contenido, todos reproducibles en formato audio, que están disponibles a través de diferentes playlists (o simplemente listas de reproducción). Dicho contenido es subido por diferentes administradores que respaldan el sistema.

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

La idea planteada para describir el diseño es que a medida que se va avanzando en el documento se va entrando en mas detalle de la solución.

1.3. Diagrama general de paquetes

El siguiente diagrama muestra una vista general de como se relacionan y cuales son los paquetes creados para la solución, como vemos WebAPI conoce a BuisnessLogic y a Domain, BuisnessLogic conoce al paquete Domain, DataAccess conoce a BuisnessLogic y por ultimo domain es conocido por todos. Como se muestra en el diagrama WebAPI también referencia al paquete DataAccess, esto se debe a que en este paquete se encuentra la clase InjectionManager que WebAPI usa. A pesar de esto la webAPI no usa ninguna de las demás clases de acceso a datos directamente si no que lo hace mediante BusinessLogic.



Como vemos entonces en 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 dos carpetas distintas una que contiene los controller y otra que contiene los filters.

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, los services y las excepciones, es el paquete que contiene la lógica.

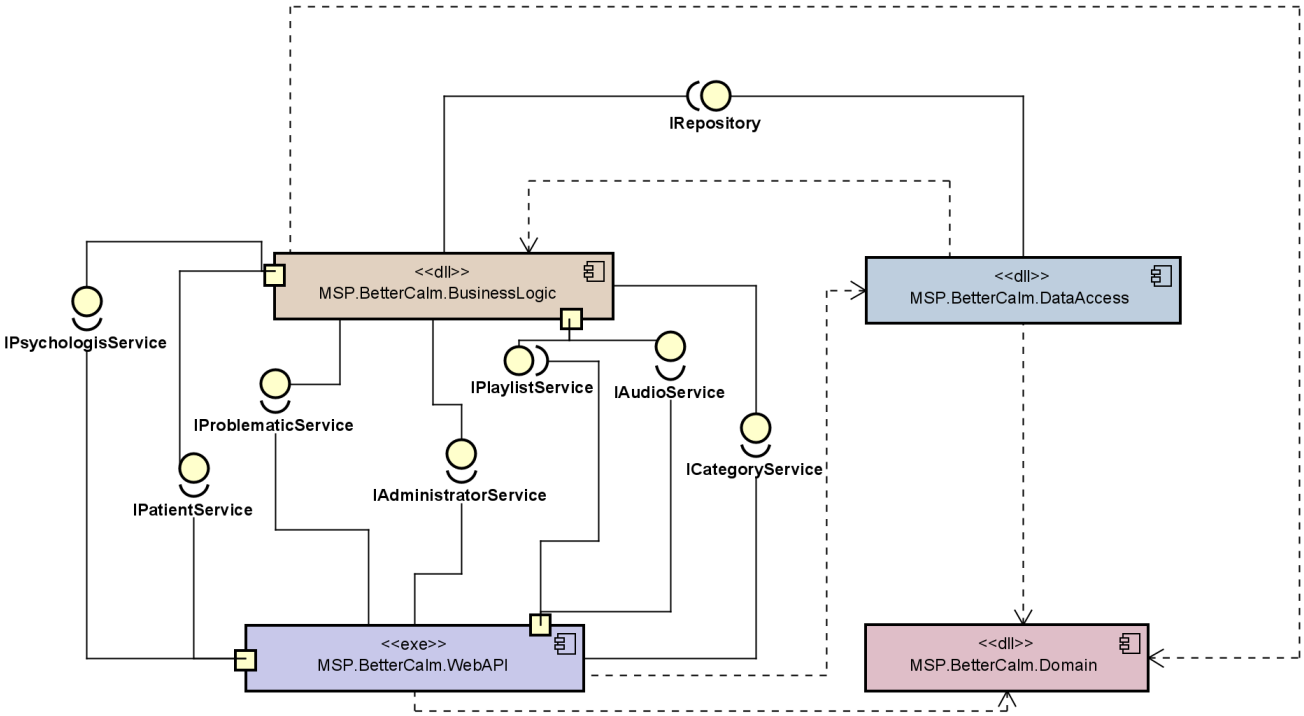
DataAccess
-Dentro de este paquete se encuentran las clases que permiten el acceso a la base de datos, los DTOs, mappers, contexo y las inyecciones de dependencia. También se encuentran en este paquete los repositories de cada entidad.

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 cases de esos paquetes.

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.

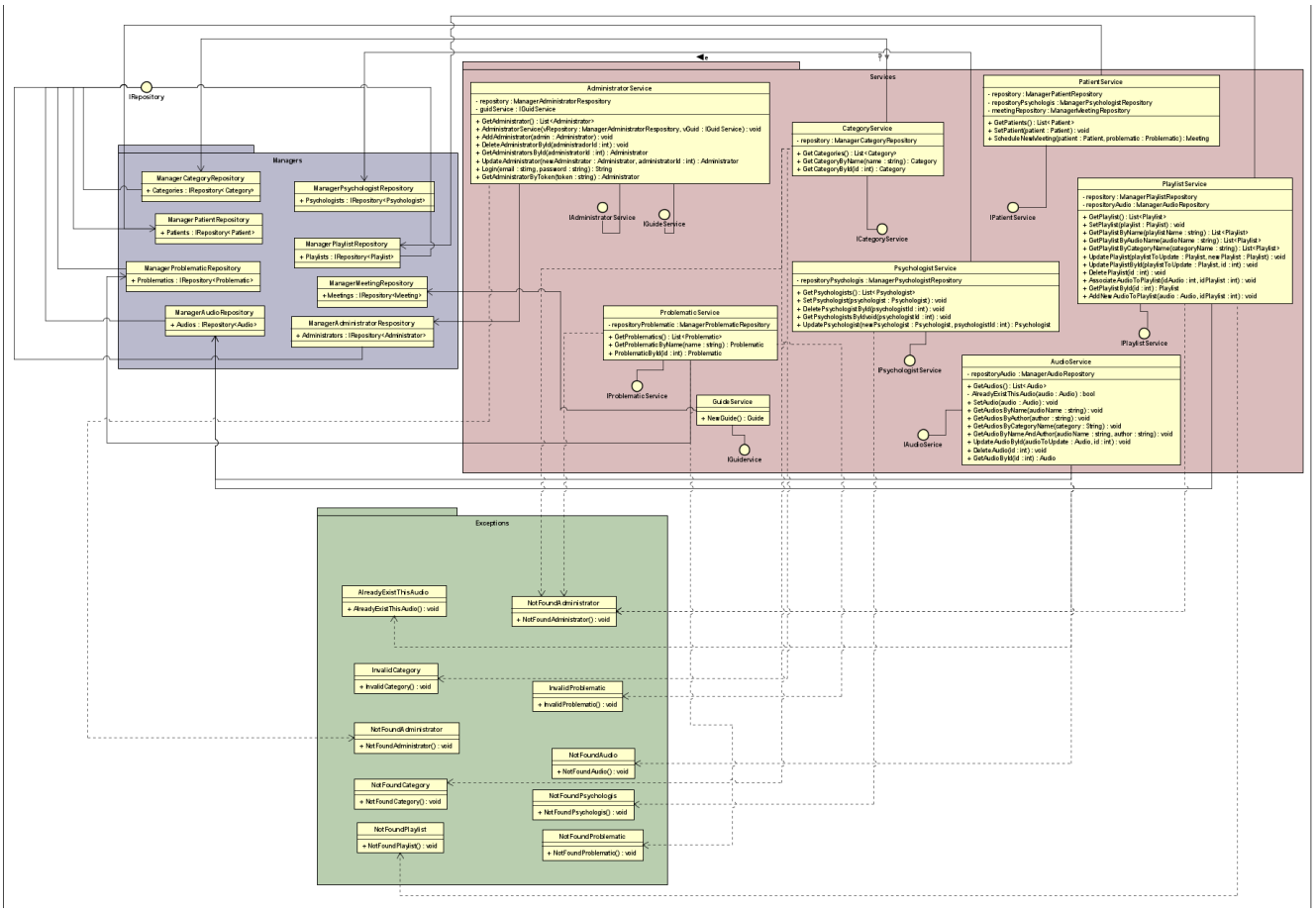
Como vemos en el diagrama WebAPi conoce todas las implementaciones de los IServices dado que es quien resuelve las dependencias, usando inyección de dependencias
Y es DataAccess quien depende de BuisnessLogic, ya que la interfaz IRepository se encuentra en Business, se aplica inversión de dependencias.



1.5. Asignación de responsabilidades

1.5.1. BusinessLogic

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



- El **namespace Services** contiene las clases e interfaces para comunicar la API con la lógica de negocio:

IAudioService y AudioService:

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

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 audio, 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 PsychologistService 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.

-El **namespace Managers** contiene la Interface IRepository generic y los managerRepository de cada entidad:

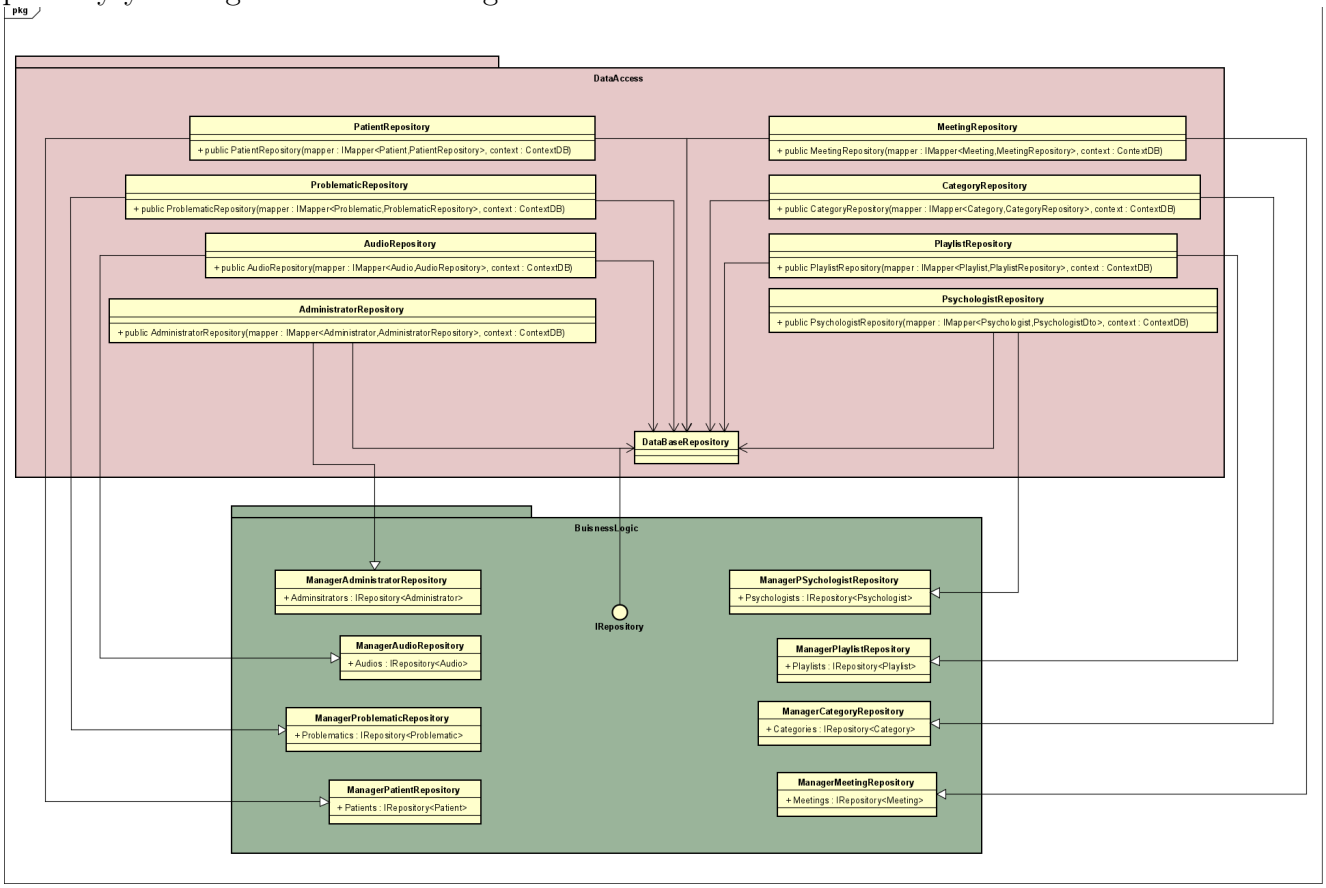
ManagerAudioRepository, ManagerPlaylistRepository, ManagerCategoryRepository,

ManagerProblematicRepository, ManagerPsychologistRepository, ManagerPatientRepository, ManagerAdministratorRepository, ManagerMeetingRepository

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

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 DataBaseRepositry la cual se encuentra en el paquete DataAcces.

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



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

NotFoundAdministrator, NotFoundAudio, NotFoundCategory, NotFoundId, NotFoundPlaylist, NotFoundProblematic, NotFoundPsychologist,

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 cu-

nado se hace la búsqueda por un Id que no se encuentra actualmente en la base de datos.

InalidCategory, InvalidProlematic

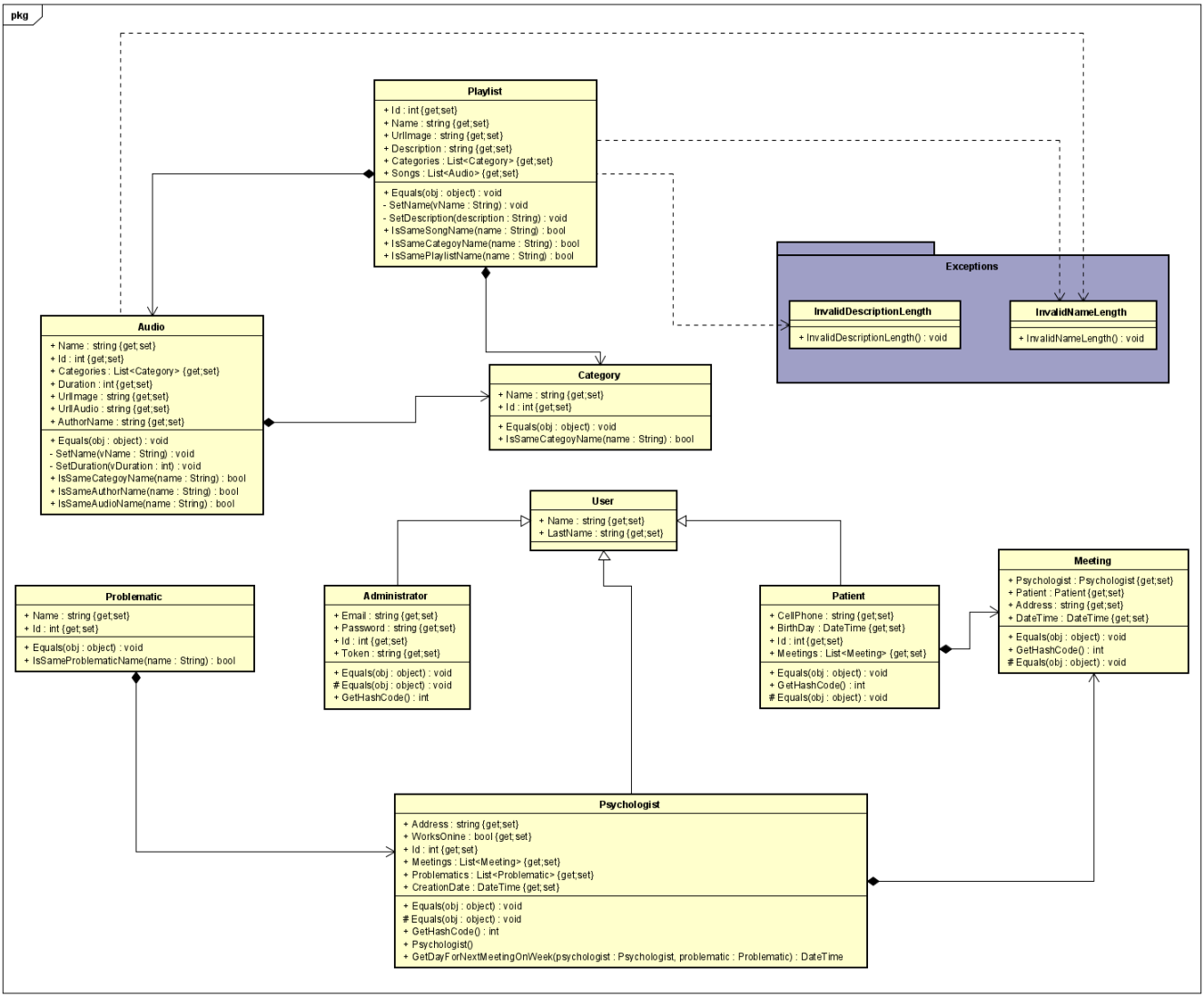
Estas clases InalidCategory y InvalidProlematic, heredan de la clase exception, y se crean para lanzar esta excepción cunado se intenta asociar una problematic o una category que no son de las fijas, que se encuentran en la base de datos.

AlreadyExistThisAudio

Esta clase InalidCategory, hereda de la clase exception, y se crea para lanzar esta excepción cuando se intenta agregar una audio con el mismo nombre y autor.

1.5.2. Domain

Dentro del namespace domain tenemos las clases correspondientes a los objetos del sistema y la carpeta Exceptions



-Clases correspondientes a los objetos del sistema:

Administrator:

Esta clase hereda de la clase User y se encarga de:
Crear los objetos de tipo Administrator
Manejar los datos del Administrator; name, lastName, id, email y password
Ver si dos Administrators son iguales

Category:
Esta clase se encarga de:
Crear los objetos de tipo Category
Manejar los datos de Category; id y name
Ver si dos categories son iguales 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; name, lastName, id, cellphone ,Birthday y meeting
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 Audios
Ver si dos Playlists son iguales
Comparar los nombres de dos Playlists
Comparar los nombres de las Audio con el nombre de una Audio 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
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
Ver si dos Psychologist son iguales
Obtener el siguiente dia libre para agendar una meeting

Audio:
Esta clase se encarga de:
Crear los objetos de tipo Audio
Manejar los datos del Audio; id, name, urlImage, duration, authorName, categories y urlAudio, associatedToPlaylist
Ver si dos Audios son iguales
Comparar los nombres de dos Audios

User:
Esta clase se encarga de:
Crear los objetos de tipo User:

Manejar los datos del Administrator; 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
Ver si dos mettings son iguales

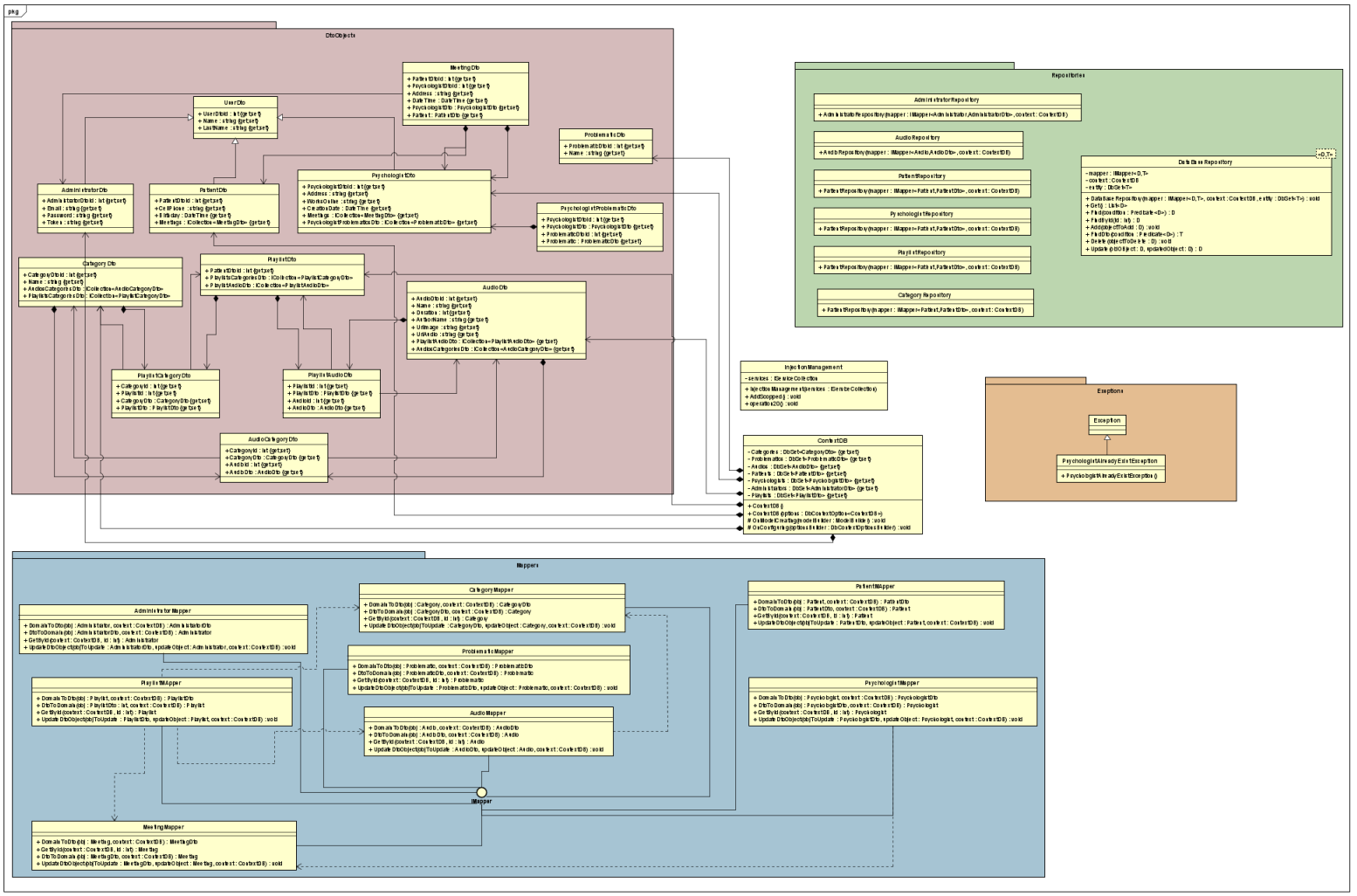
-Por ultimo tenemos dentro de este paquete el **namespace Expeptions** que contiene las clases con las excepciones creadas por nosotros.

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

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

1.5.3. DataAccess

En el paquete DataAcces creamos cuatro carpetas que agrupan las clases para facilitar la busqueda de estas, ademas se ecuentran las clases contextDB e InjectionManager



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

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 audio, esto se hace teniendo en esta clase una lista de PlaylistCategoryDto y de AudioCategoryDto

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.

PlaylistAudioDto: Esta clase es la encargada de crear en la base de datos una tabla intermedia para la relación n-n de playlist con audio, tiene una AudioDto 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 propertie PlaylistDtoID. En esta clase también se indica la relación n-n con category y audio, esto se hace teniendo en esta clase una lista de PlaylistCategoryDto y de PlaylistAudioDto

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

AudioDto: Esta clase es creada para la persistencia del objeto audio en la base de datos y tiene la misma información que el objeto audio mas una propertie AudioDtoID. 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 AudioCategoryDto y de PlaylistaudioDto

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 propertie PatientDtoId. En esta clase también se indica la relación n-n con Psychologist, esto se hace teniendo en esta clase una lista de MeetingDto

PlaylistAudioDto: 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 AudioDto 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 propertie 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 propertie 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

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.

-El **namespace Repositories** contiene los repositories de cada object y la clase DataBaseRepository.

AdministratorRepository: Esta clase hereda de la clase ManagerAdministratorRepository, y se encarga de inicializar el IRepository de administrators.

CategoryRepository: Esta clase hereda de la clase ManagerCategoryRepository, y se encarga de inicializar el IRepository de Categories.

PlaylistRepository: Esta clase hereda de la clase ManagerPlaylistRepository, y se encarga de inicializar el IRepository de Playlists.

AudioRepository: Esta clase hereda de la clase ManagerAudioRepository, y se encarga de inicializar el IRepository de audios.

ProblematicRepository: Esta clase hereda de la clase ManagerCategoryRepository, y se encarga de inicializar el IRepository de Problematics.

PatientRepository: Esta clase hereda de la clase ManagerPatientRepository, y se encarga de inicializar el IRepository de Patients.

PsychologistRepository: Esta clase hereda de la clase ManagerCategoryRepository, y se encarga de inicializar el IRepository de Psychologists.

MeetingRepository: Esta clase hereda de la clase ManagerMeetingRepository, y se encarga de inicializar el IRepository de Meetings.

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 mappeados con los métodos mappers que mencionaremos.

-El **namespace Mappers** contiene las clases mapper de cada objeto del dominio; CategoryMapper, ProblematicMapper, PlaylistMapper, AudioMApper, PatientMapper, AdministratorMApper y PsichologistMApper, estas clases heredan de la clase Imapper que también se encuentra en este paquete y se encarga de definir las 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 los métodos update que se encargan de actualizar los objetos si ya están registrados en la base de datos y los métodos GetById, que dado un id me

retorna el objeto de domino correspondiente al objeto en la base de datos con ese Id.

-La **carpeta Migrations** donde se almacenan las migraciones de los datos a la base de datos.

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

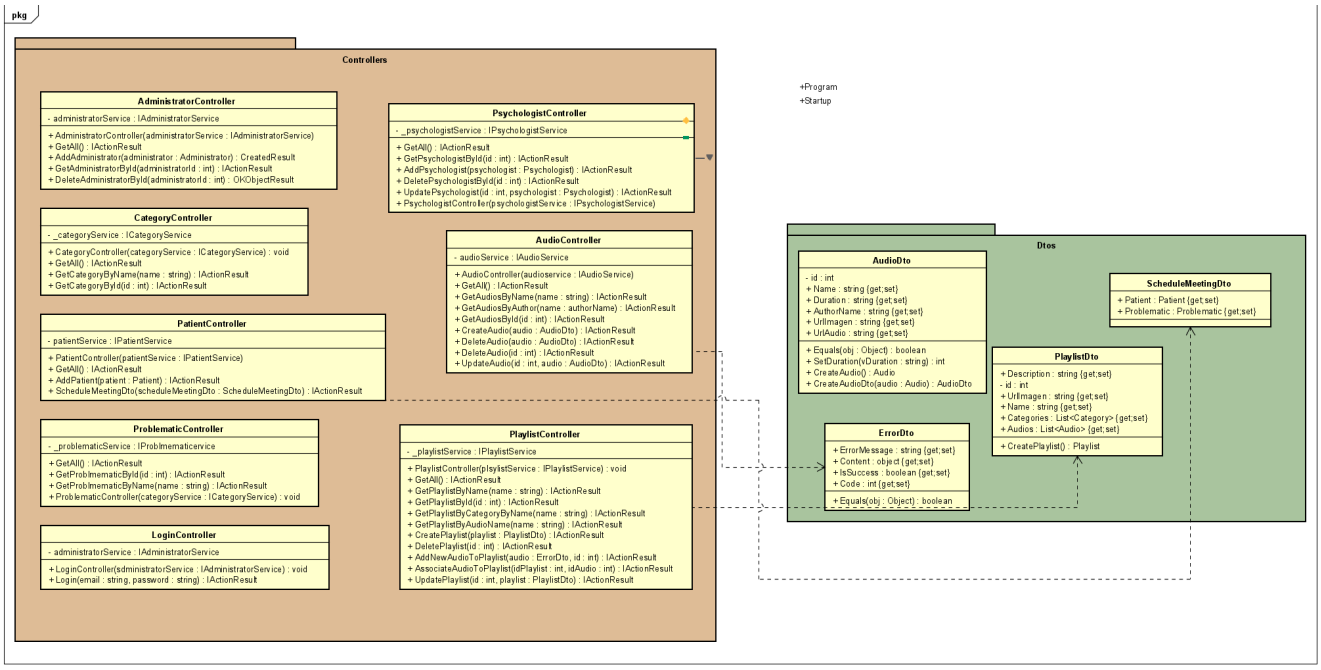
ContextDB: Esta clase hereda de la clase DbContext la cual permite consultar, crear, editar y eliminar registros en una base de datos. En esta clase se encuentran almacenados en DbSets los objetos Dto antes mencionados, respectivos a los objetos del dominio. En esta clase también se encuentra en el método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

1.5.4. WebAPI

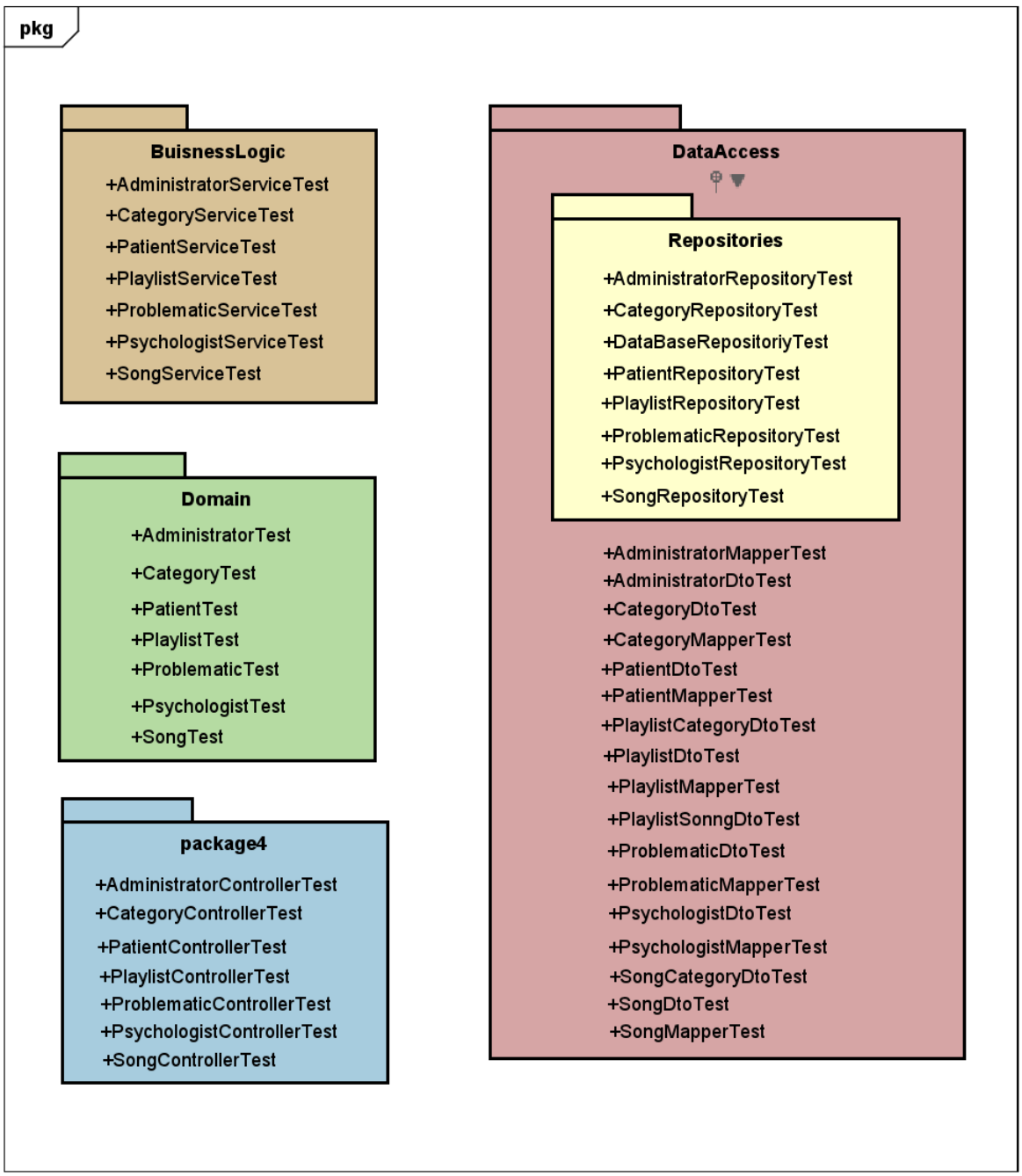
Dentro de WebAPI creamos dos carpetas para agrupar las clases, controllers y filters:

-En el namespace Controllers se encuentran los controllers con los endpoints respectivos a cada objeto del dominio.
-En el namespace Filters se encuentra las clases AuthenticationFilter y el ExceptionFilter



1.5.5. Test

Dentro del paquete Test se encuentran las clases con los test, correspondientes a cada clase de cada paquete. Es por esto que dividimos el paquete por namespaces con los mismos nombres que los paquetes antes mencionados, y las clases las llamamos igual que la clase que prueban mas la palabra test al final como se muestra en el siguiente diagrama.



1.6. Mecanismos de inyección de dependencias, fábricas, patrones y principios de diseño

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

2- Las interfaces creadas en el sistema 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 especificas en cuanto a que métodos poner como parte del contrato.

3- 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.

4- 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 responsability, además cumple con el patrón controlador ya que resuelve el problema de manejar los eventos de entrada y salida del sistema.

5- En lo que respecta a la WebApi la misma mantiene una dependencia hacia BusinessLogic, específicamente sobre los IServices, y conoce la implementación de todos ya que es quien resuelve las dependencias utilizando inyección de dependencias.

6- 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.

7- 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.

8- 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.

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

10- 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 unos e 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.

11- 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.

12- Los test fueron separados en namespaces diferenciados para cumplir Single Responsibility Principle.

13- Para el manejo de excepciones creamos la clase FilterExceptions, para juntar en una única clase la responsabilidad del manejo de errores.

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

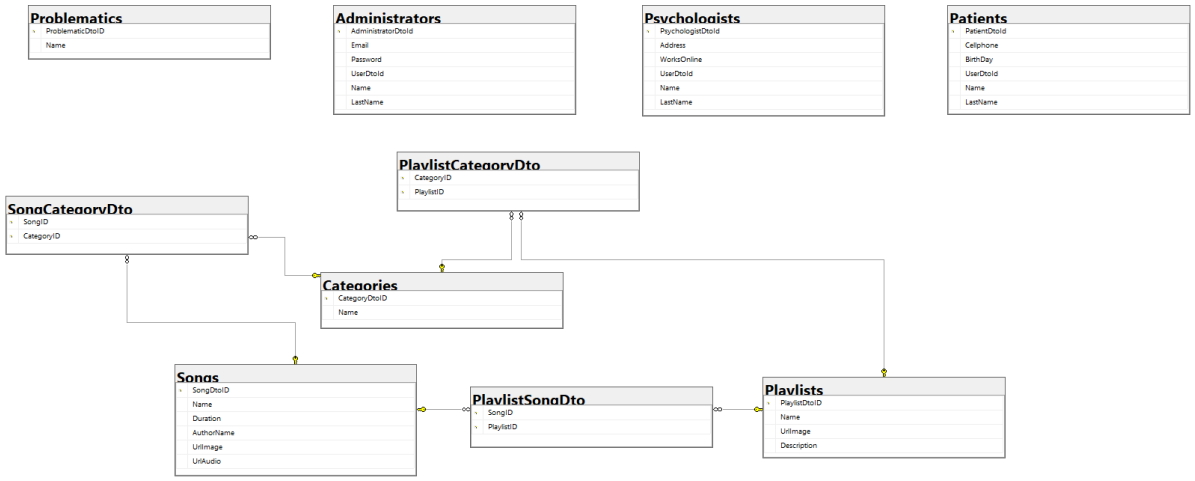
1.7. Descripción del mecanismo de acceso a datos utilizado

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.

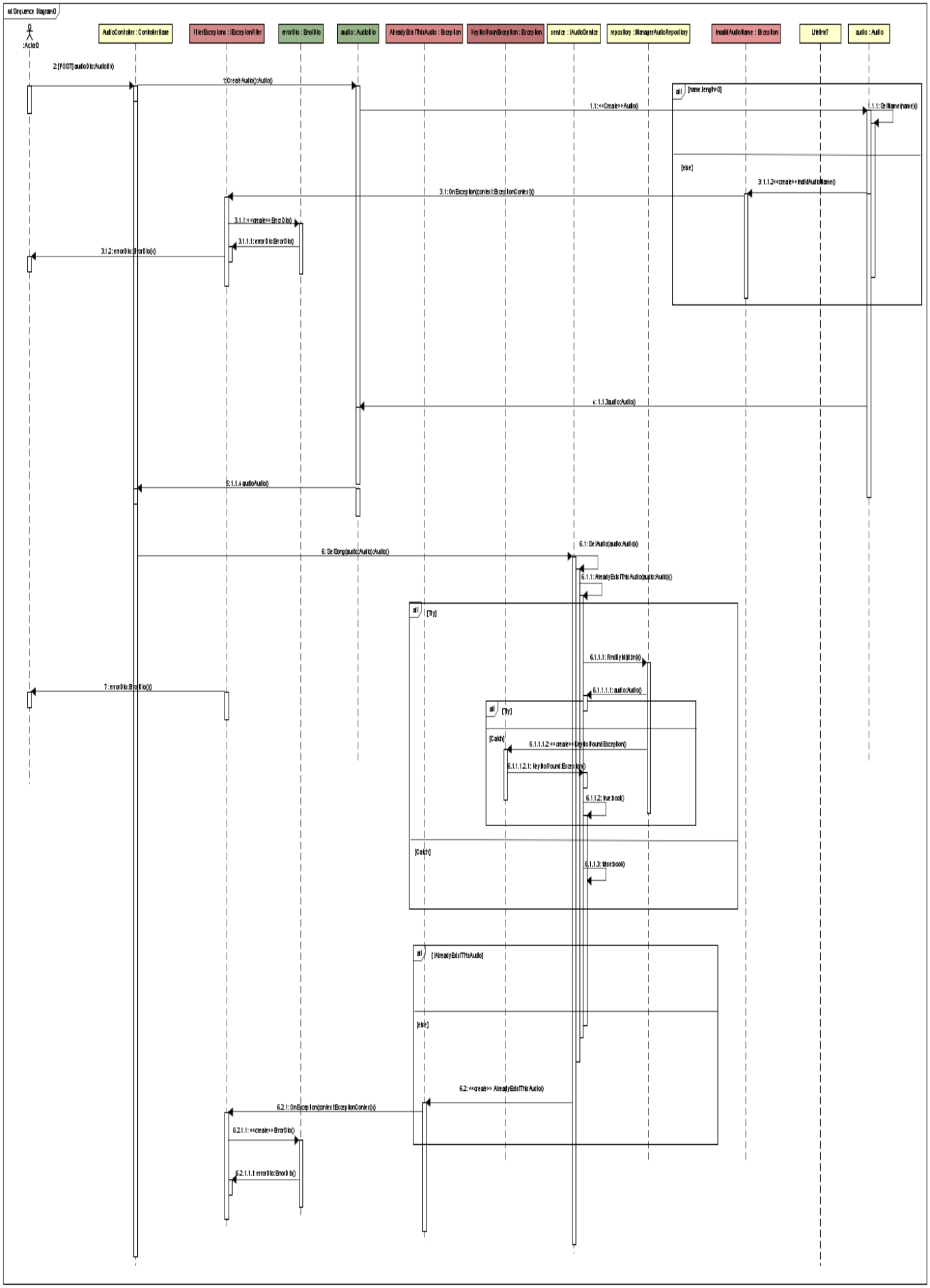
Como se ve en el diagrama además de tablas para los objetos del dominio; CategoryDto, ProblematicDto, SongDto, PlaylistDto, PsychologistDto, PatientDto y AdministratorDto. También se crean tablas para la relación N a N entre playlist y song, playlist y category, song y category; PlaylistCategoryDto,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


sd Sequence Diagram0
:Actor
AudioController : ControllerBase
FilterExceptions : IExceptionFilter
errorBlo : ErrorBlo
audio : AudioBlo
AlreadyExistThisAudio : Exception
KeyNotFoundException : Exception
service : IAudioService
repository : ManagerAudioRepository
InvalidAudioName : Exception
Utime?
audio : Audio
2:[POST] audioBlo:AudioBlo()
1:CreateAudio():Audio()
alt
[name.length>0]
1.1:<<Create>>Audio()
1.1.1:GetName(name)()
[else]
3.1.1.2:<<create>>InvalidAudioName()
3.1:OnException(context:ExceptionContext)()
3.1.1:<<create>>ErrorBlo()
3.1.1.1:errorBlo:ErrorBlo()
3.1.2:errorBlo:ErrorBlo()()
4:1.1.3audio:Audio()
5:1.1.4:audio:Audio()
6:GetSong(audio:Audio):Audio()
6.1:GetAudio(audio:Audio)()
6.1.1:AlreadyExistThisAudio(audio:Audio)()
alt
[Try]
6.1.1.1:FindById(idUnit)()
6.1.1.1.1:audio:Audio()
alt
[Try]
[Catch]
6.1.1.1.2:<<create>>KeyNotFoundException()
6.1.1.1.2.1:KeyNotFoundException()
6.1.1.2:true:bool()
[Catch]
6.1.1.3:false:bool()
7:errorBlo:ErrorBlo()()
alt
[!AlreadyExistThisAudio]
[else]
6.2:<<create>> AlreadyExistThisAudio()
6.2.1:OnException(context:ExceptionContext)()
6.2.1.1:<<create>>ErrorBlo()
6.2.1.1.1:errorBlo:ErrorBlo()

1.9. 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)