Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de Aplicaciones 2
Obligatorio 1

Descripción del diseño

Mauro Teixeira(211753) Rodrigo Hirigoyen (197396)

Docentes: Gabriel Piffaretti, Nicolas Fierro, Nicolas Hernandez

https://github.com/mauroteix/Teixeira-Hirigoyen

Entregado como requisito de la materia Diseño de Aplicaciones 2

6 de mayo de 2021

Resumen

En la siguiente documentación se explicará y justificará el diseño y arquitectura de nuestra solución relacionando las decisiones tomadas con distintos principios y patrones de diseño vistos en clase.

Para esto incluimos distintos diagramas que ayudarán a entender por completo la solución propuesta. Tratamos de diseñar nuestra aplicación para que sea lo más sencilla posible, pero a la vez que pueda el día de mañana tener una buena mantenibilidad.
La misma es una guía para entender el código y los aspectos más relevantes de nuestro diseño y la implementación.

Índice

Descripción y justificación general **3**

Vista lógica **5**

Diagramas de clases 5

Dominio 5

Manejo de errores 6

Capa lógica 7

Capa de acceso a datos 8

Diagramas de secuencia 9

Vista de desarrollo **10**

Diagramas de Paquetes 11

Diagrama de componentes 11

Vista de proceso **12**

Diagrama de Actividad 13

Vista física **14**

Modelo entidad relación **14**

Arquitectura completa **15**

Descripción y justificación general

Nuestro proyecto es una aplicación para el Ministerio de Salud Pública (MSP) que permita a las personas aprender a sobrellevar el estrés de una manera más sana para que cada persona y sus seres queridos puedan desarrollar una mejor capacidad de adaptación y resiliencia. Dicha aplicación es denominada BetterCalm.
A continuación haremos un punteo a alto nivel de las principales funcionalidades. Las mismas son:
- 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.

- 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.

Todas las funcionalidades fueron implementadas correctamente, no se encontró ningún bug relacionado a las mismas.

Un punto clave es el mantenimiento, para que la aplicación sea mantenible también debe serlo el código de la misma, por lo que una parte importante de nuestra solución fue buscar una solución que sea fácil de mantener. La aplicación está desarrollada en .NET Core, utilizando tecnologías como Entity Framework Core y SQL Server para la persistencia de los datos y arquitecturas web REST que brindan los servicios web que la plataforma utiliza. Por motivos de tiempo el trabajo realizado hasta el momento contempla únicamente el backend.

Este incluye desde adentro hacia afuera la base de datos junto con la capa de acceso a datos, las entidades y tablas junto con la lógica de negocio para cada una de ellas y finalmente el extremo de conexión con el mundo exterior, la Web Api. También vale la pena mencionar que la aplicación fue desarrollada utilizando TDD, por lo que aparte de los proyectos mencionados de manejo de datos creamos un proyecto de pruebas para cada uno de los proyectos de nuestro proyecto. De todas formas de los tests y metodología TDD se hace énfasis en el documento de TDD y Clean Code.
Como ya mencionamos, hoy en día la aplicación está siendo implementada de esta forma, pero uno de los requerimientos del cliente es que esta sea lo más mantenible posible y, en caso de el día de mañana querer cambiar el motor de base de datos o la lógica de negocio, poder hacerlo sin mayor impacto en el sistema.
En este punto tuvimos que evaluar alternativas para desarrollar un sistema que en mayor o menor medida con algunos patrones conocidos que justamente nos ayudan a diseñar sistemas

extensibles y mantenibles. Creamos una aplicación orientada a interfaces y como consecuencia se hizo fuerte énfasis en los principios SOLID y GRASP.

Alto acoplamiento y baja cohesión: Como mencionamos anteriormente, tenemos 9 módulos los cuales trabajan en forma desacoplada entre sí. Si bien obviamente existen dependencias entre paquetes, las responsabilidades están claramente divididas en paquetes y dentro de cada paquete, cada clase cumple un rol en el sistema que no es ambiguo con el de otra clase, generando un nivel alto de cohesión entre las clases de un mismo paquete. El uso de interfaces con inyección de dependencias nos permite bajar el grado de acoplamiento.

Single Responsibility Principle: Para reducir el acoplamiento entre clases intentamos que las clases tengan únicamente una responsabilidad, por este motivo nuestra solución contiene una cantidad relativamente alta de clases, las cuales poseen pocas líneas de código dentro. Es una forma de separar responsabilidades y que cada clase tenga bien claro qué hace y el desarrollador pueda mapear funcionalidades con posiciones físicas dentro del proyecto fácilmente.

Open closed: Mediante la utilización de interfaces generamos un sistema en el cual las clases de alto nivel no dependen de implementaciones concretas, siendo nuestras clases abiertas para la extensión pero cerradas para la modificación.

Dependency inversion: Este principio sí lo aplicamos rigurosamente. Para aumentar la mantenibilidad logramos que nuestros módulos de alto nivel no dependan de módulos de bajo nivel. Utilizamos inyección de dependencias para que proyectos como la api o la capa de acceso a datos dependen de interfaces y no de implementaciones. ¿Cómo favorece este aspecto a la mantenibilidad? De esta forma podríamos crear otra clase de implementación que cumpla con el contrato de la interfaz y simplemente cambiar la inyección a la nueva clase y así de fácil cambiamos la funcionalidad.

Inyección de dependencias: Como mencionamos previamente, los principales módulos de nuestro sistema se conectan entre sí por medio de interfaces y no implementaciones concretas de clases. Las implementaciones están definidas en paquetes desacoplados del resto del sistema y son inyectados en las dependencias en la clase Startup del proyecto MSP.BetterCalm.API. Presentamos la documentación utilizando las Vistas del modelo 4 + 1 (no se agregaron casos de uso en este documento).

Vista lógica

Diagramas de clases

Representaremos los diagramas de clases de a paquetes e iremos explicando la información pertinente de cada diagrama.

Dominio

En primer lugar creamos Categorías(Category) la cual como la letra indicaba no se esperaba realizar mantenimiento aunque sí se esperaba un diseño de entities para ser soportados a nivel de base de datos, lo cual nos llevó a crear esta clase que también nos sirve para en un futuro poder ampliar esta gama de categorías y no limitarlas.

Luego creamos la entidad Playlist en la cual guardamos las playlist cuenta con un nombre, una descripción (Con máximo 150 caracteres) y un link a una imagen, esta clase tiene al menos una categoría como lo fue indicado, por lo que creamos una lista con una tabla intermedia PlaylistCategory, la misma tiene ambos id, tanto de categoría como de playlist como identificadores y a su vez cuenta con los objetos de cada una.

Para el contenido reproducible creamos la entidad Track, la misma tiene un nombre, autor, imagen, en la duración (Tomamos una decisión para facilitar al momento de mostrar la duración en el futuro por lo que separamos en dos partes), por un lado las horas y por el otro los minutos segundos, y también cuenta con un sonido. Esta clase debe tener al menos una categoría, por lo que le agregamos una lista con la tabla intermedia CategoryTrack, que cuenta con ambos id, tanto de categoría como de track como identificadores y a su vez cuenta con los objetos de cada una.
Por letra un contenido reproducible puede estar o no asociado a una playlist por lo que decidimos crear una lista con otra tabla intermedia PlaylistTrack, la cual tiene ambos id como identificadores tanto de playlist como de track y a su vez los objetos de ambas.En este caso la misma puede estar vacía ya que por letra no es necesario que esté asociado a una playlist un contenido reproducible.

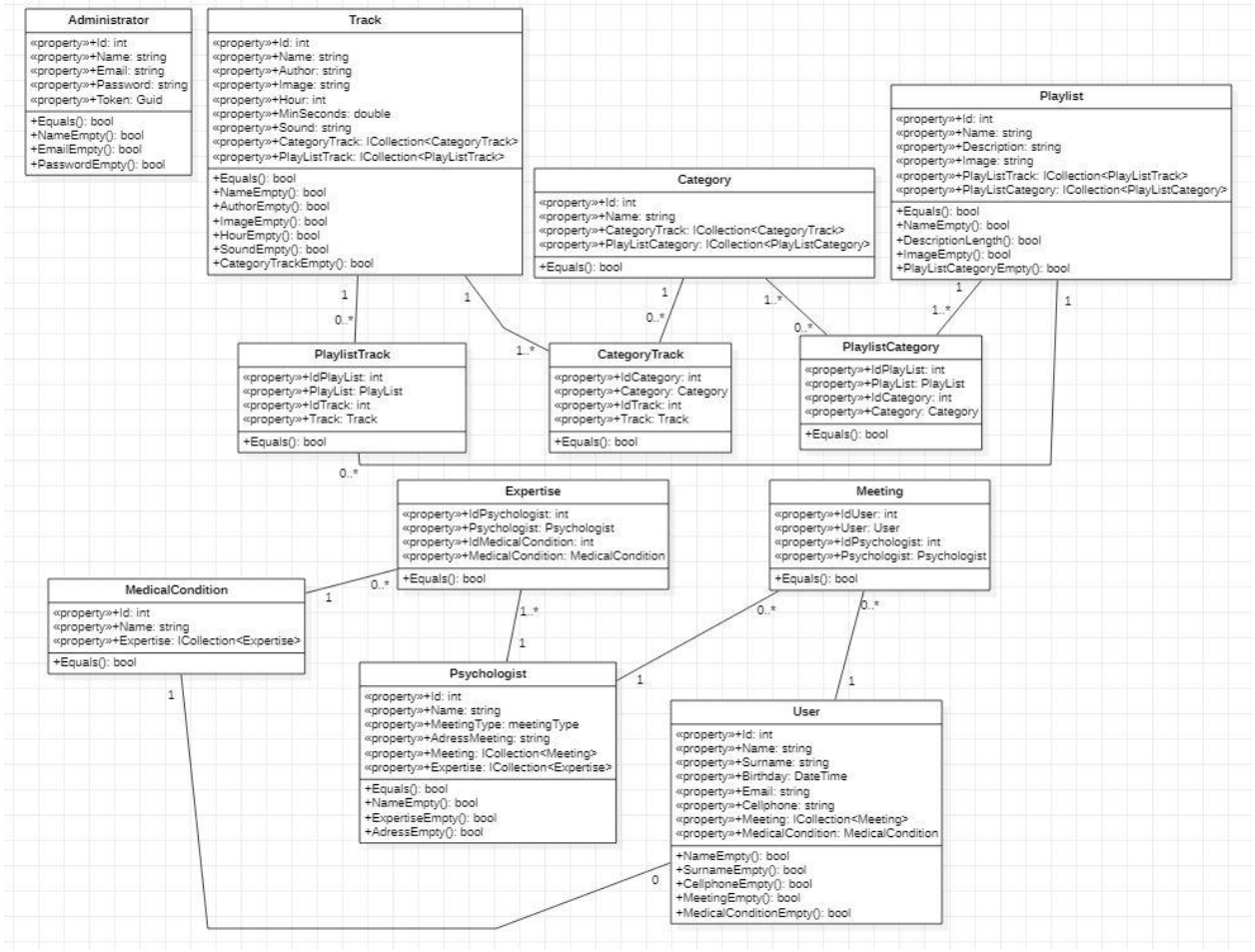
Luego seguimos con la otra funcionalidad y comenzamos con Problemáticas(Medical Condition), la cual como la letra indicaba no se esperaba realizar mantenimiento aunque sí se esperaba un diseño de entities para ser soportados a nivel de base de datos, lo cual nos llevó a crear esta clase que también nos sirve para en un futuro poder ampliar esta gama de problemáticas y no limitarlas.

Seguimos con la entidad Psychologist, la misma tiene un nombre, tipo de reunión que tomamos la decisión de hacer un enum (Face to face o virtual), una dirección de la reunión, por letra cada psicólogo cuenta con 3 principales problemáticas en la que es experto, en este caso decidimos

crear una lista con otra tabla intermedia Expertise la misma cuenta con los id de psicologo y problemática como identificador y a su vez cuenta con los objetos de ambas.

Para registrar la reunión primero creamos la entidad User, la cual contiene nombre, apellido, fecha de nacimiento, email , un celular y una problemática la cual es seleccionada previamente por el usuario. A su vez creamos una lista con una tabla intermedia Meeting, la misma contiene id de user y de psicólogo como identificadores, esta tabla también contiene ambos objetos de los identificadores, una fecha y una dirección de reunión.

Para finalizar nuestro dominio vamos a tener una entidad Administrator, la cual contendrá un nombre, email, contraseña y un token, el cual se le dará en la aplicación permisos para utilizar diferentes endpoints que un usuario común(no registrado) no podrá acceder.

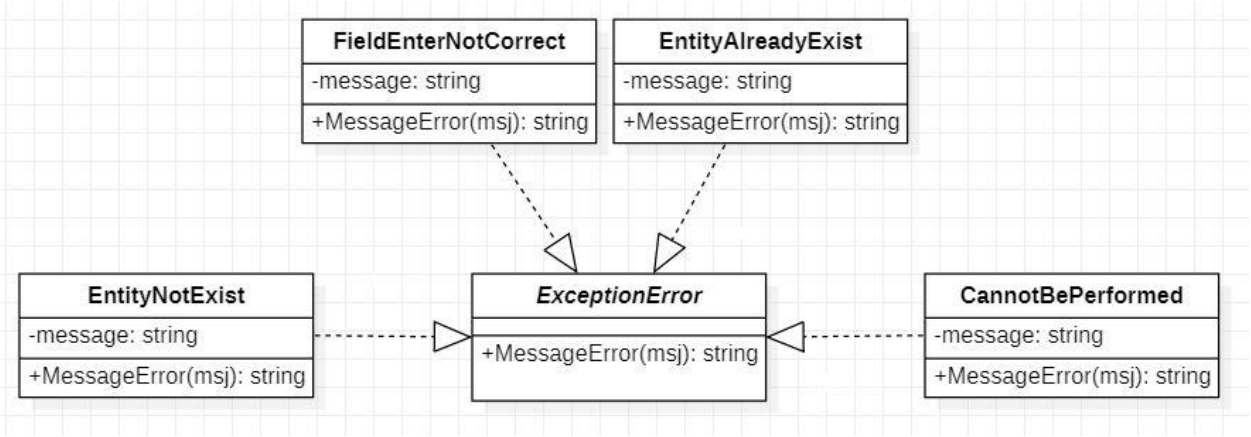


Manejo de errores

Para reforzar el concepto de división de responsabilidades, notamos que en diversos lugares de nuestra aplicación se lanzaban las excepciones, que al fin y al cabo no eran más que las mismas excepciones por situaciones en nuestros casos de uso. Por este motivo decidimos empaquetar las mismas dentro de un proyecto de manejo de excepciones, representando cada excepción común en forma de clase la cual recibe por parámetro el mensaje de error.

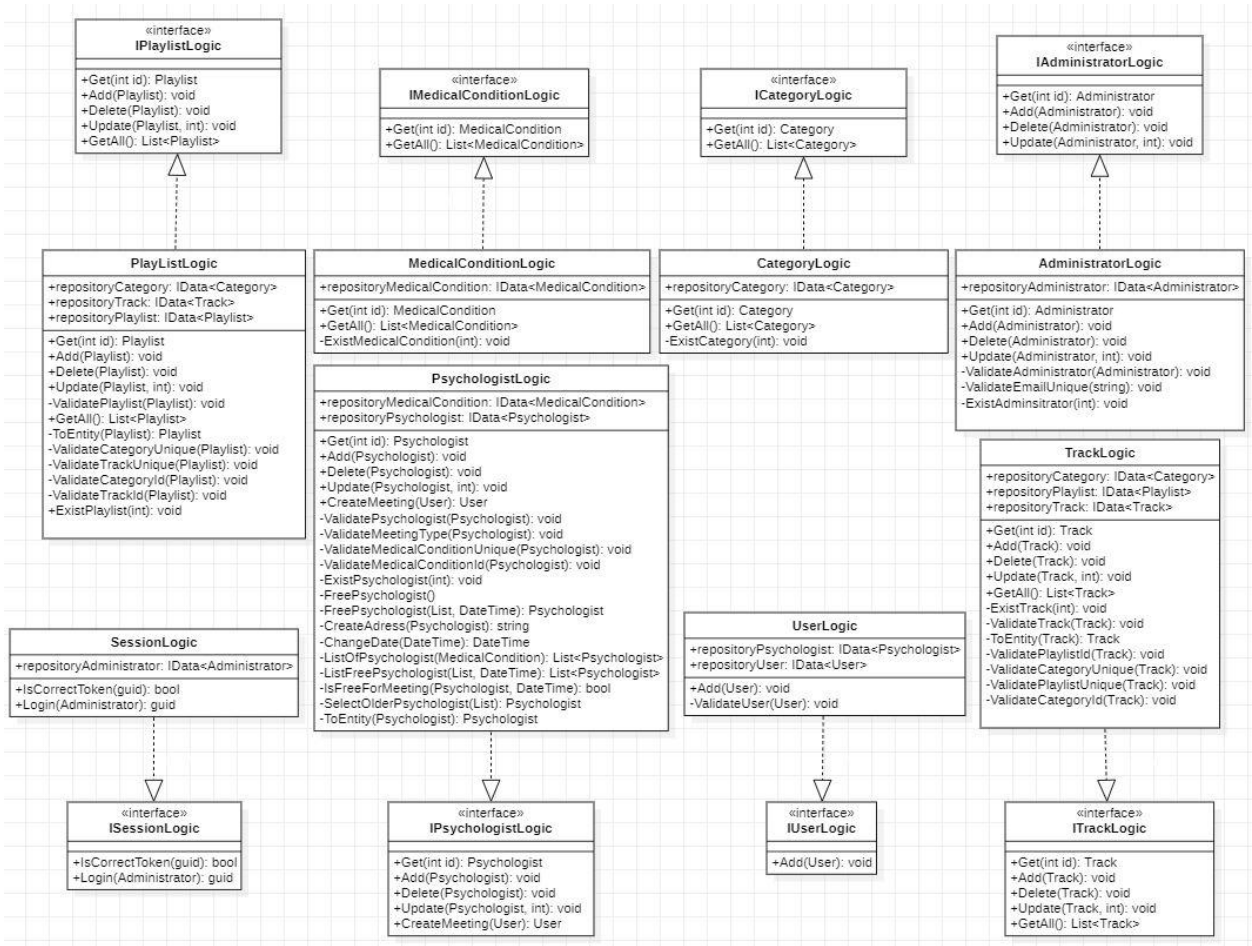
Como existen varios códigos de error relacionados al mundo de las api, tuvimos que tener mucho cuidado de cómo manejamos nuestros errores internamente para que al usuario que está consumiendo nuestra api le pueda llegar el error específico que sucedió y además un texto con una descripción lo más acertada posible para que pueda darse cuenta de que fue lo que realmente sucedió.

Por este motivo decidimos crear una biblioteca de clases en nuestra solución, la cual cada clase que creemos será una error diferente, estos errores heredarán de una clase abstracta la cual la creamos para que todos los errores tengan un mensaje de error (lo cual la clase abstracta los obliga a esto) y además esta clase abstracta hereda de Exception ya que en nuestro sistema, cada vez que necesitamos tirar un error el throw exception necesita que se herede de Exception lo cual es evidente. Las excepciones son lanzadas en la capa de acceso a datos o en la lógica de negocio dependiendo el caso, y capturados en la api para lanzar mensajes de error al usuario lo más acertados posibles. A continuación mostramos como esta armada nuestra arquitectura para el manejo de errores.



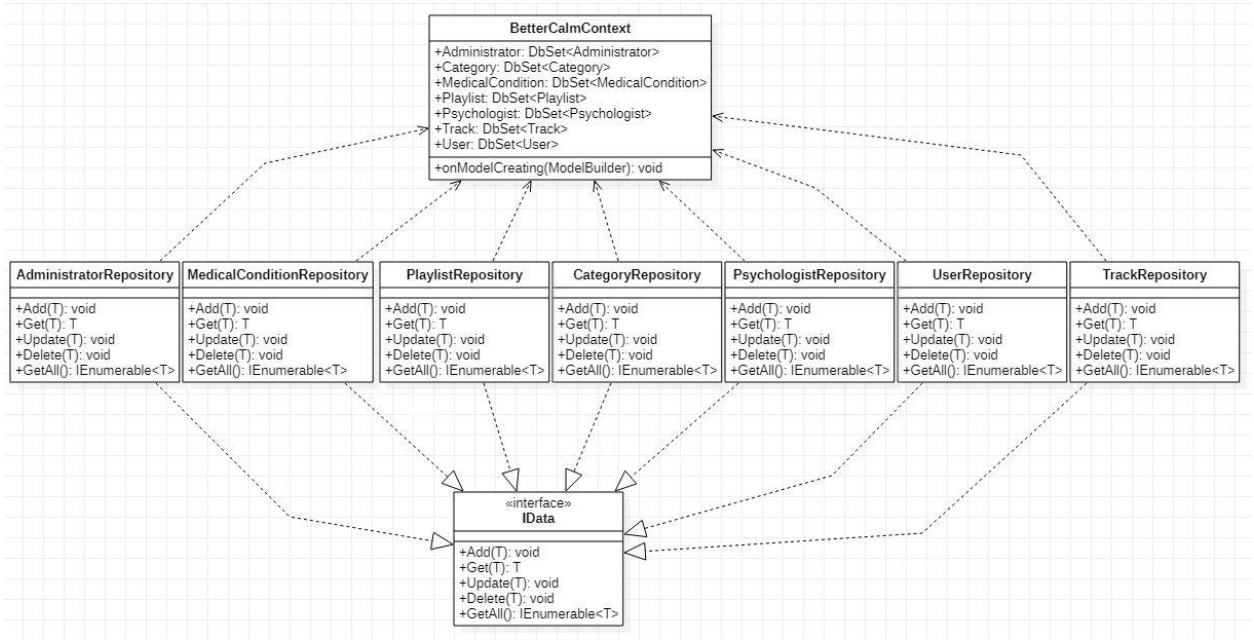
Capa lógica

Nuestra lógica de negocio se encarga de definir el comportamiento que debe tener cada una de nuestras entidades. definimos interfaces las cuales son requeridas por la API para realizar la validación de los datos recibidos y actuar como con la capa de acceso a datos (se puede observar gráficamente en el diagrama de componentes).
Por decisión del equipo optamos por encapsular la lógica de validación de campos en cada una de las BusinessLogic de las entidades, ya que consideramos que debe ser la entidad misma que necesita para considerarse válida.
Por más que las interfaces tienen varios métodos en común, no consideramos correcta la implementación de una lógica general para todas las clases debido a que iba a ser necesario aplicar el principio de segregación de interfaces para evitar que las implementaciones se vean obligadas a implementar métodos que no sean utilizados en la lógica de negocios actual.



Capa de acceso a datos

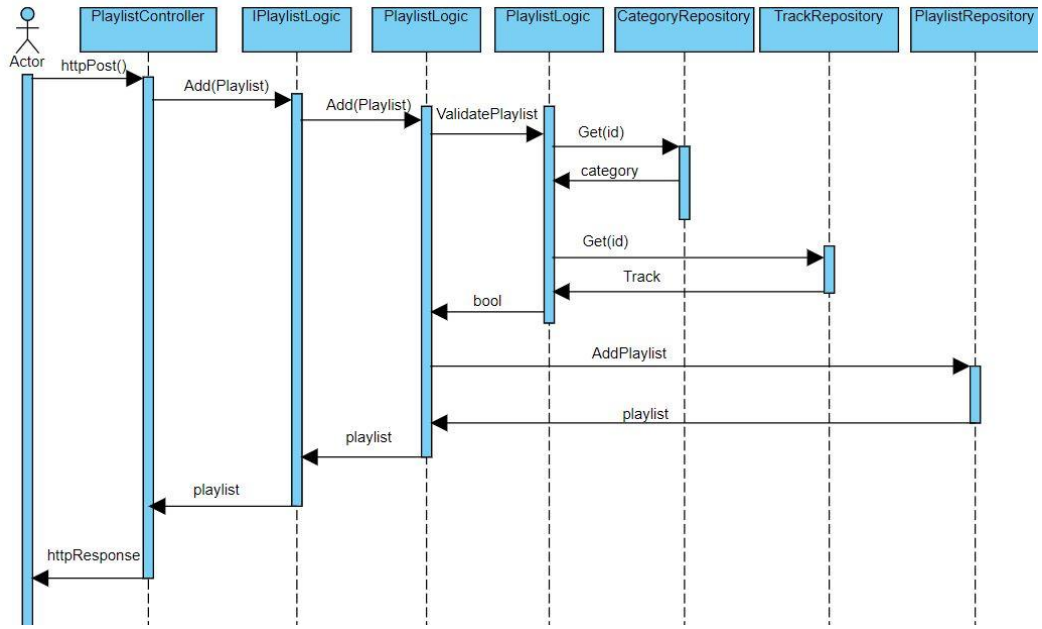
Para el acceso a datos se utilizó SQLServer con Entity Framework Core Code-First, mantuvimos la base de datos actualizada mediante el uso de migrations. Como no queremos depender necesariamente de la base de datos Sql Server que utilizamos, porque el día de mañana podría cambiar, creamos una interfaz para depender de la misma y no importarnos que es lo que sucede por atras . Como vimos que se repetían muchas operaciones entre las distintas base de datos, pensamos en crear una clase abstracta, para dos cosas: La primera tener un contrato la cual toda clase que implemente un repositorio deba cumplir con las operaciones que se le especificaron y la segunda compartir código entre todas, ya que como hablábamos anteriormente, mucho codigo se repite. Cada uno de los repositorios implementa a IData. Notamos que las entidades del sistema tenían siempre el mismo comportamiento CRUD, por lo que creamos una interfaz para que dichas clases se implementan, haciendo intercambiable la implementación concreta siempre y cuando se cumpla con el contrato



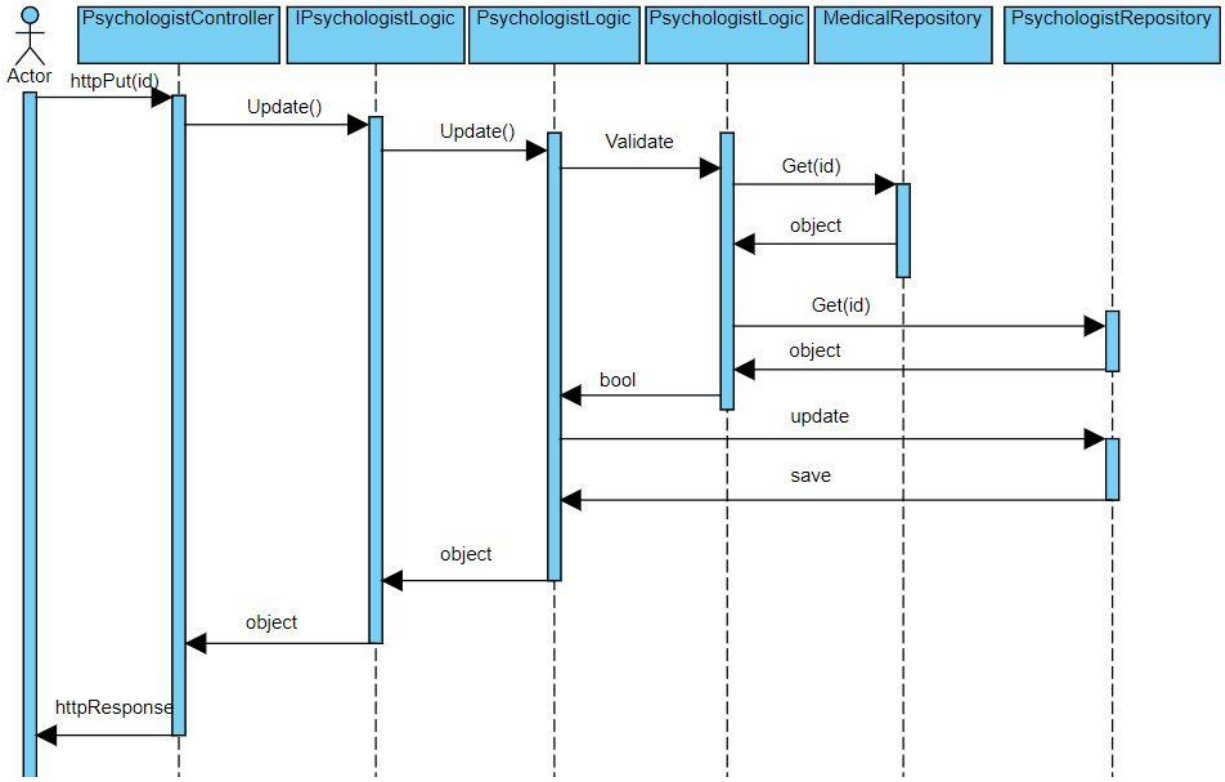
Diagramas de secuencia

Agregar Playlist

En este diagrama representamos como agregar una playlist. El controller de playlist llama a la interface IPlaylistLogic que luego llama a la playlistLogic para hacer el add de playlist. Tenemos dos playlistlogic porque el método validatePlaylist tiene varias llamadas a otras clases, tanto categoryrepositroy y tackrespository para validar y verificar que existan las clases asociadas a esta playlist. Para finalizar crea y agrega la playlist desde playlistRepository y devuelve la respuesta.



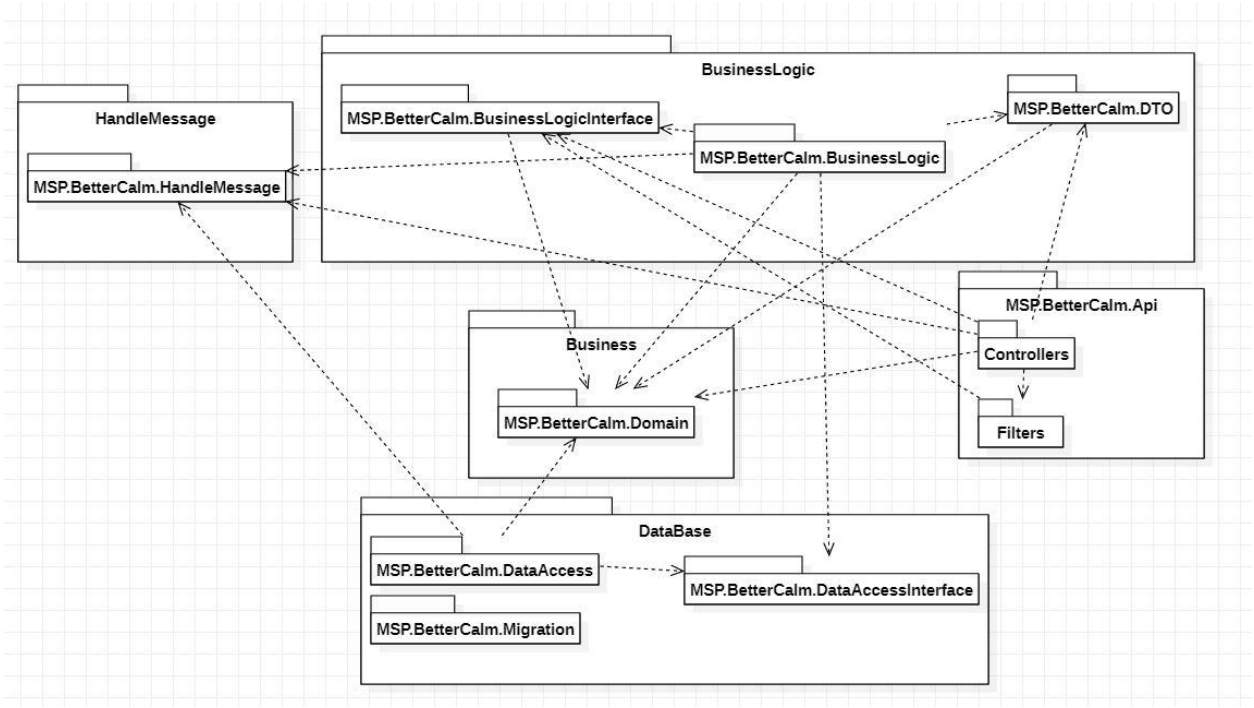
Update Psychologist
Este diagrama representa la secuencia de como hace update nuestro programa. Primero llama a la interface de la lógica para luego validar sus elementos de medical condition,luego valida el psicólogo si existe y si tiene bien los datos. Para finalizar hace el update desde psychologist Repository y devuelve la Response.



Vista de desarrollo

La vista de desarrollo ilustra el sistema de la perspectiva del programador y está enfocado en la administración de los artefactos de software. Esta vista también se conoce como vista de implementación. Utiliza el Diagrama de Componentes UML para describir los componentes del sistema. Otro diagrama UML que se utiliza en la vista de desarrollo es el Diagrama de Paquetes. Enfocándonos en nuestra solución, para poder definir los distintos diagramas debemos saber que el sistema que realizamos consta de 9 proyectos, 4 de ellos refiere a los test del sistema.

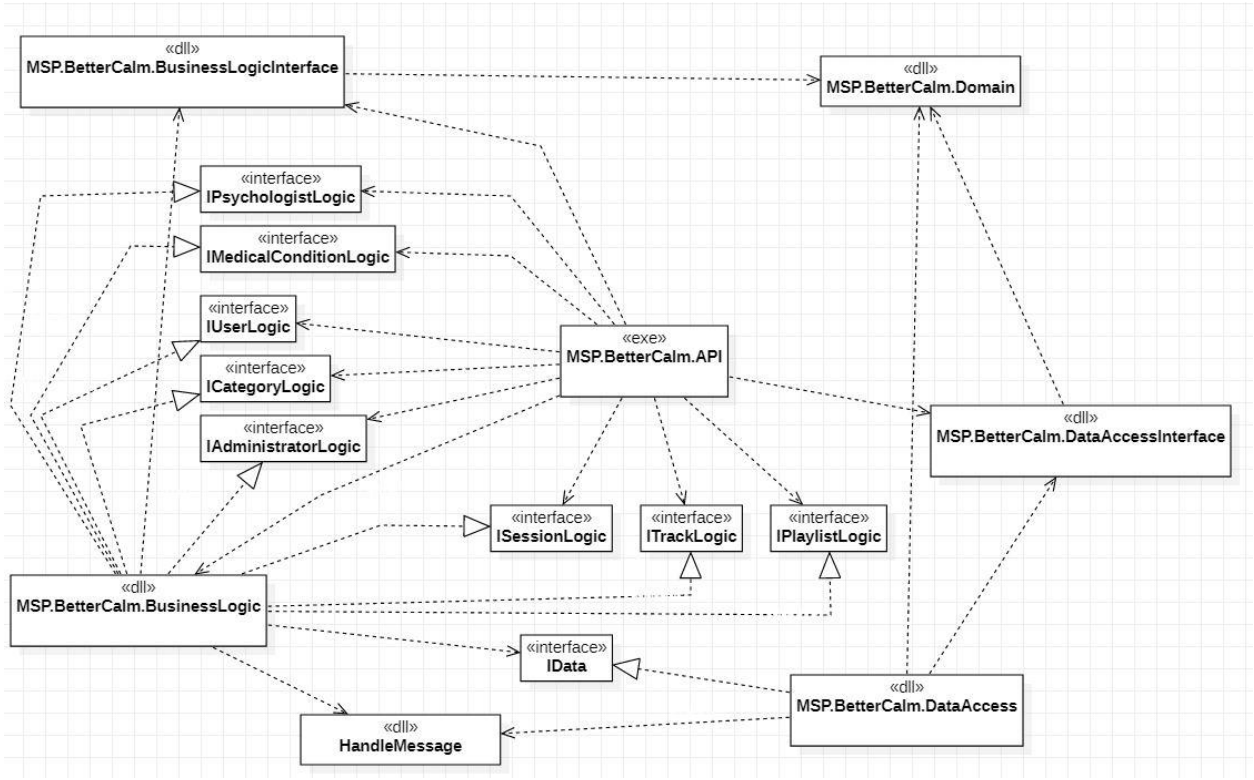
Diagramas de Paquetes



Como podemos ver, el patrón SRP lo aplicamos para los paquetes en sí mismos. La idea fue tener paquetes que cumplan una funcionalidad concreta, que dentro del mismo se encuentren clases con alta cohesión y mantener un acoplamiento entre las clases de los paquetes lo más baja posible. A su vez los nombres a utilizar cumplen con el estándar es decir nombre empresa/nombre aplicación / nombre paquete, seguimos toda esa lógica en todo los paquetes creados, en nuestro caso MSP.BetterCalm. Y el nombre del paquete.

Diagrama de componentes

Representamos mediante un diagrama de componentes la estructura estática de la arquitectura de nuestra solución, representamos cada componente como un .exe o .dll, el cual se puede relacionar con otro componente del sistema por medio de relaciones o interfaces, las cuales son representadas con notación de Hooks. Como podemos ver, más allá de las relaciones entre componentes que son bastante similares a las del diagrama de paquetes, agregamos la notación de Hooks con interfaz requerida e implementada.



Hook DataAccess-BusinessLogic:
DataAccess provee la interfaz IData la cual es requerida por las clases de la lógica de negocio que como mencionamos anteriormente, ejecutan operaciones CRUD por lo que consideramos adecuado generar ese contrato entre la lógica y la capa de acceso a datos. De esta forma, las clases de lógica van a depender de una interfaz en lugar de implementaciones concretas, lo que nos permite tener una capa de acceso a datos intercambiable y módulos poco acoplados entre ellos.

Hook BusinessLogic- WebApi:
El componente BusinessLogic provee interfaces a la WebApi, con el objetivo de que la misma pueda acceder a la información necesaria sobre las entidades que se utilizan para manipular los recursos. Estas interfaces no solo permiten acceder a la lógica de cada una de estas entidades, sino también a los repositorios para el acceso a los datos.

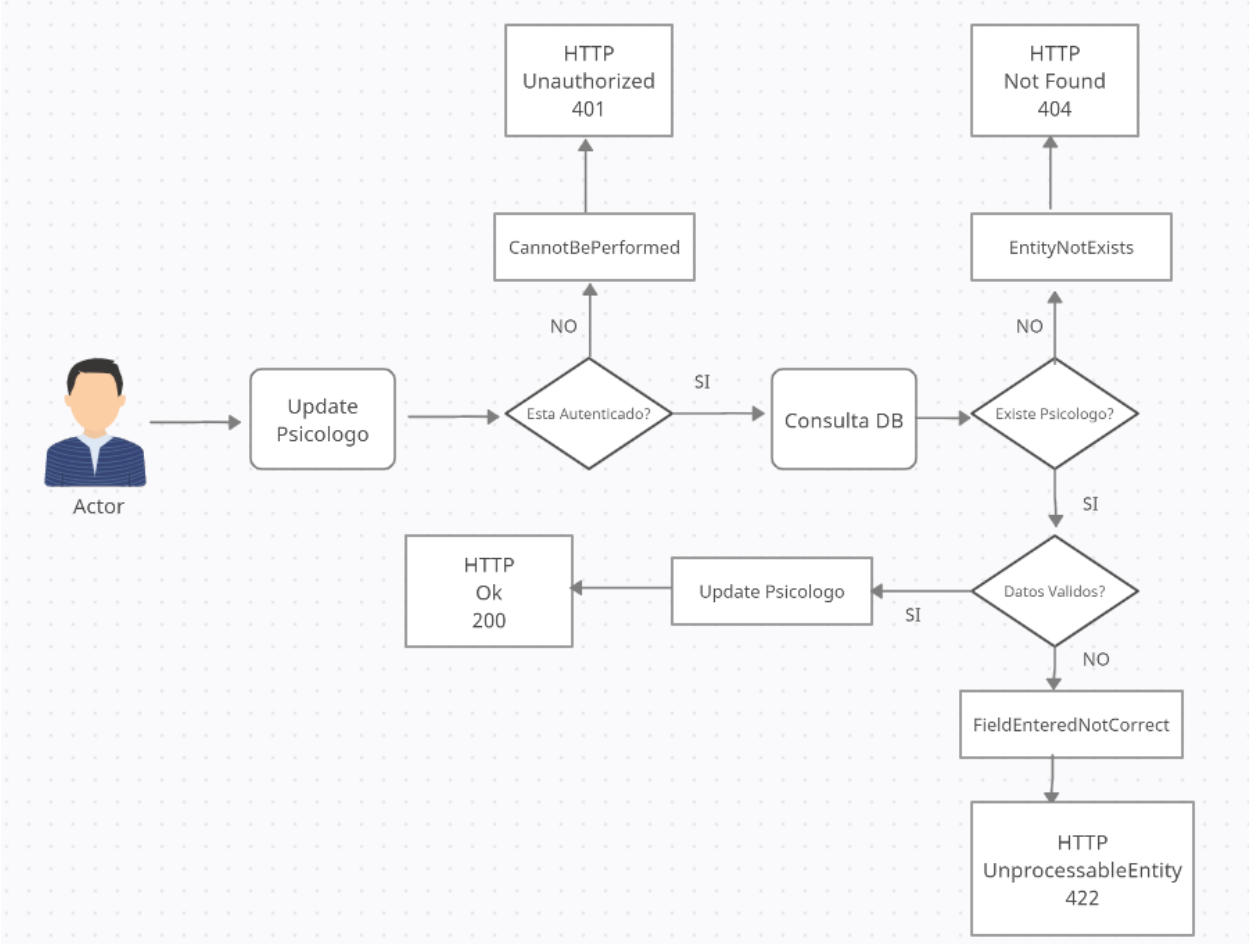
Vista de proceso

Representa los aspectos dinámicos de nuestro sistema, cómo funcionan los procesos y cómo se comunican entre ellos. Vamos a explicar mediante un diagrama de actividad para representar una funcionalidad para administradores en nuestro sistema.Cabe aclarar que en general la mayoría de los procesos de nuestra aplicación funcionan de esa forma. Una request a una api, la cual puede o no requerir permisos de administrador, seguida de validaciones de campos a cargo de la lógica de negocio de la entidad que representa al recurso y consultas a la base de datos para verificar la existencia y generar retorno de los datos o en otro caso

mensajes HTTP con códigos de errores pertinentes, las excepciones son lanzadas en la lógica de negocio y utilizadas por la api para enviar mensajes de error acordes al caso.

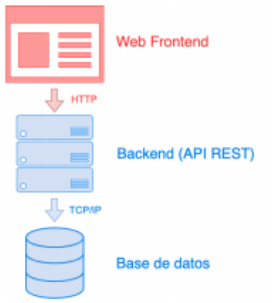
Diagrama de Actividad

Realizamos diagrama de actividad de la funcionalidad actualizar psicólogo. Utiliza distintos tipos de excepciones y respuestas HTTP, 401 para cuando el usuario no está autenticado 404 para cuando el psicólogo que queremos actualizar no existe, 422 para cuando los campos ingresados no son correctos y 200 cuando todo se realizó correctamente.

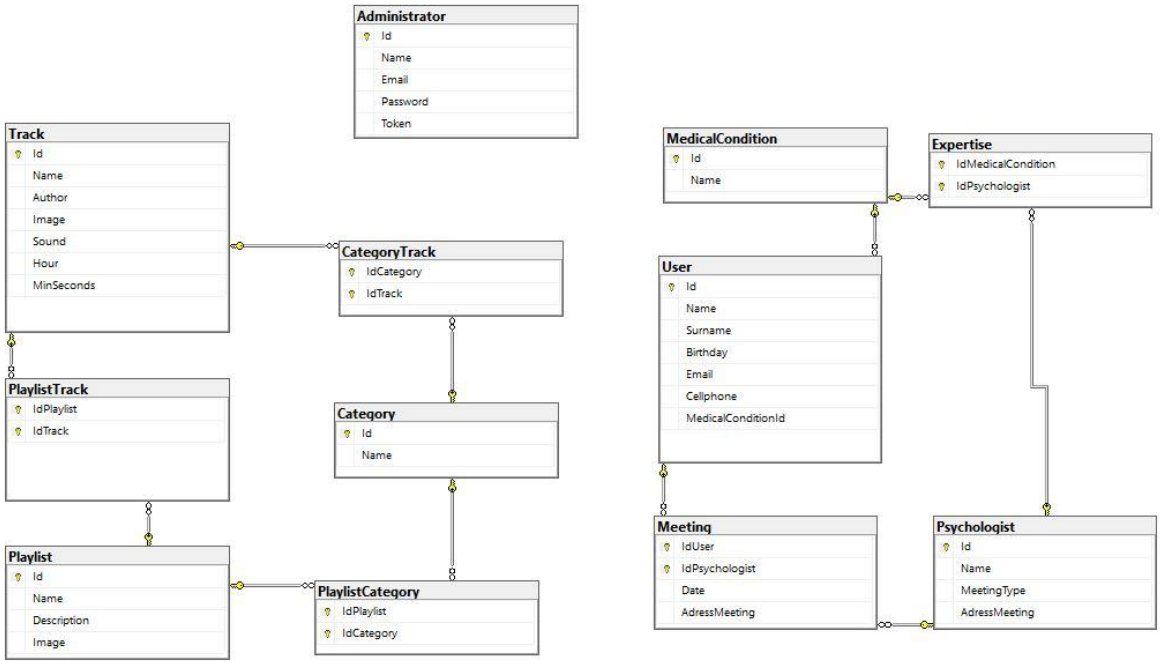


Vista física

Teniendo en cuenta que la comunicación de nuestra API hacia el mundo exterior se realiza mediante protocolo HTTP utilizando arquitectura REST y la conexión entre nuestro backend y la base de datos se realiza mediante el protocolo TCP-IP, tal como lo indica el siguiente diagrama:



Modelo entidad relación



Arquitectura completa

Como se puede apreciar en la arquitectura completa tenemos separados los niveles en 3 colores, siendo rojo el nivel de la capa de acceso de datos, amarillo la capa logica y en azul los controller.