Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Obligatorio

Diseño de aplicaciones 2

Integrantes

- Juan Pablo Sobral - 192247
- Chiara Di Marco - 186017

Links

- GitHub: https://github.com/ORT-DA2/BetterCalm-DiMarco-Sobral

Docentes

- Ignacio Valle
- Gabriel Piffaretti
- Nicolás Fierro

Descripción general de la solución

La solución desarrollada es una API REST cuyo objetivo es permitir que los usuarios accedan a contenidos de audio como podcasts o música, y que puedan coordinar consultas psicológicas.

La solución fue desarrollada utilizando el framework WebApi de .NET core, utilizando C# y SQL Server 2017 como motor de base de datos.

En esta iteración la única forma que van a tener los usuarios de interactuar con la aplicación va a ser mediante un cliente HTTP, como Curl o Postman, pero el plan es agregar un cliente web en la siguiente iteración. Una de las ventajas de haber desarrollado la solución como una API REST es que luego vamos a poder desarrollar clientes para todo tipo de plataformas, los cuales consuman esa capa de servicios.

Errores y bugs conocidos

- No se realizan validaciones tan exhaustivas como nos gustaría.
- Debido a un problema ocurrido cerca de la fecha de entrega, no se pudo implementar el filter ExceptionFilter. De todos modos, no se retiró del proyecto dado que en la segunda instancia de obligatorio este va a ser utilizado.
    - Posible implementación:

Se crea un paquete nombrado Filters en el cual se almacenarán todos los distintos tipos de filters que se utilicen en nuestro programa. Luego se crea la clase que implementa dicho filter, en nuestro caso sería ExceptionFilter. Se agrega el decorador en algunos métodos en particular o en todo el controller y se realizan las pruebas necesarias.

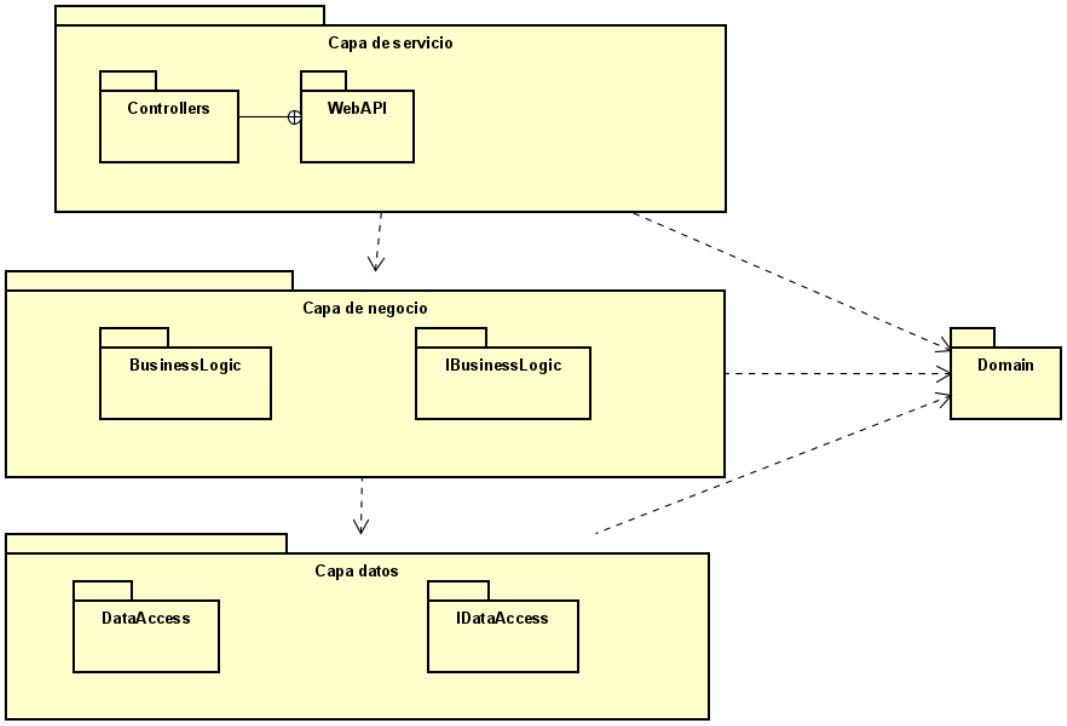
- Debido a la cantidad de cambios que implicaba esta implementación y considerando que la fecha de entrega se encontraba muy cerca, no logramos realizar la restricción de hardcordear el id en todas las clases de BusinessLogic.
    - Posible implementación:

Se agrega el siguiente método en todas aquellas clases de BusinessLogic que creen un objeto y luego se deberían cambiar las respectivas pruebas en el paquete UnitTest que prueben dicho método.

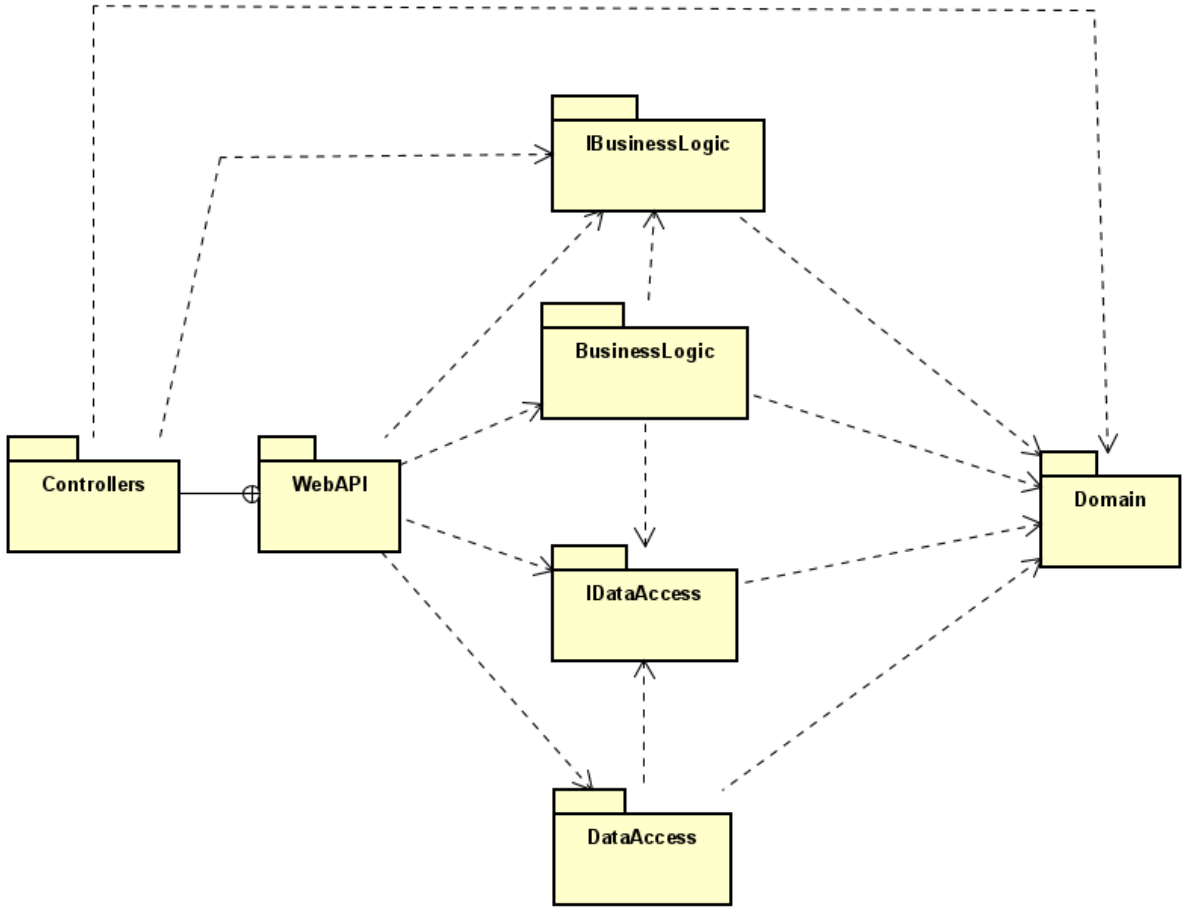
```
public void ValidId(int id)
{
    if (id != 0)
    {
        throw new Exception("Do not hardcode contentId, it is an autogenerated value.");
    }
}
```

Diagramas de paquetes

Primero que nada, vamos a dividir los paquetes en capas y mostrar las dependencias entre estas:



Luego un diagrama de paquetes representando de manera explícita las dependencias entre ellos:



Descripción de responsabilidades de los paquetes

Domain

El paquete Domain agrupa las clases que representan a las entidades con las que trabaja la solución. Es por este motivo que se ven dependencias desde todos los otros paquetes hacia este, dado que la gran mayoría de las clases en nuestra solución trabajan con al menos una de las entidades de nuestro dominio.

DataAccess

El paquete DataAccess agrupa las clases que van a controlar el acceso a la base de datos, es decir, cualquier operación que implique leer o escribir a la base de datos va a utilizar una de las clases de este paquete.
Este paquete depende del dominio dado que trabaja con esas entidades, y del paquete IDataAccess dado que éste contiene la definición de la interfaz que las clases de este paquete van a implementar. Luego el paquete WebApi depende de DataAccess, dado que va a necesitar conocer las clases del mismo para poder realizar inyección de dependencias.

BusinessLogic

El paquete BusinessLogic agrupa las clases que van a estar encargadas de llevar a cabo las funcionalidades que el negocio requiere de la solución. Es decir, si nuestra aplicación va a permitir por ejemplo que un usuario pueda ver las categorías de contenidos, va a haber una función dentro de una clase en este paquete cuya responsabilidad sea retornar las categorías de contenidos.
Este paquete depende del dominio dado que trabaja con esas entidades, y de los paquetes IDataAccess e IBusinessLogic. Depende del paquete IDataAccess para permitir un desacople de la lógica de negocios de la implementación de acceso a datos realizada, y logra que las clases de lógica de negocio no tengan que conocer la implementación de las operaciones de acceso a datos. Por otro lado, la mayoría de las clases en este paquete van a implementar interfaces definidas en el paquete IBusinessLogic, justificando esa dependencia.

WebApi.Controllers

Este paquete agrupa los controllers que se van a encargar de responder las consultas a la API. En el diagrama quisimos mostrar por separado las dependencias de este paquete de las del paquete WebApi, porque más allá de potencialmente tener acceso a las clases de todos los paquetes de la solución, nuestros Controllers únicamente utilizan las interfaces de lógica de negocios definidas en el paquete IBusinessLogic y entidades del Dominio.

WebApi

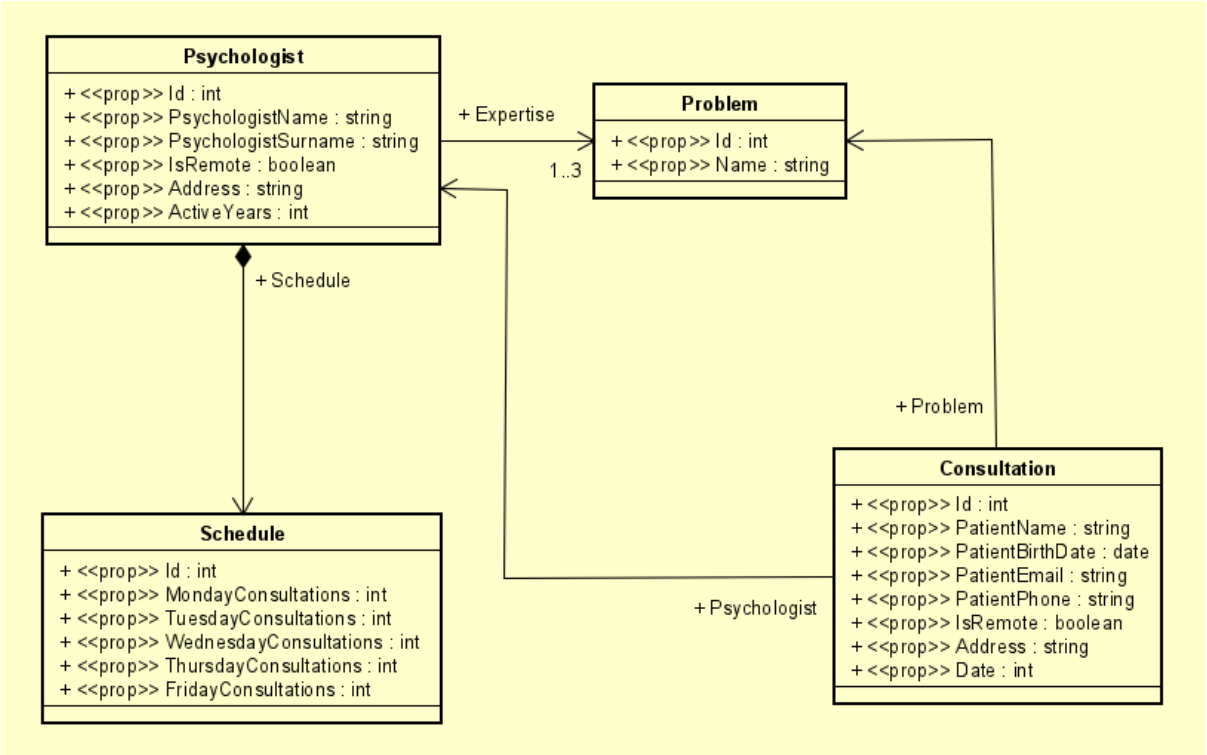
Este paquete agrupa las clases Startup y Program que nos provee el framework webapi de .net core. Estas clases nos van a ayudar con la configuración básica de servicios que va a utilizar nuestra API, como por ejemplo autenticación.

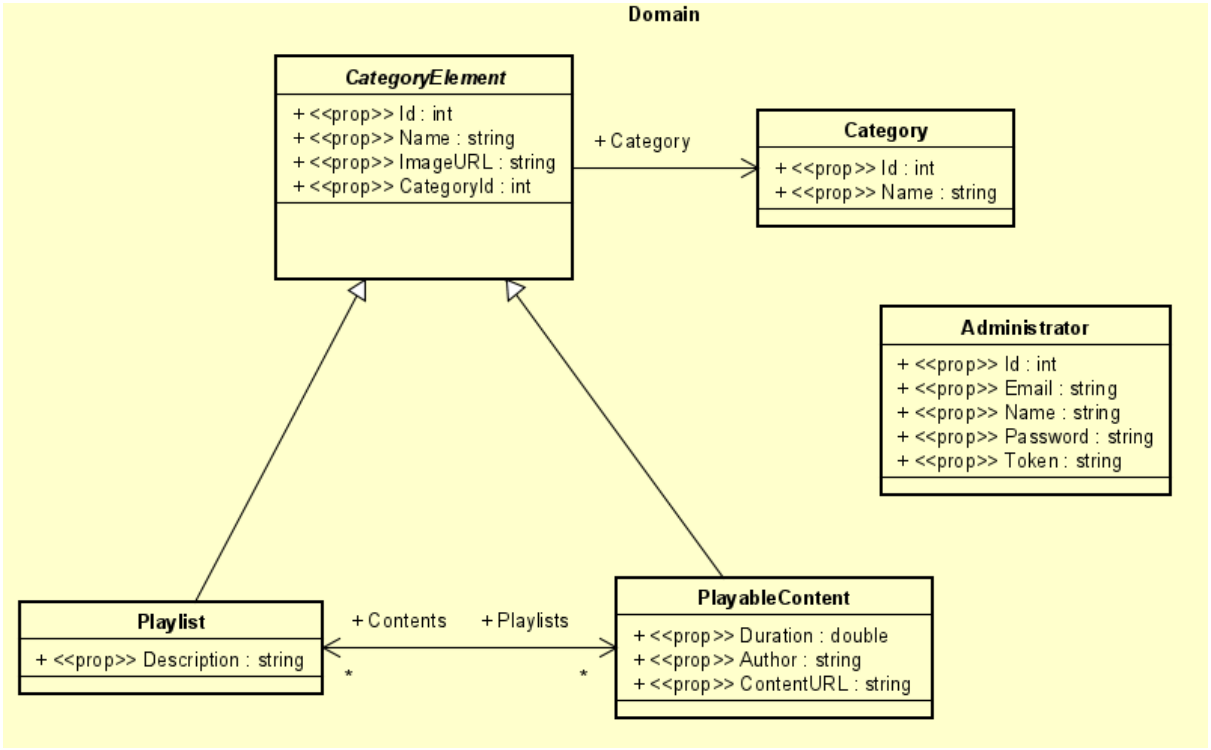
Se puede ver que este paquete depende tanto de paquetes de interfaz como de paquetes de implementación, y esto tiene una justificación válida: inyección de dependencias. En la clase Startup.cs, además de configuración de servicios se va a realizar inyección de dependencias utilizando interfaces y clases concretas.

Diagramas de clase por paquetes

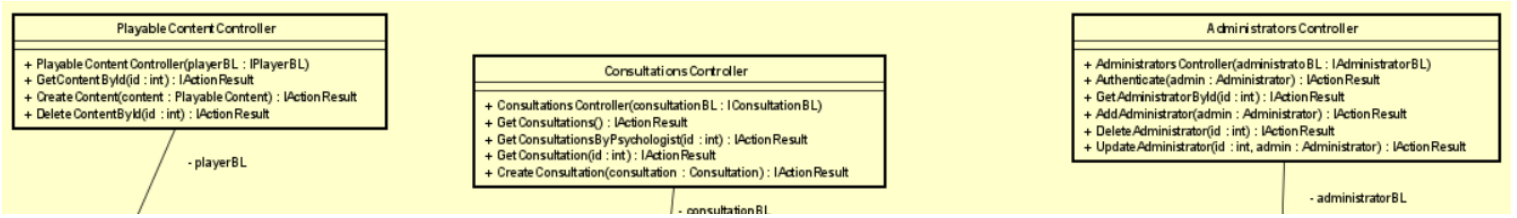
La complejidad de la solución y el formato del documento no colaboran con esta sección por lo cual además de dejar capturas aquí vamos a adjuntar el diagrama en formato .asta que puede ser abierto usando Astah.

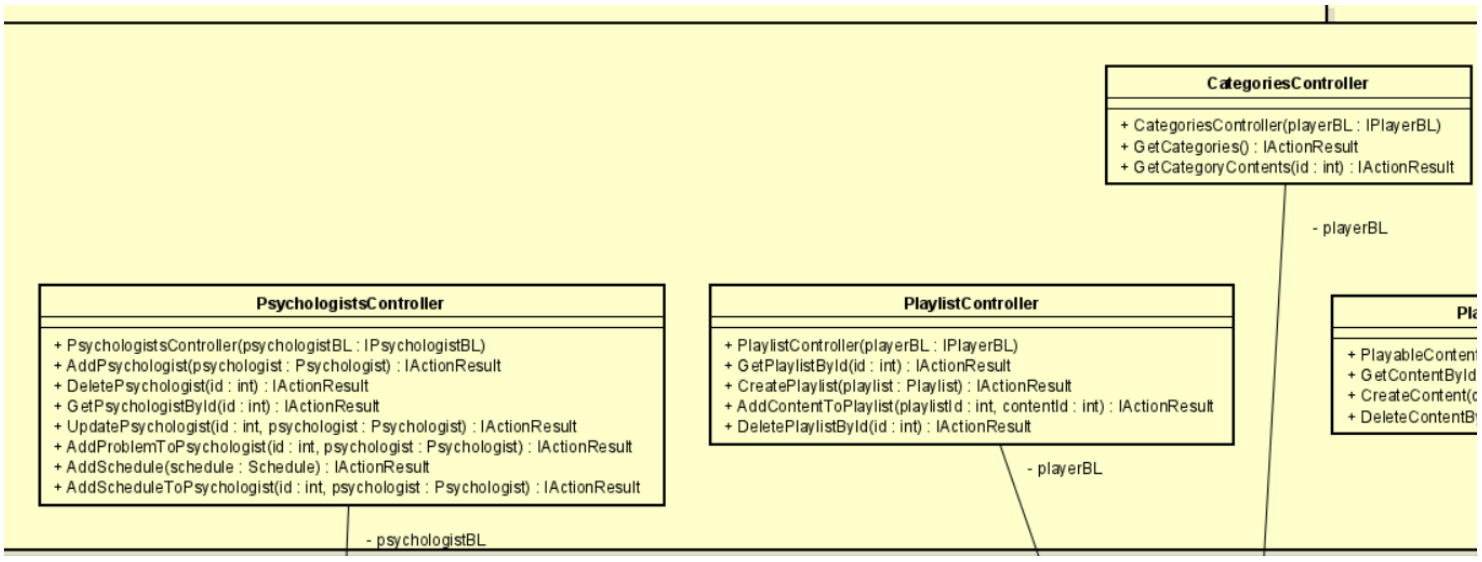
Domain



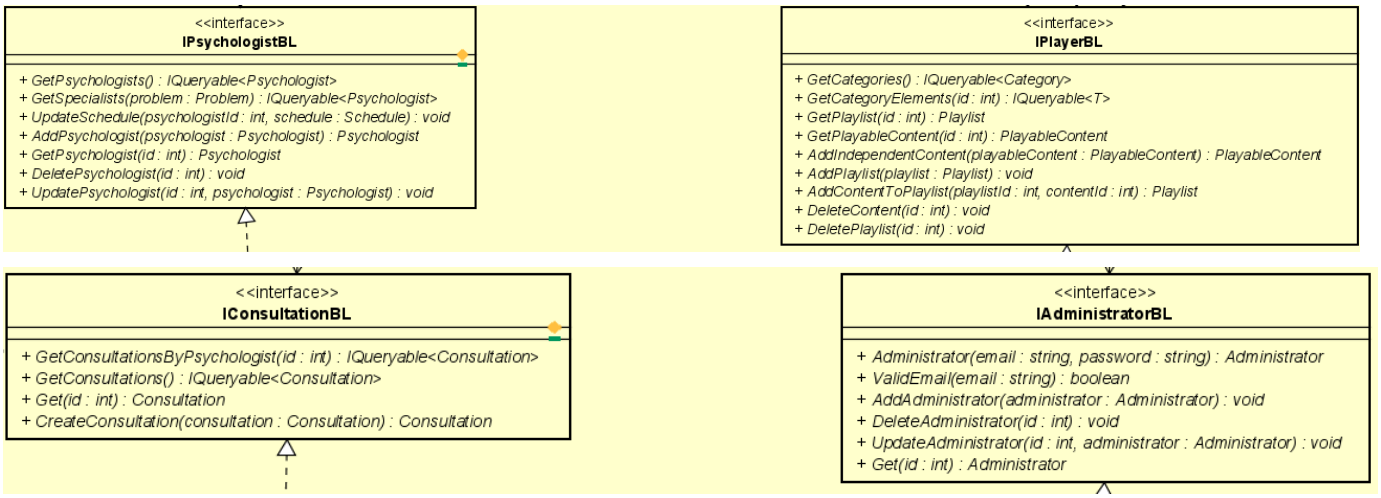


WebAPI.Controllers

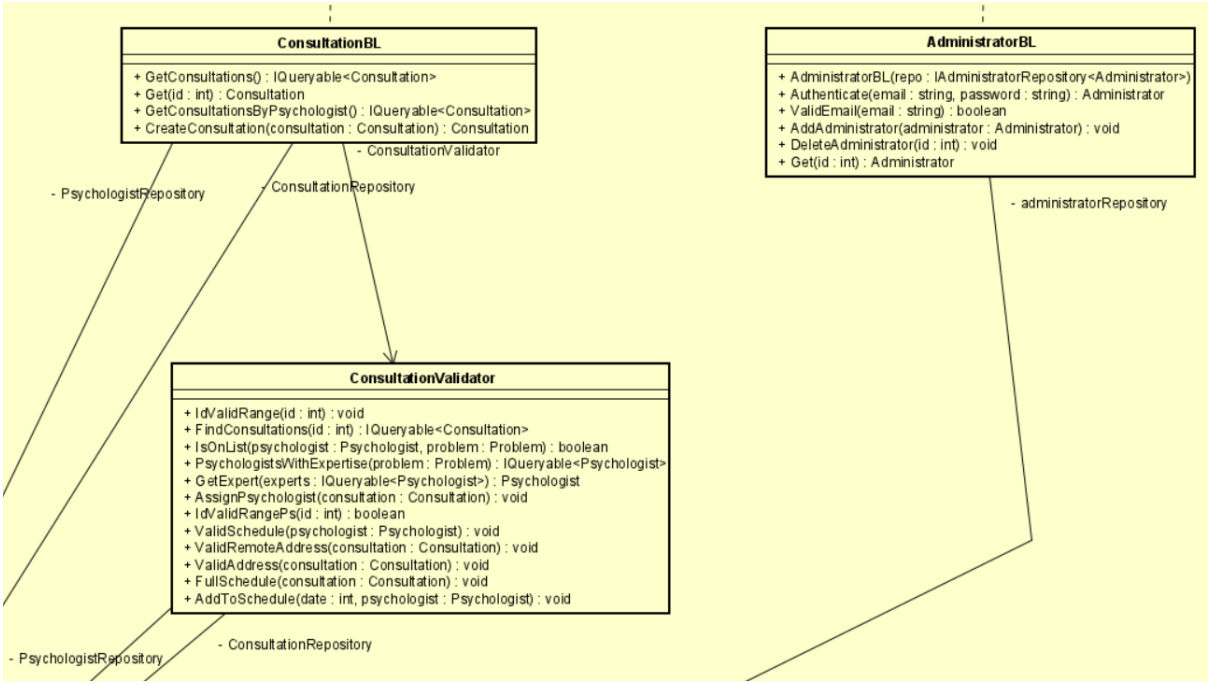


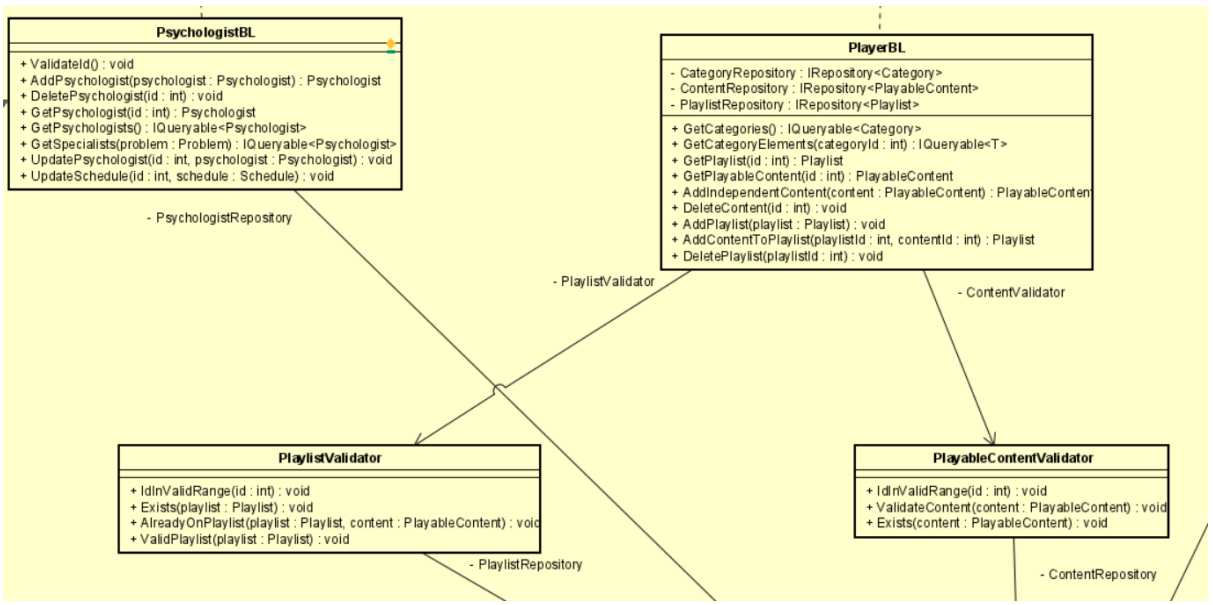


IBusinessLogic

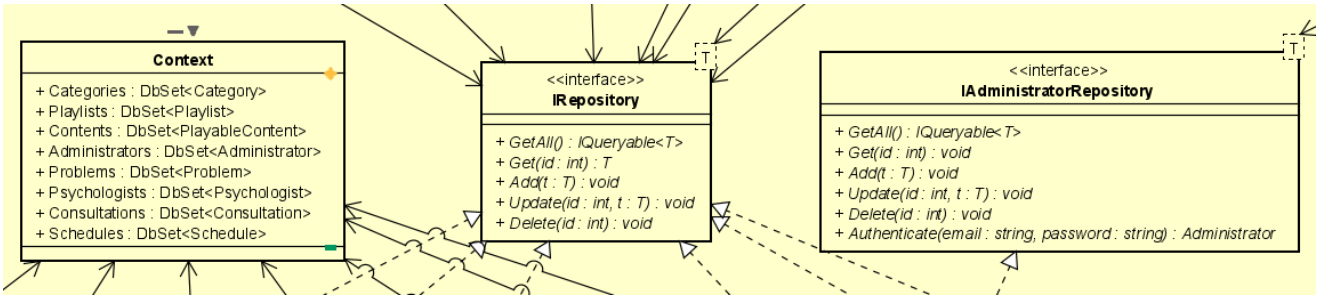


BusinessLogic

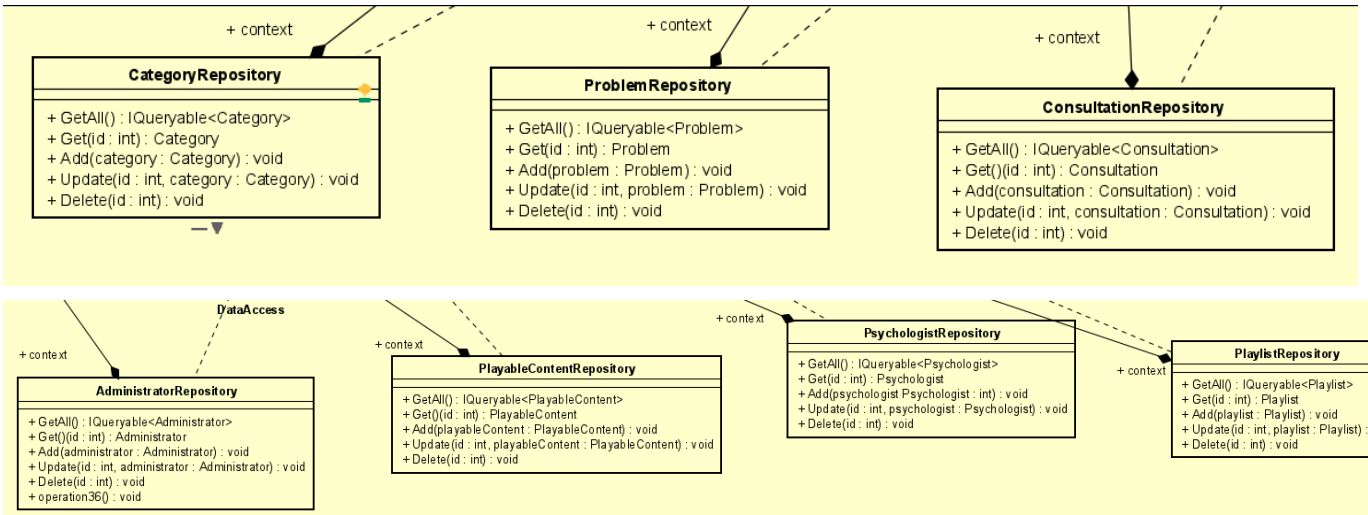




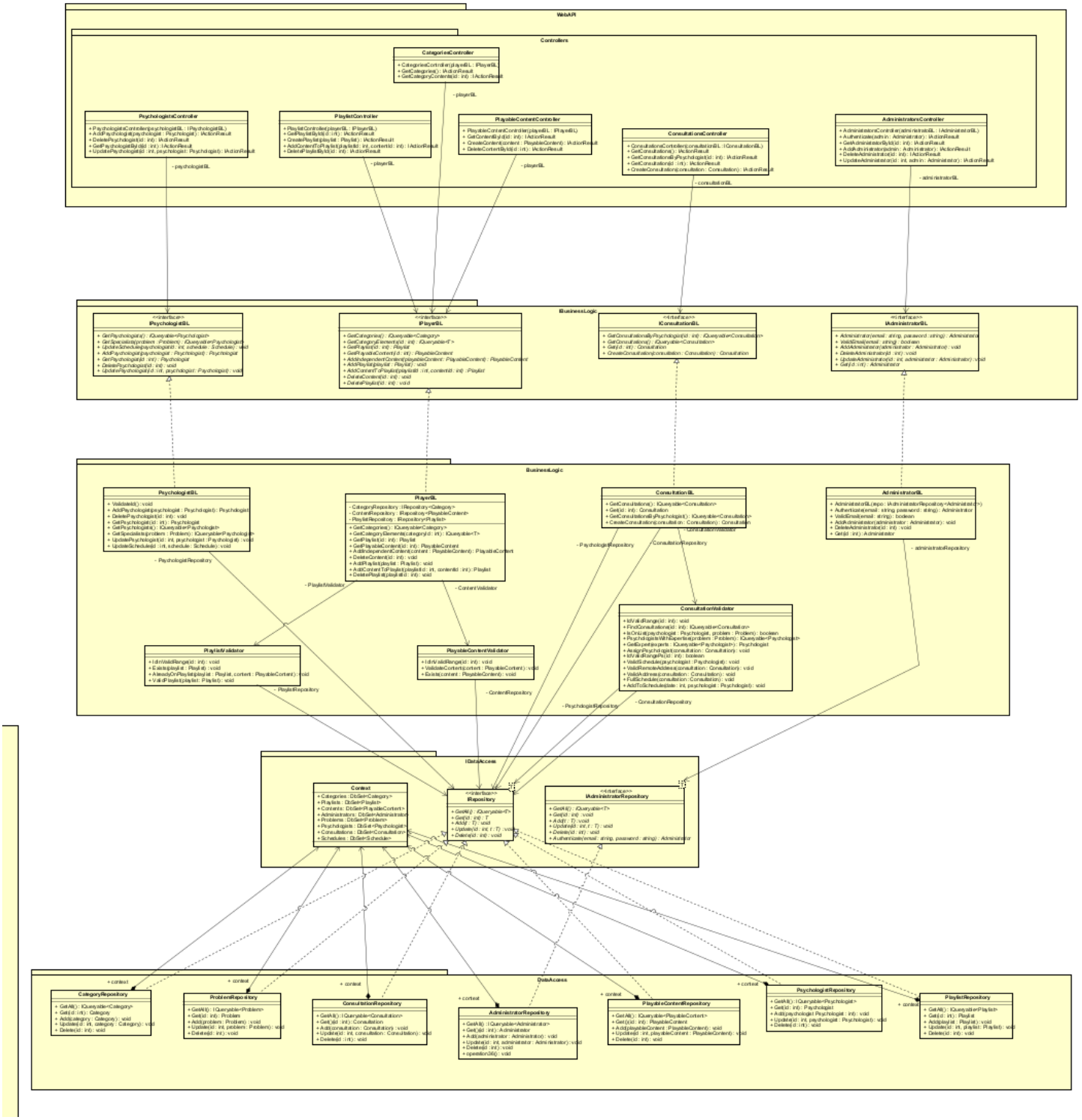
IDataAccess



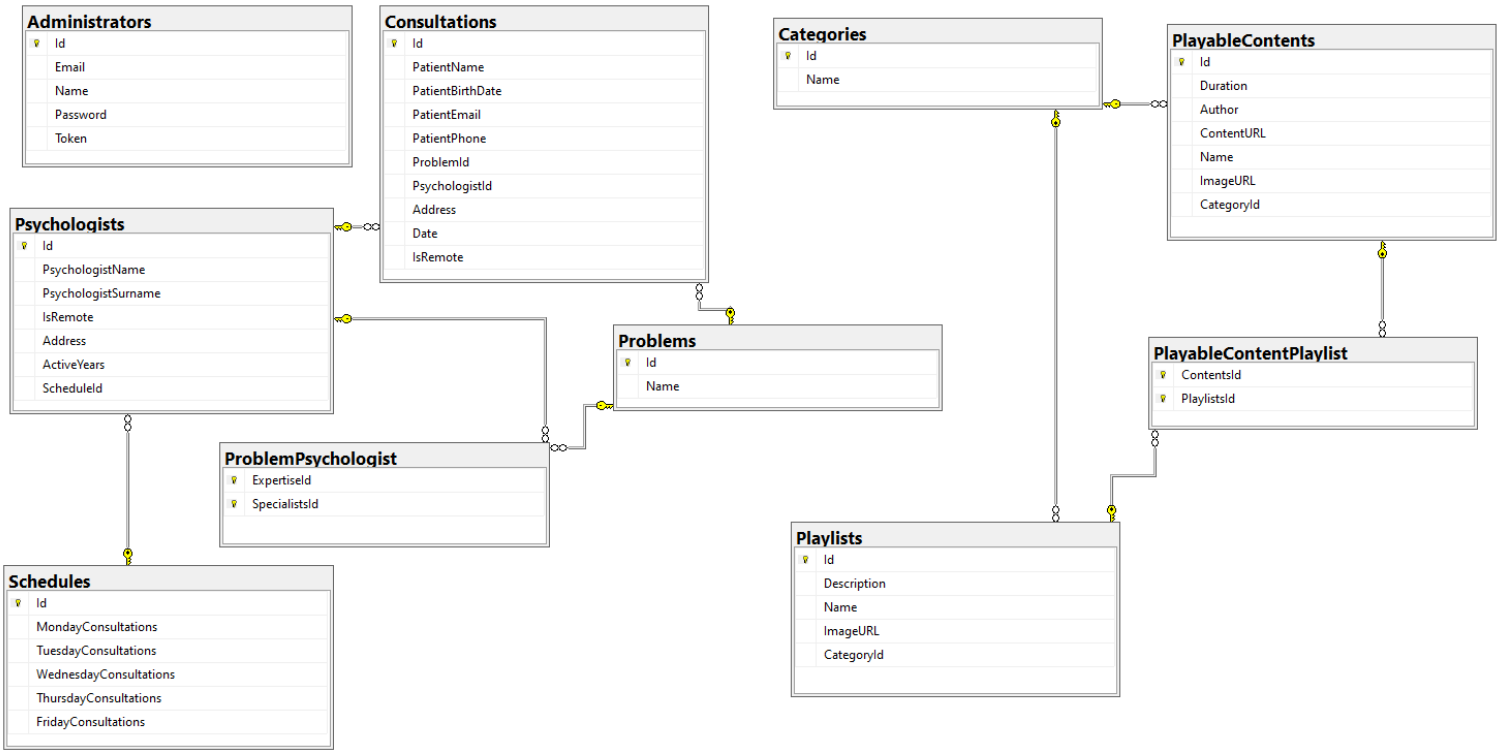
DataAccess



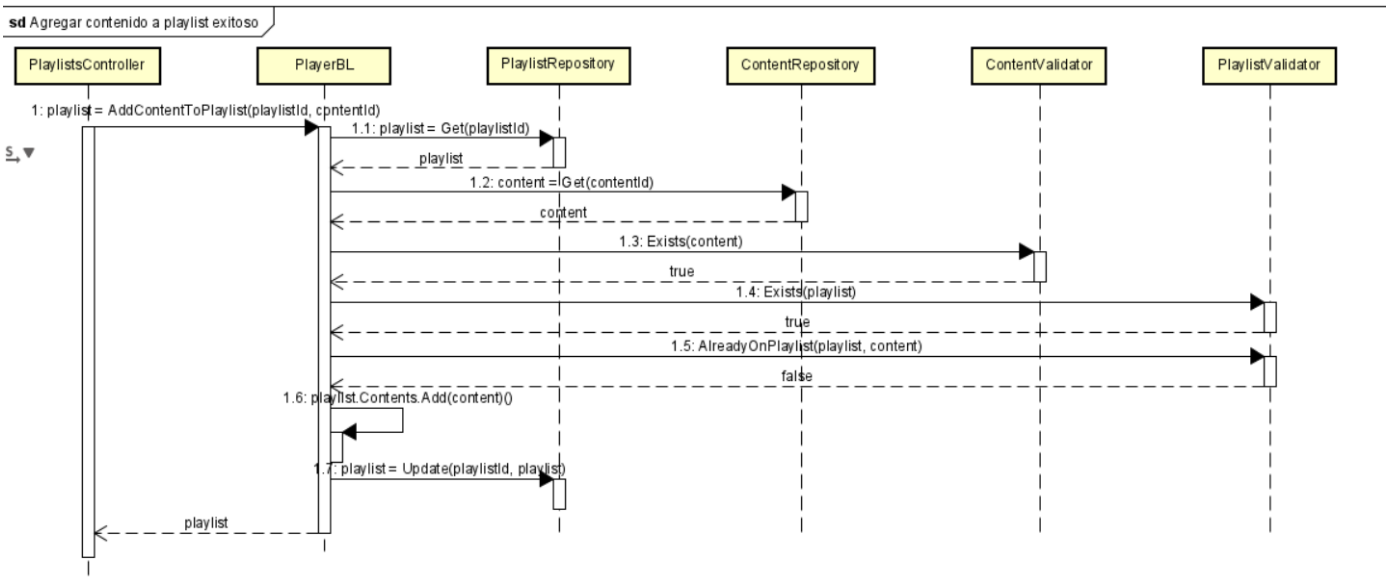
Vista con relaciones

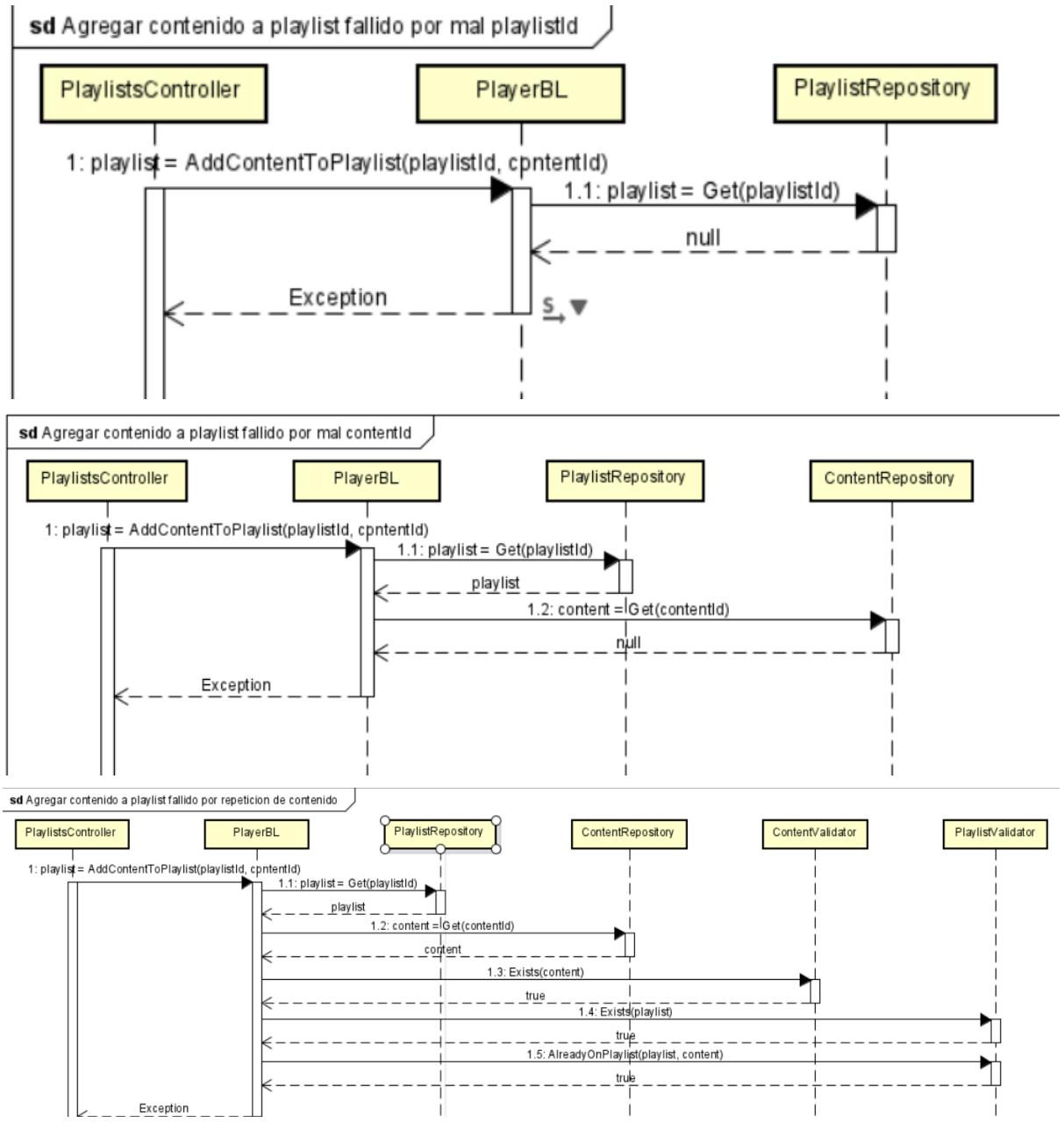


Modelo de tablas de BD



Diagramas de interacción caso agregar contenido a playlist





Justificación del diseño

Uso de patrones y principios de diseño

Dentro de lo que es patrones de diseño, no utilizamos demasiados dado que nos costó un poco identificar con claridad alguno a utilizar en ciertas secciones, y preferimos no forzar su uso.

Dentro de los implementados, hicimos algo similar a Facade por ejemplo, unificando una serie de clases detrás de una misma interfaz en el uso de una interfaz de repositorio

genérica la cual luego implementaban una serie de repositorios de clases concretas. También hicimos uso de inyección de dependencias dentro de la clase Startup en la WebApi, pero eso fue más que una decisión nuestra simplemente un aprovechamiento de una herramienta que el framework nos ofrecía.

A nivel decisiones de diseño en general, tomamos varias en función de qué nos iba a facilitar más el trabajo en partes separadas de la aplicación para poder ir realizando avances en paralelo. Lo primero fue separar el desarrollo en dos aplicaciones distintas a nivel conceptual. Como la parte de reproducción de contenidos y la de agenda de consulta con psicólogos no compartían casi nada sin ser la parte de autenticación, decidimos primero hacer una aplicación completa para luego implementar la otra a mayor velocidad y ya poniendo en práctica los aprendizajes del primer desarrollo.

Además, nos apoyamos mucho en el DIP, para permitirnos trabajar en paralelo de mejor forma. Por ejemplo, nos reuníamos, definíamos la interfaz que fuese a implementar una de las clases de lógica de negocio, y luego uno podía implementar un controller que fuese a hacer uso de una clase que implemente esa interfaz y el otro implementar la clase de lógica que la implemente. Luego de tener probado esto a nivel unitario, podíamos mergear sin conflictos y comenzar a trabajar en pruebas de integración. Ese apego al DIP creo que es evidente hasta en la estructura de paquetes, donde los paquetes o clases concretos dependen siempre de una interfaz.

También nos apoyamos en el SRP, teniendo ejemplos claros en el caso de las clases de lógica de negocio. A medida que las desarrollábamos, nos dimos cuenta que había una serie de funciones auxiliares para validaciones que estábamos utilizando que en realidad no tenía sentido que se encuentren por ejemplo en la misma clase que se encarga de agregar un contenido reproducible al sistema. En esos casos, hicimos una serie de clases con nombre {entity}Validator, cuya responsabilidad era validar por ejemplo los parámetros que se recibían a la hora de agregar un contenido reproducible.

El ISP fue tenido en cuenta pero creemos que es algo que a esta altura tenemos internalizado, es decir, podríamos por ejemplo unificar las interfaces de lógica de negocio bajo una gran interfaz pero no ganamos nada con eso. Por otro lado, al tener distintos controllers dependiendo de distintas interfaces, si un día los requerimientos de uno cambian puede cambiarse esa interfaz también sin interferir con los otros.

Creemos que estas decisiones aumentan la mantenibilidad de la solución, además de otras tomadas. Ejemplos de estas serían el uso de un repositorio genérico en la definición de nuestra interfaz de acceso a datos. Si el día de mañana se quiere agregar otra entidad y arrancar a utilizarla en otras capas, se puede por ejemplo escribir y probar a nivel unitario su lógica de negocio sin necesitar en ningún momento implementar el repositorio concreto, y también llegada la hora implementarlo con facilidad.

Otros atributos de la solución que sin lugar a duda aumentan su mantenibilidad, son el apego a los principios SOLID de diseño, y el grado (95%) de cobertura de pruebas unitarias alcanzado, al igual que el alto grado de cobertura de pruebas de integración. Esto permite embarcarse en un refactor o una extensión de la solución con un grado de seguridad mayor.

Mecanismo de acceso a datos

Para el mecanismo de acceso a datos se utilizó Entity Framework como ORM del lado de la aplicación. Esto nos permitió tomar el approach Code First y definir bien las entidades de dominio y sus relaciones y luego simplemente dejar que EF se encargue de crear y organizar las tablas.

El motor de base de datos utilizado es SQL Server 2017 como se pide en la letra.

A nivel código, utilizamos una interfaz de repositorio genérica y varias concretas que la implementan, de manera de tener una forma estandarizada de acceder a la base de datos.

Manejo de excepciones

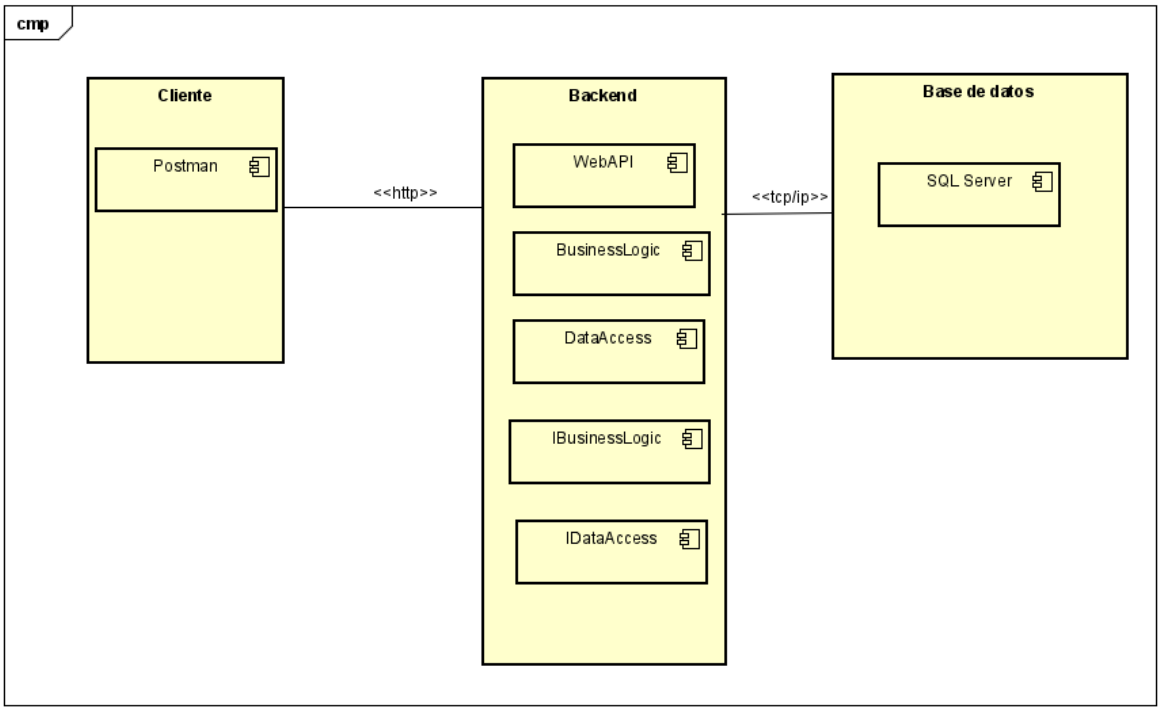
En cuanto al manejo de excepciones, no incluimos nada demasiado complejo en nuestra solución (como creación de clases de excepciones propias), pero aún así logramos un manejo satisfactorio.

Nuestro enfoque a la hora de manejar excepciones fue intentar que haya la menor cantidad posible de excepciones sin controlar y que ocurran al nivel más alto posible.
Es decir, hay casos típicos de excepciones que se podrían manejar en cualquiera de los niveles de la aplicación, pero que intencionalmente manejamos al nivel más alto posible.

Un ejemplo sería si se quisiese dar de alta una entidad con un identificador hardcodeado duplicado, es decir, con un identificador brindado por el usuario y ya existente en el sistema. Podríamos dejar que la excepción la levante entity framework al intentar de agregar al usuario a nuestro contexto, pero preferimos no desperdiciar tiempo del usuario ni cómputo de nuestro servidor, y realizar ese chequeo a un nivel más alto como por ejemplo en la capa de lógica de negocio, y de ser necesario elevar la excepción. De todas formas, todas las operaciones que puedan llegar a generar excepciones están envueltas en un try catch.

Otro ejemplo de prevención de excepciones del estilo es cuando un usuario intenta acceder por id a alguna entidad en el sistema. lo que hacemos en esos casos es asegurarnos antes de ir contra la base de datos que ese identificador esté dentro del rango que maneja en el momento la BD, de forma de prevenir nuevamente una excepción de ese tipo, y poder brindar al usuario un mensaje de error más claro (no hay una entidad asociada al identificador provisto) que simplemente un error de SQL.

Diagrama de componentes



Es interesante cómo estos componentes pueden residir físicamente en el mismo equipo o en distintos, facilitado por los mecanismos de comunicación que utilizan. Idealmente en un ambiente productivo, puede ser más seguro separar el servidor web del servidor que contenga la base de datos. Luego el cliente que en este caso es postman, puede estar en cualquier servidor siempre y cuando pueda acceder vía http al servidor web.