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.

Construimos también un frontend en Angular cuyo objetivo es permitir a los usuarios acceder a nuestra aplicación sin tener que depender de un cliente HTTP, utilizando una SPA.

La solución fue desarrollada utilizando el framework WebApi de .NET core, utilizando C# y SQL Server 2017 como motor de base de datos. Para el frontend se usó Typescript y Angular.

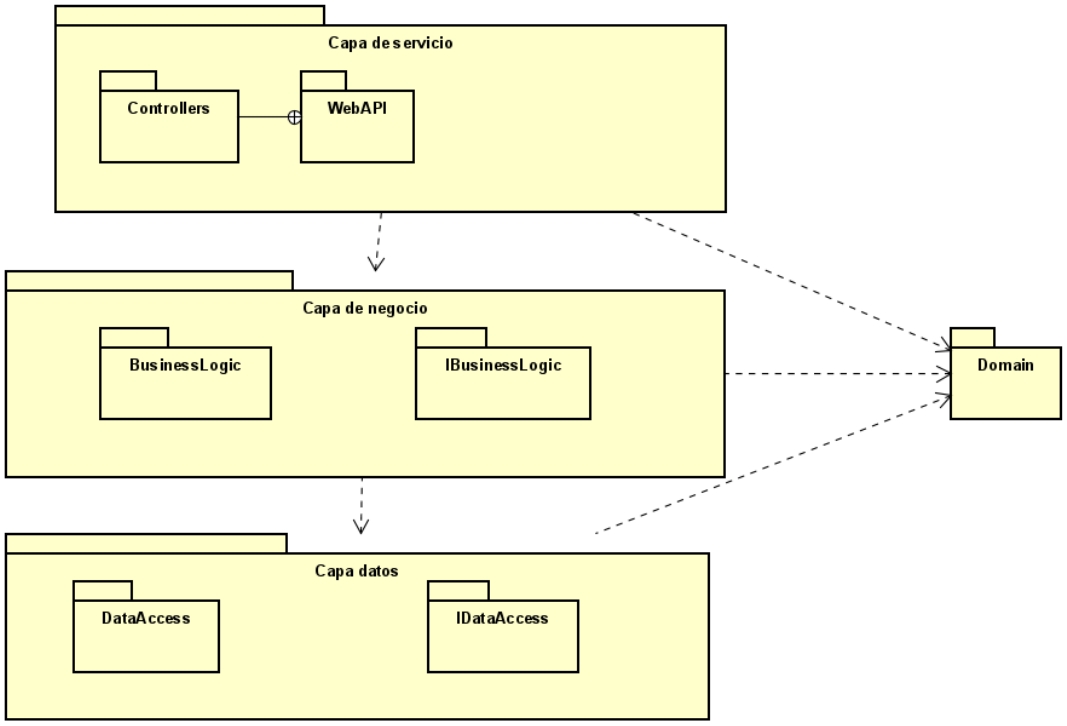
En esta iteración los usuarios podrían interactuar con la aplicación tanto mediante un cliente HTTP como podría ser Postman o Curl, como mediante el frontend en Angular.

Errores conocidos

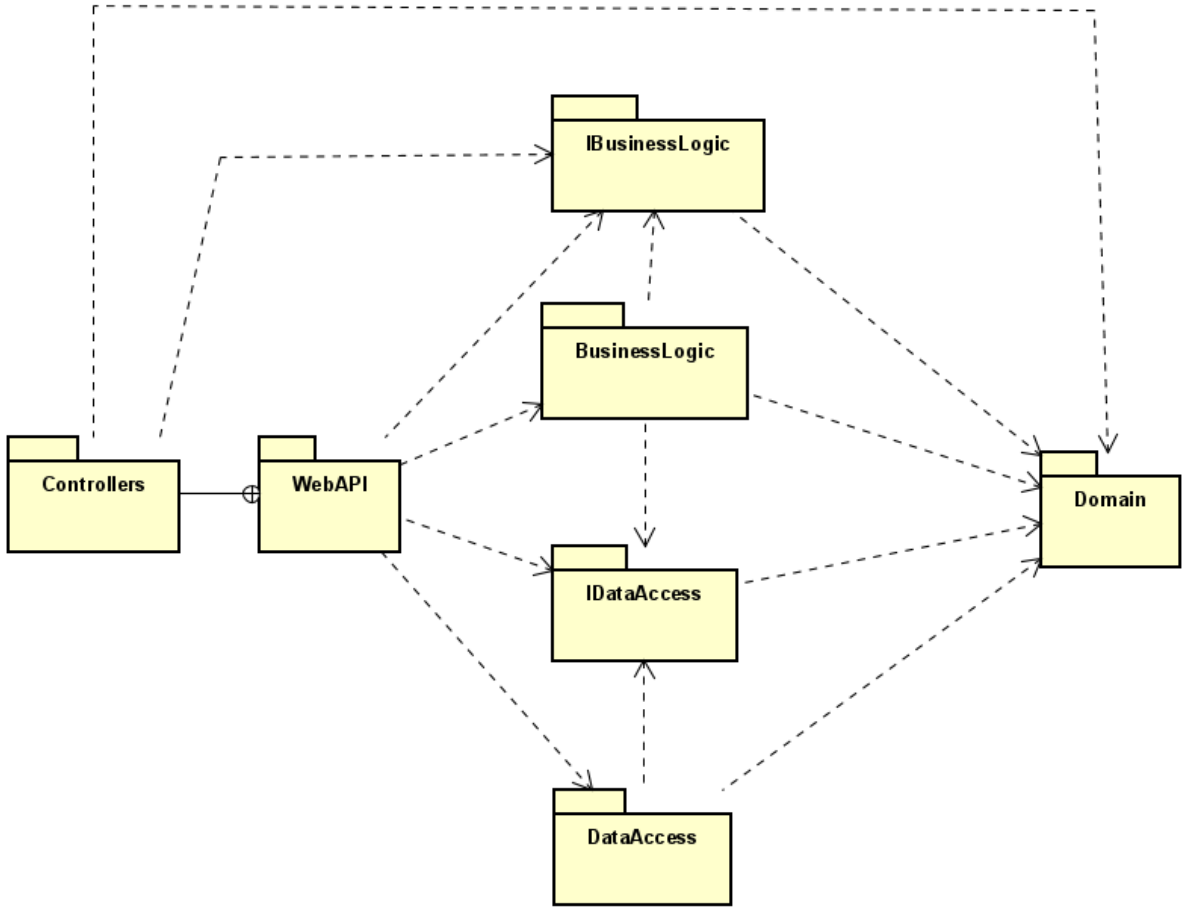
En cuanto a errores conocidos, no se logró culminar la integración del frontend con la funcionalidad de importación, es decir, no quedó funcional vía Angular. No se pudo encontrar la razón del mismo y por temas de tiempo se tuvo que dejar de esta forma.

Diagramas de paquetes

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

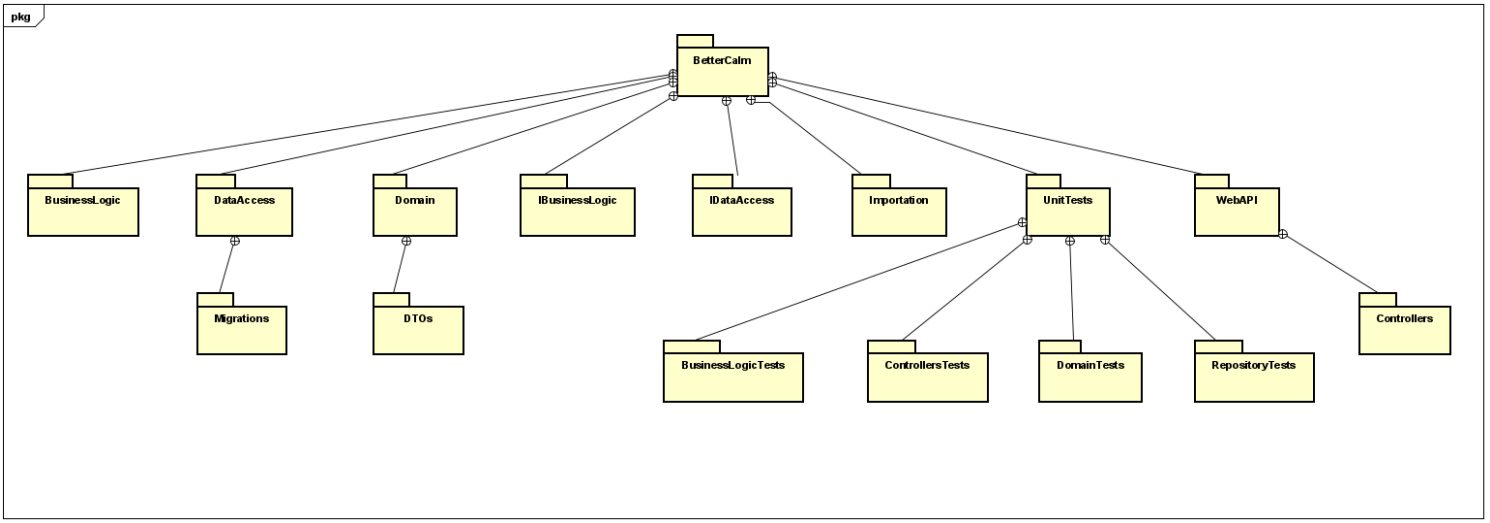


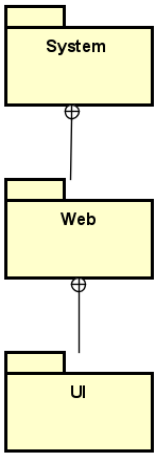
Luego un diagrama de paquetes representando de manera explícita las dependencias entre ellos:



Descripción de responsabilidades de los paquetes

En el siguiente diagrama de descomposición de paquetes se puede observar la distribución general de responsabilidades.

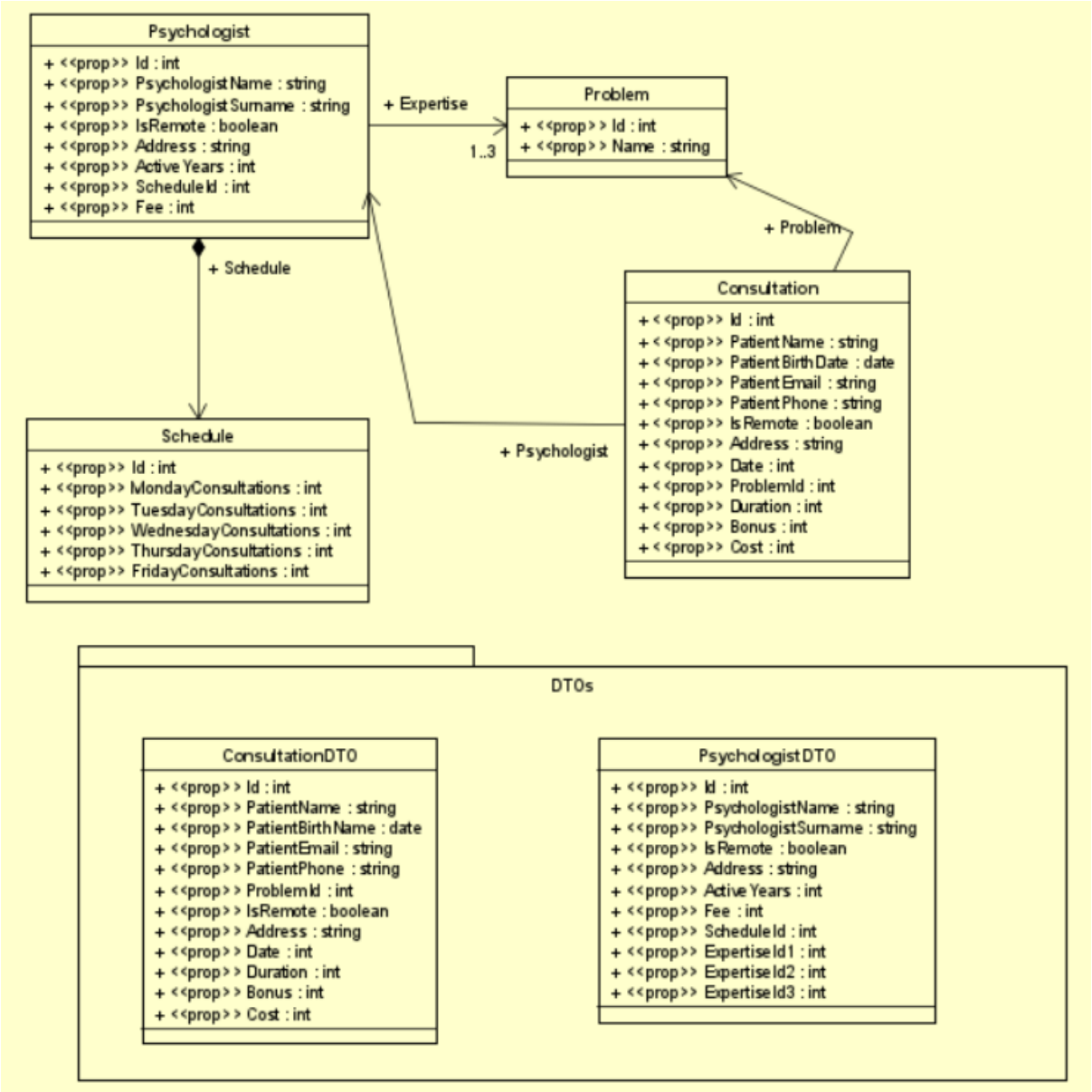




Domain

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.

Aquí, a diferencia de la entrega anterior, se puede observar una nueva entidad denominada VideoContent. Esta fue utilizada para la implementación del nuevo tipo de contenido, video. A su vez, se creó un paquete DTOs en el cual se almacenaron todos los DTO utilizados en el proyecto, en nuestro caso, ConsultationDTO y PsychologistDTO.



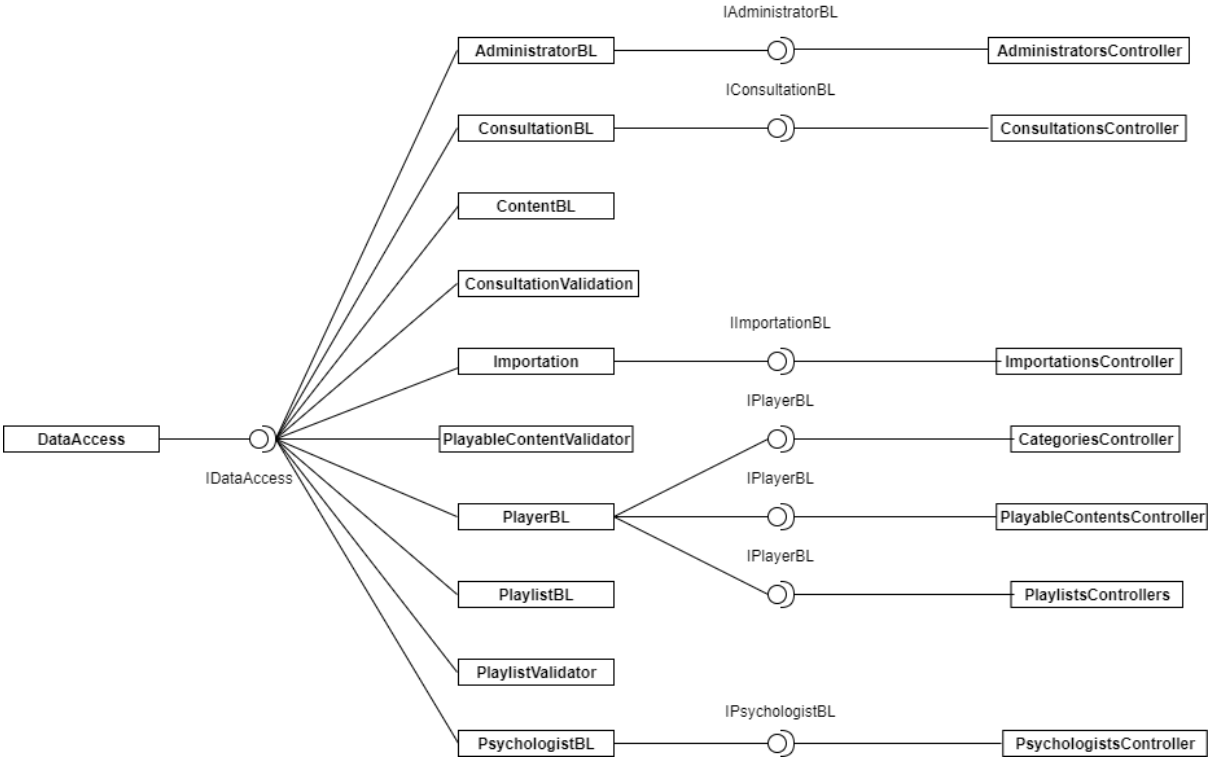
DataAccess

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.

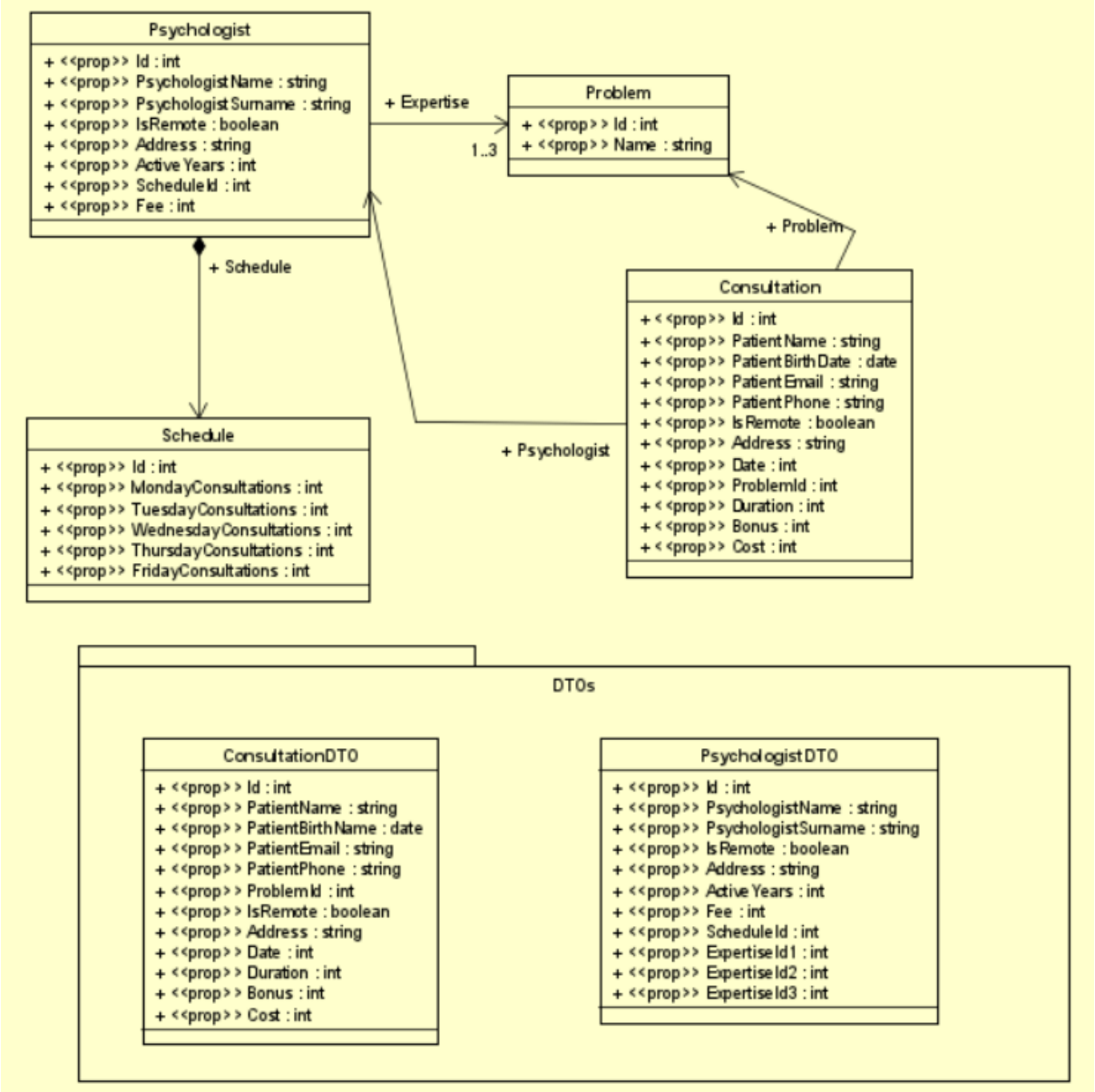
Se puede ver que este paquete depende tanto de paquetes de interfaz como de paquetes de implementación, y esto tiene una justificación válida: inyección de dependencias. En la

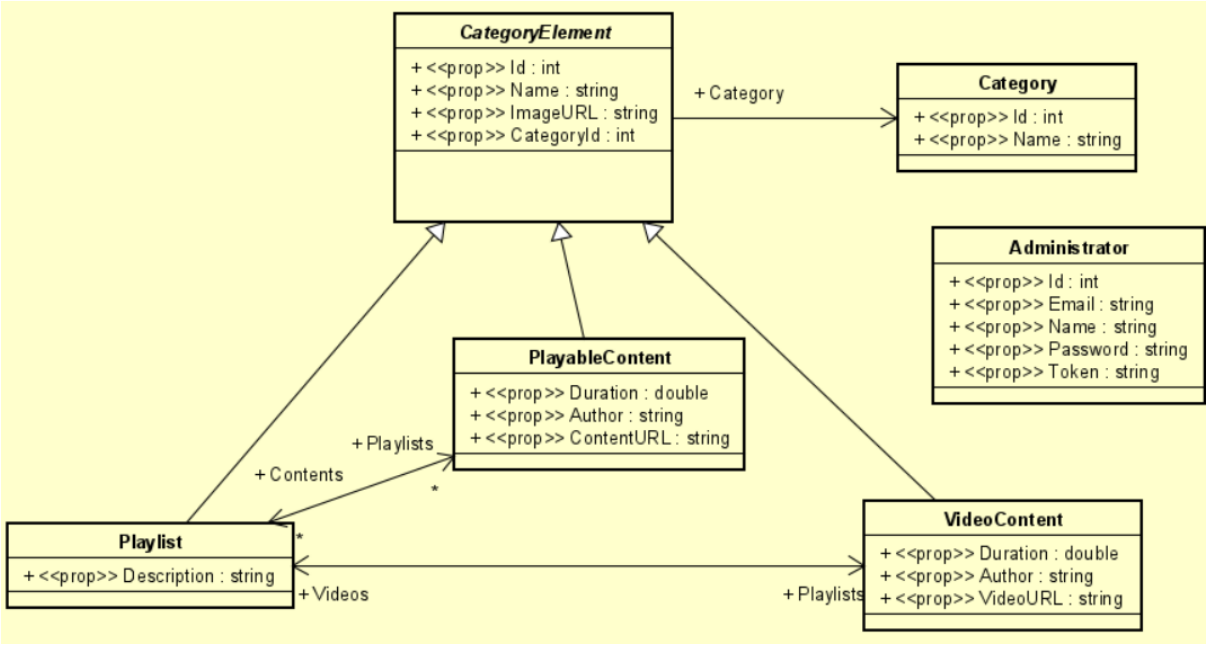
clase Startup.cs, además de configuración de servicios se va a realizar inyección de dependencias utilizando interfaces y clases concretas.

Diagramas de clase por paquetes

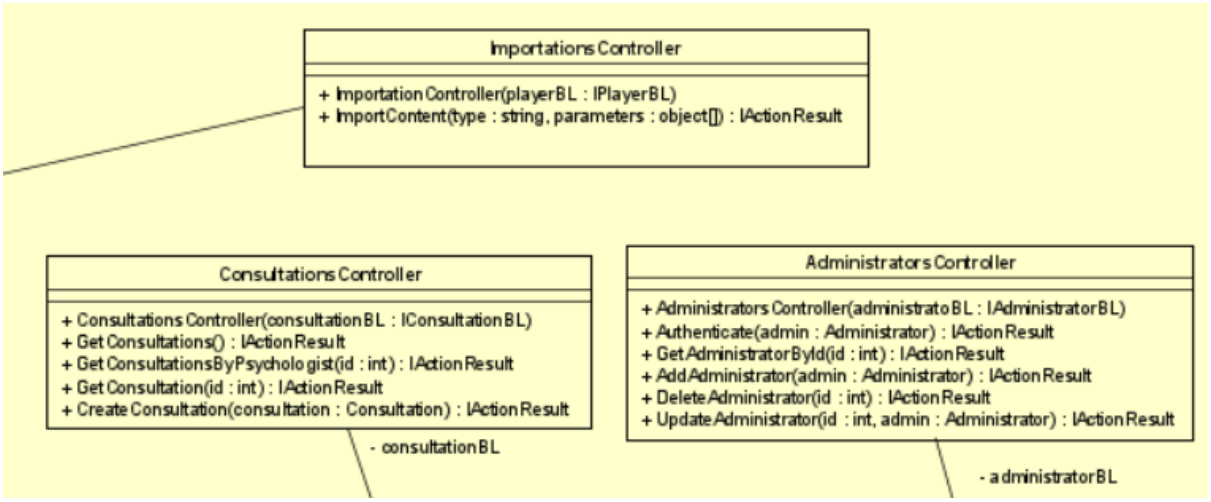
La complejidad de la solución y el formato del documento no colaboran con esta sección por lo cual además de dejar capturas aquí vamos a adjuntar el diagrama en formato .asta que puede ser abierto usando Astah.

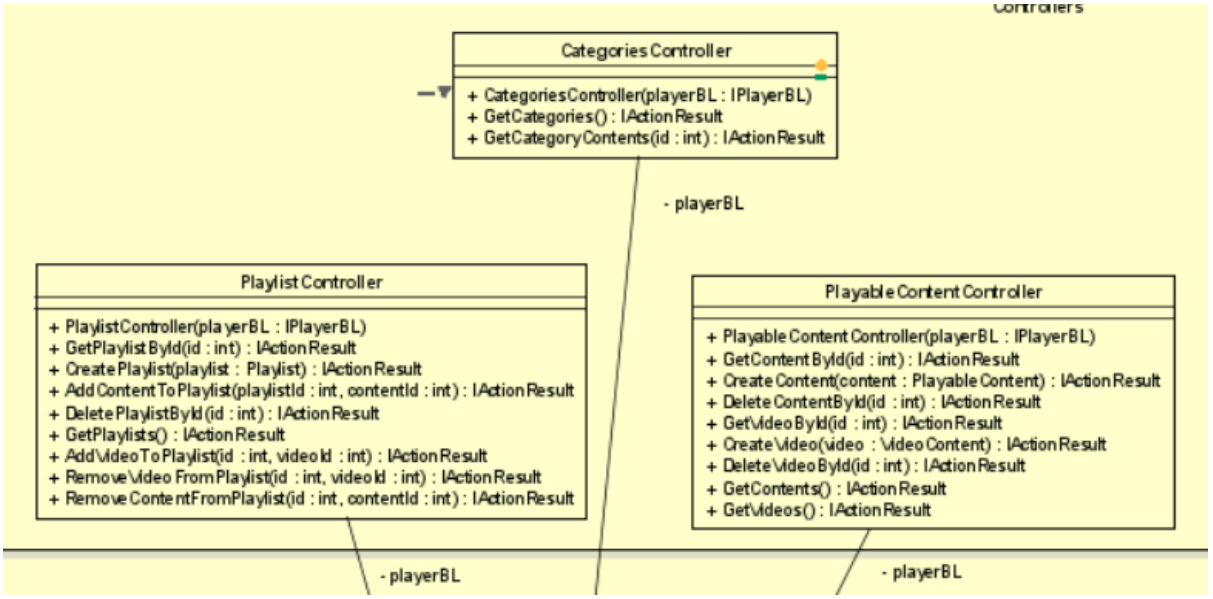
Domain

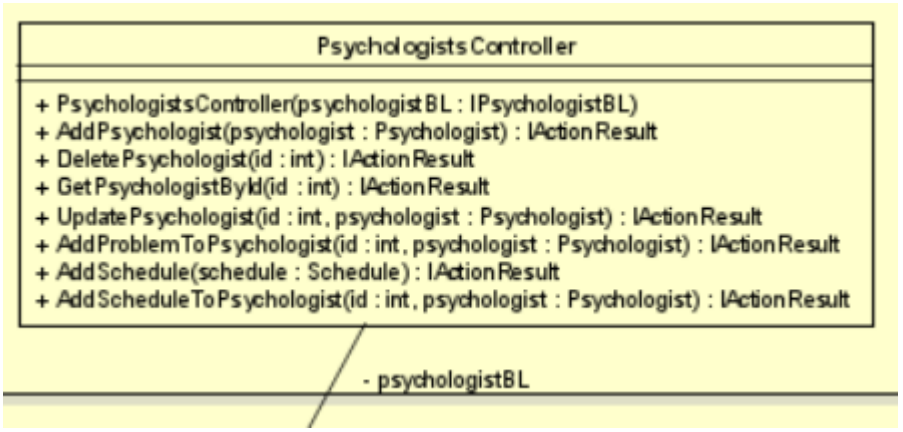




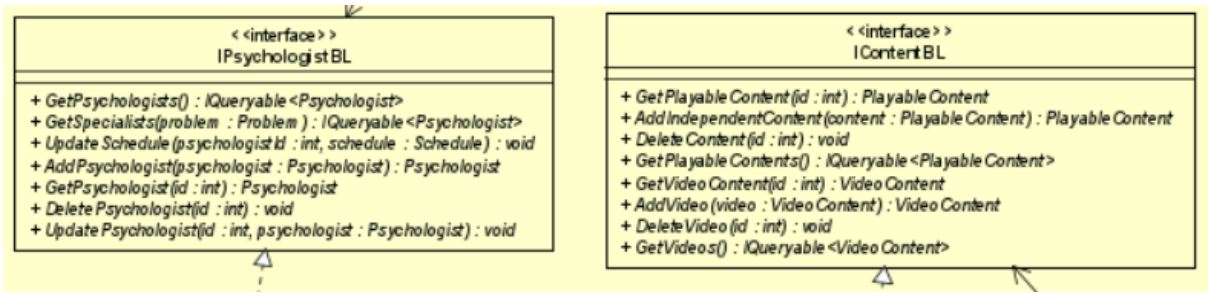
WebAPI.Controllers

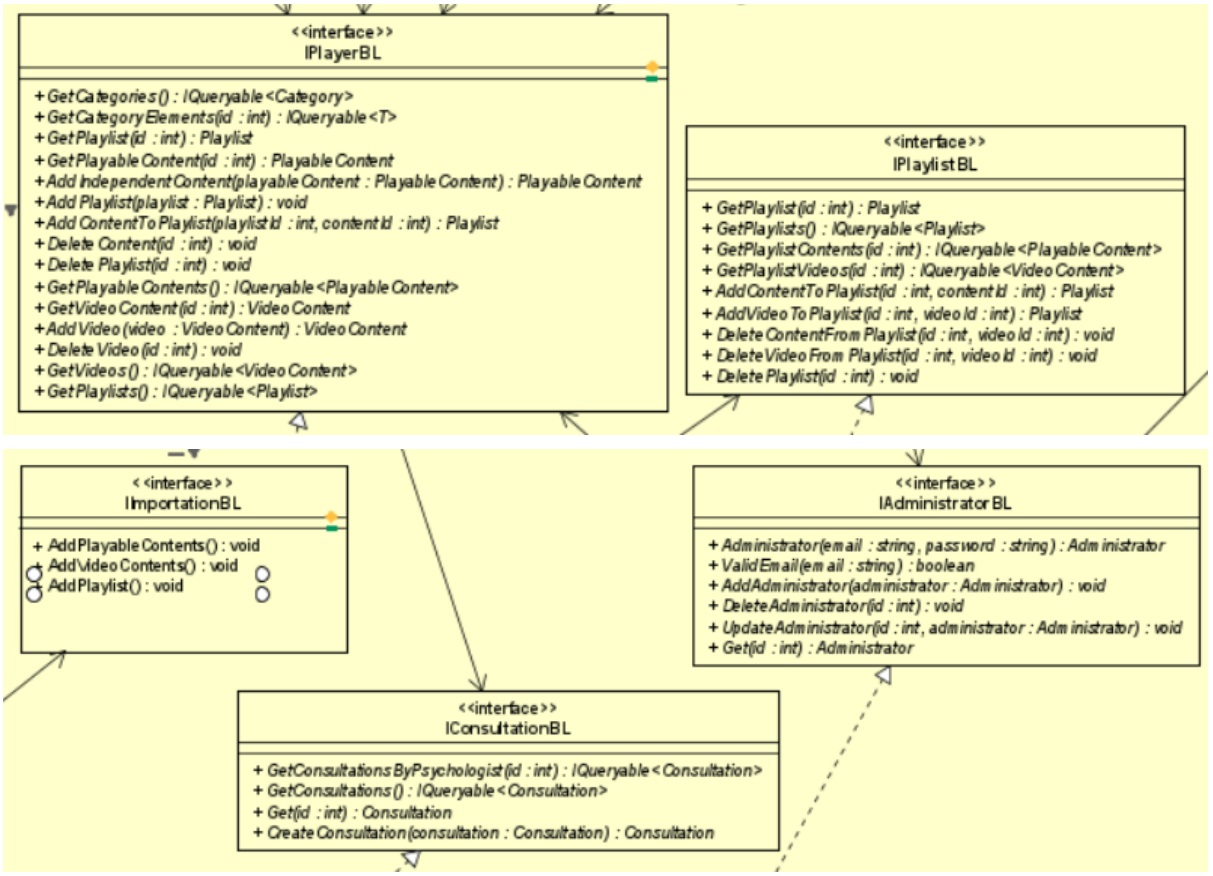




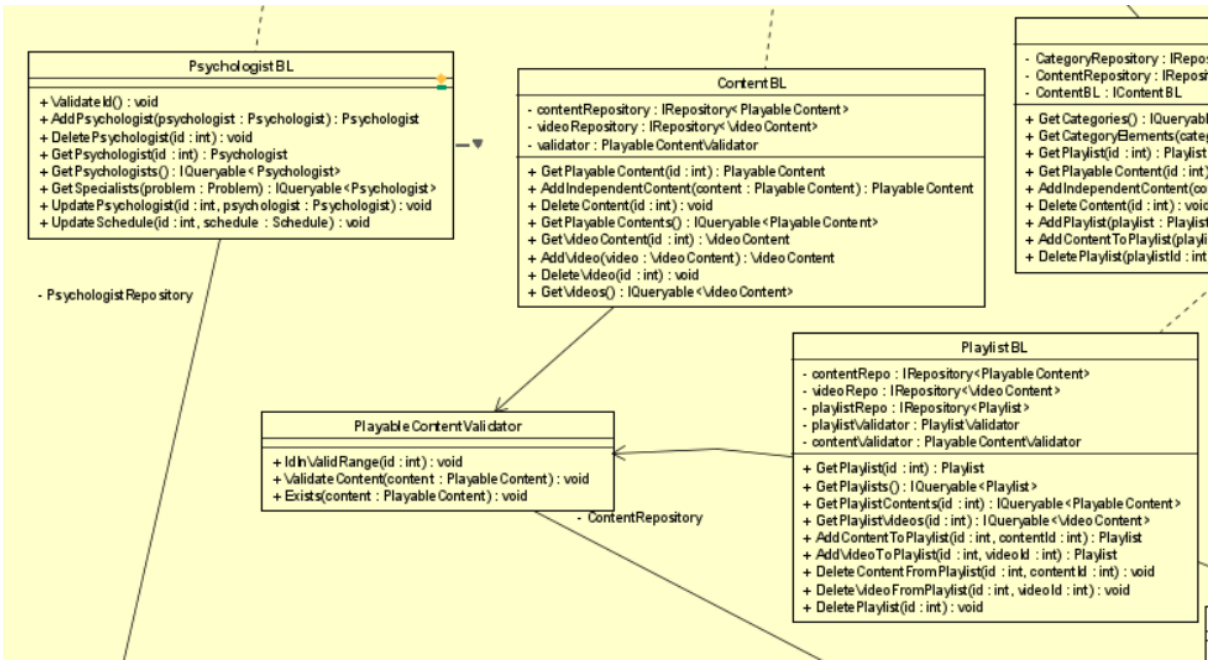


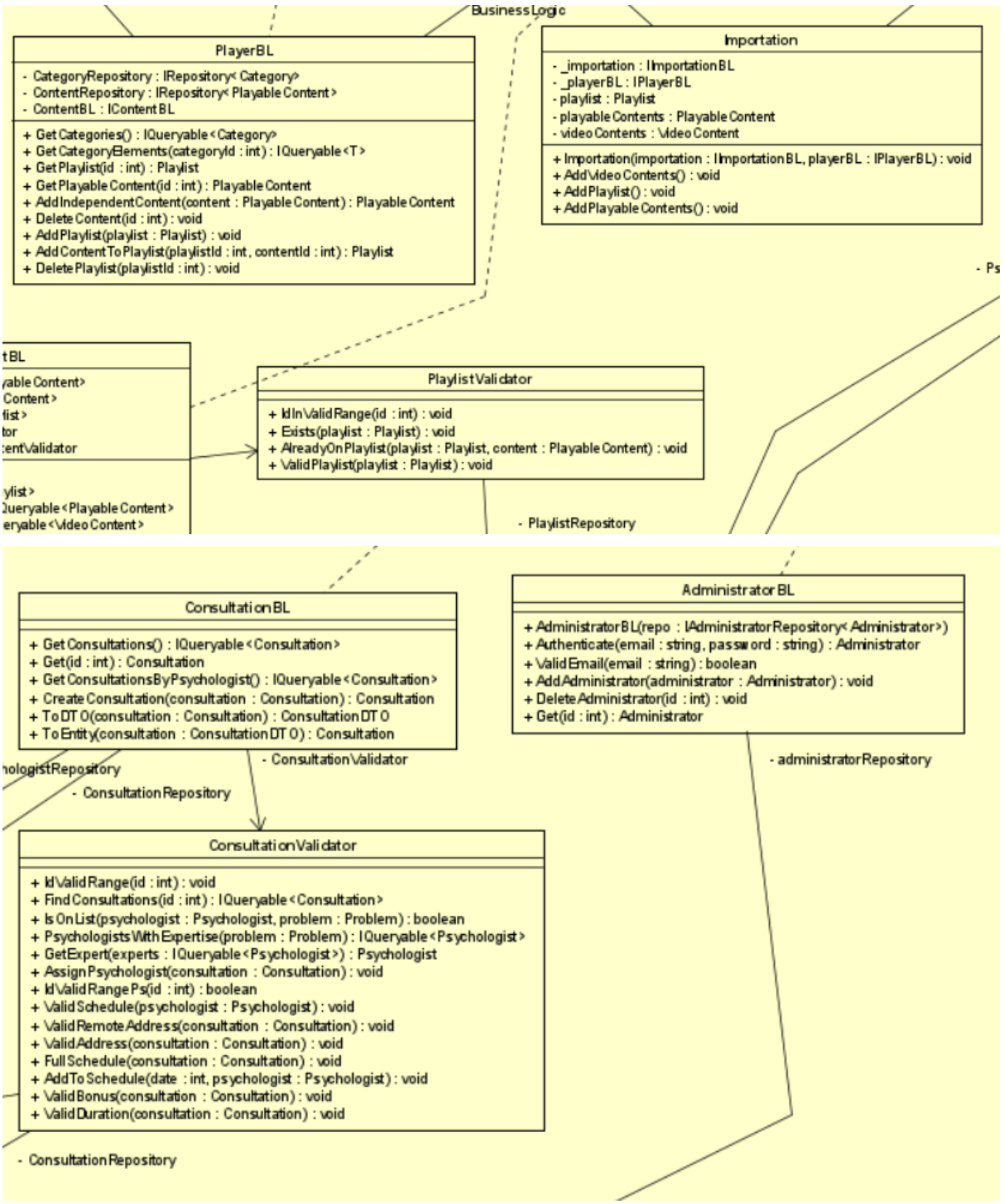
IBusinessLogic



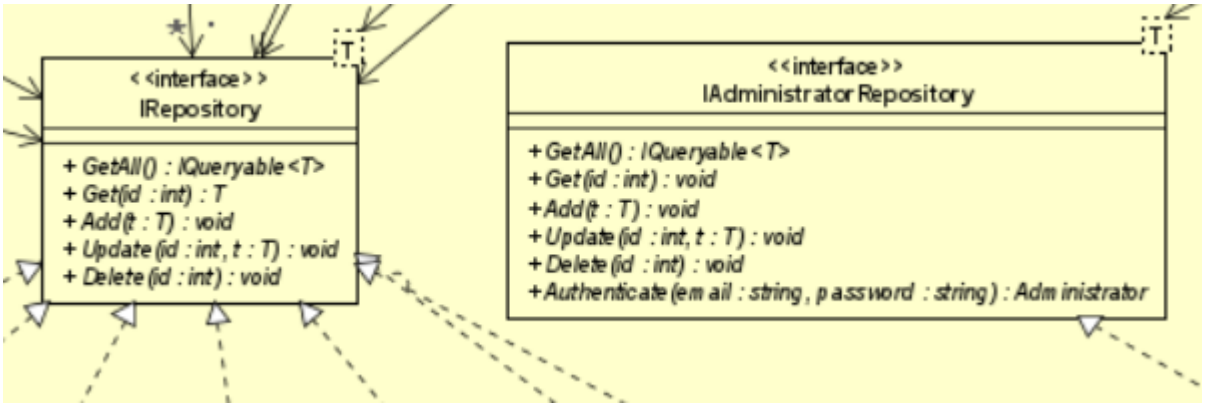


BusinessLogic

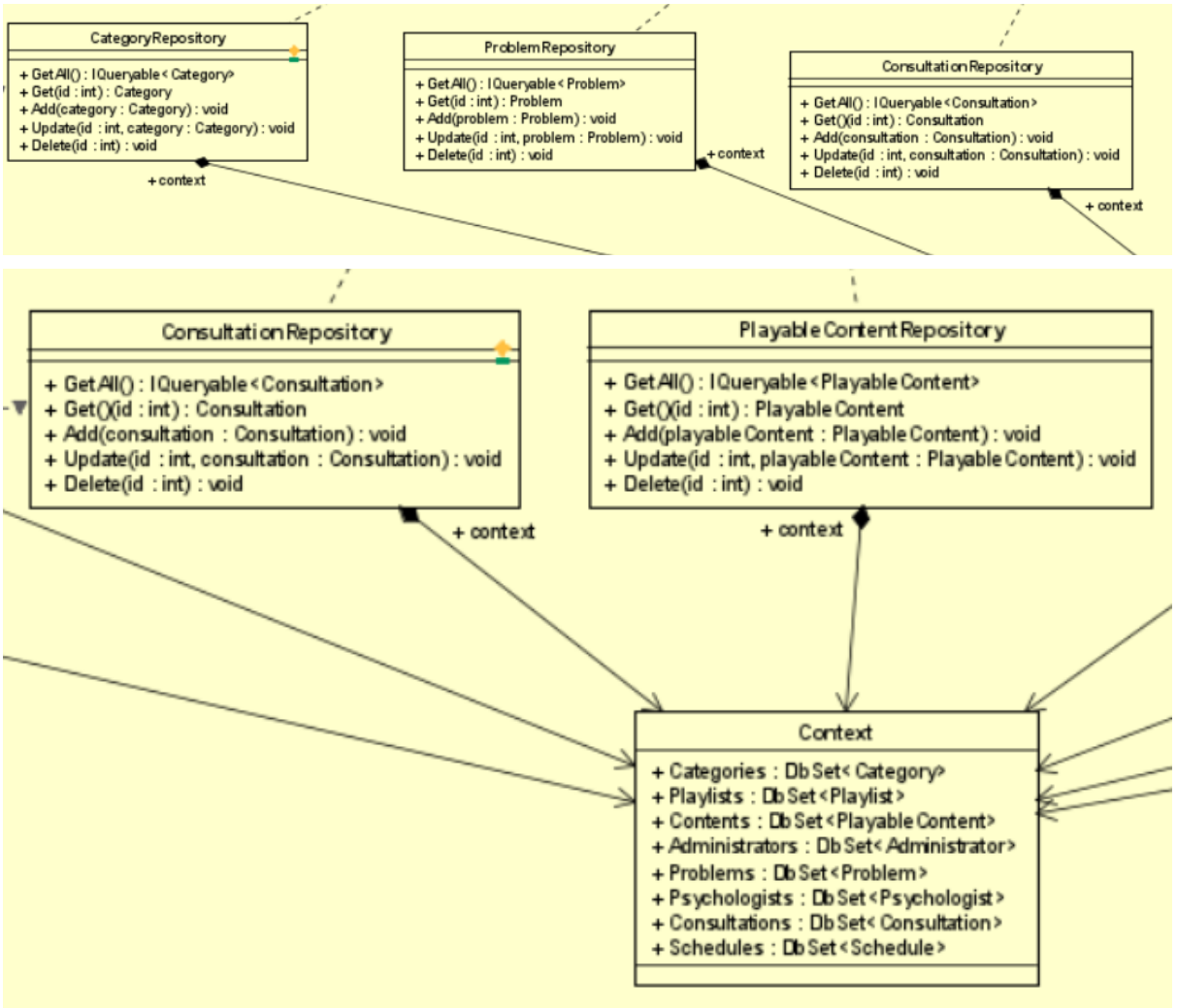


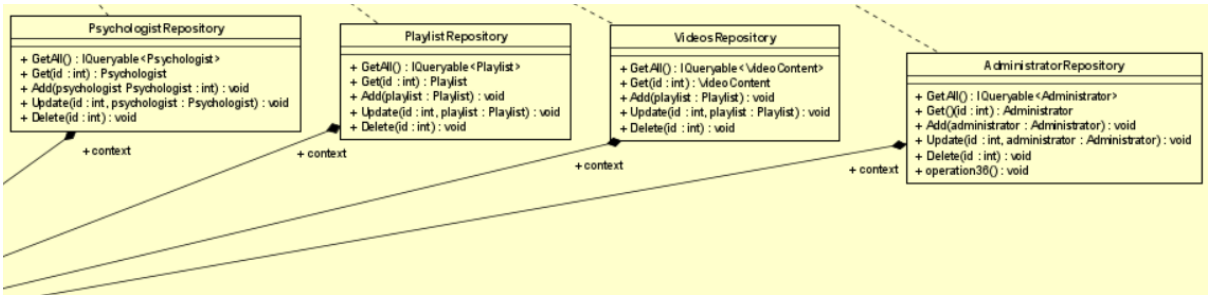

Business Logic
PlayerBL
- CategoryRepository : IRepository< Category>
- ContentRepository : IRepository< Playable Content >
- ContentBL : IContent BL
+ Get Categories() : IQueryable < Category>
+ Get CategoryElements(categoryId : int) : IQueryable <T >
+ Get Playlist(id : int) : Playlist
+ Get Playable Content(id : int) : Playable Content
+ Add Independent Content(content : Playable Content) : Playable Content
+ Delete Content(id : int) : void
+ Add Playlist(playlist : Playlist) : void
+ Add Content To Playlist(playlistId : int, contentId : int) : Playlist
+ Delete Playlist(playlistId : int) : void
Importation
- _importation : IImportation BL
- _playerBL : IPlayer BL
- playlist : Playlist
- playable Contents : Playable Content
- video Contents : Video Content
+ Importation(importation : IImportation BL, playerBL : IPlayer BL) : void
+ Add Video Contents() : void
+ Add Playlist() : void
+ Add Playable Contents() : void
PlaylistValidator
+ IdIn\Valid Range(id : int) : void
+ Exists(playlist : Playlist) : void
+ Already On Playlist(playlist : Playlist, content : Playable Content) : void
+ Valid Playlist(playlist : Playlist) : void
- PlaylistRepository
ConsultationBL
+ Get Consultations() : IQueryable < Consultation >
+ Get(id : int) : Consultation
+ Get Consultations By Psychologist() : IQueryable < Consultation >
+ Create Consultation(consultation : Consultation) : Consultation
+ To DT O(consultation : Consultation) : Consultation DT O
+ To Entity(consultation : Consultation DT O) : Consultation
- Consultation Validator
- Consultation Repository
Administrator BL
+ Administrator BL(repo : IAdministrator Repository< Administrator>)
+ Authenticate(email : string, password : string) : Administrator
+ Valid Email(email : string) : boolean
+ Add Administrator(administrator : Administrator) : void
+ Delete Administrator(id : int) : void
+ Get(id : int) : Administrator
- administrator Repository
Consultation Validator
+ Id\Valid Range(id : int) : void
+ Find Consultations(id : int) : IQueryable < Consultation >
+ Is On List(psychologist : Psychologist, problem : Problem) : boolean
+ Psychologists With Expertise(problem : Problem) : IQueryable < Psychologist >
+ Get Expert(experts : IQueryable < Psychologist >) : Psychologist
+ Assign Psychologist(consultation : Consultation) : void
+ Id\Valid Range Ps(id : int) : boolean
+ Valid Schedule(psychologist : Psychologist) : void
+ Valid Remote Address(consultation : Consultation) : void
+ Valid Address(consultation : Consultation) : void
+ Full Schedule(consultation : Consultation) : void
+ Add To Schedule(date : int, psychologist : Psychologist) : void
+ Valid Bonus(consultation : Consultation) : void
+ Valid Duration(consultation : Consultation) : void
- Consultation Repository

IDataAccess

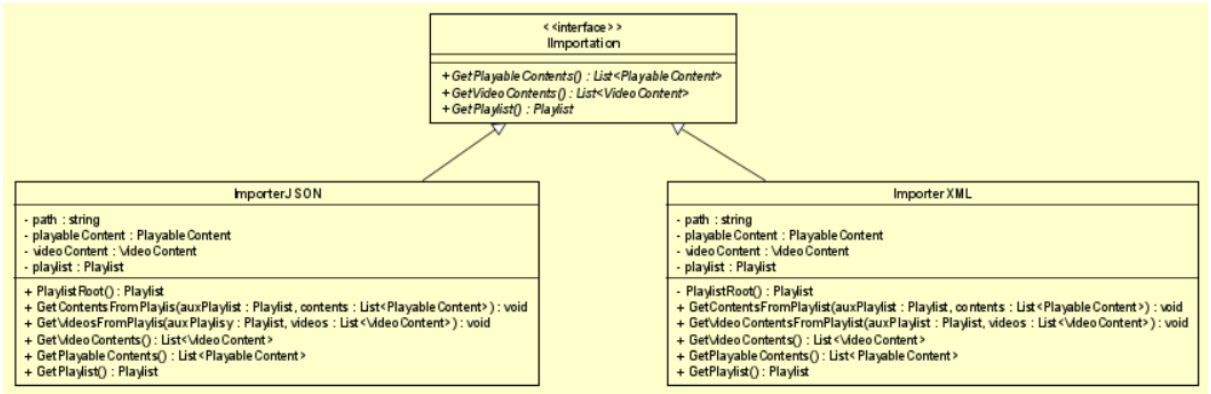


DataAccess

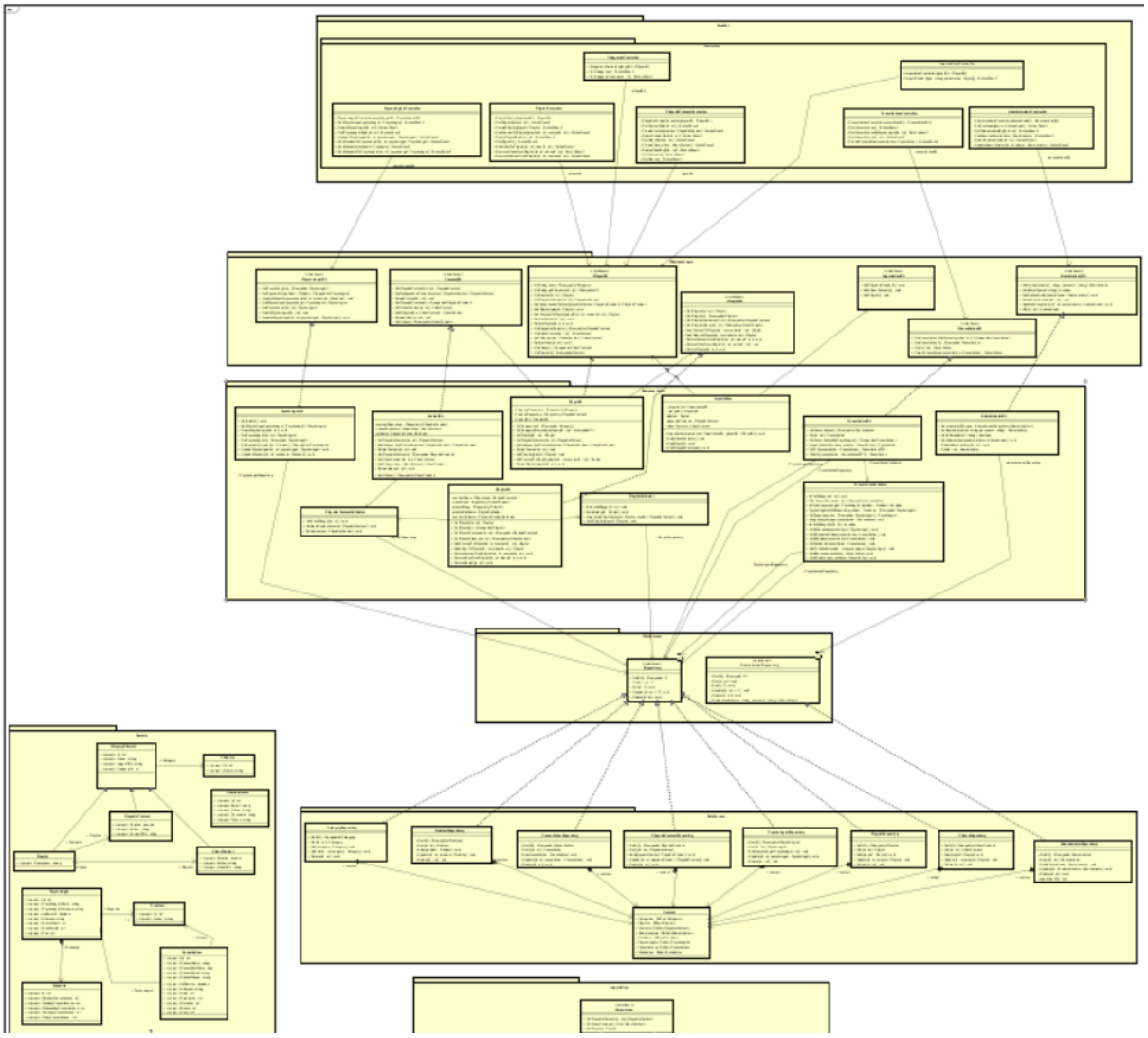




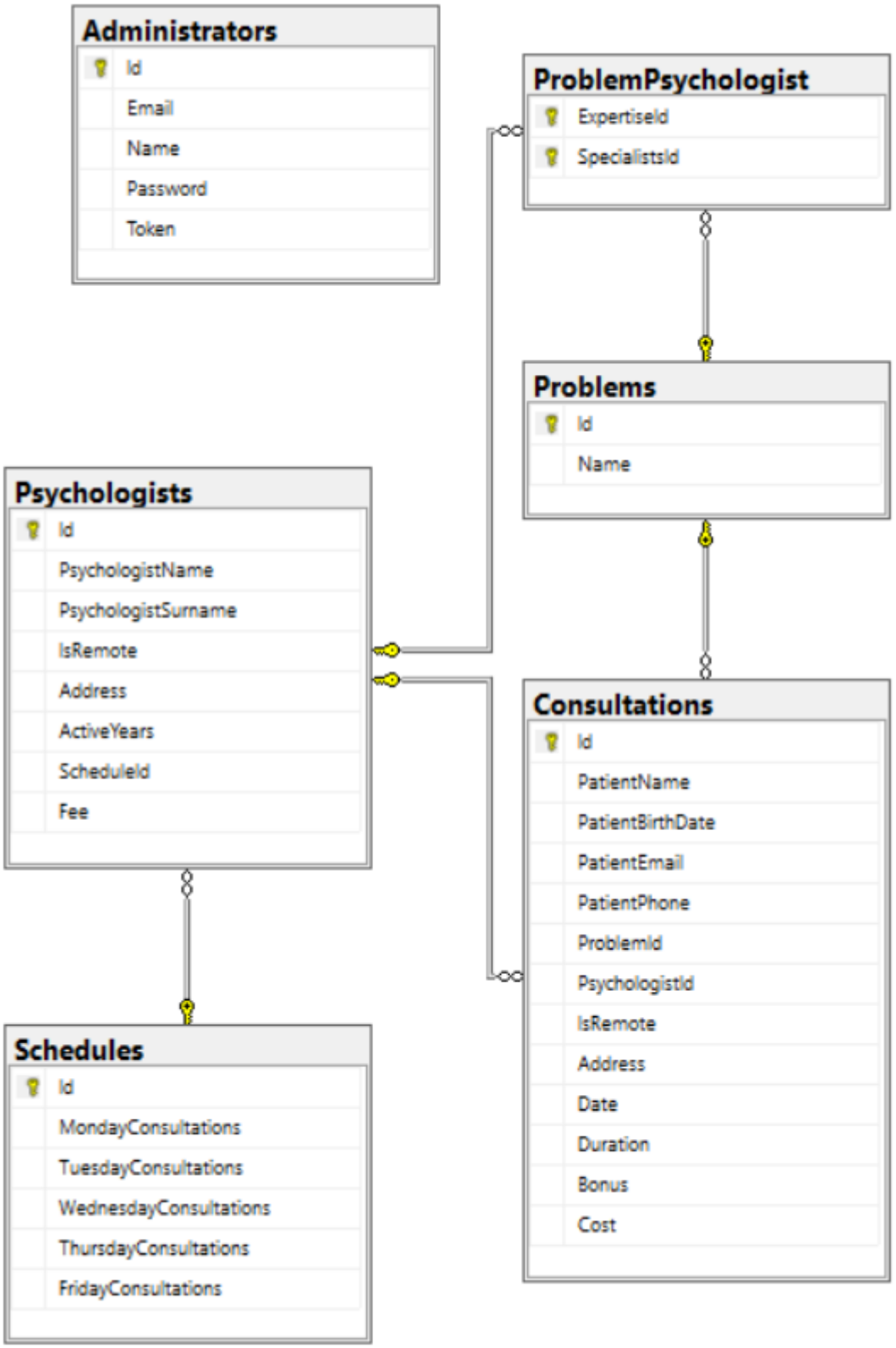
Importation

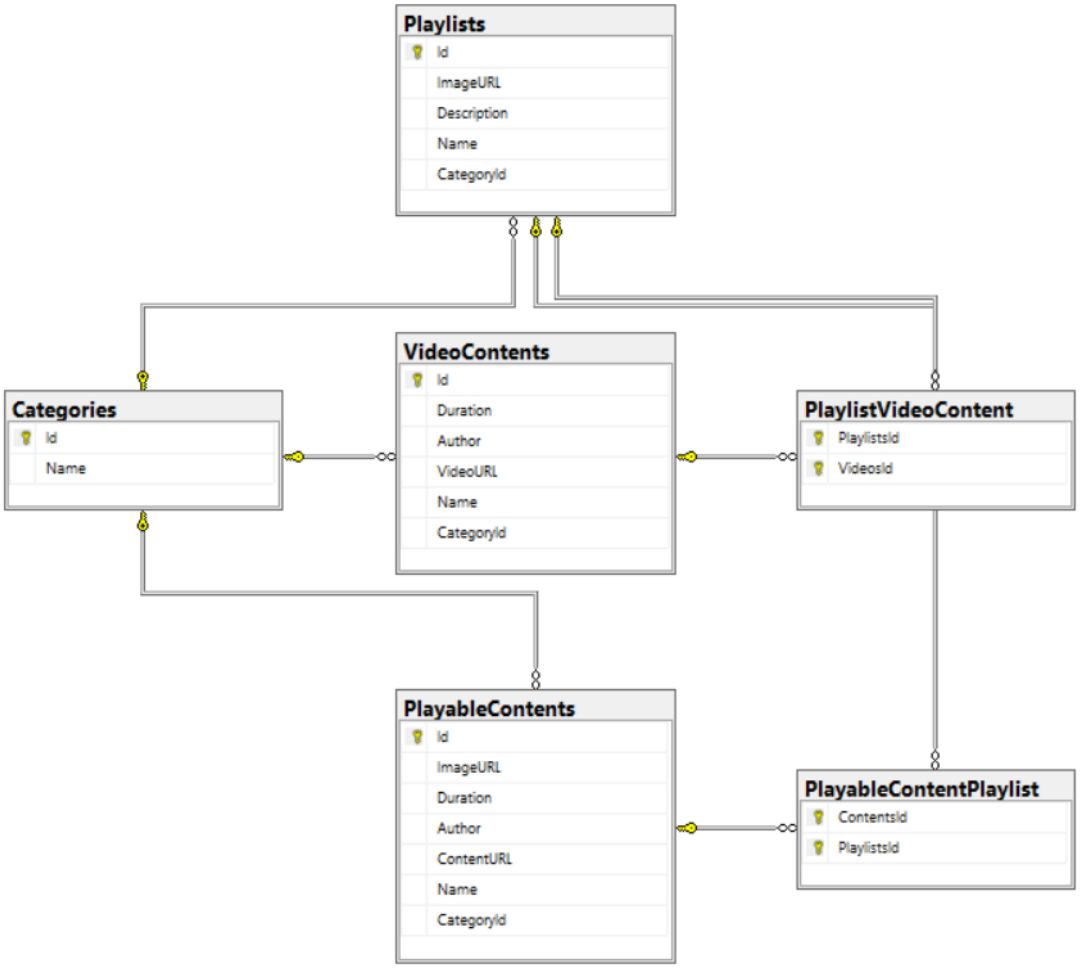


Vista con relaciones

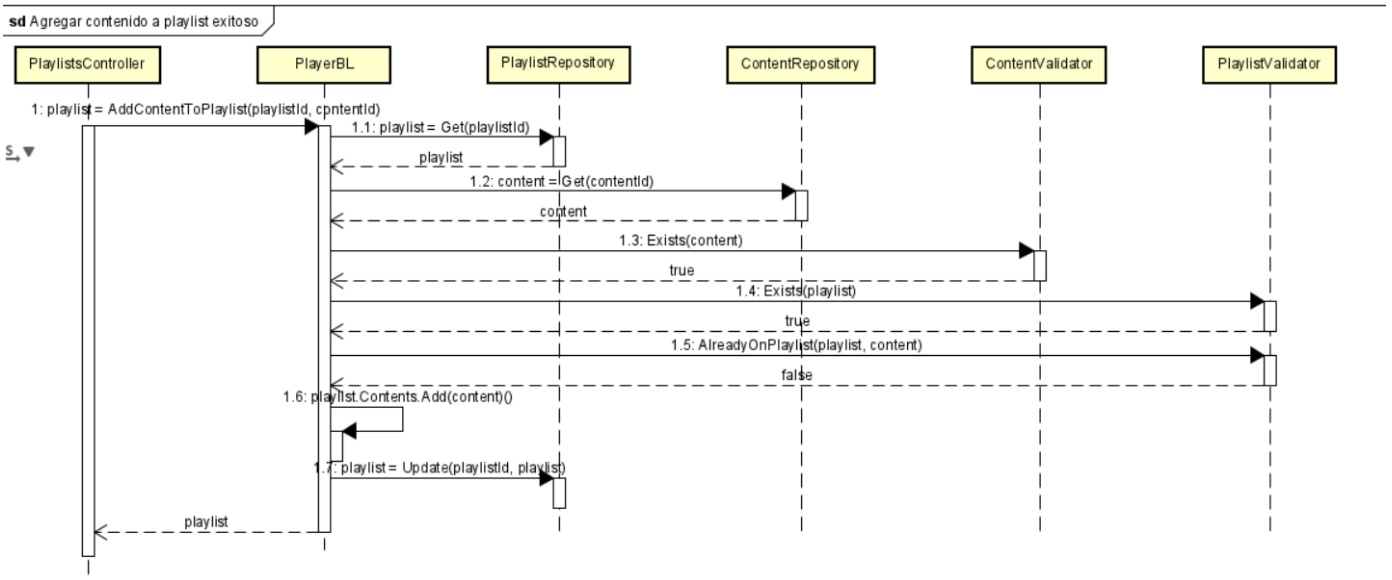


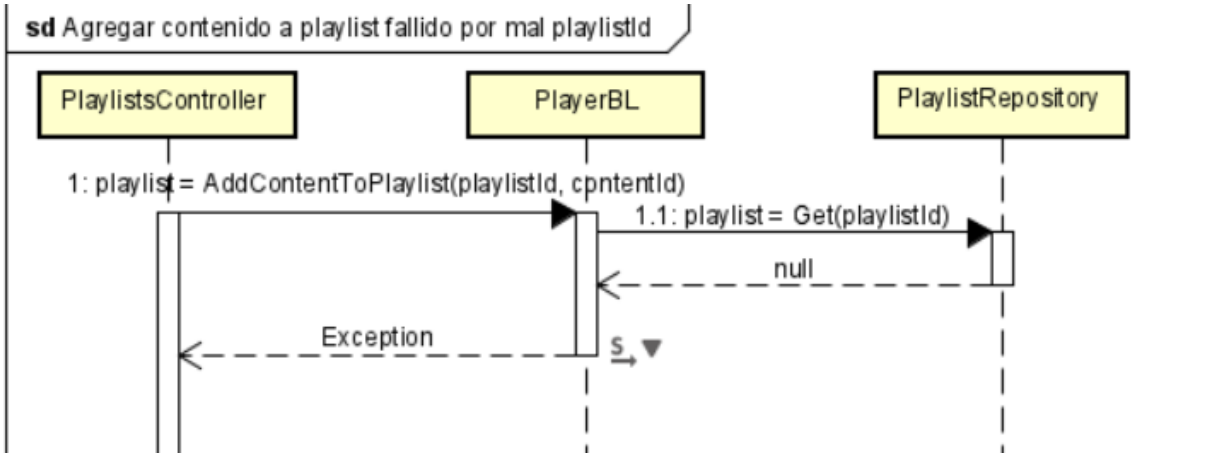
Modelo de tablas de BD

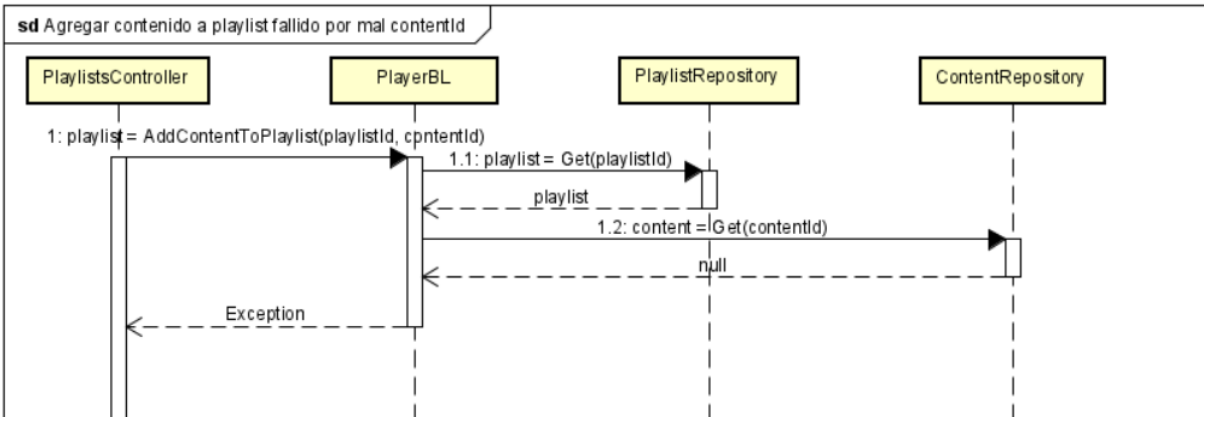


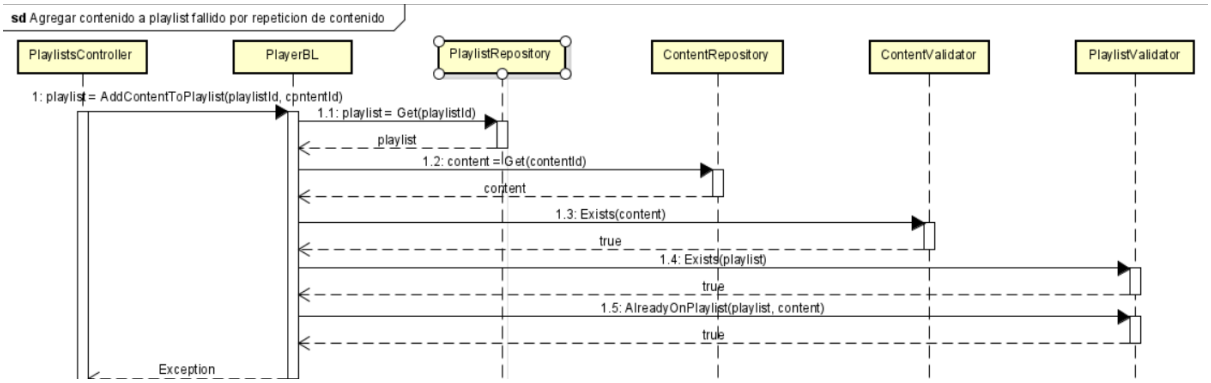


Diagramas de interacción para el caso agregar contenido a playlist


sd Agregar contenido a playlist exitoso
PlaylistsController
PlayerBL
PlaylistRepository
ContentRepository
ContentValidator
PlaylistValidator
1: playlist = AddContentToPlaylist(playlistId, cpntentId)
1.1: playlist = Get(playlistId)
playlist
1.2: content = Get(contentId)
content
1.3: Exists(content)
true
1.4: Exists(playlist)
true
1.5: AlreadyOnPlaylist(playlist, content)
false
1.6: playlist.Contents.Add(content)()
1.7: playlist = Update(playlistId, playlist)
playlist


sd Agregar contenido a playlist fallido por mal playlistId
PlaylistsController
PlayerBL
PlaylistRepository
1: playlist = AddContentToPlaylist(playlistId, cpntentId)
1.1: playlist = Get(playlistId)
null
Exception


sd Agregar contenido a playlist fallido por mal contentId
PlaylistsController
PlayerBL
PlaylistRepository
ContentRepository
1: playlist = AddContentToPlaylist(playlistId, cpntentId)
1.1: playlist = Get(playlistId)
playlist
1.2: content = Get(contentId)
null
Exception



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 estos 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 (83%) 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.

Implementación nuevas funcionalidades

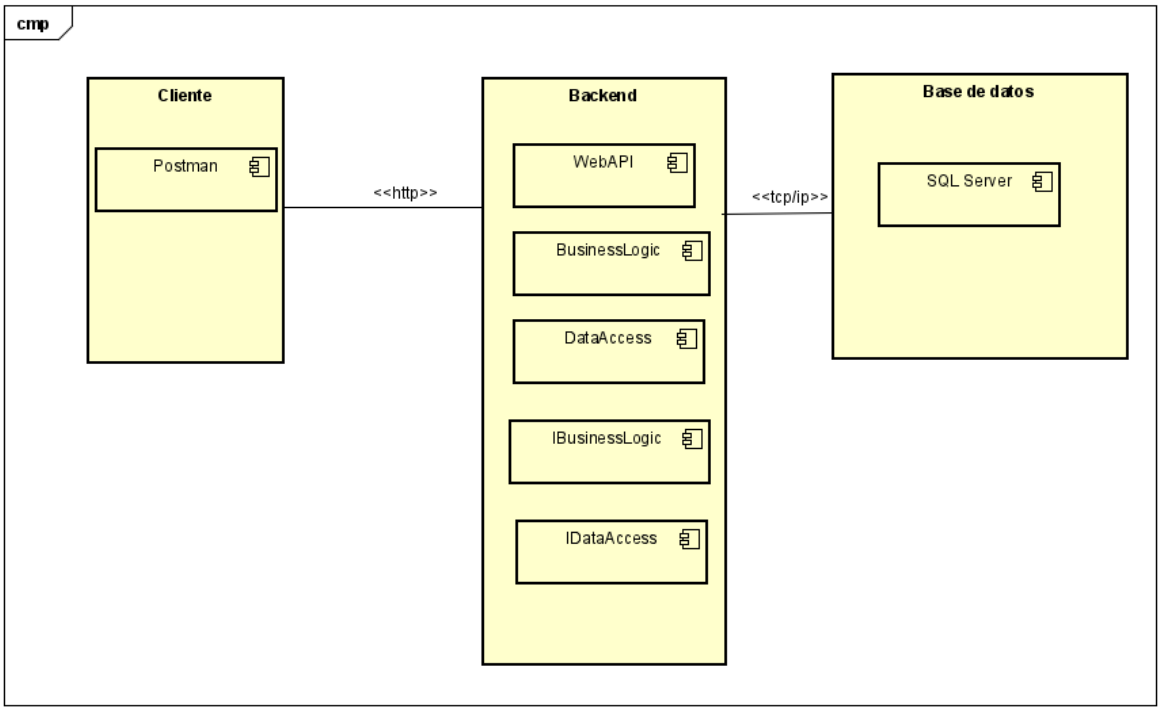
Para este segundo obligatorio se plantearon tres nuevo requerimientos:

- Importación de contenidos
- Soporte de un nuevo tipo de contenido
- Costos en las consultas con psicólogos

Para la implementación del nuevo tipo de contenido, se creó la entidad VideoContent la cual hereda de otra clase denominada CategoryElement. Esta misma es también heredada por las entidades PlayableContent y Playlist. Aquí podemos observar el cumplimiento del principio Open Close. Este plantea que las clases deben ser abiertas a la extensión y cerradas a la modificación, es por esto que si en un futuro se desea agregar otro tipo de contenido, simplemente se deberá agregar una nueva clase la cual implemente el nuevo tipo.

Por otro lado, para la implementación del “requerimiento costos de consultas para psicologos”, se crearon properties en las entidades Psychologist y Consultation. También debido a esto, se tuvo que implementar el DTO ConsultationDTO el cual contiene todas aquellas properties de la entidad Consultation qué necesito para implementar el front end del mismo.

Diagrama de componentes

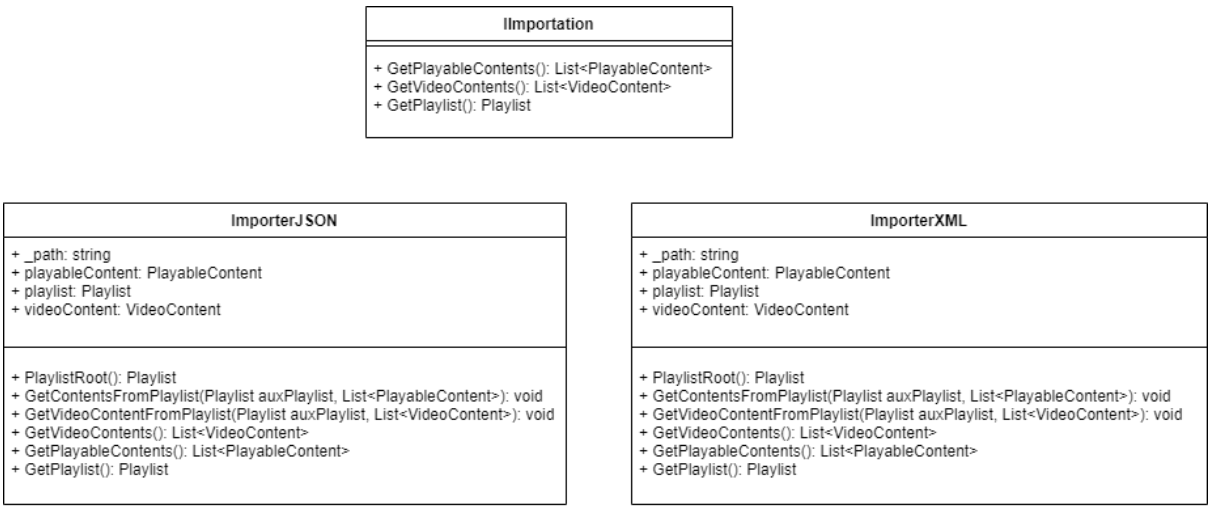


Es interesante cómo estos componentes pueden residir físicamente en el mismo equipo o en distintos, facilitado por los mecanismos de comunicación que utilizan. Idealmente en un ambiente productivo, puede ser más seguro separar el servidor web del servidor que contenga la base de datos. Luego el cliente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

Implementación Reflection

Para la implementación de esta nueva funcionalidad, se creó la librería Importation, la cual contiene la interfaz que va a ser necesaria implementar en el caso que se desee extender la misma. Además de la interfaz, se encuentran dos clases: ImportationJSON y ImportationXML. Estos dos son los tipo de importación que el sistema soporta hasta el día. En el caso que en un futuro se desee incorporar una importación .csv u otra, se deberá crear una clase que implemente las funciones establecidas en la interfaz. Específicamente:

- GetPlayableContents(): Me devuelve una lista de todos los PlayableContents que se encuentren en el archivo.
- GetVideoContents(): Me devuelve una lista de todos los VideoContents que se encuentren en el archivo.
- GetPlaylist(): Me devuelve la playlist recibida.



Por otro lado, se creó el controller ImportationsController el cual contiene el endpoint ImportContent que importa el contenido, este mismo recibirá el tipo de importación, siendo los soportados XML y JSON, y una lista de parámetros. En esta se encontraran todas los parámetros que dicha importación requiere, por ejemplo, en el caso de un archivo de tipo JSON, simplemente será necesario un campo de texto donde se colocará la ruta.

Por tanto, también nuestra lógica de negocio se vio modificada. Aquí se creó la interfaz IImportationBL, esta contiene los métodos necesarios para agregar los contenidos obtenidos. La entidad Importation implementa esta misma.

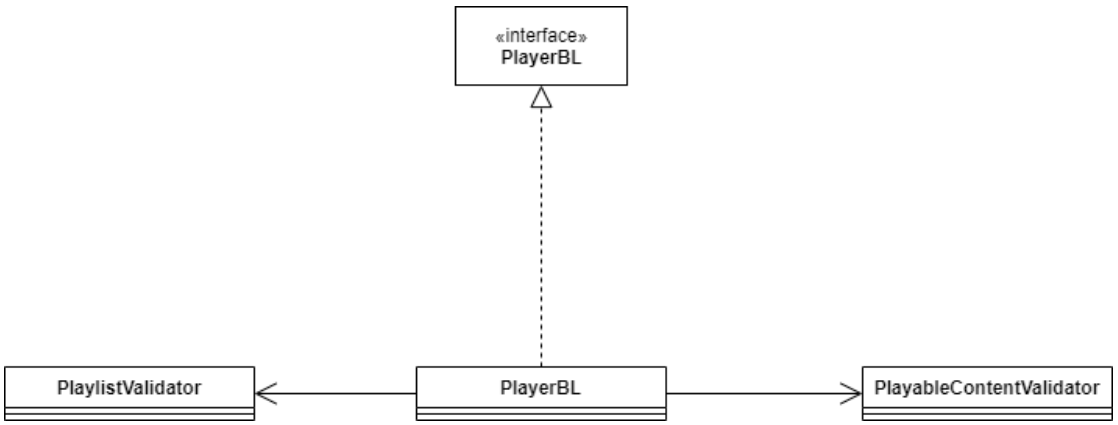
Mejoras realizadas

Implementación patrón Facade

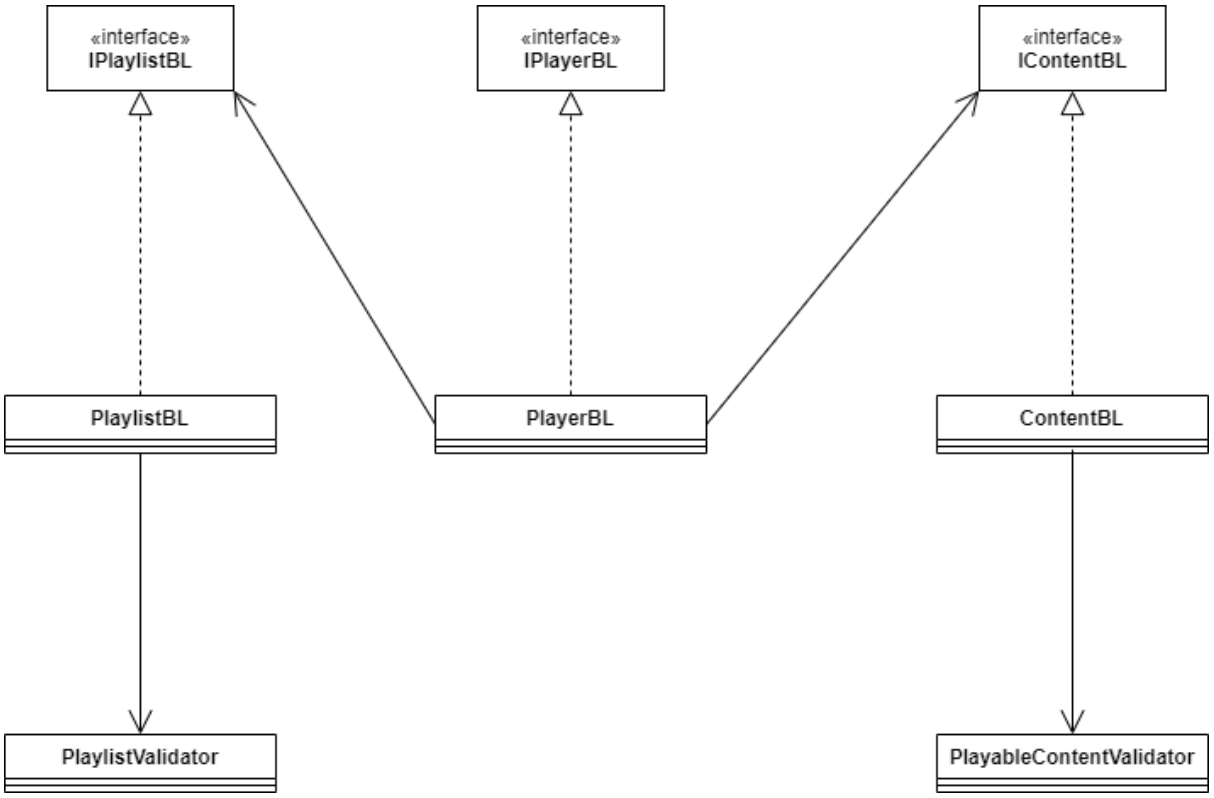
Para esta iteración decidimos utilizar el patrón Facade, dado que encajaba con uno de los cambios que queríamos implementar. Nuestra clase PlayerBL con el tiempo había ido acumulando una serie de responsabilidades muy grande, lo cual hacía que presentase necesidad de cambio con frecuencia, además de estar alcanzando un largo considerable.

Pensamos que Facade nos iba a resultar una manera sencilla de descomponer las distintas responsabilidades que había ido acumulando PlayerBL en un par de clases, y que PlayerBL actúe simplemente como una fachada que llame a las mismas. Además, esto nos permitió desacoplar PlayerBL de los validators de Content y Playlist, y hacer que simplemente dependa de las interfaces que luego implementaron ContentBL y PlaylistBL.

El diseño original que teníamos era este:



Y luego de implementado el refactor quedó así:



Ahora PlayerBL actúa simplemente como una fachada que unifica las interfaces PlayerBL y PlaylistBL, además de haberse desacoplado de PlaylistValidator y PlayableContentValidator.

Si queremos modificar algo sobre la lógica de negocios, esto puede afectar a PlayerBL pero por lo menos los cambios no va a ser tocada constantemente, lo cual es riesgoso con la cantidad de clases (controllers) que dependían de ella.

Traslado de Context a DataAccess

Si bien esta mejora no es muy grande, de todos modos, nos brindó una visión y entendimiento más claro del diseño, no solo para nosotros sino que también para terceros que deseen trabajar con la misma. En el obligatorio pasado teníamos la clase Context en el paquete IDataAccess, el cual fue trasladado al paquete DataAccess.

Esto implicó una serie de mejoras en cuanto al diseño que tal vez no sean evidentes.

Como primer punto a destacar, esto aumentó la abstracción del paquete IDataAccess, lo cual es bueno para cumplir el principio de dependencias estatales (el paquete de lógica de negocio depende de este).

Por otro lado, desacopla el paquete DataAccess del paquete IDataAccess, dado que antes dependía de este para conocer las interfaces que debía implementar, y además para hacer uso del Contexto. Ahora la dependencia tiene más sentido.

Finalmente, este desacoplamiento alcanzado se ve reflejado en el diagrama de clases, en el que ahora no va a quedar tan confusas las relaciones entre las clases de estos dos paquetes. Vale la pena aclarar que notamos el error en el diseño cuando realizabamos la documentación para el primer obligatorio y vimos cómo quedaba el diagrama.

Por último, esto también aumentó la cohesión del paquete de buena manera.

Uso de DTOs

Uno de los cambios que quisimos implementar en el primer obligatorio pero no llegamos por motivos de tiempo fue el uso de DTOs. Esto nos permite controlar el formato en el que enviamos y recibimos datos desde y hacia la API, por lo cual redujo una gran cantidad de potenciales errores que podían ocurrir al enviar objetos complejos.

Para este obligatorio implementamos el uso de DTOs en los tipos Psychologist y Consultation, lo cual nos facilitó muchísimo el desarrollo del cliente en Angular por un lado, y por otro nos redujo la cantidad de chequeos y potenciales errores que implicaba enviar y recibir objetos complejos.

Análisis de métricas

Los resultados de la ejecución de NDepend sobre el obligatorio fueron los siguientes:

Assemblies	# lines of code	# IL instruction	# Types	# Abstract Types	# lines of comment	% Comment	% Coverage	Afferent Coupling	Efferent Coupling	Relational Cohesion
Domain v1.0.0.0	160	608	12	1	0	0	-	37	14	1.17
IDataAccess v1.0.0.0	0	0	2	2	-	-	-	19	5	0.5
DataAccess v1.0.0.0	508	10062	28	0	8	1.55	-	1	93	1
IBusinessLogic v1.0.0.0	0	0	7	7	-	-	-	15	17	0.14
Importation v1.0.0.0	125	595	5	1	0	0	-	2	23	1
BusinessLogic v1.0.0.0	433	2902	11	0	0	0	-	1	62	0.82
WebAPI v1.0.0.0	224	1325	10	0	5	2.18	-	0	81	0.2

No logramos que ninguno de los paquetes quede con el valor ideal para cohesión relacional ($1,5 < x < 4$), pero esto tiene su justificación.

Por el lado de los paquetes de interfaces, no sorprende demasiado que no tengan gran cantidad de dependencias entre sí.

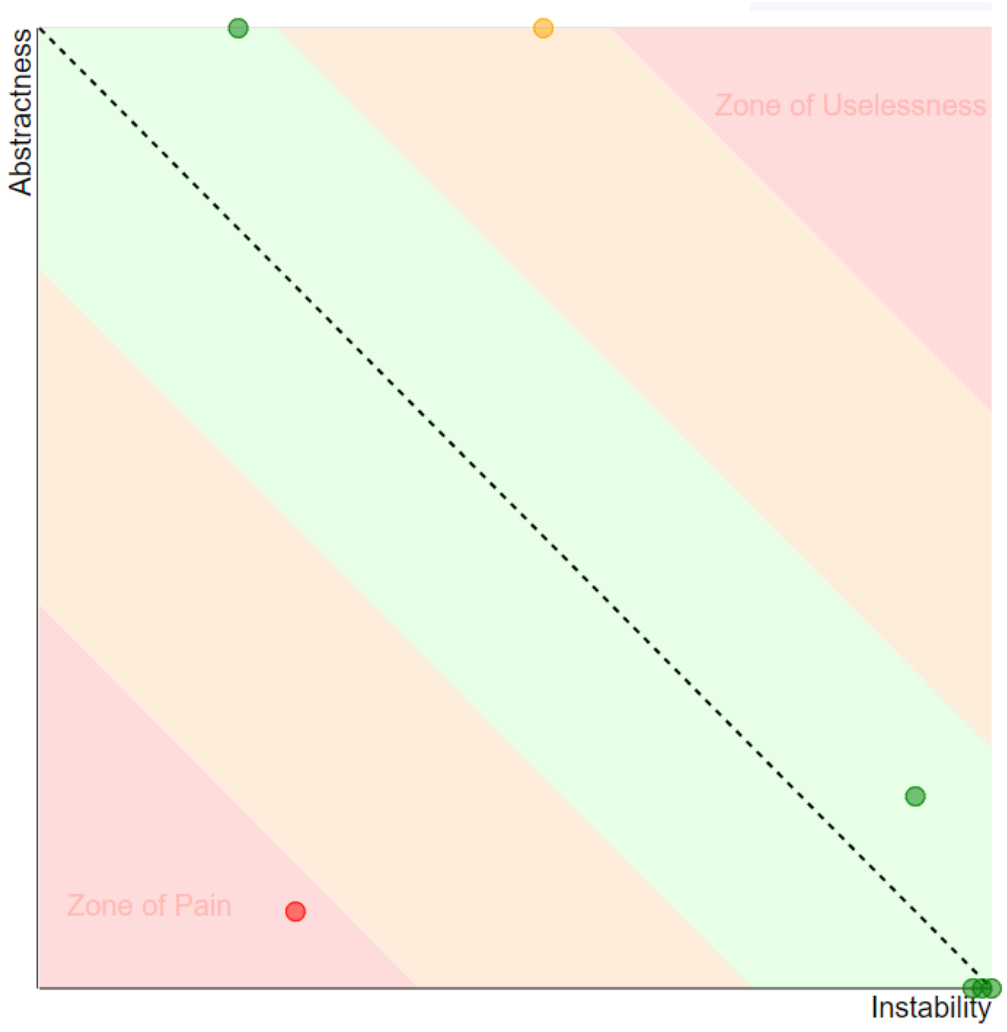
En cambio, se puede ver por ejemplo que BusinessLogic, Importation, DataAccess y Domain presentan valores más cercanos a los ideales, pero tampoco los alcanzan. Esto es porque esta aplicación se puede ver, o por lo menos nosotros así la concebimos, como dos aplicaciones distintas. Es decir, hay todo un grupo de entidades de dominio (psicologo, problema, consulta) que no tiene nada que ver con playlist, video, etc. Esta fragmentación se puede ver a lo largo de toda la aplicación, llevando a estos valores.

Finalmente, el paquete WebAPI agrupa controllers, que no deberían tener dependencia alguna entre sí, por lo cual el hecho de que tenga baja cohesión relacional es perfectamente comprensible.

# Types	# Abstract Types	# lines of comment	% Comment	% Coverage	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
12	1	0	0	-	37	14	1.17	0.27	0.08	0.45
2	2	-	-	-	19	5	0.5	0.21	1	0.15
28	0	8	1.55	-	1	93	1	0.99	0	0.01
7	7	-	-	-	15	17	0.14	0.53	1	0.38
5	1	0	0	-	2	23	1	0.92	0.2	0.08
11	0	0	0	-	1	62	0.82	0.98	0	0.01
10	0	5	2.18	-	0	81	0.2	1	0	0

Así quedan estos valores graficados en la gráfica de abstracción vs inestabilidad.

Se ve claramente que se cumple el principio de dependencias estables, con la pequeña excepción de que WebAPI,que depende de dataAccess, tiene un valor mayor por 0,02.



La mayoría de nuestros paquetes se encuentran en el área deseada, a excepción de Domain (zona de dolor) y IBusinessLogic (casi zona de inutilidad).

El motivo por el cual IBusinessLogic se encuentra mucho más a la derecha que IDataAccess (misma línea a la izquierda) es el hecho de que todas las clases de IBusinessLogic presentan acoplamiento a las entidades de dominio. A futuro se podría intentar mejorar esto, pero viendo que se encuentra solo un paquete con este tipo de problemas no lo interpretamos como una situación grave.

Por el lado de Domain, al ser muy concreto y además tener una cantidad de clases e interfaces que dependen de sus clases, lo mueven a la zona de dolor, y lo sentimos cada vez que quisimos hacer un cambio a una entidad de dominio.