Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de aplicaciones II

Obligatorio I

Descripción del diseño

Juan Manuel Gallicchio 233335

Federico Carbonell 224359

https://github.com/ORT-DA2/CarbonellGallicchio

Descripción General

Se entrega el BackEnd del sistema BetterCalm, la finalidad de esta sería ayudar a las personas a sobrellevar el estrés de una manera más sana.

Está compuesto por dos grandes funcionalidades, un reproductor de contenido reproducible y un sistema de agendamiento de consultas con psicólogos. El desarolla se llevó a cabo utilizando el framework .Net Core.

La API se realizó cumpliendo el estándar REST. Este estándar aprovecha los headers HTTP para darle un nombre a las consultas. Esta API a su vez no posee lógica y su función principal es atender peticiones y devolver una respuesta.

Los verbos principales y utilizados de REST son GET, POST, PUT y DELETE. A pesar de ser solo 4, mediante la especificación de distintas URIs la API logra tener más de 4 métodos por controller.

Los retornos de la API pueden contener un mensaje de error u éxito pero su parte principal que indica que sucedió con la consulta es el Status Code. Estos errores se clasifican en cinco "familias":

- 100 – Informational
- 200 – Success
- 300 – Redirections
- 400 – Client Errors
- 500 – Server Errors

Para el envío de entidades en el body de una consulta se utiliza JSON.

Para el acceso a datos utilizamos el ORM Entity Framework Core realizando la técnica de Code First.

La aplicación fue realizada utilizando la metodología TDD, implica primero codificar las pruebas, luego implementar las funciones y por último realizar un refactor de las mismas. Por lo que la aplicación presenta sus proyectos de pruebas. Estos son MSTest.

Decisiones Generales

Luego de analizar el código hecho inicialmente donde el mapeo de los modelos a las clases del dominio se realizaba en los controller decidimos quitarle la responsabilidad y crear una nueva capa denominada Adapter. Esta se encargará de dicho proceso y de negociador entre los controllers y la lógica de negocio.

Para el mapeo del adapter utilizamos AutoMapper, consideramos que realizar el mapeo de las clases era algo sencillo y no aportaba un gran conocimiento sino más una pérdida de tiempo. Para esto se crearon Profiles por cada tipo de clase a mapear. Esto nos genera una mayor mantenibilidad.

Se utilizaron filters para la autentificación de los administradores, esto nos evita la duplicación de código. Este se encarga de tomar el token desde el Header y validarlo, se encapsula la lógica en un solo lugar y este se ejecuta antes de llegar al controller. Por lo que si el token no es válido o inexistente no se accede al controller.

Separamos los modelos en dos paquetes, los de entrada y los de salida. El propósito fue limitar la información de estos mismos ya que la creación de un objeto puede no ser igual al retorno del mismo.

Se implementaron distintos proyectos de excepciones, dependiendo la capa donde habitara. De esta forma evitamos generar dependencias de todo el sistema sobre un proyecto.

Para las validaciones creamos una interfaz con un método genérico, luego cada clase la implementa con su validación específica y sus excepciones.

Al igual que las validaciones, para el acceso a datos decidimos utilizar un repositorio genérico, nos dimos cuenta que de esta forma lograremos abstraer la lógica en un solo lugar. Que para este caso es muy similar en todos los escenarios. Gracias a esto logramos optimizar la cantidad de código y de pruebas del mismo, solamente debemos implementar una única vez cada método.

Bugs o Defectos conocidos:

Debido a las limitaciones de Entity Framework Core, debimos realizar clases intermedias para realizar las relaciones Many-to-Many entre las entidades. Esto hace que nuestro dominio de cierta forma depende de la base de datos ya que esa clase no forma parte de nuestro dominio de negocio. Deberíamos haber creado entidades para persistirlas y separar el dominio del negocio de estas evitando las dependencias.

Descripciones de la API

Como hemos dicho anteriormente es una API que cumple con el estándar REST, por lo que se utiliza los verbos HTTP. Además, se utilizan distintas URIs para así poder acceder a distintos recursos mediante el mismo verbo (en las URIs no se utilizan verbos).

La API no interactúa con las clases del dominio. Simplemente mediante la inyección de dependencias utiliza las funciones de la capa Adapter que esta se encarga de realizar el mapeo de modelos a dominio para enviar a la capa lógica de negocio o a la inversa. Estas clases de dominio se mapean con Models(In) y BasicInfoModels(Out). Estas clases son modelos simplificados con los que interactúa el cliente.

El propósito de los modelos es limitar la información, ya que no siempre se desea enviar la misma información al cliente. De esta forma se mejora la performance de las consultas que con pocos datos no se aprecia, pero en bases de datos grandes aspectos como este juegan un rol muy importante.

EndPoints y Justificación

AdministratorController

- GET administrator/{id}: retorna la información del administrador con igual id al enviado.
- POST administrator: crea un administrador, recibe un nombre, un mail y una contraseña en el Body.
- DELETE administrator/{administratorId}: elimina el administrador con igual id al enviado.
- PUT administrator modifica el contenido de un administrador, recibe un id, un nombre, un mail y una contraseña en el Body.
-

CategoryController

- GET category: retorna la información de todas las categorías del sistema.
- GET category/{categoryId}/playlist: retorna la información de todas las playlist asociadas a la categoría de igual id al enviado.

AudioContentController

- GET audioContent/{id}: retorna la información de un contenido reproducible con igual id al enviado.
- POST audioContent: crea un contenido reproducible y es retornado al usuario. Es necesario un token de administrador que es enviado en el Header.
- DELETE audioContent/{audioContentId}: elimina el contenido reproducible con igual id al enviado. Es necesario un token de administrador que es enviado en el Header.
- PUT administrator modifica el contenido de un contenido reproducible, es necesario un token de administrador que es enviado en el Header.

ConsultationController

- POST consultation: sirve para la creación de una consulta con un psicólogo. En caso de agendarse es devuelta la información del psicólogo asignado a la consulta.

ProblematicController

- GET problematic: retorna la información de todas las problemáticas del sistema.

PsychologistController

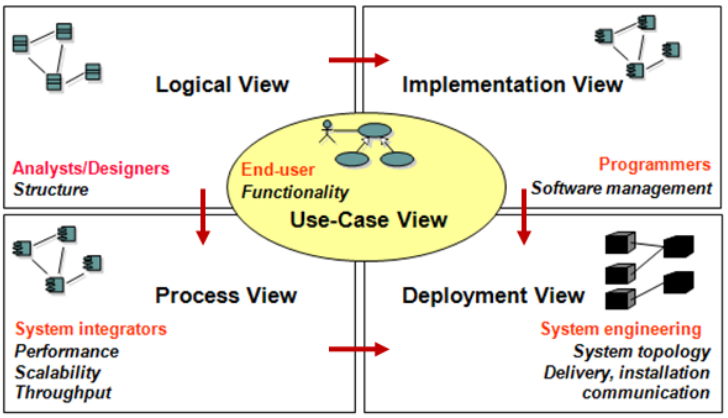
- GET psychologist/{id}: retorna la información de un psicólogo con igual id al enviado.
- POST psychologist: crea un psicólogo y es retornado al usuario. Es necesario un token de administrador que es enviado en el Header.
- DELETE psychologist/{id}: elimina un psicologo con igual id al enviado. Es necesario un token de administrador que es enviado en el Header.
- PUT: psychologist: modifica el contenido de un psicólogo, es necesario un token de administrador que es enviado en el Header.

SessionController

- POST session: sirve para la creación de una sesión de un administrador. Si los datos enviados son correctos le es devuelto al cliente su correo y un token de administrador.

Descripción y Justificación de Diseño

Seguiremos las vistas del Modelo 4+1 para describir y justificar nuestro diseño. Este modelo sirve para describir la arquitectura de un sistema de software basado en varias vistas concurrentes. El siguiente diagrama describe al modelo 4+1.

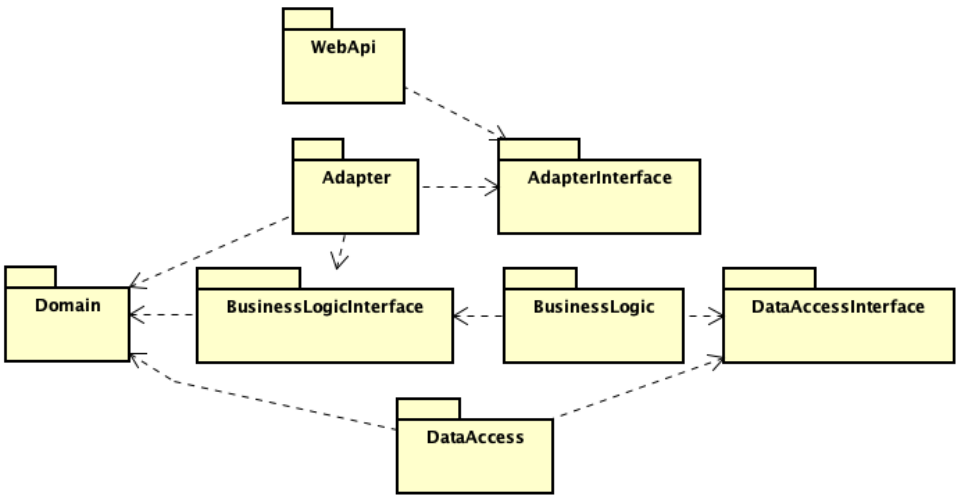


Vista Lógica

Esta vista describe el modelo desde los elementos de diseño que componen el sistema y también cómo interactúan entre ellos.

El siguiente diagrama ilustra la división de capas y las dependencias de nuestro proyecto, decidimos omitir paquetes que no son relevantes para esta instancia como el paquete de test , validaciones, sesión y excepciones.

Se puede apreciar que el proyecto está dividido en cuatro capas distintas, no se realizan atajos entre capas. Es decir, en ningún momento la capa de Web Api interactúa directamente con la Business Logic ni viceversa.



Paquete	Responsabilidad
WebApi	Posee nuestra Web Api. Su responsabilidad es recibir las consultas del cliente, esto se realiza a través de controllers y luego debe derivarlas a la lógica de negocio correspondiente. La entrada y salida de datos se realizan a través de modelos.
AdapterInterface	Poseen las interfaces de nuestros adapters, estos se encargan de mappear los modelos a objetos del dominio entre los controllers y la business logic.
Adapter	Posee las implementaciones de las clases del paquete AdapterInterface.
BusinessLogicInterface	Poseen las interfaces de nuestra lógica de negocio. Estas interfaces son utilizadas por las clases del paquete Adapter.
BusinessLogic	Poseen las implementaciones de la clase del paquete BusinessLogicInterface. Utilizan la interfaz IRepository para interactuar con la base de datos.
Domain	Posee nuestras entidades. Estas mismas persisten en la base de datos.
DataAccessInterface	Posee la interfaz de los métodos relacionados al acceso a datos.
DataAccess	Posee la implementación de la clase del paquete DataAccessInterface. También contiene el contexto de nuestra base de datos.

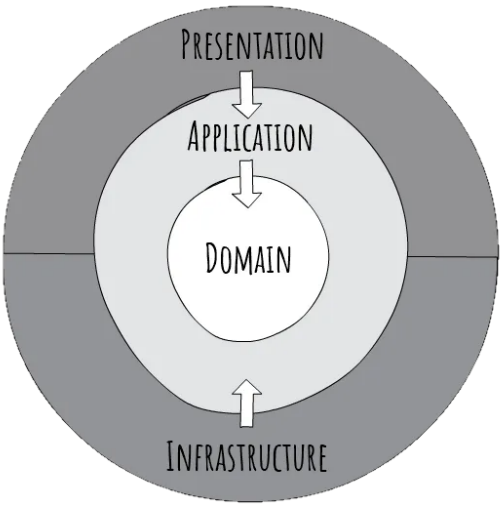
Como se puede apreciar en el diagrama ambas capas de bajo nivel (Web Api y Data Access) dependen de capas de alto nivel (Data Access Interface y Adapter Interface). Esto es sumamente importante y reutilizable ya que las componentes de bajo nivel están más dispuestos a los cambios, por lo que sí dependen de capas de alto nivel (menor probabilidad de cambio) nos genera estabilidad evitando tener que modificar gran parte de la aplicación. Si el día de mañana se desea cambiar la implementación de la Web Api o de Data Access solamente deberíamos integrarlo a la lógica ya hecha. Para realizar el diseño nos basamos en el principio SOLID de inversión de dependencias.

También se puede observar cómo se cumple con Clean Architecture (Robert C. Martin).

Este concepto plantea ciertas reglas:

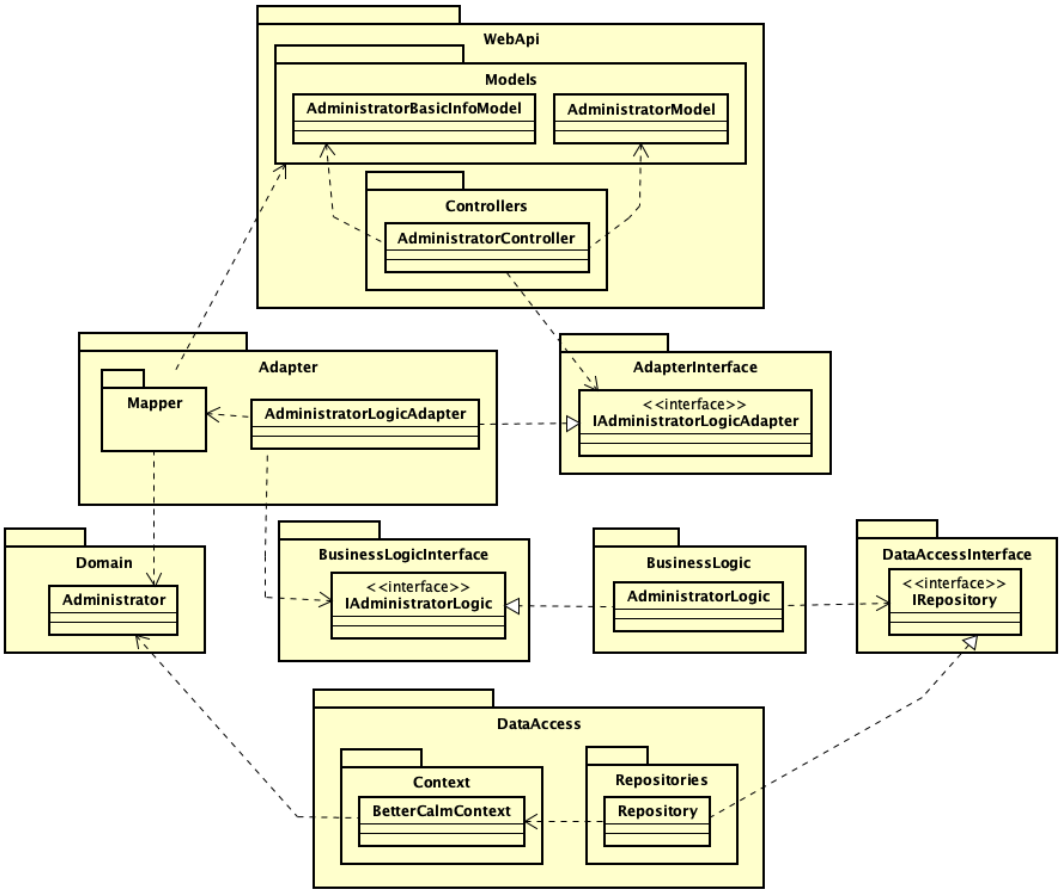
- La aplicación no debe depender de la interfaz de usuario
- La aplicación no debe depender a la base de datos
- Todas las capas deben poder probarse de forma independiente
- No se debe depender de frameworks específicos

En conclusión lo que nos indica las reglas es que las dependencias deben ser hacia adentro, es decir a capas que sean menos propensas al cambio. No se puede depender de capas externas ya que si se desea modificar el motor de base de datos esto no nos genera un gran impacto en el sistema.



Si observamos nuestro diagrama de capas se puede notar que todas nuestras dependencias van hacia el centro. Es cierto que dependemos del dominio, pero este posee una baja probabilidad de cambio en comparación con el resto por lo que es una buena práctica.

Para mejorar el entendimiento y la visibilidad decidimos realizar el siguiente diagrama para ilustrar las dependencias de la clase Administrator. Decidimos representarlo de esta forma ya que las dependencias entre los controllers, business logic, models y domain son iguales para todas las entidades.



Siguiendo el principio Single Responsability de SOLID es que se creó una clase de Business Logic por entidad. Esto nos garantiza que solamente se realicen acciones sobre cada entidad mediante su propia lógica, ésta será modificada solamente si la entidad como es afectada por cambios. Además de esto, cada lógica implementa una interfaz que expone las funciones requeridas para los respectivos controllers. Con esto estamos invirtiendo las dependencias y además la segregando las interfaces ya que el cliente implementa lo mínimo indispensable para su uso.

Esto igualmente es complementado con la inyección de dependencias. Este concepto se basa en un contenedor en el que se asocian interfaces con su implementación. Una vez declarada esta asociación en el contenedor, cuando una clase posee en el constructor de su clase la interfaz declarada, el contenedor se encarga de realizar el new de la implementación

e inyectarsela a la clase que la requiere. De esta forma, como el new no se realiza dentro de la clase que lo necesita, no se obtiene la dependencia entre la clase. La inyección de dependencias se utilizó tanto para la lógica, el adapter, el repositorio y el mapper.

Para evitar la dependencia desde la lógica hacia el acceso a datos, decidimos crear una interfaz que dentro se encuentre la interfaz del repositorio.

Para la implementación del acceso a datos creamos una interfaz de repositorio genérico y la implementación de la misma genérica. Esta decisión fue tomada luego de empezar a codificar y visualizar la repetición de lógica de la gran mayoría de los métodos de acceso a datos, gracias a esto logramos disminuir la cantidad de líneas de código y pruebas. Por lo que una mejora en performance y mantenibilidad.

Diagrama de clases del paquete Adapter

Poseen las implementaciones de las interfaces de nuestros adapters, estos se encargan de mappear los modelos a objetos del dominio entre los controllers y la business logic.

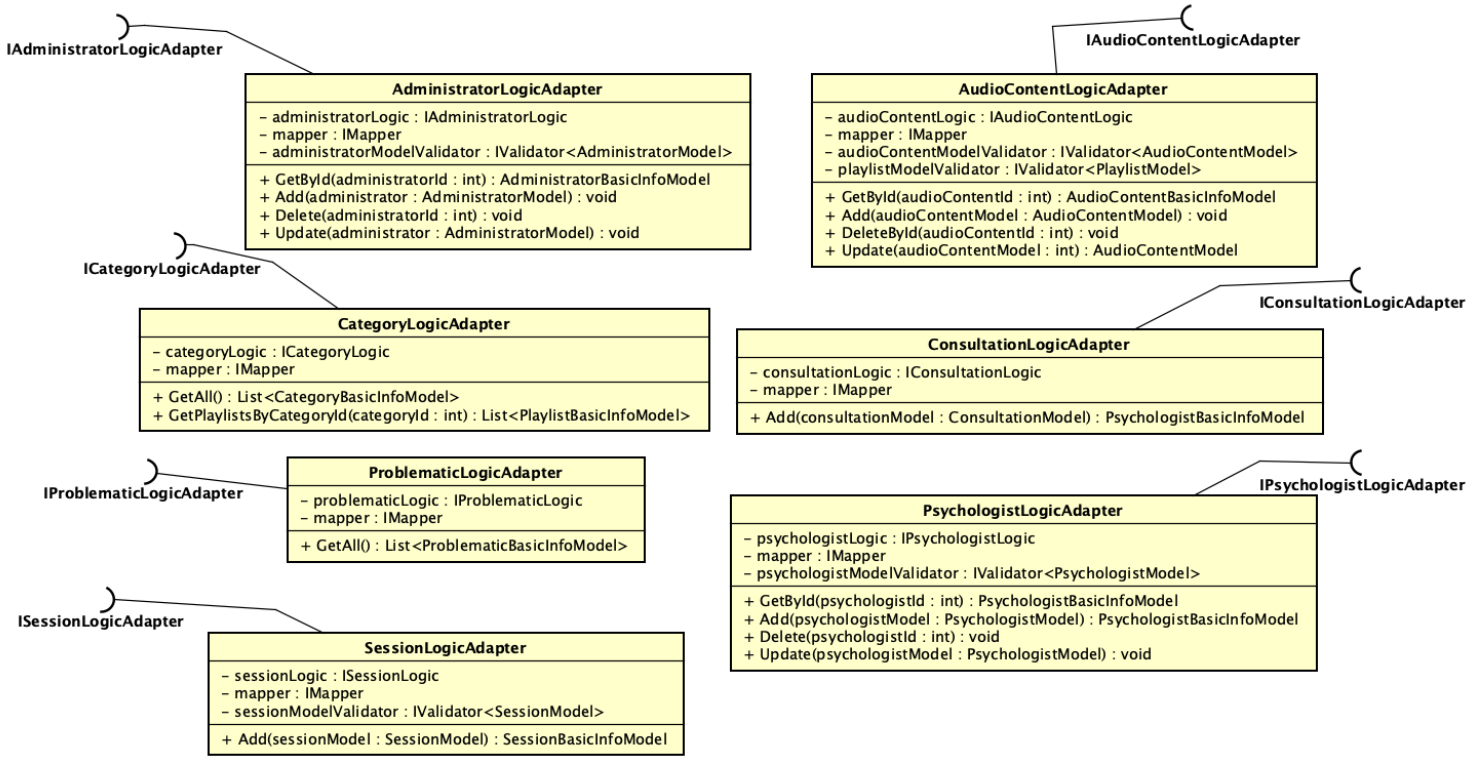


Diagrama de clases del paquete Business Logic

Poseen las implementaciones de la clase del paquete BusinessLogicInterface. Utilizan la interfaz IRepository para interactuar con la base de datos.

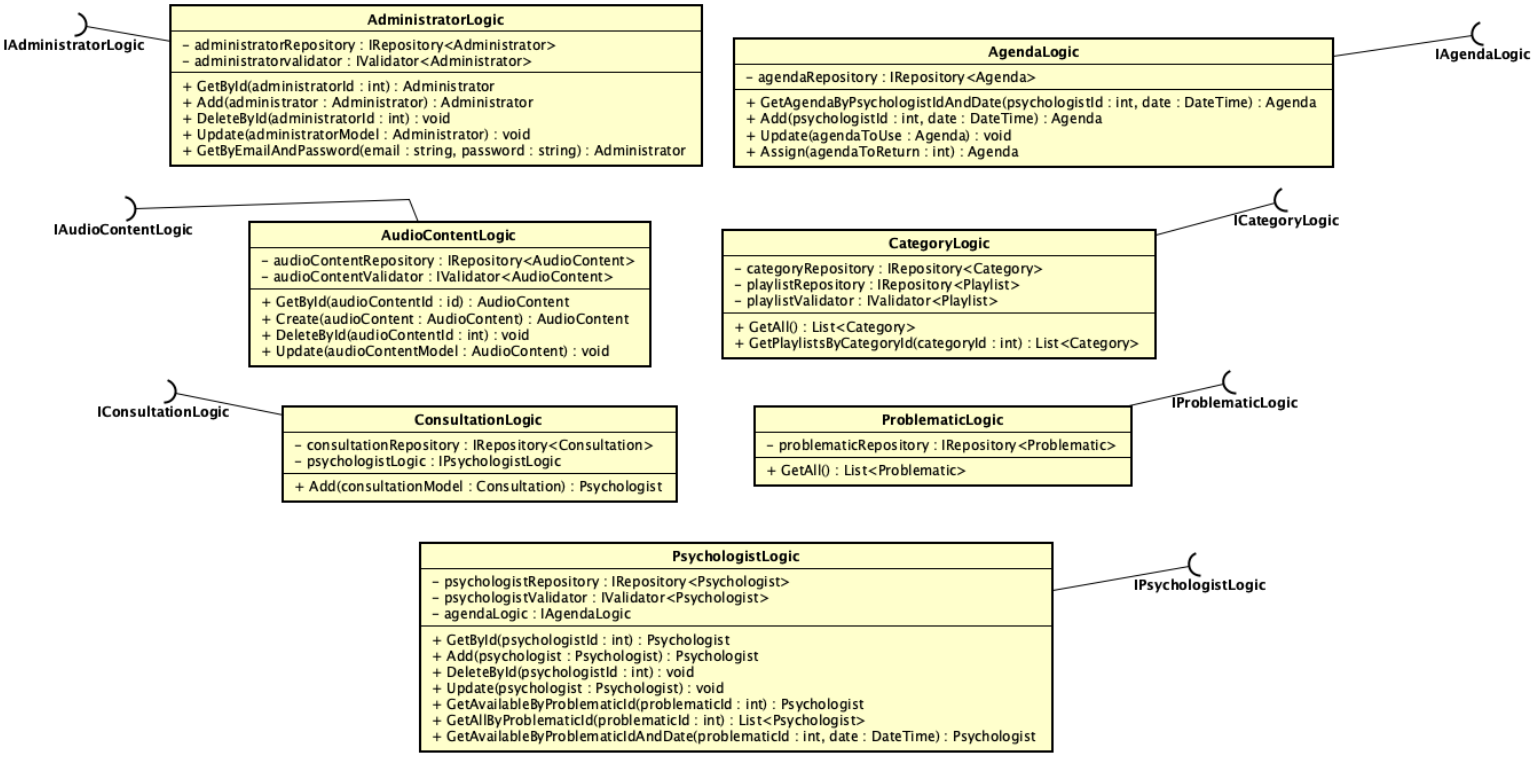


Diagrama de clases del paquete Domain

Posee nuestras entidades. Estas mismas persisten en la base de datos.

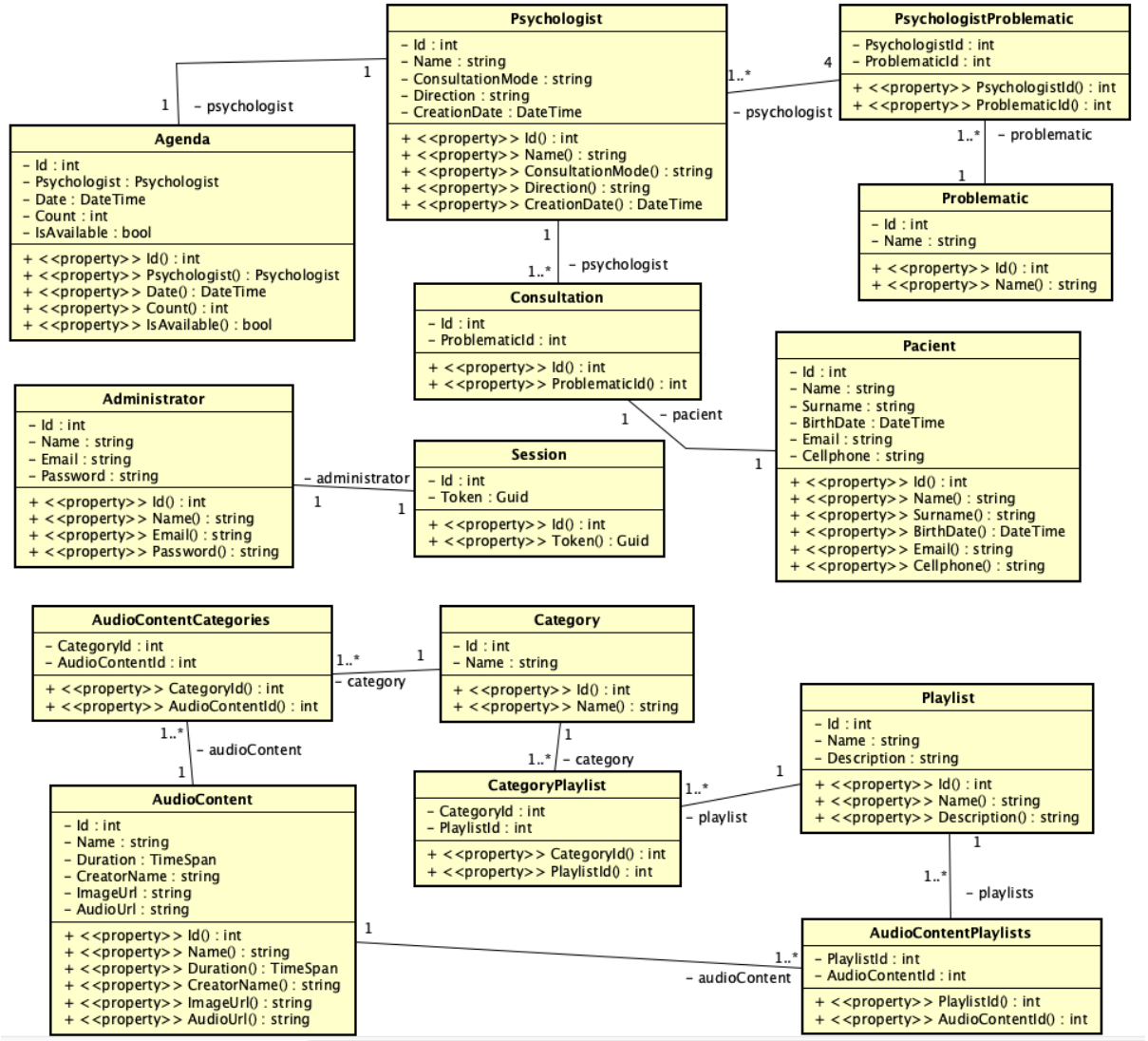
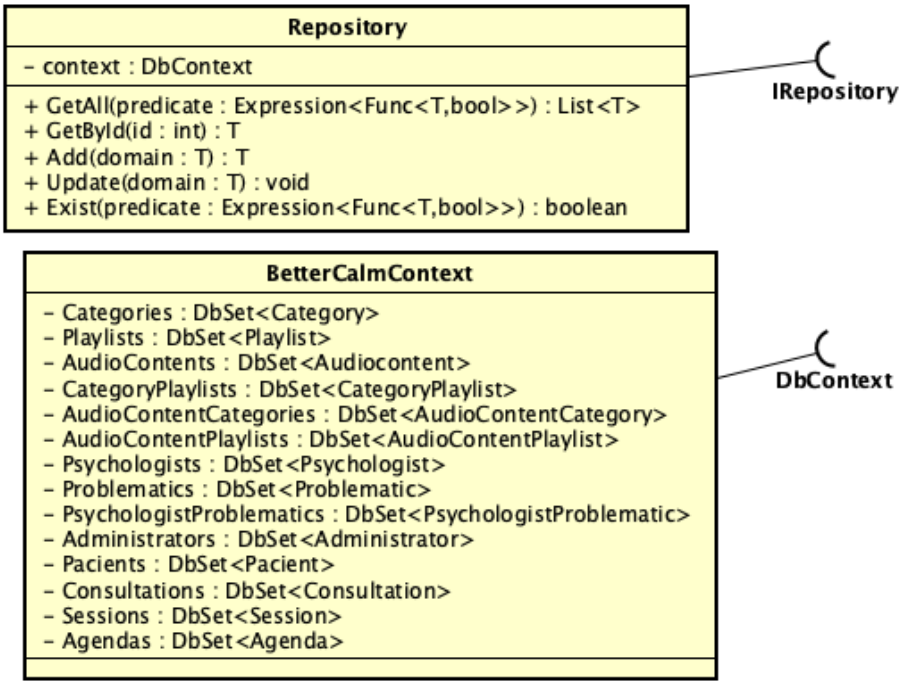


Diagrama de clases del paquete Data Access

Posee la implementación de la clase del paquete DataAccessInterface. También contiene el contexto de nuestra base de datos.

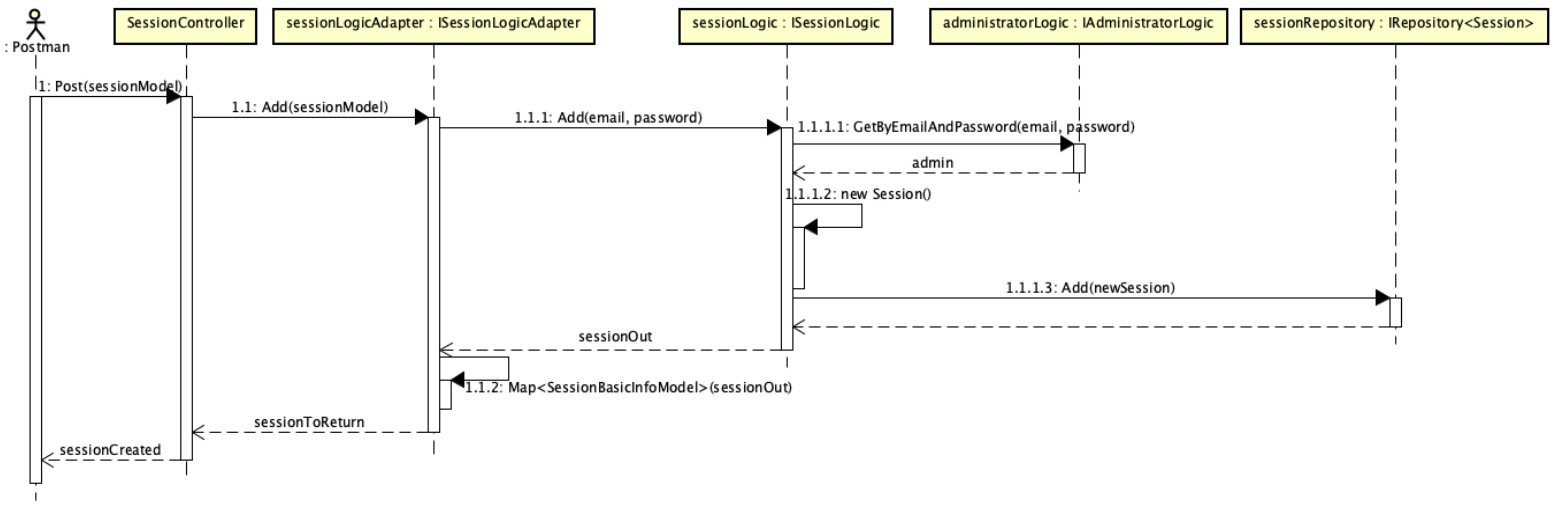


Vista de Proceso

Aquí mostraremos el comportamiento de algunas funciones de nuestro sistema en tiempo de ejecución. Decidimos realizar de la obtención de un token de administración y la lógica de la agenda de una consulta con un psicólogo ya que están compuestas por la lógica más abstracta, las demás funciones leyendo el código se logran entender sencillamente. El propósito de este documento es darle al lector una ayuda para entender el funcionamiento general del sistema, por esto decidimos no abrumarlo con diagramas.

Obtención de token

Para la creación de un token lo primero que verificamos es que los datos enviados por el cliente son válidos, luego de verificarlos el adapter se encarga de realizar el mapeo al tipo de datos utilizado por la lógica de negocio. Ya en la lógica de negocio se obtiene el administrador con los datos proporcionados por el adapter. Con este y un nuevo token se crea una instancia de sesión que es enviada al repositorio. De ser satisfactoria la creación se le devuelve al cliente el correo proporcionado y el token de privilegio administrador.



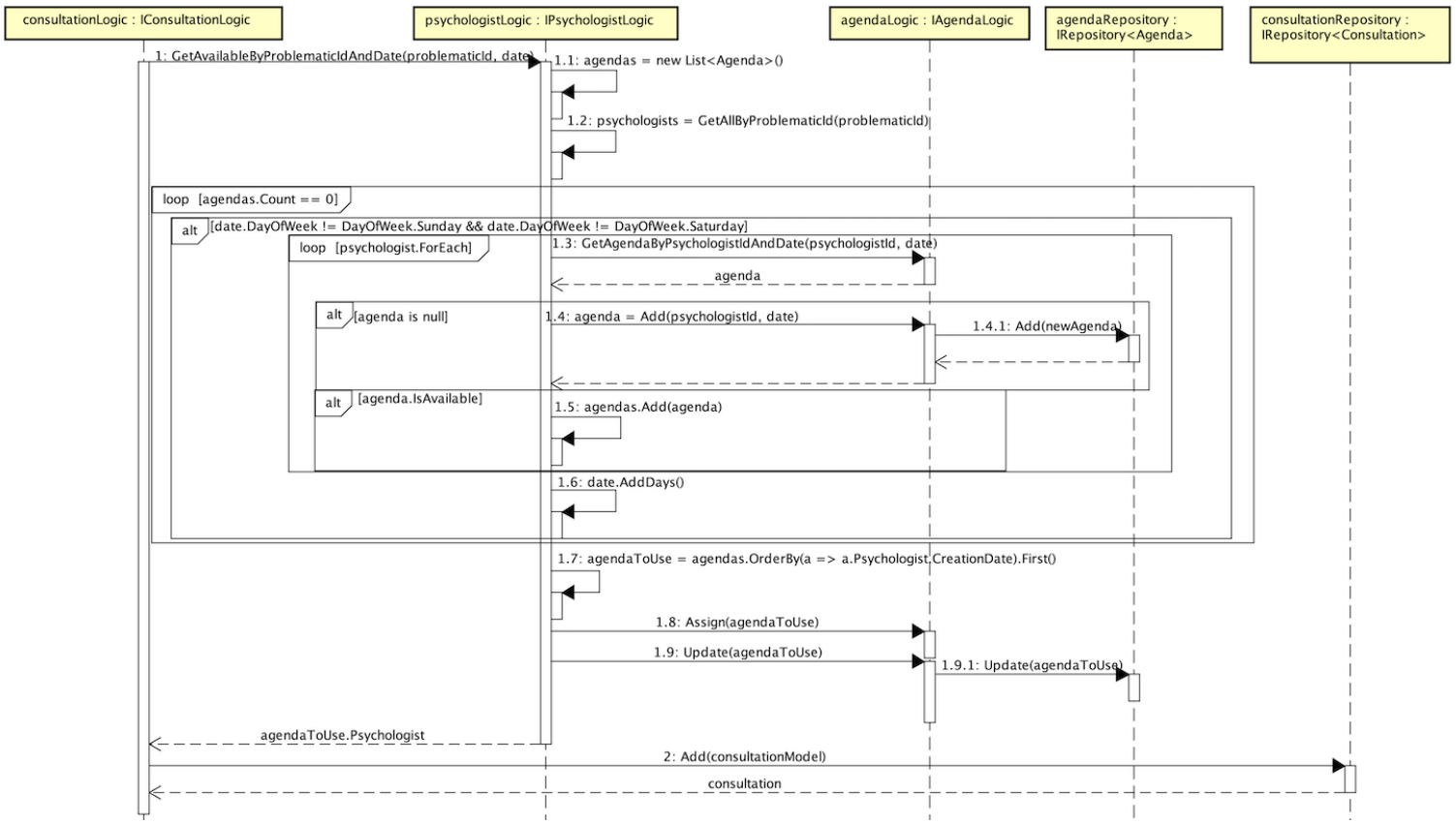
Lógica de agendamiento de consulta con psicólogo

No incluimos la ejecución desde el controller a la lógica ya que su finalidad es realizar los mapeos correspondientes y no aportaba información relevante. La finalidad del siguiente diagrama es aclarar la lógica que se realiza a la hora de agendar una consulta.

Inicialmente se ejecuta el método de la lógica de negocios de psicólogo, este creará una lista de tipo agenda que se compone por un Psicólogo, una fecha, la cantidad de consultas y una variable que indica si el psicólogo se encuentra disponible. Luego obtendrá todos los psicólogos que son expertos en la problemática indicada en la consulta.

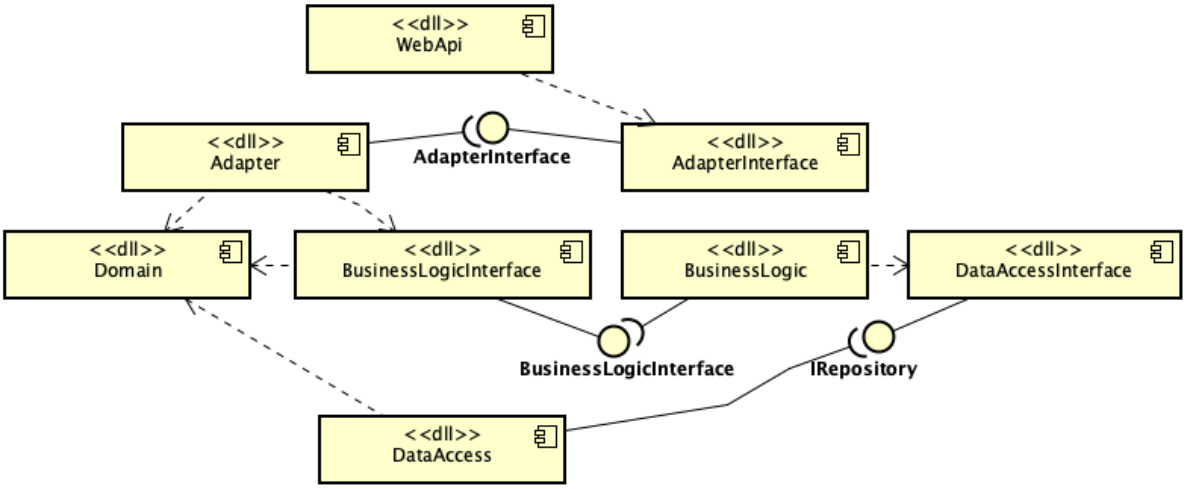
Al tener las dos listas creadas se ejecutará un while recorriendo la agenda a partir de la fecha hasta encontrar un psicólogo que sea experto en la problemática indicada y tenga menos de cinco consultas al día.

Al finalizar la iteración se ordenará la lista obtenida para obtener el psicólogo más antiguo en el sistema, se asignará en la agenda y se modificará en el repositorio. Si el proceso fue satisfactorio se le retornará al usuario el psicólogo que lo atenderá en la consulta.



Vista de Desarrollo

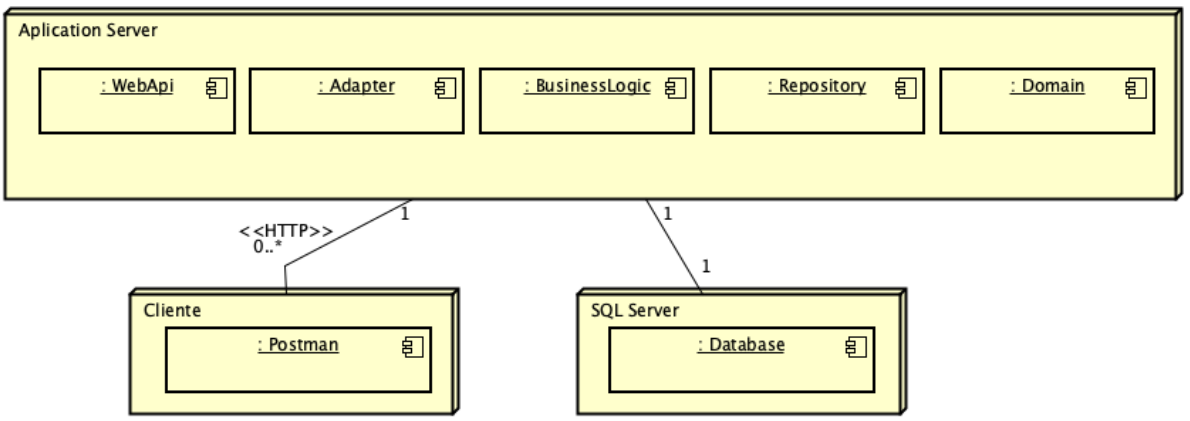
Aquí se mostrará el diagrama de los componentes de nuestro proyecto. Estos son los elementos físicos que se manifiestan en tiempo de ejecución.



Aquí se puede observar como el componente AdapterInterface provee interfaces que son requeridas por Adapter, asimismo BusinessLogicInterface provee interfaces que son requeridas por BusinessLogic y DataAccessInterface provee la interfaz del repositorio a DataAccess.

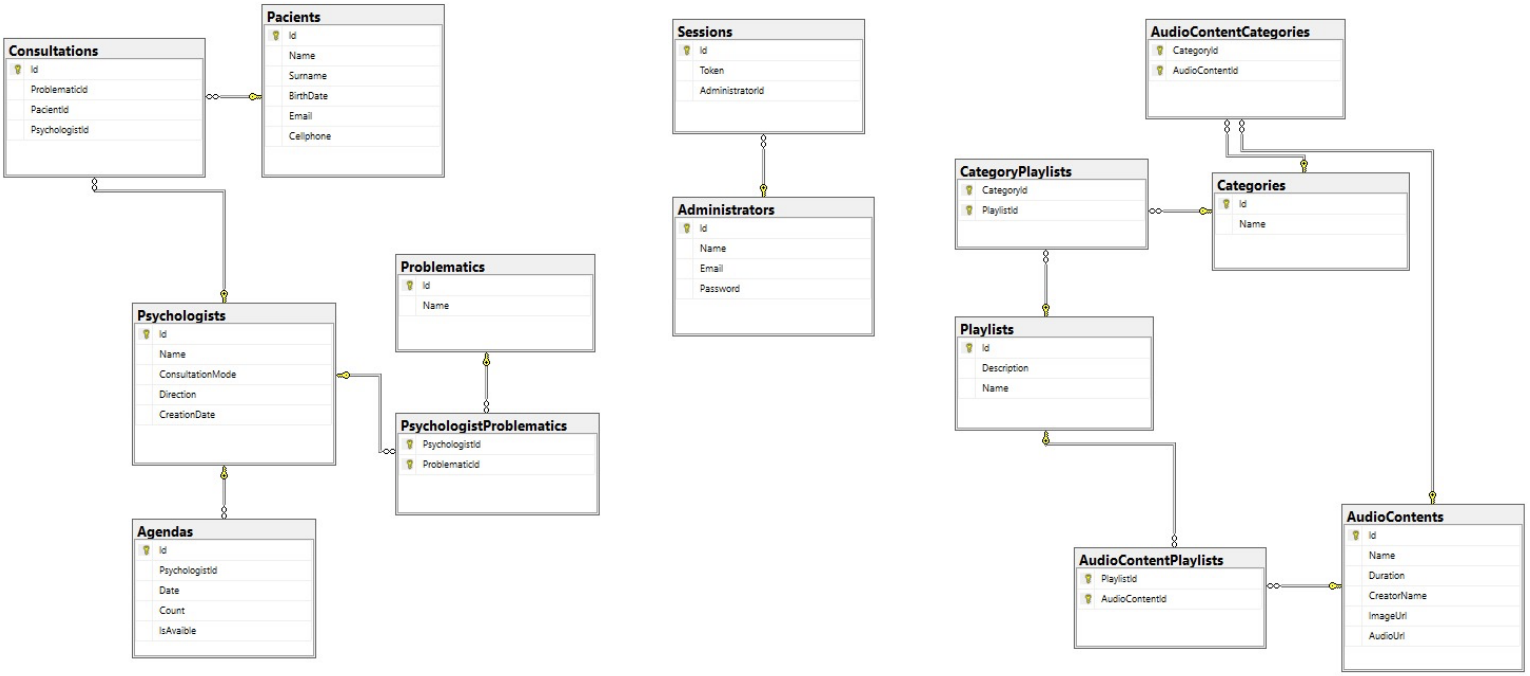
Vista de Despliegue

Aquí mostraremos los artefactos de nuestra aplicación en su ambiente de ejecución.



A simple vista se puede notar que la potencia de nuestro hardware no deberá ser muy grande por el momento.

Modelo de Base de Datos



Anexo

Link a repositorio

https://github.com/ORT-DA2/CarbonellGallicchio.git

Super-Admin

El usuario de correo: admin@gmail.com y contraseña 1234 tiene permisos super admin.