Facultad de Ingeniería

Bernard Wand Polak

Obligatorio 1

Diseño de aplicaciones 2

Lucas Castro - Nº 218709

Ricardo Poladura - Nº 238052

Grupo: N5A

Docentes: Ignacio Valle – Nicolás Fierro – Nicolás Blanco

DESCRIPCIÓN DEL DISEÑO

Introducción

TicketPal[1] es una aplicación que permite administrar las compras de tickets a conciertos en el Antel Arena. Este sistema es utilizado por personas con diferentes perfiles.

El sistema permitirá tener una lista de usuarios donde, según su rol, podrá realizar determinadas acciones. El usuario que tenga un rol de administración podrá realizar el mantenimiento (altas, bajas y modificaciones) de los diferentes datos que se manejan en el sistema (géneros musicales, artistas, conciertos, y usuarios); el que tenga un rol de vendedor, podrán registrar la venta de tickets; el que tenga rol de acomodador, podrá cambiar el estado de un ticket a modificado; y el que tenga rol de espectador, podrá ver datos de los conciertos (aplicando distintos filtros), comprar tickets, ver los tickets que compró y modificar los datos de su cuenta.

Especificaciones técnicas

La aplicación está desarrollada en lenguaje de programación C# utilizando Visual Studio Code y Visual Studio 2022, y para compartir, versionar, actualizar y mantener el código entre los programadores se utiliza GitHub como repositorio. Se utilizó Entity Framework Core para mapear objetos a la base de datos, y para ello, se utilizó la opción Code First para hacerlo, que consiste en escribir el código primero, y luego, a partir del mismo, crear la base de datos de la aplicación.

El sistema operativo utilizado en la implementación es Windows.

El idioma utilizado en el desarrollo de todos los objetos es el inglés.

Alcance

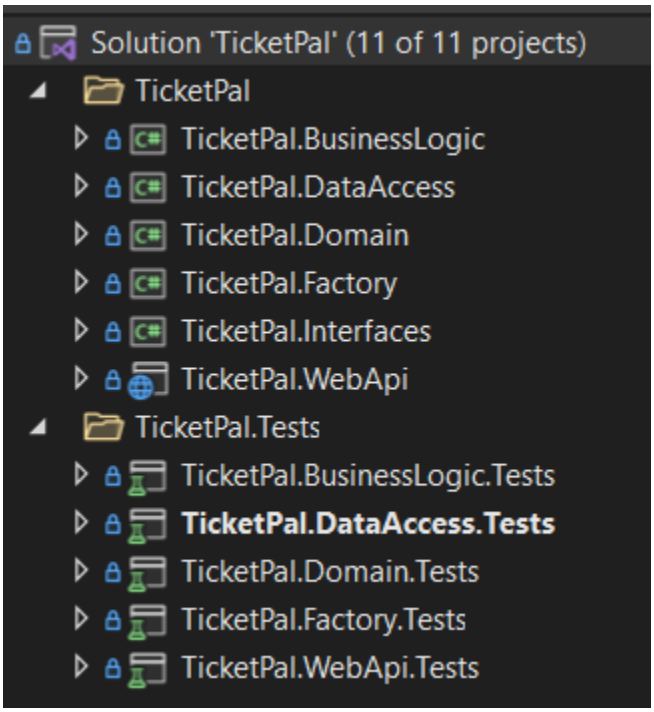
Las funcionalidades de la aplicación son:

- Alta, baja y modificación de usuarios
- Iniciar sesión
- Alta, baja y modificación de conciertos
- Alta, baja y modificación de géneros musicales
- Alta, baja y modificación de artistas
- Alta, baja y modificación de tickets
- Visualización de usuarios, conciertos, géneros musicales, artistas y tickets

[1] https://github.com/ORT-DA2/Castro-Poladura

Diseño y decisiones tomadas respecto al mismo (justificación del diseño)

Para implementar nuestro sistema, dividimos el proyecto en dos módulos (TicketPal y TicketPal.Tests). De esta forma, mantuvimos una clara separación de los proyectos de prueba con los de la solución.



Creamos las clases de tests para cada uno de los módulos de TicketPal, y las usamos como guía en la implementación, de manera que primero generamos las pruebas para cada método y luego implementamos el código, aplicando TDD.

Decidimos separar el módulo TicketPal en otros módulos, para mantener un orden y una estructura más mantenible. Se buscó mantener un bajo acoplamiento y una alta cohesión.

En nuestra solución, utilizamos el patrón de Diseño Factory Method, creando una clase llamada “ServiceFactory”, que nos permite crear objetos sin especificar sus clases concretas; de esta forma, evitamos un acoplamiento fuerte entre el creador y los objetos o productos concretos. El uso de este patrón nos permite cumplir con los principios SOLID de Responsabilidad Única y de Abierto/Cerrado, ya que, al tener una clase para crear objetos, hace que el código sea más fácil de mantener, y se puede incorporar nuevos objetos en el programa sin tener que modificar el código cliente existente.

Ejemplo de utilización de la clase ServiceFactory para cargar una instancia de objeto de TicketCode, que permita generar el código único para el ticket que se va a crear:

```
5 references | 3/3 passing
public OperationResult AddTicket(AddTicketRequest model)
{
    try
    {
        EventEntity newEvent = concertRepository.Get(model.Event);

        TicketEntity found = ticketRepository.Get(t => t.Buyer.Email == model.User.Email && t.Event.Id == model.Event);
        var ticketCode = serviceFactory.GetService(typeof(ITicketCode)) as ITicketCode;
```

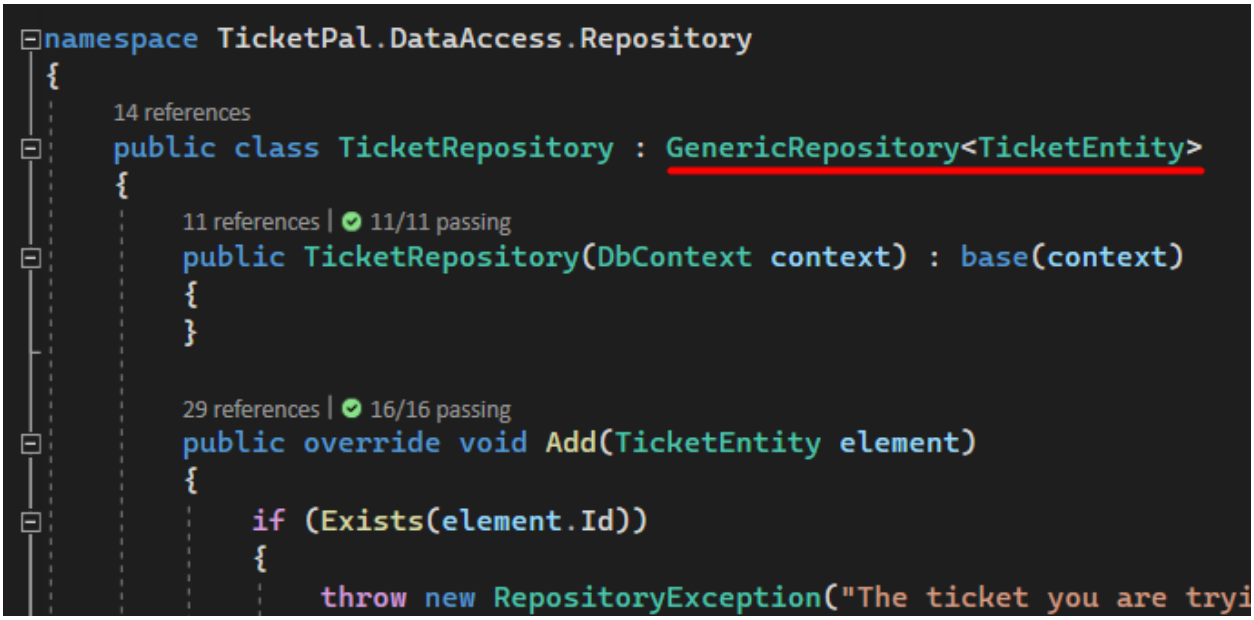
Las clases que consideramos más importantes (por ejemplo, la clase ServiceFactory, las clases de Services en BusinessLogic, las de Repository en DataAccess, y las de Entity en Domain, entre otras), y las que pudiesen tener una razón para cambiar posteriormente, dependen de interfaces o de clases abstractas, y no de implementaciones concretas, para poder garantizar una extensibilidad mayor. Esto nos permite cumplir con el Principio de Segregación de Interfaz y con el de Inversión de Dependencias.

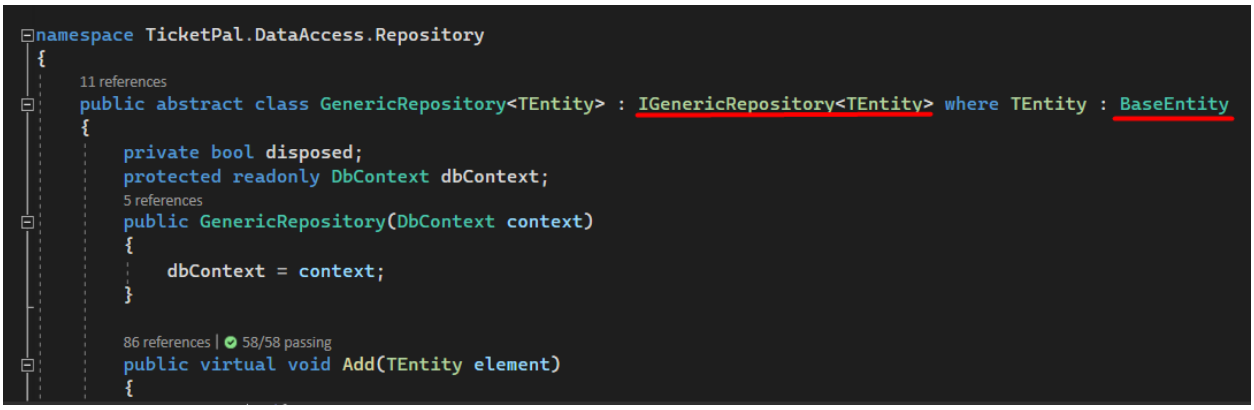
Ejemplos de utilización de interfaces y clases abstractas:

```
namespace TicketPal.BusinessLogic.Services.Performers
{

    public class PerformerService : IPerformerService
    {
        private readonly IServiceFactory serviceFactory;
        private readonly IMapper mapper;
        public IGenericRepository<PerformerEntity> performerRepository;
        public IGenericRepository<GenreEntity> genreRepository;


        public PerformerService(IServiceFactory factory, IMapper mapper)
        {
            this.mapper = mapper;
            this.serviceFactory = factory;
            this.performerRepository = serviceFactory.GetRepository(typeof(
            this.genreRepository = serviceFactory.GetRepository(typeof(Genr
        }
```

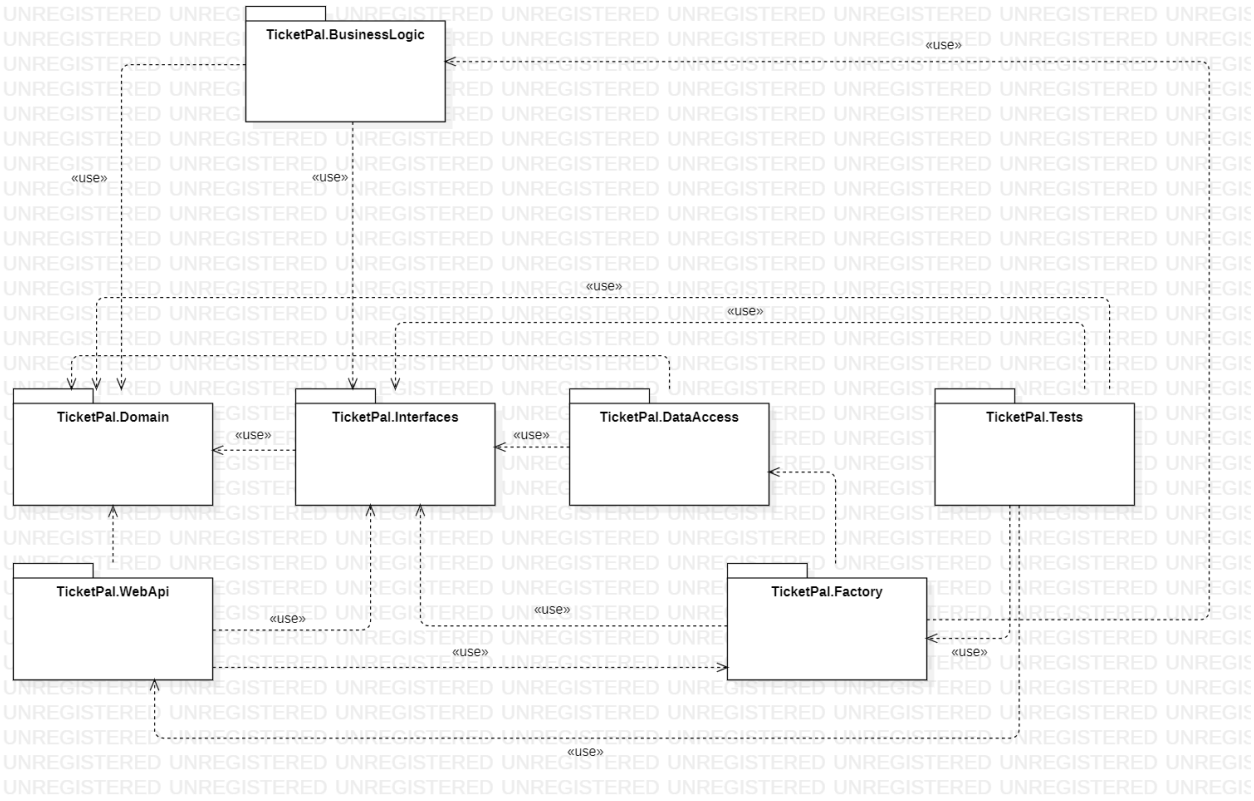




Otro de los principios SOLID aplicados a nuestro obligatorio fue el Principio de Sustitución de Liskov, ya que, al implementar las entidades, creamos una clase abstracta base (llamada BaseEntity), y de la que heredan el resto de las entities creadas. Esto nos permite crear instancias de los subtipos de la clase base, sin alterar el correcto funcionamiento del sistema y pudiendo usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.

Dentro de los bugs o funcionalidades no implementadas se encuentra la de los tickets, ya que por falta de tiempo, no se pudo implementar el controlador correspondiente. Dicha funcionalidad será agregada en la siguiente entrega. También, debido a un problema con los AppSettings, se debió hardcodear el string de JWSecret para poder generar el token, funcionalidad que también se implementará por completo en la siguiente entrega.

Diagrama de paquetes



El módulo de **TicketPal** se compone de varios módulos, que son **BusinessLogic**, **DataAccess**, **Domain**, **Factory**, **Interfaces**, y **WebApi**.

El módulo de **BusinessLogic** es el que contiene la lógica de negocios del sistema, y se compone de varios paquetes, que son Mapper, Services, y Utils.

Mapper es el encargado de mapear las clases de Servicios con las clases de Responses.

Services está subdividido en otros paquetes, que son Concerts, Genres, Jwt, Performers, Settings, Tickets y Users, que contienen cada uno de los servicios correspondientes. Los servicios Concerts, Genres, Performers, Tickets y Users, son los que contienen la lógica de negocios en la que se basa el sistema. Jwt y Settings tienen una función de configuración para el funcionamiento de la aplicación.

Y por último, Utils contiene el paquete TicketsCode, que es el encargado de generar el identificador único que tienen los tickets.

El módulo de **DataAccess** es el que vincula el sistema con la base de datos, y se compone del paquete Repository, que es donde están cada uno de los repositorios, encargados de llevar las consultas del sistema hacia la base de datos, y recibir las respuestas solicitadas.

El módulo de **Domain** es el encargado de las entities que van a estar conteniendo los datos que devuelva el módulo DataAccess cuando se acceda a la base de datos, así como también contiene las clases de Requests y Responses, la clase de enumerables que se necesitan para el sistema, y la clase de manejo de excepciones. Se compone de los paquetes Constants (que contiene la clase de enumerables), Entity (que contiene los DTOs), Exceptions, y Models, que a su vez, se compone de los paquetes Requests y Responses.

El módulo Factory es el encargado de crear objetos del sistema, sin especificar sus clases concretas, lo que nos permite evitar un acoplamiento fuerte entre el creador y los objetos concretos.

El módulo Interfaces es el encargado de contener todas las interfaces usadas en el sistema, y se compone de los paquetes Factory, Repository, Services y Utils. En cada uno de estos paquetes, se encuentran los repositorios correspondientes a los respectivos paquetes de otros módulos del sistema. El paquete Services, a su vez, se compone de otros paquetes que son Concerts, Genres, Jwt, Performers, Settings, Tickets y Users. Y el paquete Utils, se compone del paquete TicketsCode.

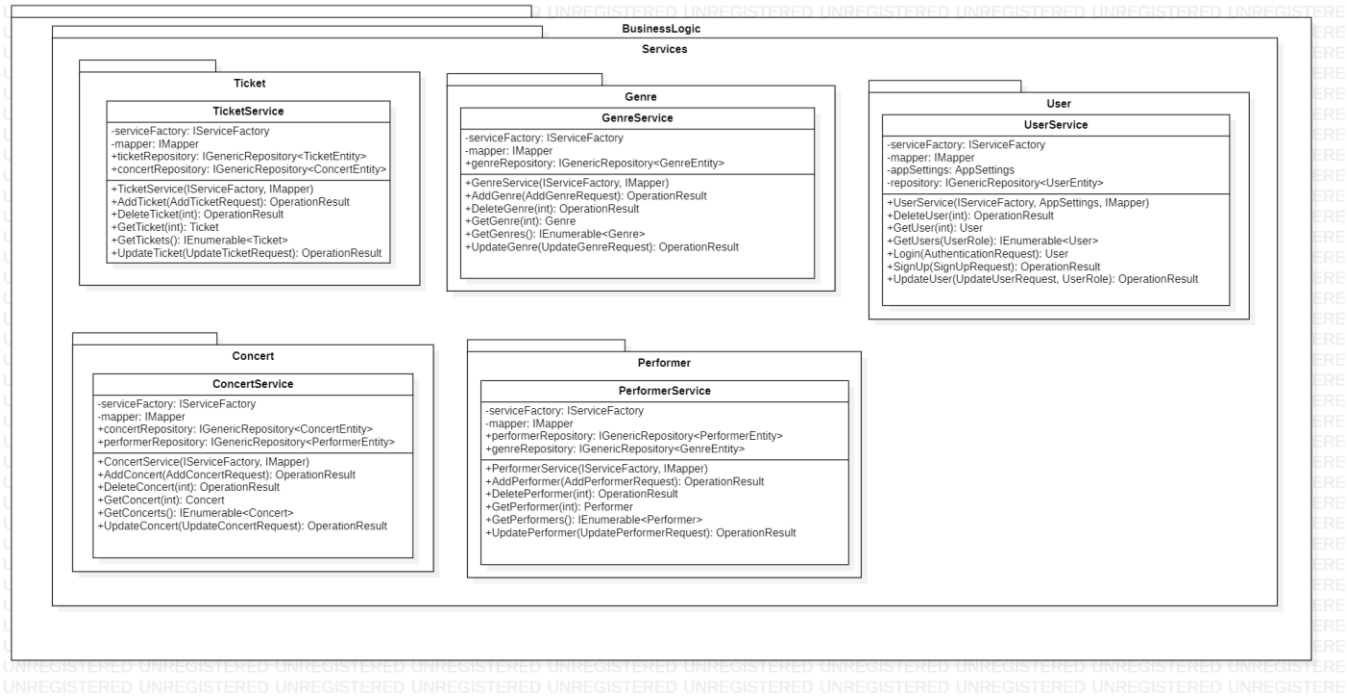
Y por último, el módulo WebApi es el que contiene todo lo relativo a la webapi, y es la responsable de manejar los filtros, los controladores, y los diferentes roles de usuarios. Se compone de los paquetes Properties, Constants, Controllers y Filters, que a su vez, se compone de los paquetes Auth y Model.

A su vez, como módulo aparte de TicketPal, tenemos el módulo de tests (**TicketPal.Tests**), que se compone por **BusinessLogic.Tests**, **DataAccess.Tests**, **Domain.Tests**, **Factory.Tests**, y **WebApi.Tests**, y éstos contienen los tests de cada uno de los módulos asociados de TicketPal.

Diagrama de clases UML

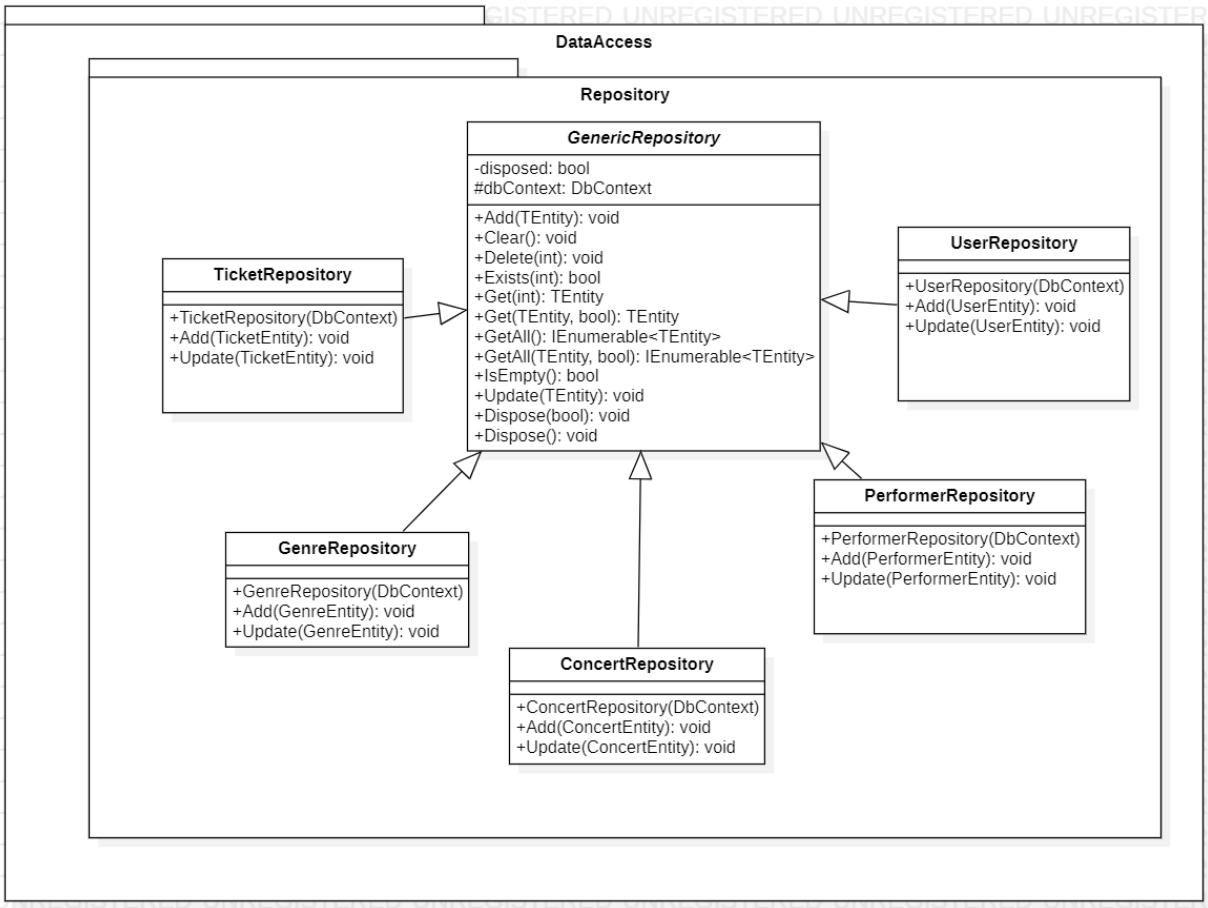
Teniendo en cuenta los paquetes más relevantes para el sistema, se realizaron los siguientes diagramas de clases[2]:

Módulo BusinessLogic:

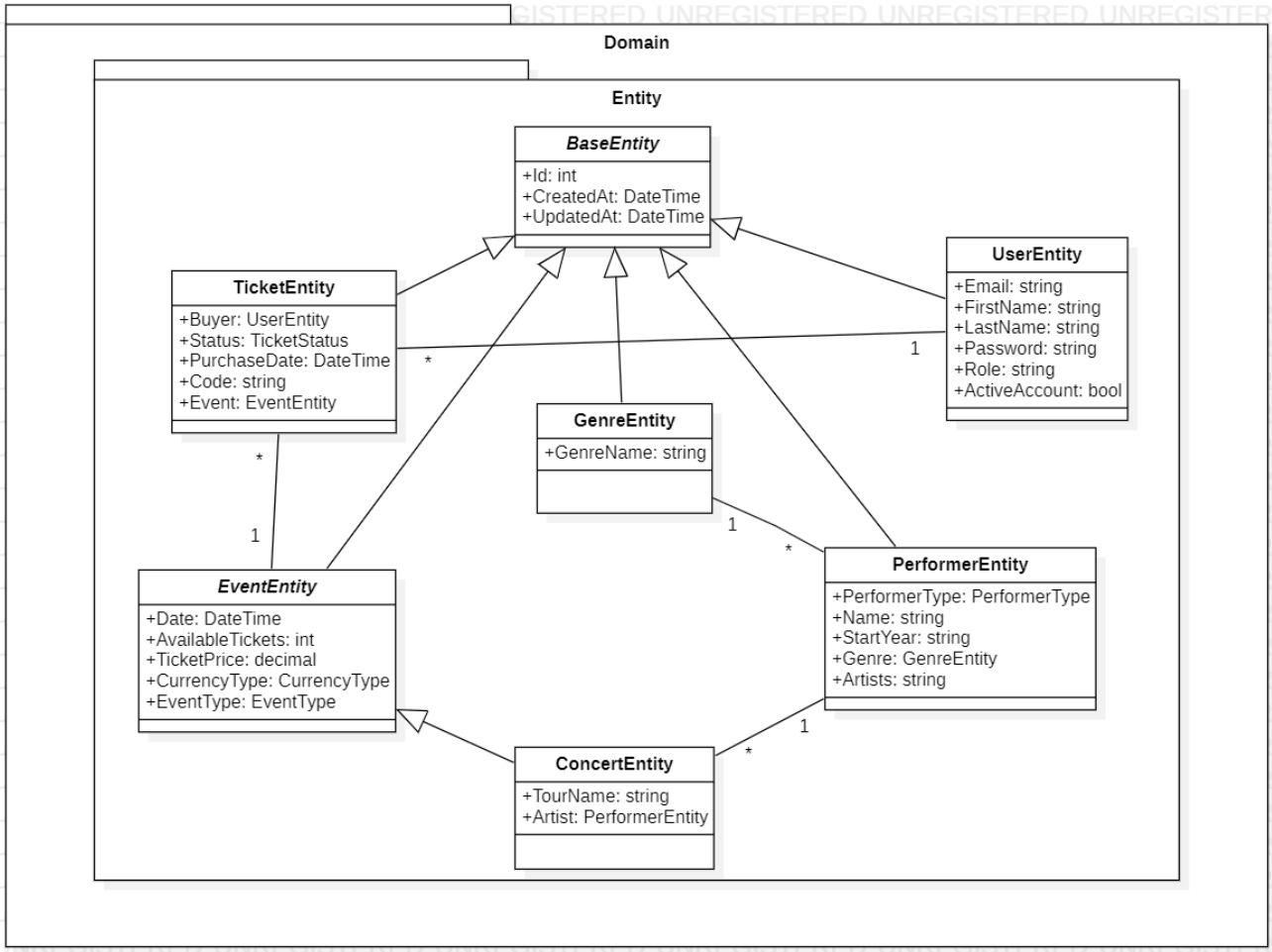


[2] Los diagramas de clases pueden verse más claramente aquí: https://github.com/ORT-DA2/Castro-Poladura/tree/develop/Documentación/UML

Módulo DataAccess:

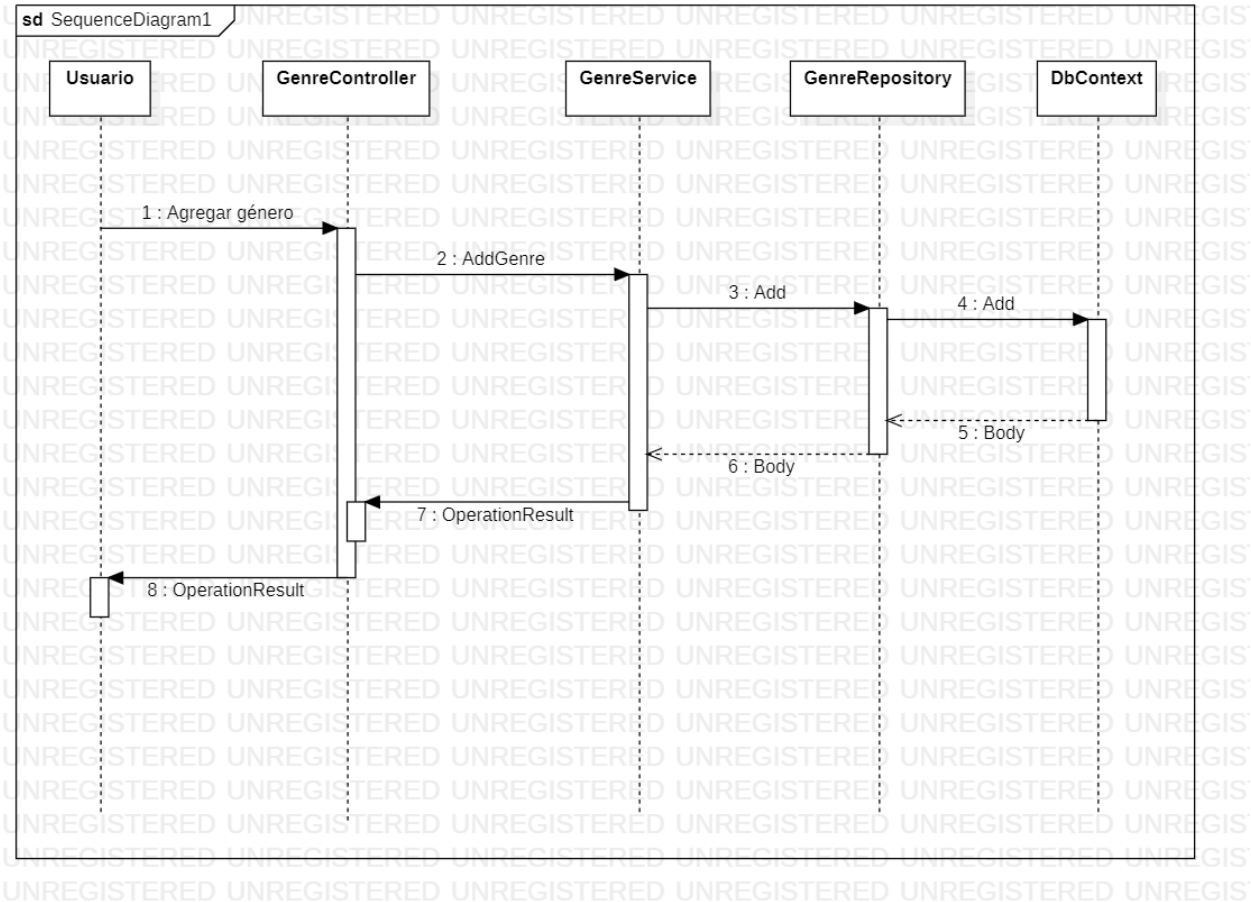


Módulo Domain:



Diagramas de Interacción

Por falta de tiempo, se agrega el diagrama de interacción de una de las funcionalidades, que es la de agregar un género musical, pero que resume la secuencia de las demás funcionalidades, ya que la implementación fue similar para todas.



Modelo de la estructura de tablas de la base de datos