Facultad de Ingeniería

Bernard Wand Polak

Obligatorio 2

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; 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; y el que tenga un rol de artista, puede ver todos los conciertos que realizó y ver si ese concierto vendió todas las entradas que tenía previsto o no.

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

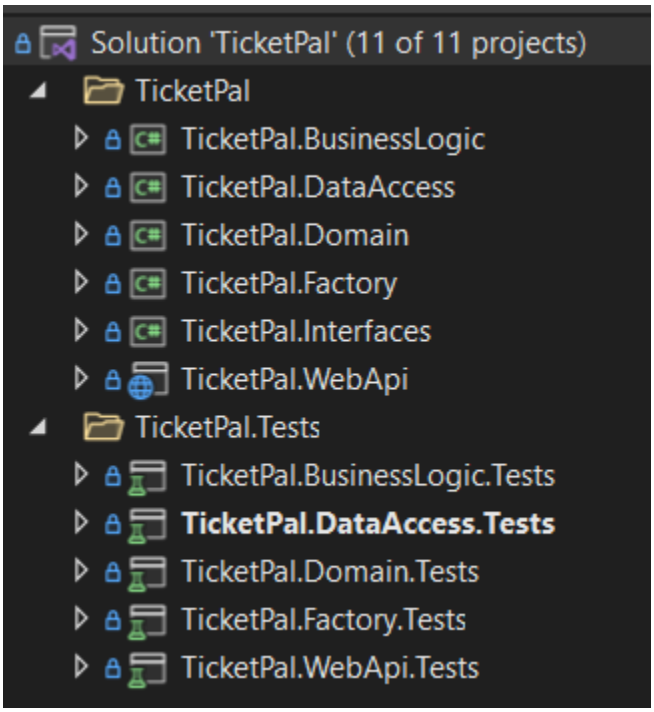
Las funcionalidades de la aplicación son:

- 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
        }
```

```
namespace TicketPal.DataAccess.Repository
{
    14 references
    public class TicketRepository : GenericRepository<TicketEntity>
    {
        11 references | 11/11 passing
        public TicketRepository(DbContext context) : base(context)
        {
        }

        29 references | 16/16 passing
        public override void Add(TicketEntity element)
        {
            if (Exists(element.Id))
            {
                throw new RepositoryException("The ticket you are tryi
```

```
namespace TicketPal.DataAccess.Repository
{
    11 references
    public abstract class GenericRepository<TEntity> : IGenericRepository<TEntity> where TEntity : BaseEntity
    {
        private bool disposed;
        protected readonly DbContext dbContext;
        5 references
        public GenericRepository(DbContext context)
        {
            dbContext = context;
        }

        86 references | 58/58 passing
        public virtual void Add(TEntity element)
        {
```

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.

Con respecto a la extensibilidad que se solicitó para la exportación e importación de conciertos, creamos una interfaz llamada IExportImport, que es la que actúa como “contrato” entre nuestra API y las aplicaciones de terceros que quieran implementar esta nueva función de exportación/importación. Esta interfaz es una dll externa, tanto a nuestro proyecto como al de terceros, lo que nos permite escribir código que puede ser reusado por otra app, sin necesidad de conocer su código.

```
public interface IExportImport<T>
{
    0 references
    string Name { get; set; }
    0 references
    void Export(string destinationPath, List<T> export);
    0 references
    List<T> Import(string importPath);
}
```

Como la interfaz utiliza una lista de objetos “T”, los terceros no tienen que conocer que existe un objeto Concierto, y aún así, pueden aplicar su lógica de implementación de los métodos de Exportación e Importación.

Nuestra clase ExportImportDelegator, que es la encargada de “conectar” con la API de terceros, tiene como referencia a la dll “ExportImport”. A través del método ExportImportConcerts, que recibe como parámetros la acción a realizarse (exportar o importar), y el formato solicitado a través del frontend, se obtiene la instancia de la implementación que se necesita, buscando el assembly que coincida su propiedad “Name” con el parámetro “Format”, que se le pasa a la función. Ahí se obtiene la instancia de la implementación correcta, y de acuerdo a la acción solicitada (exportar o importar), se llama al correspondiente método desde la dll de terceros.

De esta forma, se cumple con: el Principio de Responsabilidad Única, ya que la clase ExportImportDelegator sólo se encarga de la función de exportación e importación; con el Principio de Abierto/Cerrado, ya que se pueden agregar nuevas dlls de terceros, y eso aumentaría las funcionalidades de nuestra app, sin tener que modificar código; con el Principio de segregación de interfaz, porque las aplicaciones de terceros no van a depender de interfaces que no utilizan, sino que por el contrario, dicha interfaz es la que oficia como contrato entre ellas y nuestra aplicación; con el Principio de inversión de dependencia, ya que no se depende de implementaciones concretas, sino de abstracciones.

Todo esto nos permite lograr un incremento de la cohesión, un bajo acoplamiento, tener un código extensible y robusto, y con menos posibilidades de introducir errores en la aplicación existente, además de una menor complejidad del código.

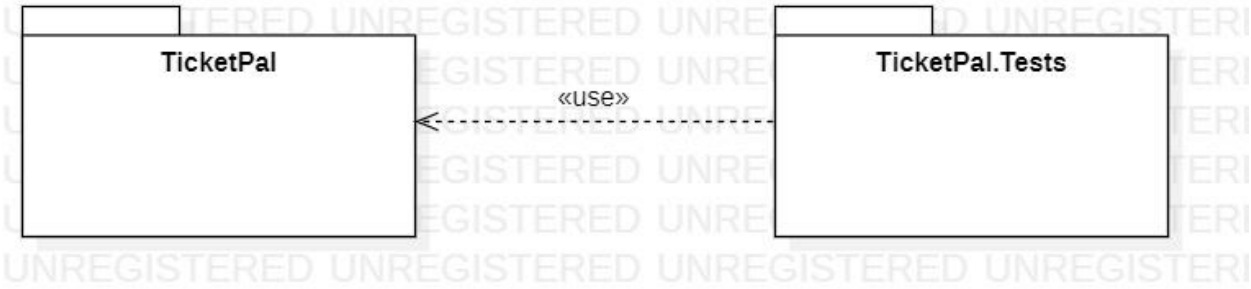
Con respecto a los bugs conocidos o a las funcionalidades no implementadas, por falta de tiempo, no se pudo implementar el crear conciertos y performers a través del frontend. En el caso de los conciertos, tampoco se pudo implementar el borrado de los mismos, por una falla que no nos fue posible solucionar antes de la entrega.

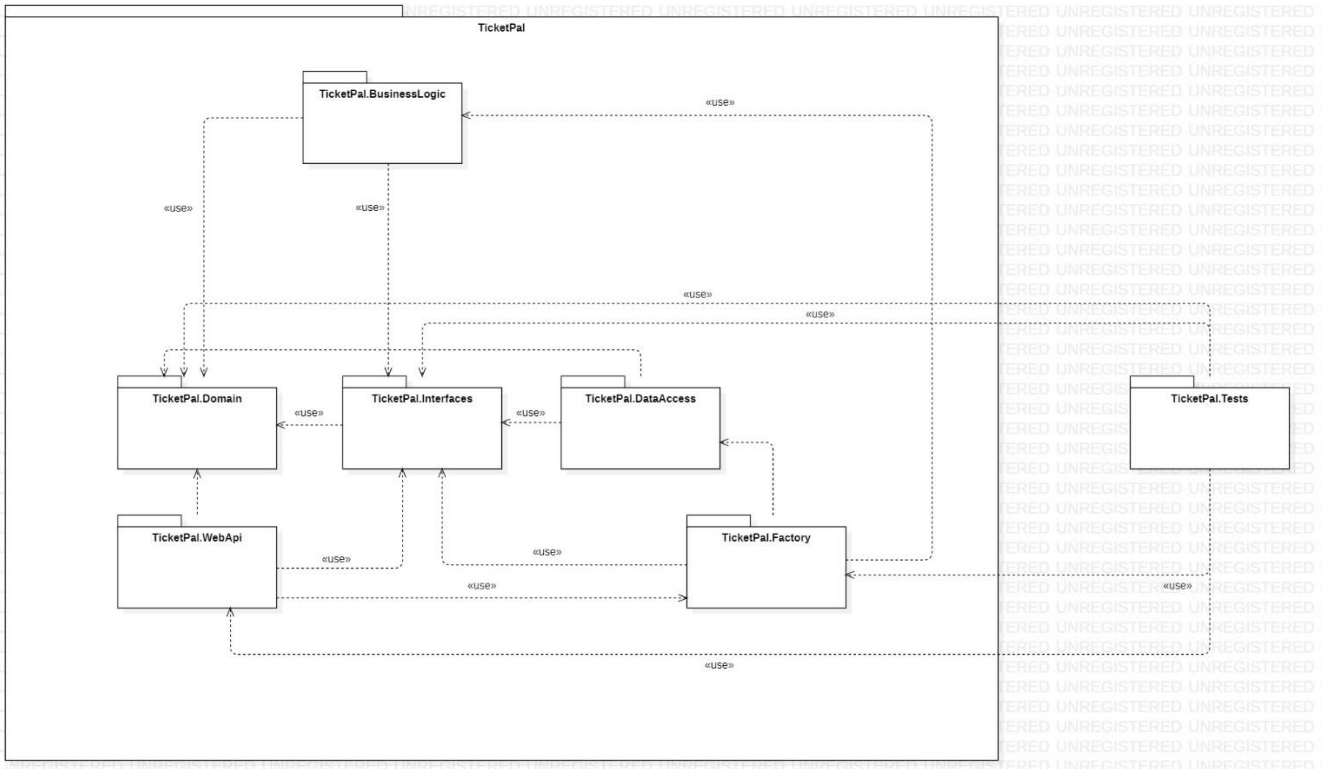
También por falta de tiempo, tampoco fue posible implementar el frontend de la feature de exportación/importación, aunque sí se realizó el desarrollo de la misma en backend, y se agregó a una carpeta “lib” las dlls del contrato que utilizamos (ExportImport), y de la funcionalidad de exportación/importación en formato Json, realizada por “terceros”.

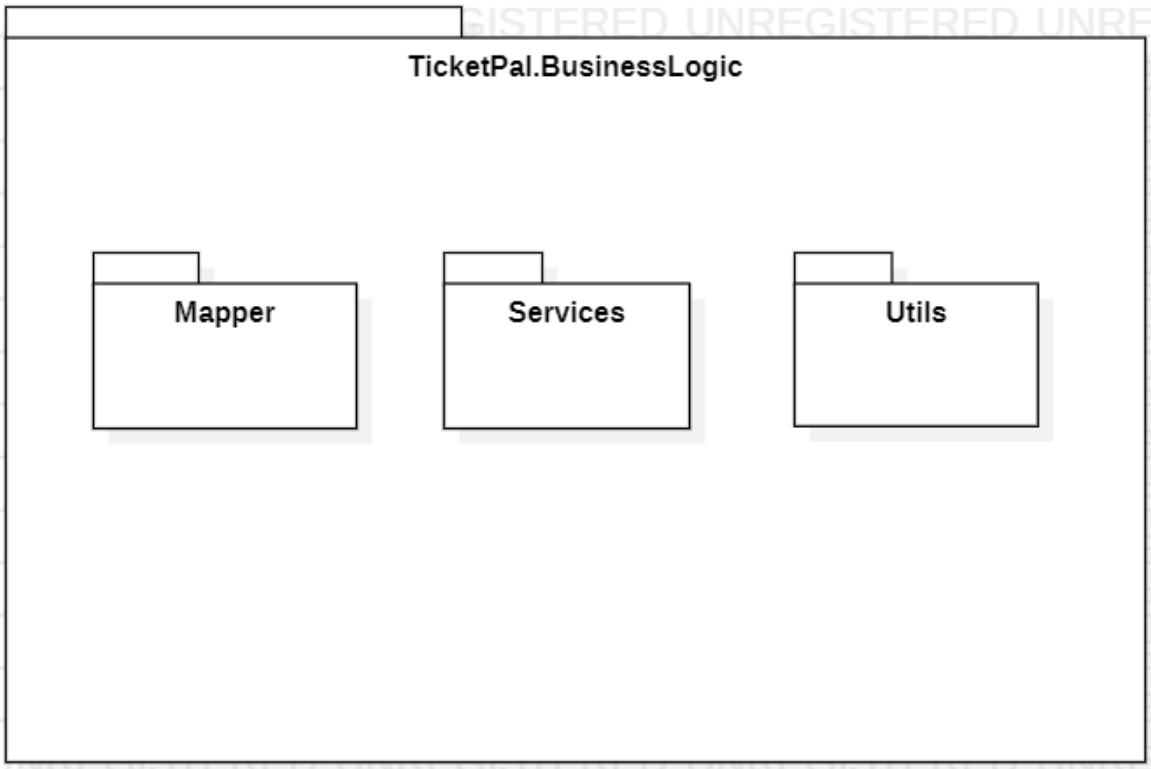
Otro bug que encontramos al actualizar la especificación de la API fue que el endpoint de “Obtener eventos de un performer” no cumple con REST, ya que tendría que ser “performer/events”, en vez

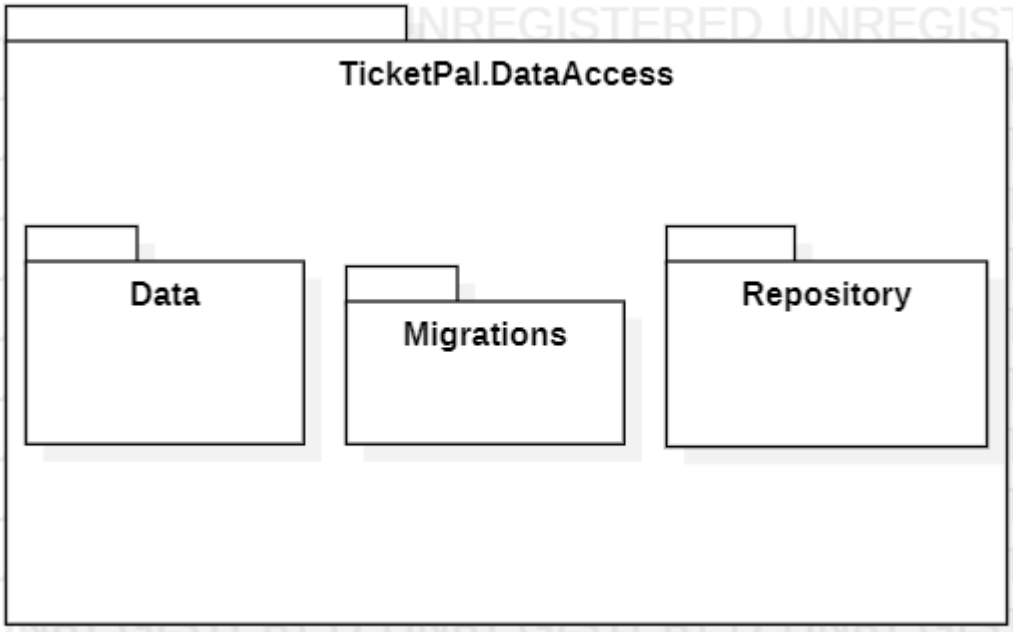
de “events/performers”. Esto se debió a un error de tipeo y se corregiría para una siguiente versión, debido a que, por plazo de la entrega, no podemos realizar el cambio.

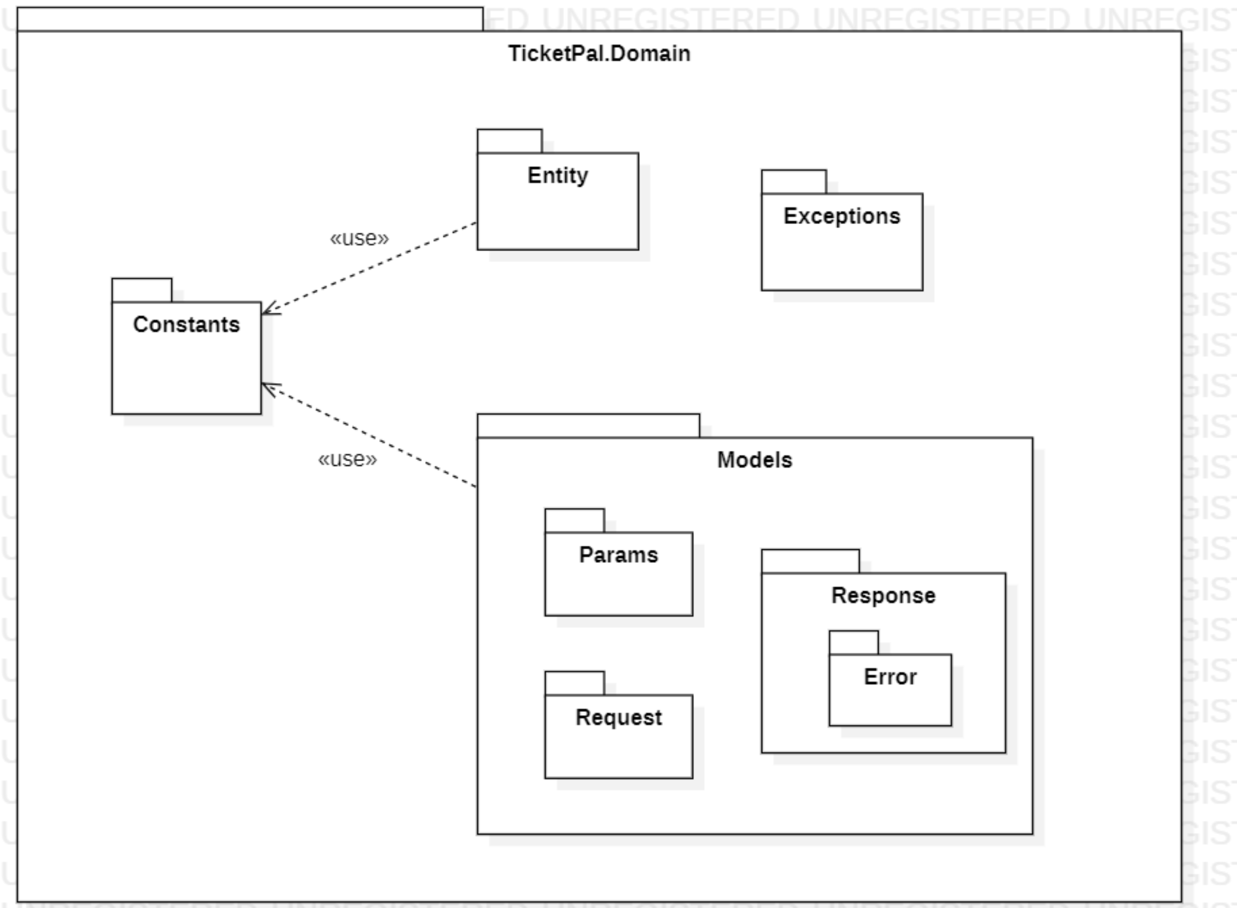
Diagrama de paquetes

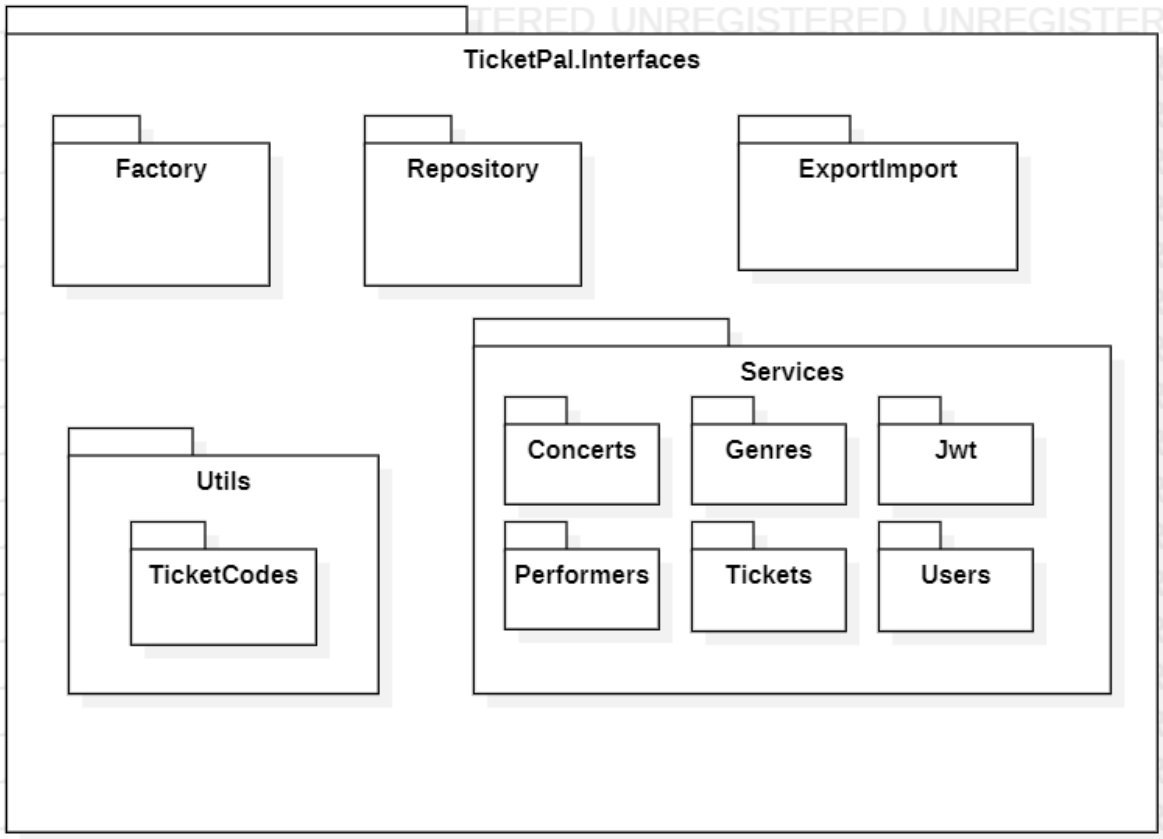


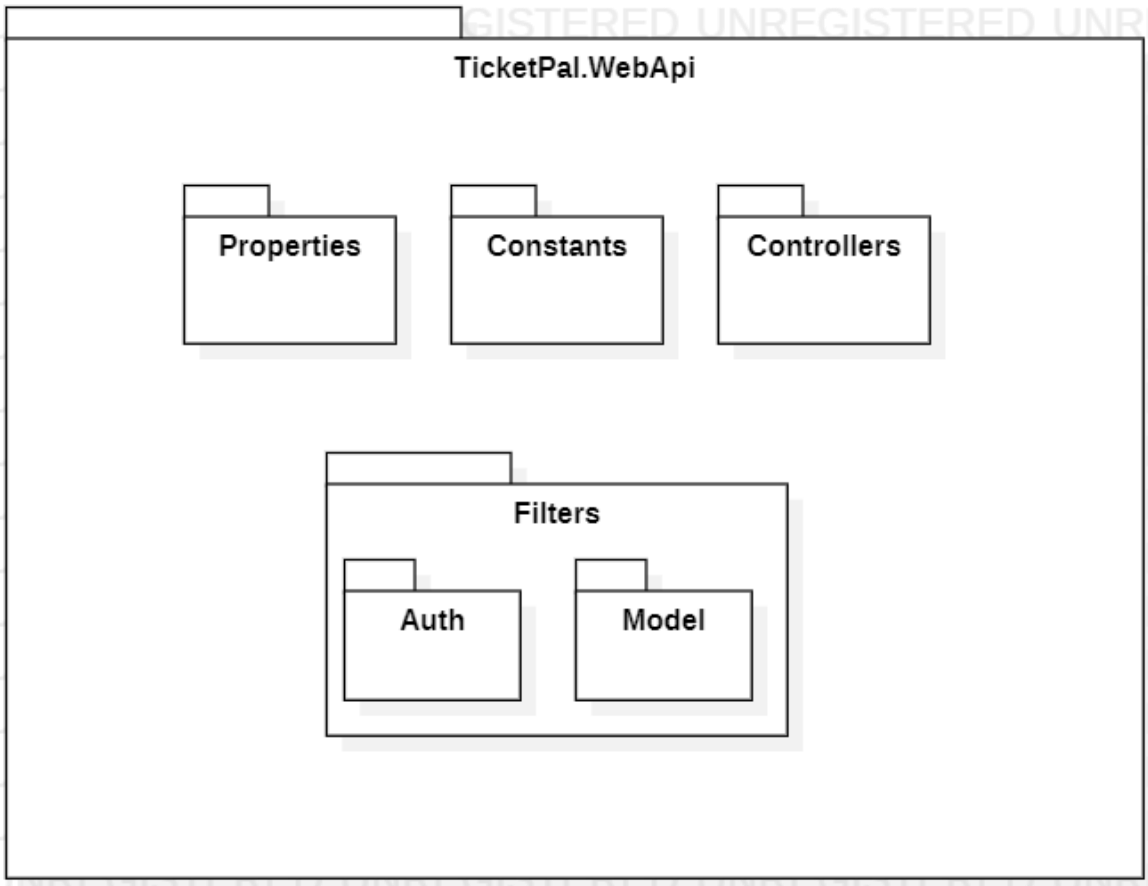



TicketPal.BusinessLogic
Mapper
Services
Utils


TicketPal.DataAccess
Data
Migrations
Repository







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, Delegator, 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 Params, 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 Params, 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 ExportImport, 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.

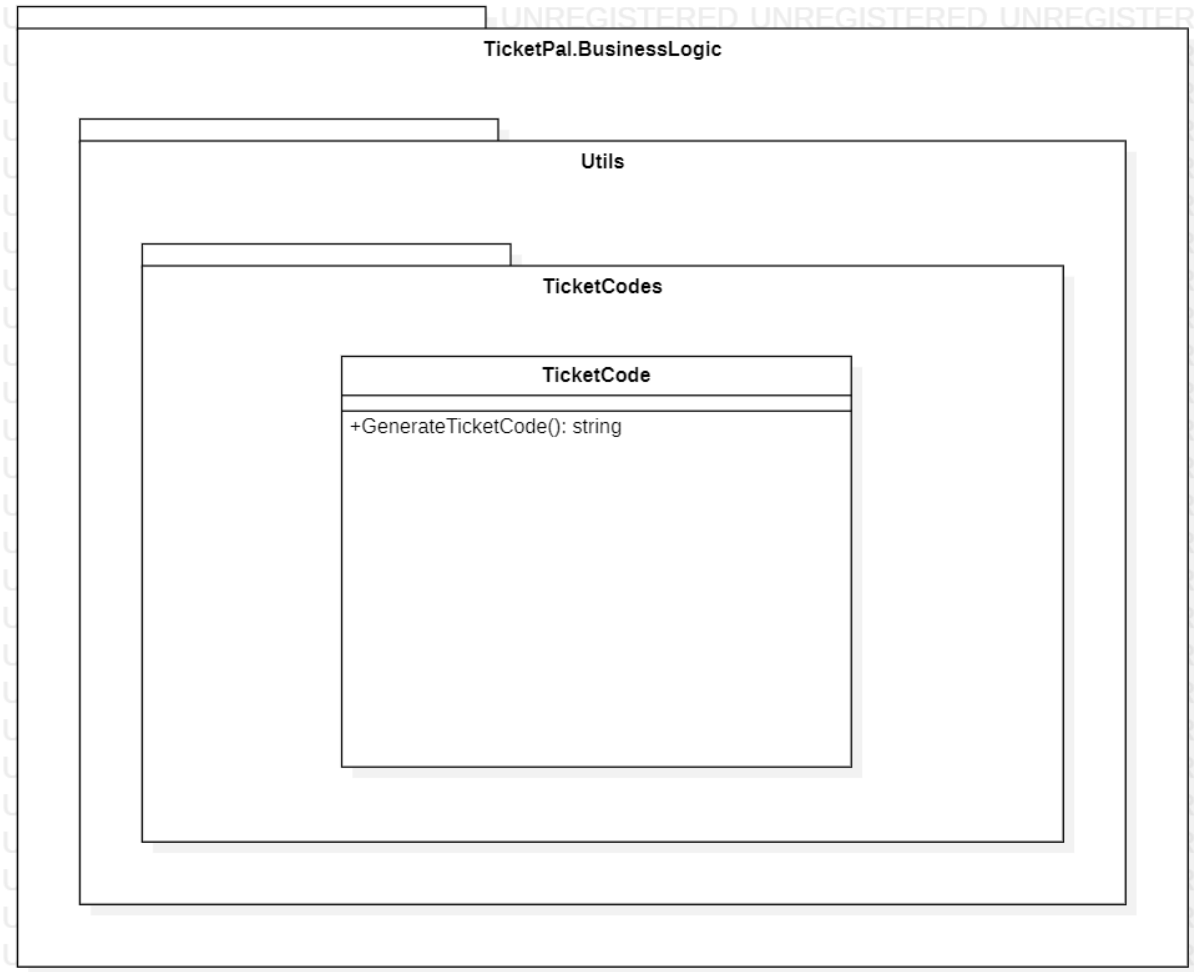
A su vez, como módulo aparte de TicketPal, tenemos el módulo de tests (**TicketPal.Tests**), que se compone por **BusinessLogic.Tests**, **DataAccess.Tests**, **Domain.Tests**, **Factory.Tests**, y **WebApi.Tests**, y éstos contienen los tests de cada uno de los módulos asociados de TicketPal.

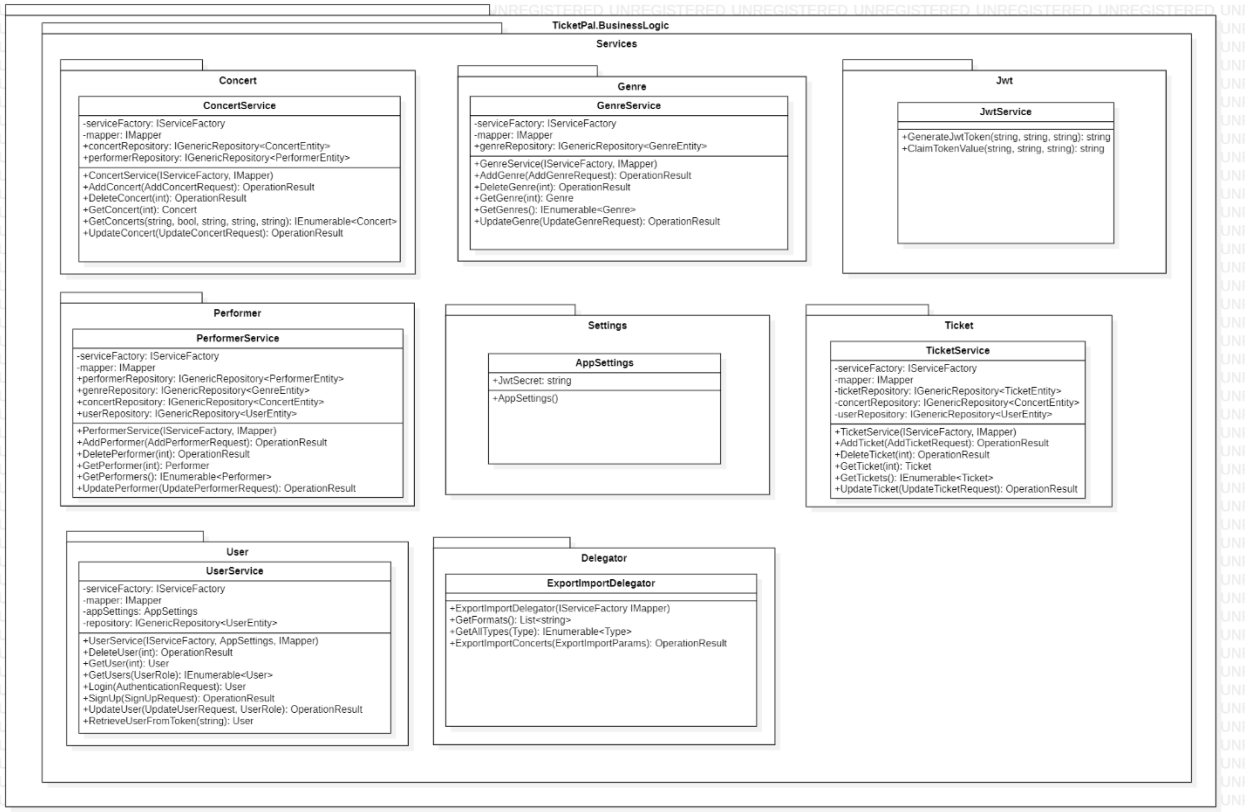
Diagrama de clases UML

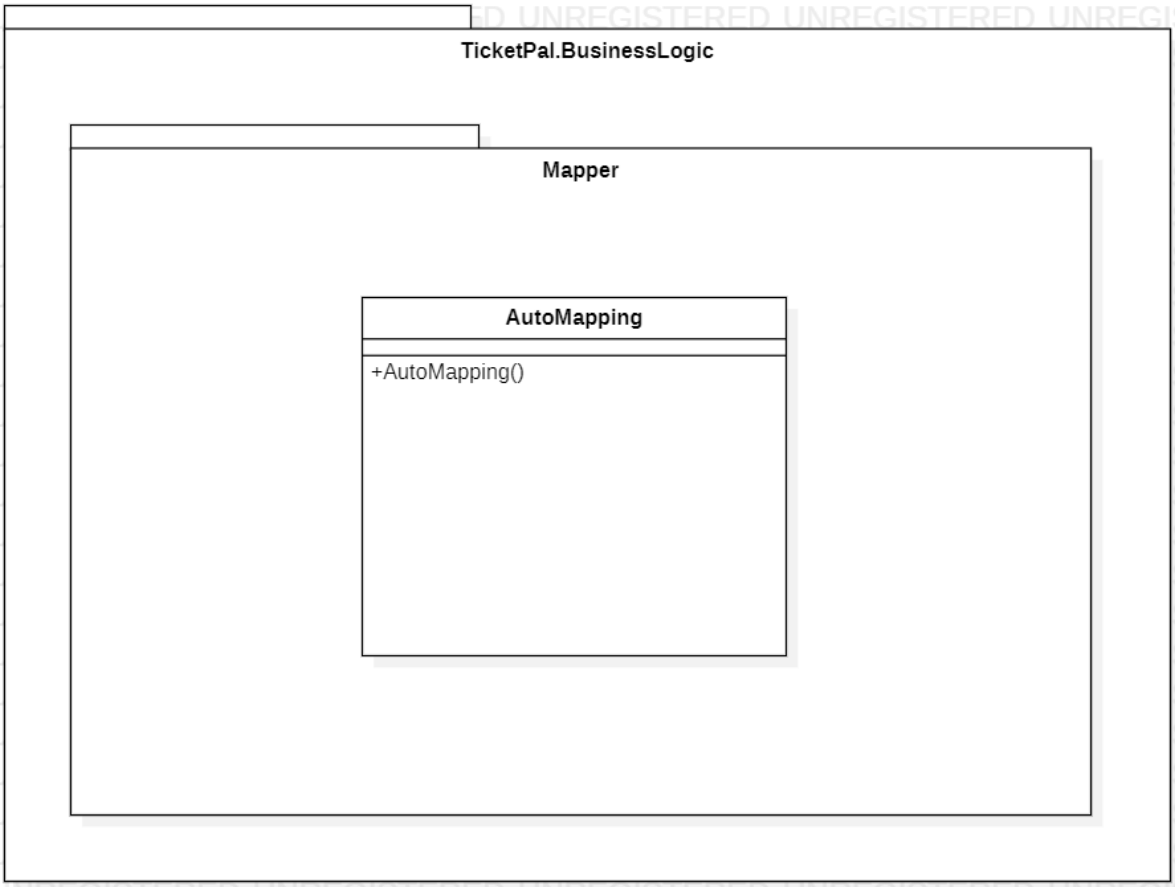
Teniendo en cuenta los paquetes más relevantes para el sistema, se realizaron los siguientes diagramas de clases[2]:

Módulo BusinessLogic:

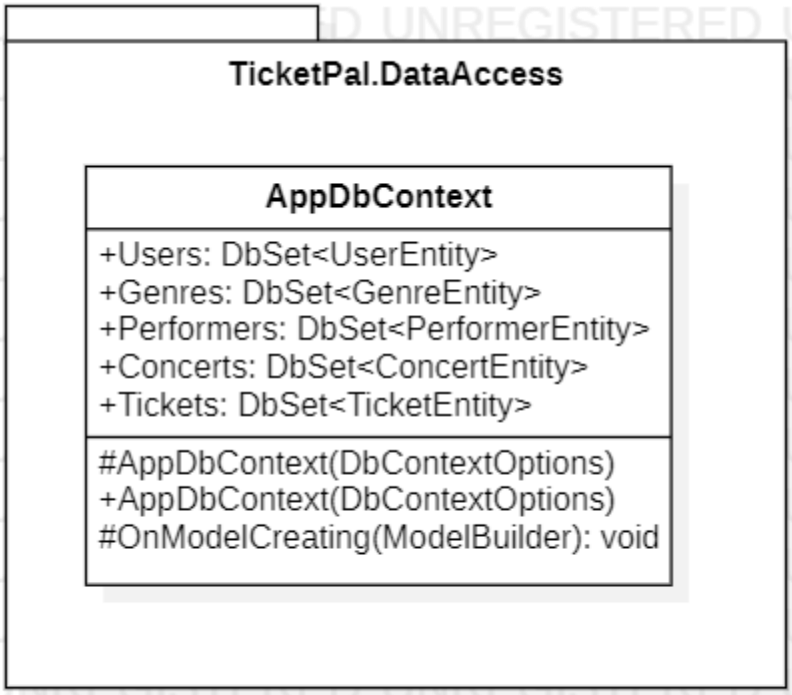
[2] Los diagramas de clases pueden verse más claramente aquí: https://github.com/ORT-DA2/Castro-Poladura/tree/develop/Documentación/UML

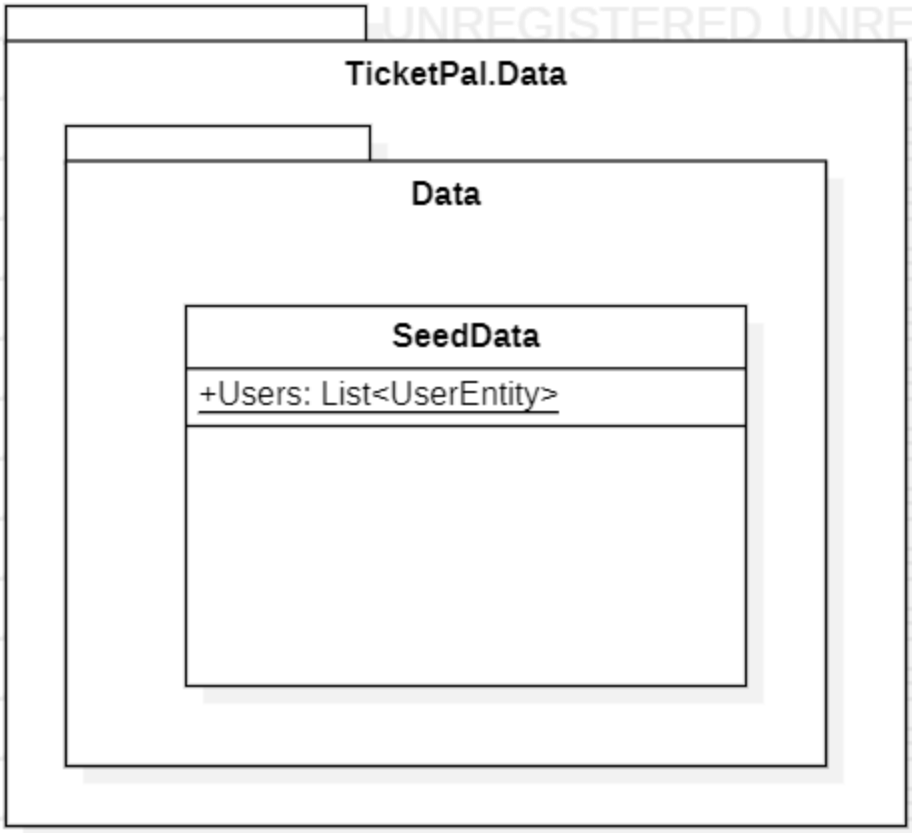


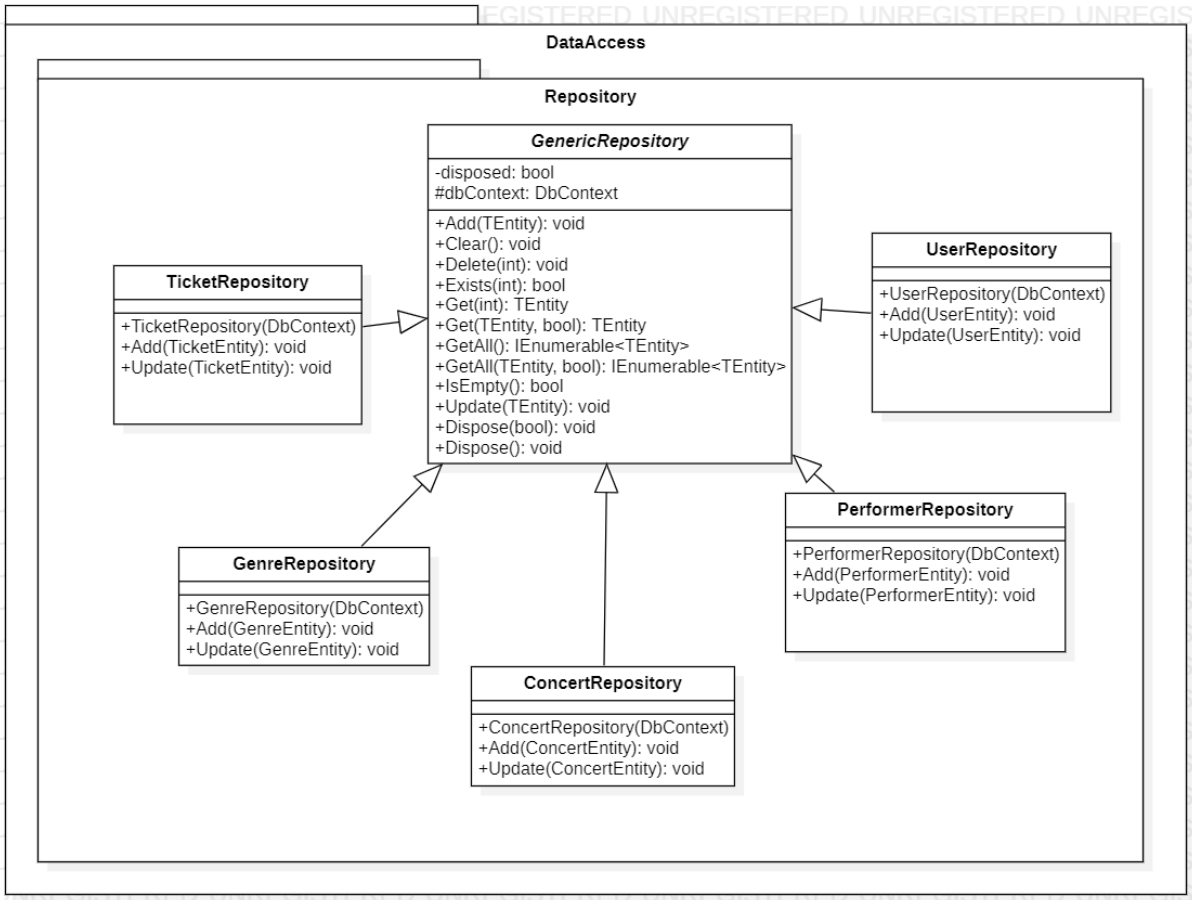

TicketPal.BusinessLogic
Services
Concert
ConcertService
-serviceFactory: IServiceFactory
-mapper: IMapper
+concertRepository: IGenericRepository<ConcertEntity>
+performerRepository: IGenericRepository<PerformerEntity>
+ConcertService(IServiceFactory, IMapper)
+AddConcert(AddConcertRequest): OperationResult
+DeleteConcert(int): OperationResult
+GetConcert(int): Concert
+GetConcerts(string, bool, string, string, string): IEnumerable<Concert>
+UpdateConcert(UpdateConcertRequest): OperationResult
Genre
GenreService
-serviceFactory: IServiceFactory
-mapper: IMapper
+genreRepository: IGenericRepository<GenreEntity>
+GenreService(IServiceFactory, IMapper)
+AddGenre(AddGenreRequest): OperationResult
+DeleteGenre(int): OperationResult
+GetGenre(int): Genre
+GetGenres(): IEnumerable<Genre>
+UpdateGenre(UpdateGenreRequest): OperationResult
Jwt
JwtService
+GenerateJwtToken(string, string, string): string
+ClaimTokenValue(string, string, string): string
Performer
PerformerService
-serviceFactory: IServiceFactory
-mapper: IMapper
+performerRepository: IGenericRepository<PerformerEntity>
+genreRepository: IGenericRepository<GenreEntity>
+concertRepository: IGenericRepository<ConcertEntity>
+userRepository: IGenericRepository<UserEntity>
+PerformerService(IServiceFactory, IMapper)
+AddPerformer(AddPerformerRequest): OperationResult
+DeletePerformer(int): OperationResult
+GetPerformer(int): Performer
+GetPerformers(): IEnumerable<Performer>
+UpdatePerformer(UpdatePerformerRequest): OperationResult
Settings
AppSettings
+JwtSecret: string
+AppSettings()
Ticket
TicketService
-serviceFactory: IServiceFactory
-mapper: IMapper
-ticketRepository: IGenericRepository<TicketEntity>
-concertRepository: IGenericRepository<ConcertEntity>
-userRepository: IGenericRepository<UserEntity>
+TicketService(IServiceFactory, IMapper)
+AddTicket(AddTicketRequest): OperationResult
+DeleteTicket(int): OperationResult
+GetTicket(int): Ticket
+GetTickets(): IEnumerable<Ticket>
+UpdateTicket(UpdateTicketRequest): OperationResult
User
UserService
-serviceFactory: IServiceFactory
-mapper: IMapper
-appSettings: AppSettings
-repository: IGenericRepository<UserEntity>
+UserService(IServiceFactory, AppSettings, IMapper)
+DeleteUser(int): OperationResult
+GetUser(int): User
+GetUsers(UserRole): IEnumerable<User>
+Login(AuthenticationRequest): User
+SignUp(SignUpRequest): OperationResult
+UpdateUser(UpdateUserRequest, UserRole): OperationResult
+RetrieveUserFromToken(string): User
Delegator
ExportImportDelegator
+ExportImportDelegator(IServiceFactory IMapper)
+GetFormats(): List<string>
+GetAllTypes(Type): IEnumerable<Type>
+ExportImportConcerts(ExportImportParams): OperationResult



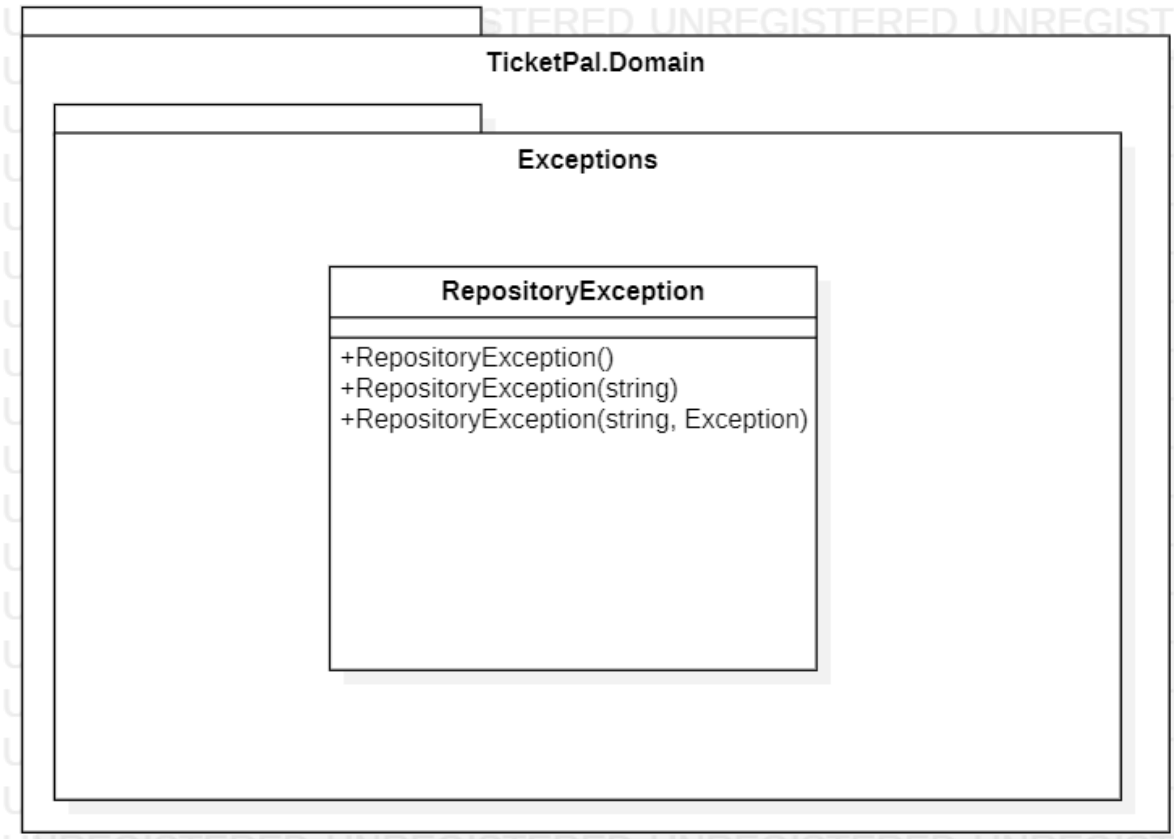
Módulo DataAccess:

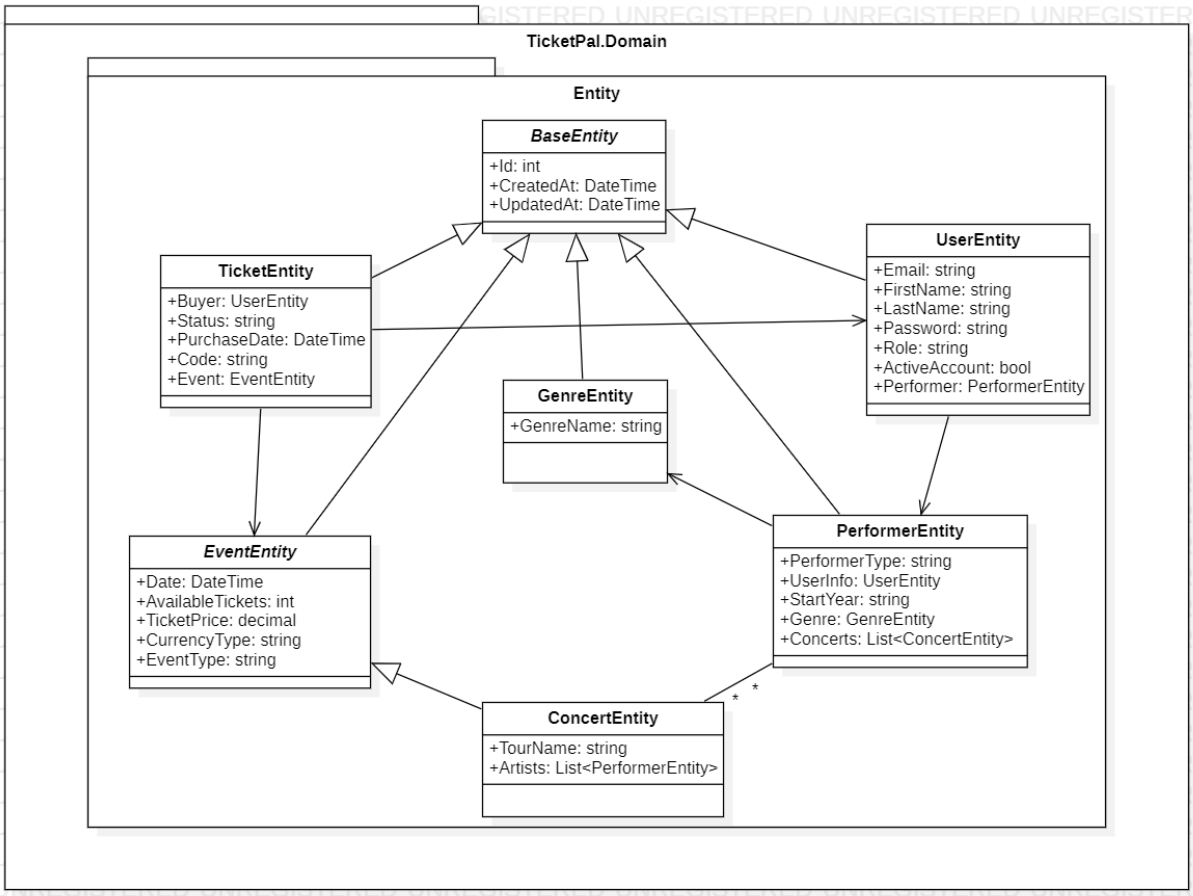

TicketPal.DataAccess
AppDbContext
+Users: DbSet<UserEntity>
+Genres: DbSet<GenreEntity>
+Performers: DbSet<PerformerEntity>
+Concerts: DbSet<ConcertEntity>
+Tickets: DbSet<TicketEntity>
#AppDbContext(DbContextOptions)
+AppDbContext(DbContextOptions)
#OnModelCreating(ModelBuilder): void

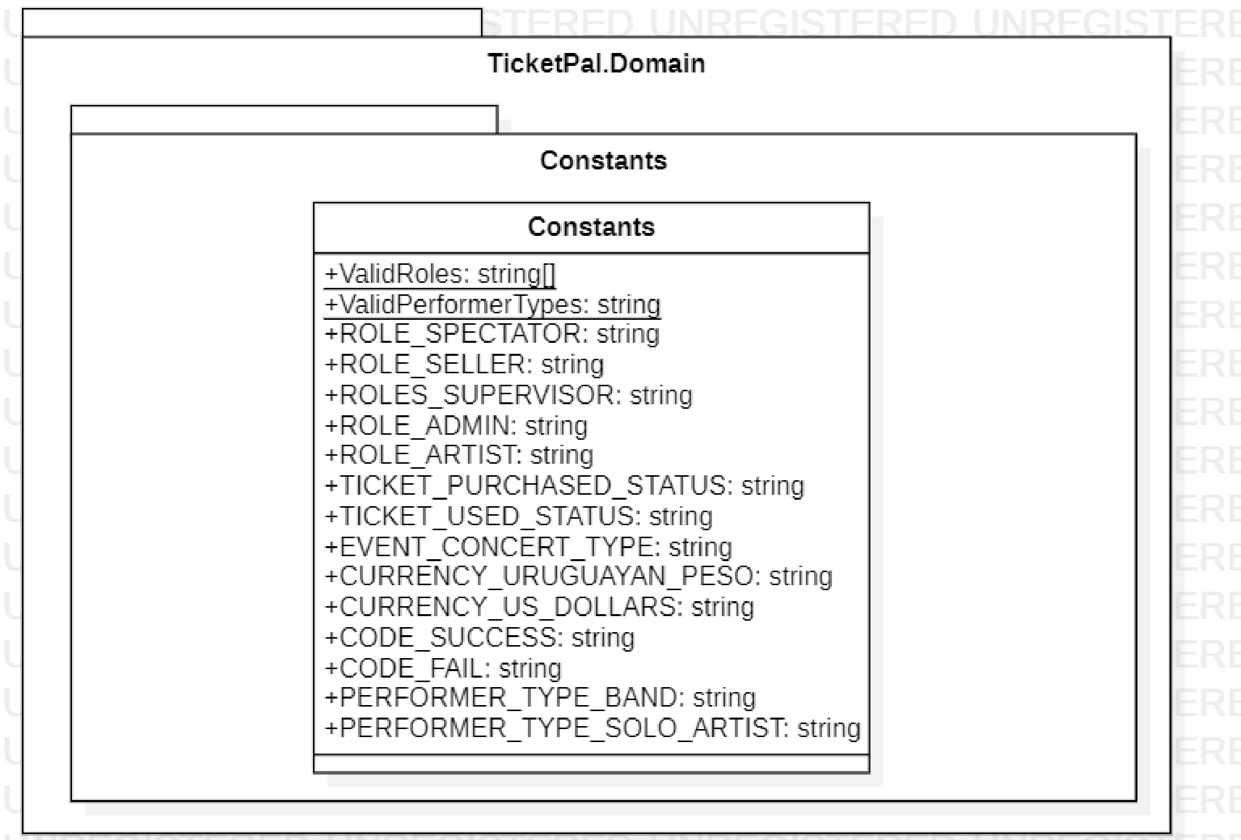

TicketPal.Data
Data
SeedData
+Users: List<UserEntity>

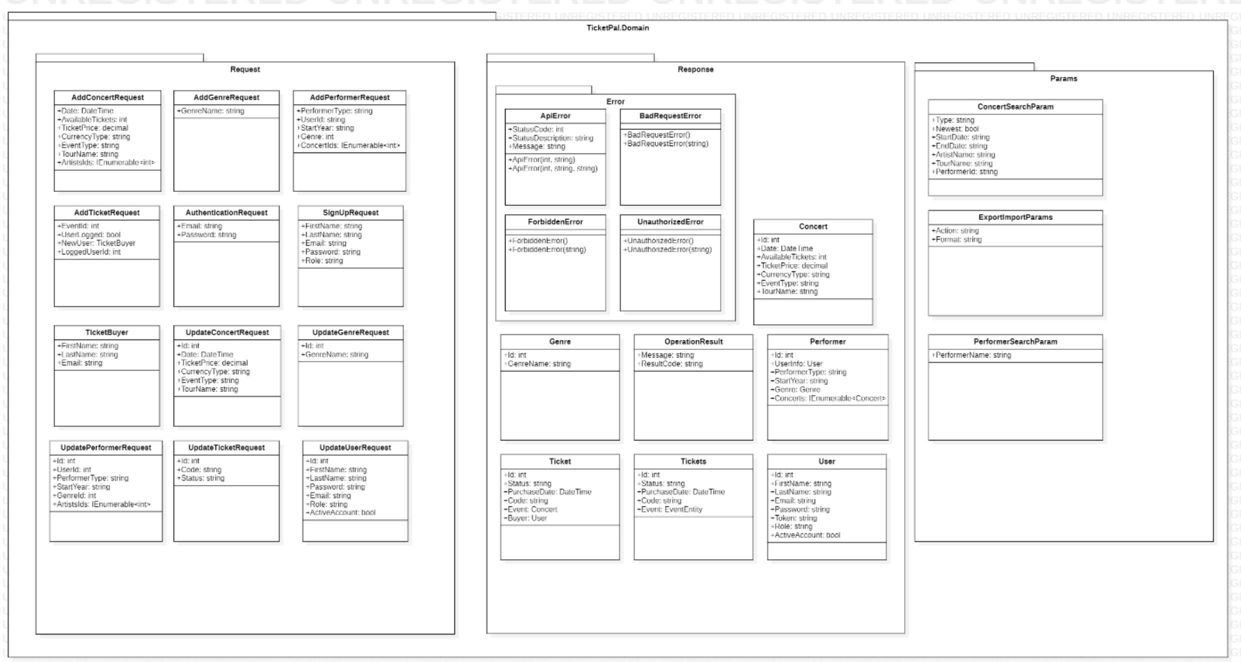


Módulo Domain:

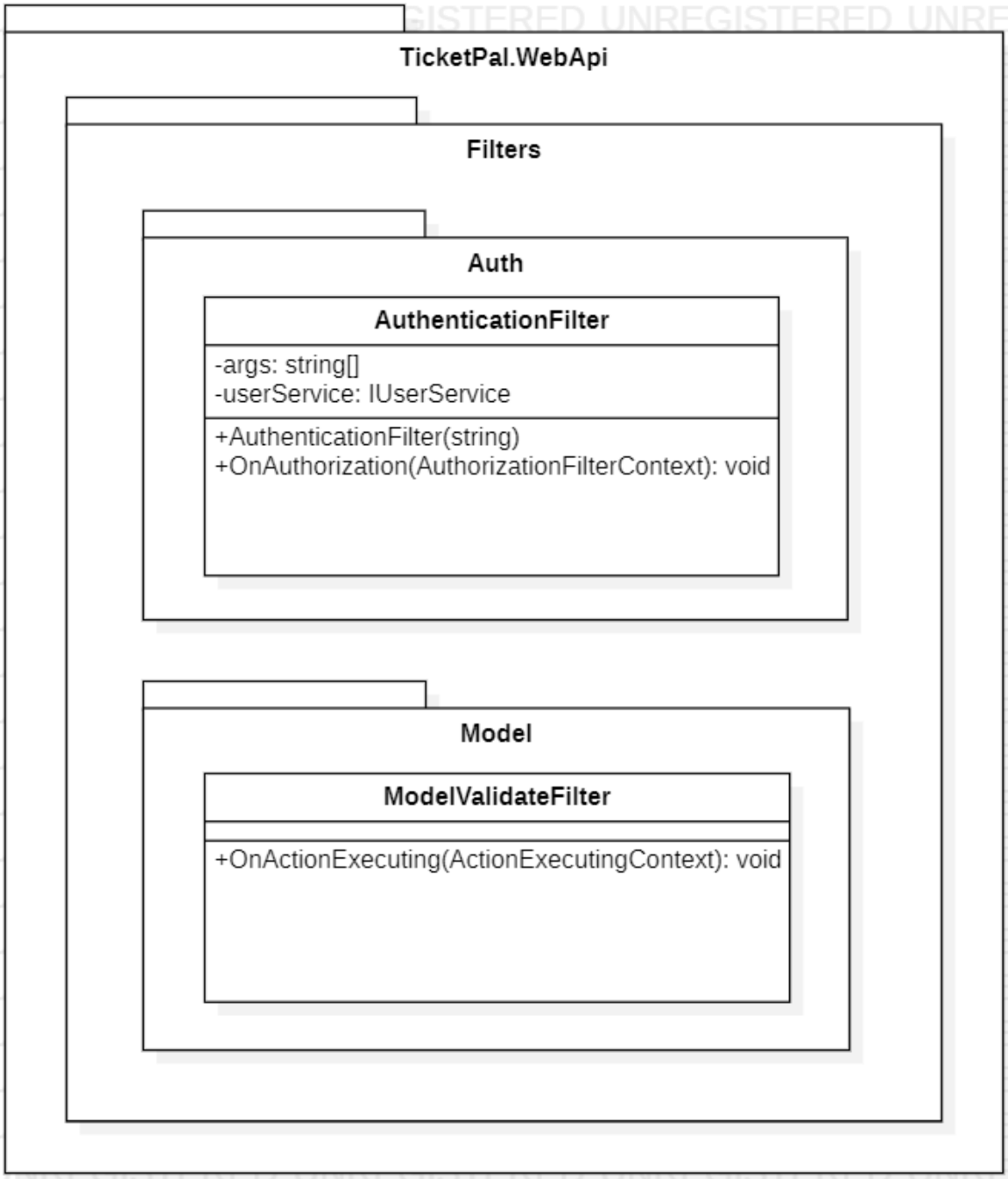


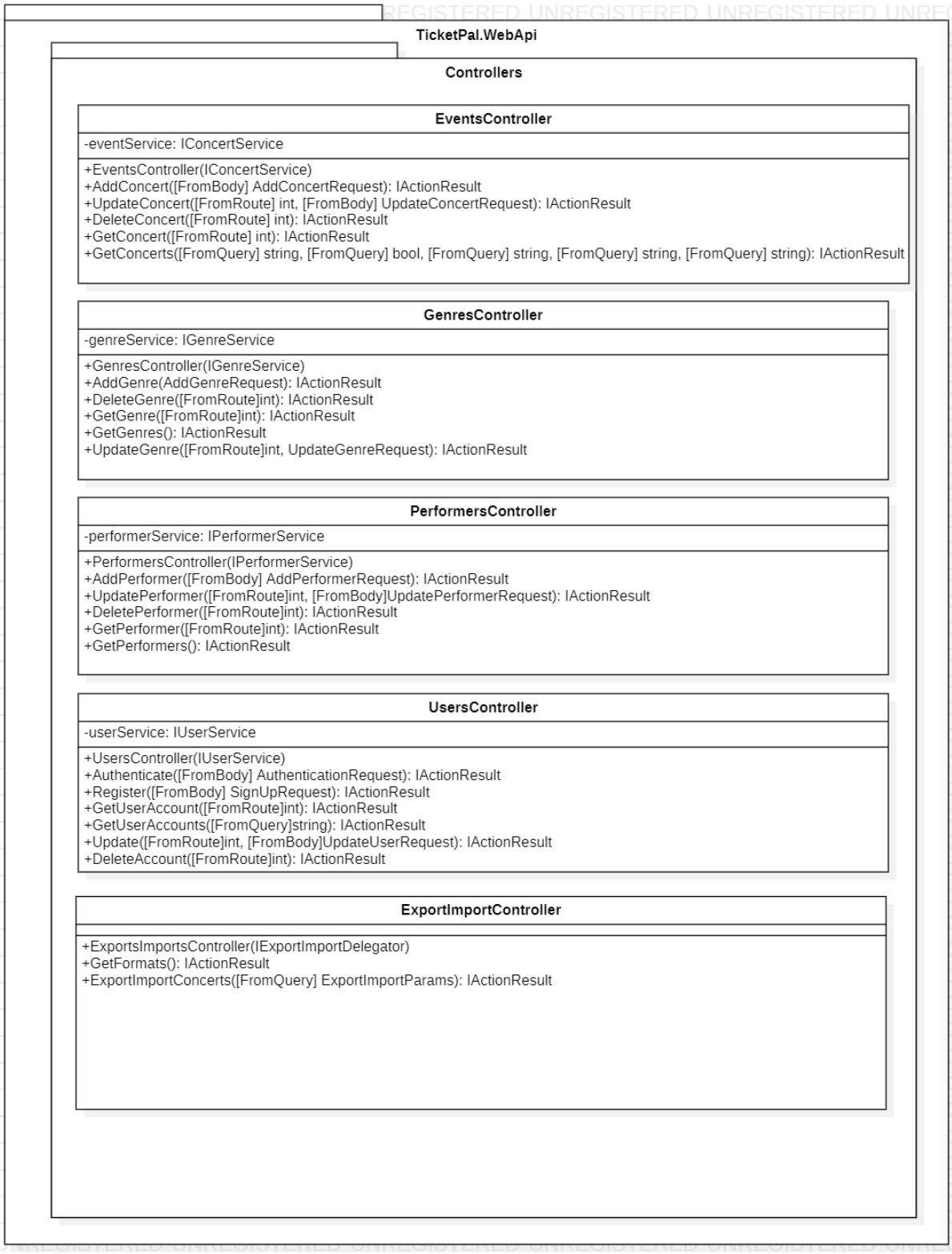

TicketPal.Domain
Entity
BaseEntity
+Id: int
+CreatedAt: DateTime
+UpdatedAt: DateTime
TicketEntity
+Buyer: UserEntity
+Status: string
+PurchaseDate: DateTime
+Code: string
+Event: EventEntity
UserEntity
+Email: string
+FirstName: string
+LastName: string
+Password: string
+Role: string
+ActiveAccount: bool
+Performer: PerformerEntity
GenreEntity
+GenreName: string
EventEntity
+Date: DateTime
+AvailableTickets: int
+TicketPrice: decimal
+CurrencyType: string
+EventType: string
PerformerEntity
+PerformerType: string
+UserInfo: UserEntity
+StartYear: string
+Genre: GenreEntity
+Concerts: List<ConcertEntity>
* *
ConcertEntity
+TourName: string
+Artists: List<PerformerEntity>

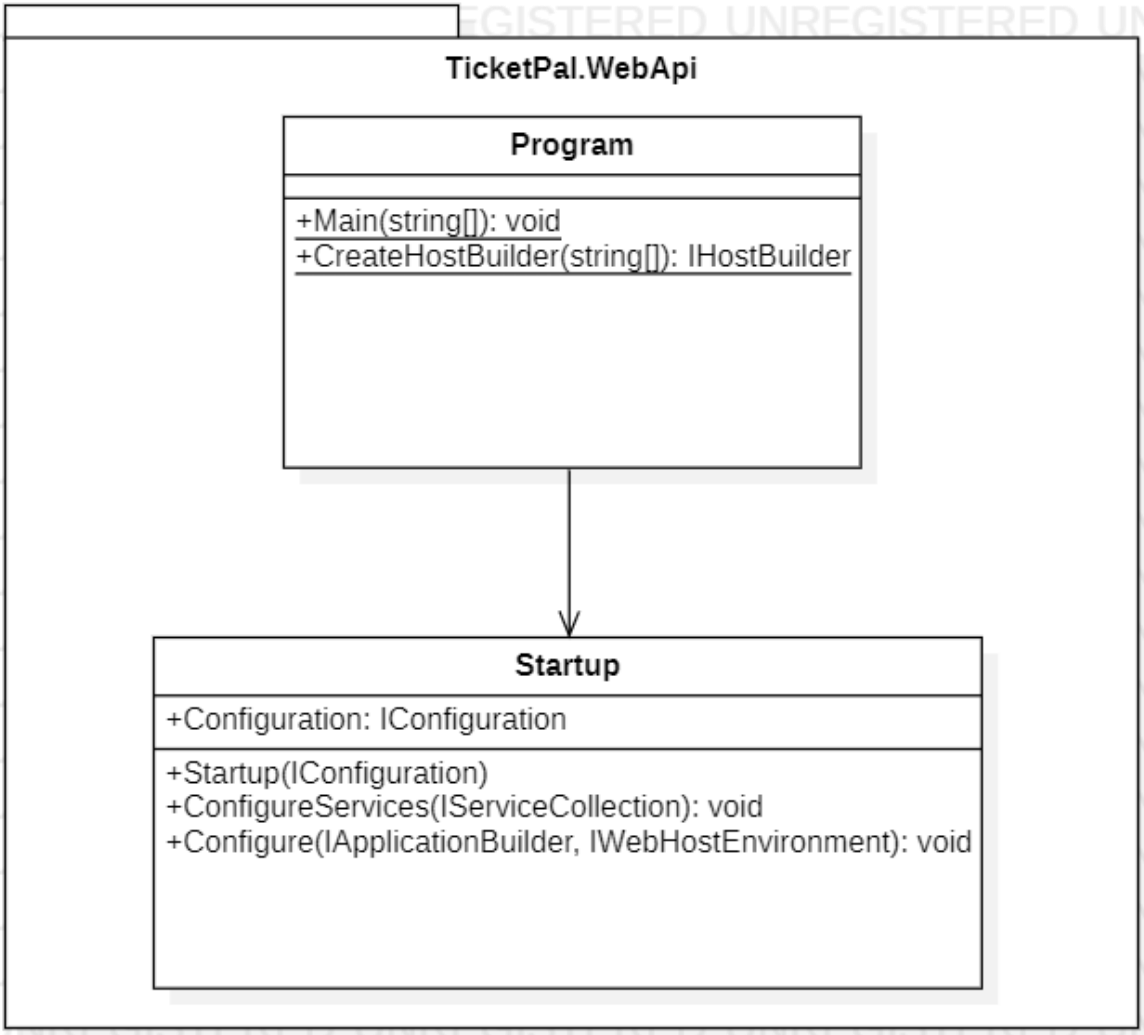

TicketPal.Domain
Constants
Constants
+ValidRoles: string[]
+ValidPerformerTypes: string
+ROLE_SPECTATOR: string
+ROLE_SELLER: string
+ROLES_SUPERVISOR: string
+ROLE_ADMIN: string
+ROLE_ARTIST: string
+TICKET_PURCHASED_STATUS: string
+TICKET_USED_STATUS: string
+EVENT_CONCERT_TYPE: string
+CURRENCY_URUGUAYAN_PESO: string
+CURRENCY_US_DOLLARS: string
+CODE_SUCCESS: string
+CODE_FAIL: string
+PERFORMER_TYPE_BAND: string
+PERFORMER_TYPE_SOLO_ARTIST: string


TicketPal.Domain
Request
AddConcertRequest
+Date: DateTime
+AvailableTickets: int
+TicketPrice: decimal
+CurrencyType: string
+EventType: string
+TourName: string
+ArtistsIds: IEnumerable<int>
AddGenreRequest
+GenreName: string
AddPerformerRequest
+PerformerType: string
+UserId: string
+StartYear: string
+Genre: int
+ConcertIds: IEnumerable<int>
AddTicketRequest
+EventId: int
+UserLogged: bool
+NewUser: TicketBuyer
+LoggedUserId: int
AuthenticationRequest
+Email: string
+Password: string
SignUpRequest
+FirstName: string
+LastName: string
+Email: string
+Password: string
+Role: string
TicketBuyer
+FirstName: string
+LastName: string
+Email: string
UpdateConcertRequest
+Id: int
+Date: DateTime
+TicketPrice: decimal
+CurrencyType: string
+EventType: string
+TourName: string
UpdateGenreRequest
+Id: int
+GenreName: string
UpdatePerformerRequest
+Id: int
+UserId: int
+PerformerType: string
+StartYear: string
+GenreId: int
+ArtistsIds: IEnumerable<int>
UpdateTicketRequest
+Id: int
+Code: string
+Status: string
UpdateUserRequest
+Id: int
+FirstName: string
+LastName: string
+Password: string
+Email: string
+Role: string
+ActiveAccount: bool
Response
Error
ApiError
+StatusCode: int
+StatusDescription: string
+Message: string
+ApiError(int, string)
+ApiError(int, string, string)
BadRequestError
+BadRequestError()
+BadRequestError(string)
ForbiddenError
+ForbiddenError()
+ForbiddenError(string)
UnauthorizedError
+UnauthorizedError()
+UnauthorizedError(string)
Concert
+Id: int
+Date: DateTime
+AvailableTickets: int
+TicketPrice: decimal
+CurrencyType: string
+EventType: string
+TourName: string
Genre
+Id: int
+GenreName: string
OperationResult
+Message: string
+ResultCode: string
Performer
+Id: int
+UserInfo: User
+PerformerType: string
+StartYear: string
+Genre: Genre
+Concerts: IEnumerable<Concert>
Ticket
+Id: int
+Status: string
+PurchaseDate: DateTime
+Code: string
+Event: Concert
+Buyer: User
Tickets
+Id: int
+Status: string
+PurchaseDate: DateTime
+Code: string
+Event: EventEntity
User
+Id: int
+FirstName: string
+LastName: string
+Email: string
+Password: string
+Token: string
+Role: string
+ActiveAccount: bool
Params
ConcertSearchParam
+Type: string
+Newest: bool
+StartDate: string
+EndDate: string
+ArtistName: string
+TourName: string
+PerformerId: string
ExportImportParams
+Action: string
+Format: string
PerformerSearchParam
+PerformerName: string

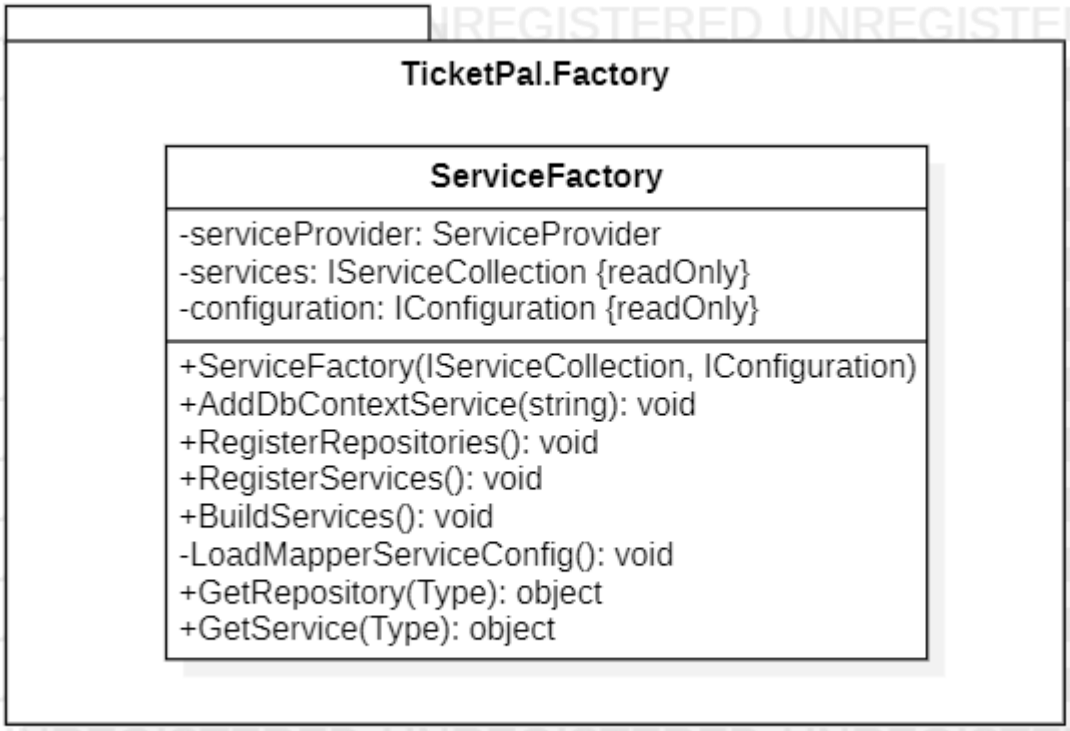
Módulo WebApi:



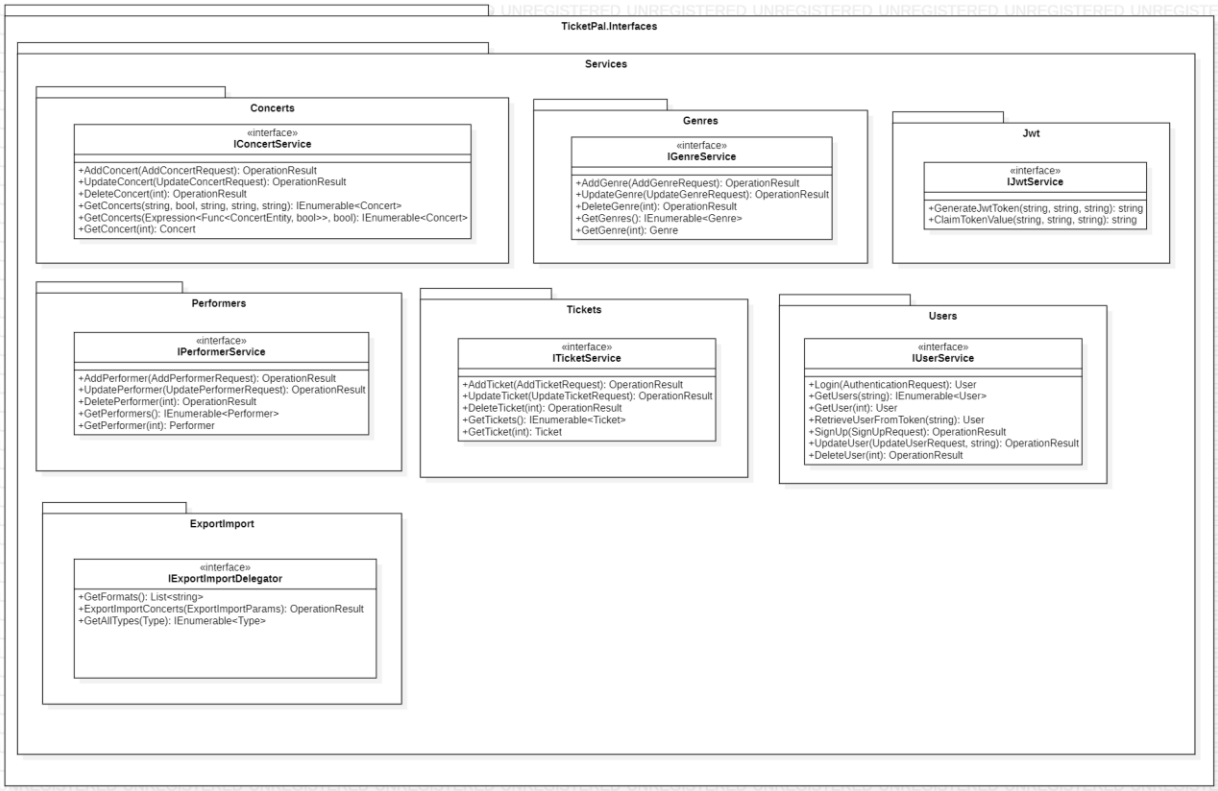

TicketPal.WebApi
Controllers
EventsController
-eventService: IConcertService
+EventsController(IConcertService)
+AddConcert([FromBody] AddConcertRequest): IActionResult
+UpdateConcert([FromRoute] int, [FromBody] UpdateConcertRequest): IActionResult
+DeleteConcert([FromRoute] int): IActionResult
+GetConcert([FromRoute] int): IActionResult
+GetConcerts([FromQuery] string, [FromQuery] bool, [FromQuery] string, [FromQuery] string, [FromQuery] string): IActionResult
GenresController
-genreService: IGenreService
+GenresController(IGenreService)
+AddGenre(AddGenreRequest): IActionResult
+DeleteGenre([FromRoute]int): IActionResult
+GetGenre([FromRoute]int): IActionResult
+GetGenres(): IActionResult
+UpdateGenre([FromRoute]int, UpdateGenreRequest): IActionResult
PerformersController
-performerService: IPerformerService
+PerformersController(IPerformerService)
+AddPerformer([FromBody] AddPerformerRequest): IActionResult
+UpdatePerformer([FromRoute]int, [FromBody]UpdatePerformerRequest): IActionResult
+DeletePerformer([FromRoute]int): IActionResult
+GetPerformer([FromRoute]int): IActionResult
+GetPerformers(): IActionResult
UsersController
-userService: IUserService
+UsersController(IUserService)
+Authenticate([FromBody] AuthenticationRequest): IActionResult
+Register([FromBody] SignUpRequest): IActionResult
+GetUserAccount([FromRoute]int): IActionResult
+GetUserAccounts([FromQuery]string): IActionResult
+Update([FromRoute]int, [FromBody]UpdateUserRequest): IActionResult
+DeleteAccount([FromRoute]int): IActionResult
ExportImportController
+ExportsImportsController(IExportImportDelegator)
+GetFormats(): IActionResult
+ExportImportConcerts([FromQuery] ExportImportParams): IActionResult

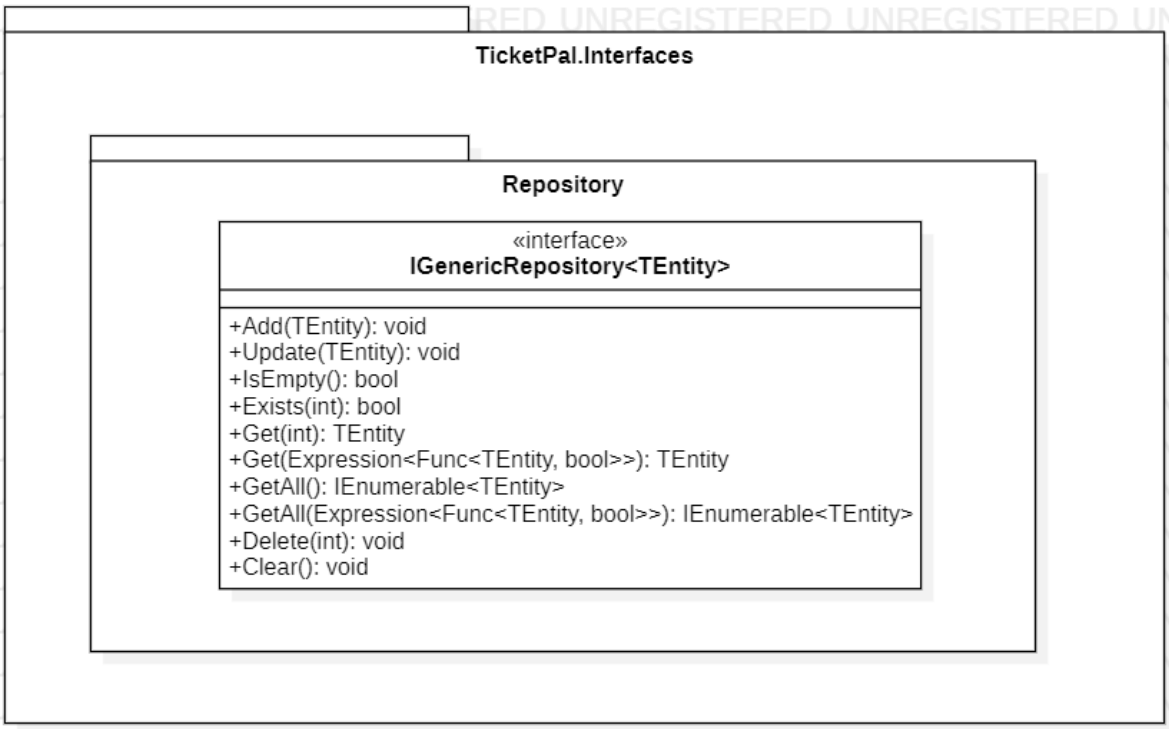


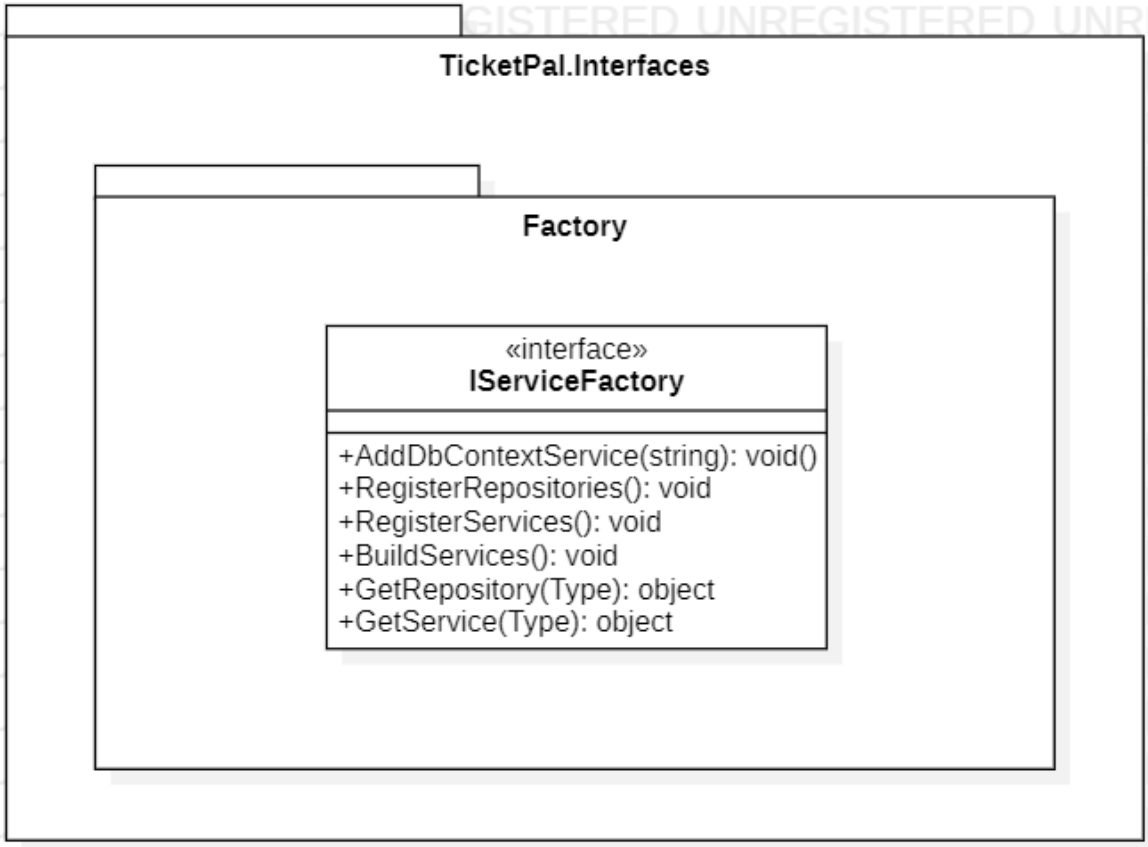
Módulo Factory:

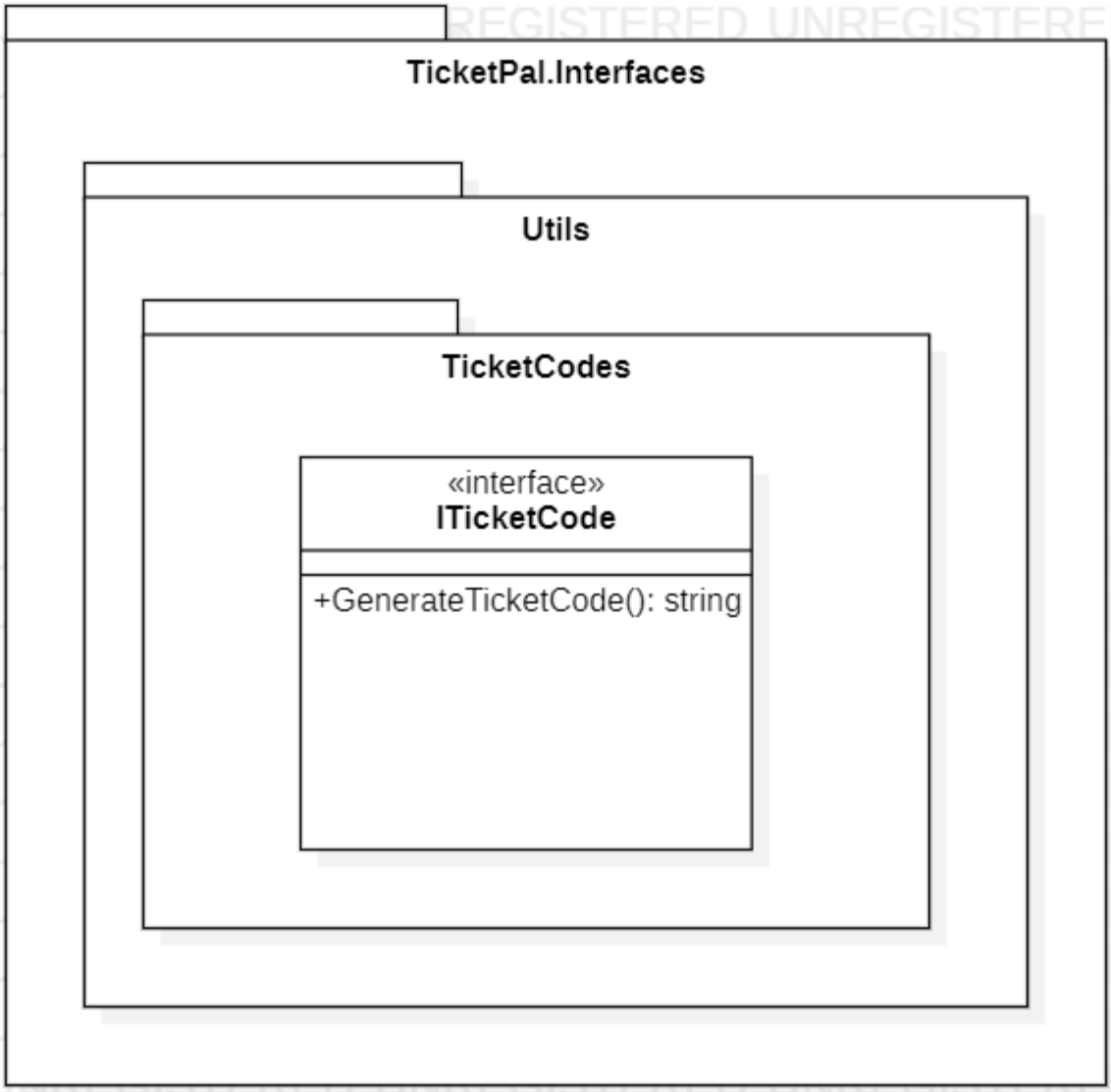


Módulo Interfaces:

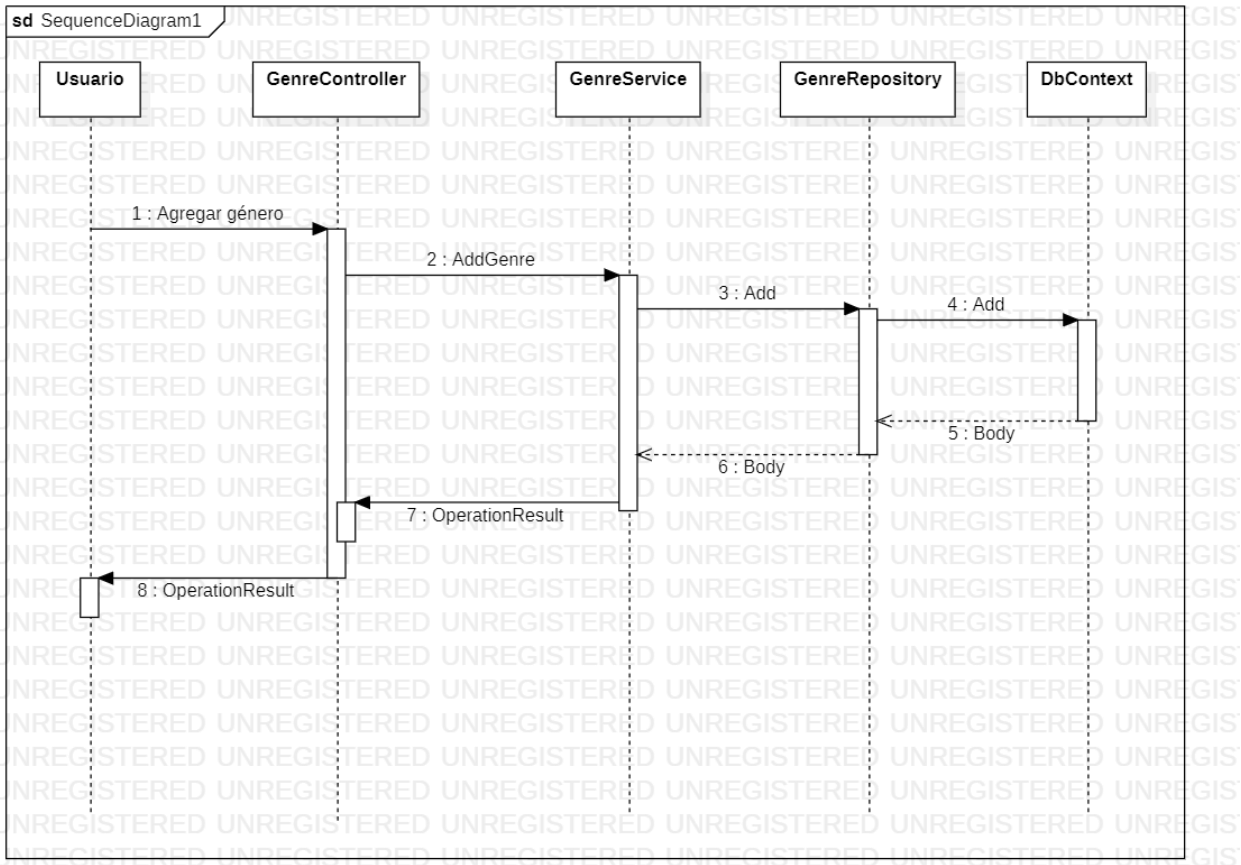




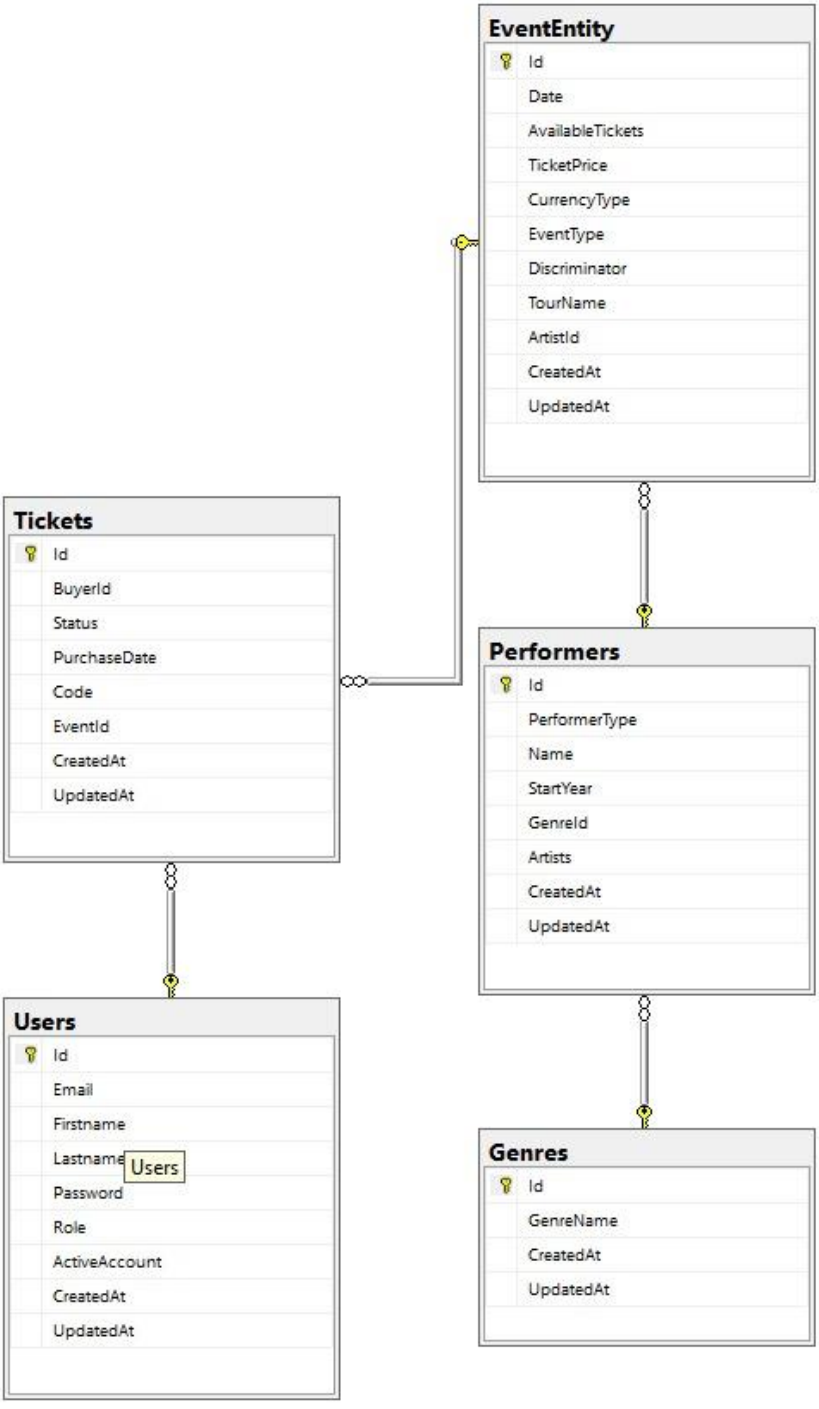




Diagramas de Interacción



Modelo de la estructura de tablas de la base de datos


EventEntity
Id
Date
AvailableTickets
TicketPrice
CurrencyType
EventType
Discriminator
TourName
ArtistId
CreatedAt
UpdatedAt
Tickets
Id
BuyerId
Status
PurchaseDate
Code
EventId
CreatedAt
UpdatedAt
Performers
Id
PerformerType
Name
StartYear
GenreId
Artists
CreatedAt
UpdatedAt
Users
Id
Email
Firstname
Lastname
Users
Password
Role
ActiveAccount
CreatedAt
UpdatedAt
Genres
Id
GenreName
CreatedAt
UpdatedAt

Análisis de métricas de diseño (diseño y calidad)

Se adjunta en la carpeta “código\TicketPal_API\NDependOut\ NDependReport.html” del obligatorio, el reporte generado por NDepend.

Tomando en cuenta los principios de Clausura Común, Reuso común, Abstracciones estables, y Dependencias estables, analizamos los resultados de las métricas, enfocándonos en las de distancia, abstracción e inestabilidad.

Conforme a ello, los resultados que obtuvimos fueron los siguientes:

	TicketPal.Interfaces	TicketPal.Domain	TicketPal.WebApi	TicketPal.Factory	TicketPal.DataAccess	TicketPal.BusinessLogic
Abstracción:	1	0,06	0	0	0,05	0
Inestabilidad:	0,58	0,42	1	0,98	0,99	0,98
Distancia:	0,41	0,37	0	0,01	0,03	0,01

El paquete TicketPal.Interfaces se ubica en el rango más alto de abstracción (debido a que es un paquete que sólo contiene interfaces), y con un grado bastante avanzado de inestabilidad, ya que tiene cierta dependencia a otros paquetes, en el sentido de que necesita de los tipos de datos de paquetes como Domain, por ejemplo.

Por otra parte, el paquete TicketPal.Domain se ubica en un rango bastante alto de concreción, y más cerca de la estabilidad. Esto ocurre porque es la clase que tiene todos los DTO’s, las constantes, las excepciones y los modelos de request y response utilizados para que la WebApi pueda relacionarse con el frontend.

Estos dos paquetes fueron los que obtuvieron los peores resultados en cuanto a abstracción e inestabilidad, en comparación con los demás paquetes (TicketPal.WebApi, TicketPal.Factory, TicketPal.DataAccess y TicketPal.BusinessLogic), que lograron una máxima inestabilidad y una máxima concreción.

Con respecto a la métrica de distancia, la situación se invertiría, y los paquetes TicketPal.Interfaces y TicketPal.Domain son los que obtienen un mejor resultado, ya que sus valores se encuentran más cerca del 0,5, lo que los aleja de la zona de dolor (es decir, que si cambian, impactan en otros) y de la zona de poca utilidad (es decir, que tienen pocos paquetes que dependan de él).

Teniendo en cuenta los principios de Clausura común, Reuso común, Abstracciones estables, y Dependencias estables, y la métrica de distancia, sumado a las dos métricas anteriores, podemos ver lo siguiente:

- Que se cumpliría el principio de Clausura común, ya que las clases que cambian juntas, se mantienen juntas (por ejemplo, todos los entities están en Domain, y todos los servicios están en BusinessLogic).

- Que se cumpliría el principio de Reuso común, ya que no hay clases que no se reusen juntas, que no estén agrupadas en un mismo paquete (por ejemplo, los repositorios se reusan juntos, y se encuentran todos en DataAccess).

- Que se cumpliría el principio de Dependencias estables, ya que, salvo excepciones, los paquetes dependen de otros paquetes que tienen una métrica de inestabilidad menores a la suya.
- Que se cumpliría el principio de Abstracciones estables, ya los paquetes más estables son abstractos y los inestables deben tender a la concreción. Sin embargo, no se nos estaría cumpliendo este principio para las clases TicketPal.Interfaces y TicketPal.Domain.

Salvo las excepciones mencionadas, entendemos que las métricas, en gran parte de los resultados, nos muestra el código está orientado a lograr una buena calidad del mismo.

Resumen de las mejoras al diseño

Con respecto a la primera entrega, no existieron demasiados cambios en el diseño, debido a que logramos que la estructura del proyecto que se implementó fuera la que queríamos para el proyecto.

Ya desde la primera entrega intentamos mantener un código extensible y que cumpliera con los principios SOLID, aplicando también patrones de diseño, como por ejemplo, el Factory Method para implementar un ServiceFactory que nos permitiera crear objetos sin especificar sus clases concretas.

Por tanto, los cambios que se agregaron para esta segunda entrega fueron para terminar de arreglar los problemas que tuvimos en la primera entrega, y en algunos casos puntuales, para agregar nuevas features para el frontend, que no habíamos previsto (ya que no se había pensado en un diseño para esa parte todavía), o que se solicitaba en la letra del segundo obligatorio.

De todas formas, sí hubieron varias mejoras e implementaciones nuevas con respecto al entregable anterior.

-**Mejoras en los diagramas e inclusión de los que faltaban:** por falta de tiempo, para la primera entrega nos vimos limitados en cuanto a la agregación de varios diagramas de clase y de

paquetes, que sí se adjuntan en esta entrega. Se hizo una división de los diagramas de clases por paquetes, para una mejor comprensión de cómo está compuesto cada paquete en su interior. Todos estos diagramas se pueden encontrar en la carpeta “Documentación\UML”.

-**Implementación de lo que no se pudo hacer antes:** para la primera entrega, no se había podido cumplir con algunas funcionalidades, debido a un problema con la clase del controlador de Tickets, y a que no se había podido generar el token sin tener que hardcodearlo en JWSecret. Sin embargo, el controlador de Tickets está completamente activo y funcionando, y los tokens que se generan cuando un usuario se loguea, ahora se está tomando desde la configuración de appsettings.json, como corresponde.

-**Proteger la aplicación de un problema de concurrencia:** cuando empezamos a implementar el frontend, empezamos a tener problemas con la concurrencia, ya que Core no permite el paralelismo, y por esa razón, cuando queríamos cargar varias tablas a la vez, la API devolvía un error. Para ello, debimos utilizar “async” y “await” en nuestros métodos, para que aquellos que debían obtener datos desde la BD, pudieran hacerlo sin problemas. Ese fue un cambio imprevisto para nosotros, ya que no habíamos tenido ese problema en la primera entrega, ejecutando de a una consulta a la vez a través de Postman, pero que permitió mejorar la performance del funcionamiento de nuestra web api.

-**Se utilizaron los valores de las métricas obtenidas para mejorar diseño:** aunque estábamos bastante conformes con el diseño de la primera entrega, se ejecutó la extensión NDepend para revisar las métricas que tenía nuestra primera entrega, para, en base a eso, apuntar a que lo nuevo que debíamos agregar, tendiera a mantener o mejorar los valores obtenidos. En algunos casos, pudimos mejorar esos índices; en otros, la complejidad de la solución y la falta de tiempo no lo permitieron. Sin embargo, el deterioro de algunos índices fue mínimo, como en el caso de los paquetes de TicketPal.Interfaces y TicketPal.Domain, como se puede ver en la tabla siguiente:

	TicketPal.Interfaces (primera entrega)	TicketPal.Interfaces (segunda entrega)
Abstracción:	1	1
Inestabilidad:	0,56	0,58
Distancia:	0,40	0,41
	TicketPal.Domain (primera entrega)	TicketPal.Domain (segunda entrega)
Abstracción:	0,06	0,06
Inestabilidad:	0,44	0,42
Distancia:	0,35	0,37

Como agregado, otra de las mejoras o cambios que se hicieron con respecto a la entrega anterior fue la de la descripción de los endpoints y la mejora del documento de especificación de la API.

ANEXO

EVIDENCIA DEL DISEÑO Y ESPECIFICACIÓN DE LA API

Se crearon 5 usuarios diferentes para poder realizar las pruebas de la API. Dichos usuarios son los siguientes:

Id = 1,
Firstname = "Lucas",
Lastname = "Castro",
Email = "lucas@example.com",
Password = "lucas1",
Role = ADMIN

Id = 2,
Firstname = "Ricardo",
Lastname = "Poladura",
Email = "ricardo@example.com",
Password = "ricardo1",
Role = ADMIN

Id = 3,
Firstname = "Spectator",
Lastname = "Test",
Email = "spectator@example.com",
Password = "spectator1",
Role = SPECTATOR

Id = 4,
Firstname = "Seller",
Lastname = "Test",
Email = "seller@example.com",
Password = "seller1",
Role = SELLER

Id = 5,
Firstname = "Supervisor",

Lastname = "Test",
Email = "supervisor@example.com",
Password = "supervisor1",
Role = SUPERVISOR

La justificación y la evidencia del diseño se encuentra aquí: https://github.com/ORT-DA2/Castro-Poladura/tree/develop/Documentación/Markdown

EVIDENCIA DE LA APLICACIÓN DE TDD Y CLEAN CODE

La estrategia utilizada para aplicar TDD en nuestro obligatorio fue la de outside-in, que consiste en escribir un test, con lo mínimo necesario, que falle, para luego escribir el código necesario para que el test pase, y luego refactorizar, volviendo a escribir otro test que falle, y así sucesivamente ("Red-Green-Refactor"). Y se van realizando las iteraciones necesarias de este bucle interno hasta conseguir pasar el test de aceptación.

Por regla general, casi toda la implementación del obligatorio se hizo aplicando TDD. Por un tema de tiempo, solamente la implementación de algunos Requests y Responses, se hizo sin aplicar TDD.

Con respecto a la cobertura de código, se adjunta reporte de cobertura en la carpeta de Documentación del repositorio.

Por falta de tiempo, quedaron los tests de la clase ExportImportDelegator comentados o sin hacer. Esta parte, como se hizo más cerca de la fecha de entrega, no se pudo hacer completamente mediante TDD (sí la parte del controller).

Pruebas de aplicación de TDD y Clean Code

Ejemplos de aplicación de TDD y Clean code en la implementación del obligatorio:

Commit 570b1ee (Primero se terminan de implementar los tests de UserServiceTests):

```
18      [TestClass]
19      public class UserServiceTest : BaseServiceTest
20      {
-           private string jwtTestSecret;
-           private string userPassword;
-
21          [TestMethod]
22          public void UserAuthenticateCorrectly()
23          {
-               int id = 1;
-               this.userPassword = "somePassword";
-               this.jwtTestSecret = "23jrb783v29fwfvfg2874gf286fce8";
-               var testAppsettings = Options.Create(new AppSettings { JwtSecret = jwtTestSecret });
-
24  +           int id = 1;
25              var authRequest = new AuthenticationRequest
26              {
27                  Email = "someone@example.com",
    @@ -50,7 +43,7 @@ public void UserAuthenticateCorrectly()
43
44              this.userService = new UserService(
45                  this.usersMock.Object,
-                   testAppsettings,
46  +               this.testAppSettings,
47                  this.mapper
48              );
49              User authenticatedUser = userService.Login(authRequest);
```

Commit e22c195 (Se implementa la clase UserService):

```
18  +     public class UserService : IUserService
19  +     {
20  +         private readonly IUserRepository repository;
21  +         private readonly IAppSettings appSettings;
22  +         private readonly IMapper mapper;
23  +         public UserService(
24  +             IUserRepository repository,
25  +             IOptions<IAppSettings> appSettings,
26  +             IMapper mapper
27  +         )
28  +         {
29  +             this.mapper = mapper;
30  +             this.repository = repository;
31  +             this.appSettings = appSettings.Value;
32  +         }
33  +         public OperationResult DeleteUser(int id)
34  +         {
35  +             try
36  +             {
37  +                 repository.Delete(id);
38  +                 return new OperationResult
39  +                 {
40  +                     ResultCode = ResultCode.SUCCESS,
41  +                     Message = "User removed successfully"
42  +                 };
43  +             }
44  +             catch (RepositoryException ex)
45  +             {
46  +                 return new OperationResult
47  +                 {
48  +                     ResultCode = ResultCode.FAIL,
49  +                     Message = ex.Message
```

```
+                    Message = ex.Message
+                };
+            }
+        }
+
+        public User GetUser(int id)
+        {
+            return mapper.Map<User>(repository.Get(id));
+        }
+
+        public IEnumerable<User> GetUsers(UserRole role = UserRole.SPECTATOR)
+        {
+            var users = repository.GetAll(u => u.Role.Equals(role.ToString()));
+            return mapper.Map<IEnumerable<UserEntity>, IEnumerable<User>>(users);
+        }
+
+        public User Login(AuthenticationRequest model)
+        {
+            var found = repository.Get(u => u.Email.Equals(model.Email));
+            if (found == null || !BC.Verify(model.Password, found.Password))
+            {
+                return null;
+            }
+            var token = JwtUtils.GenerateJwtToken(appSettings.JwtSecret, "id", found.Id.ToString());
+            var user = mapper.Map<User>(found);
+
+            return user;
+        }
```

Commit 0a22225 (Se crearon las pruebas de la clase TicketServiceTests):

Clase TicketServiceTests:

```
53  +       [TestMethod]
54  +       public void AddTicketSuccesfullyTest()
55  +       {
56  +           OperationResult result = ticketService.AddTicket(ticketRequest);
57  +
58  +           Assert.IsTrue(result.ResultCode == ResultCode.SUCCESS);
59  +       }
60  +
61  +       [TestMethod]
62  +       public void AddTicketTwiceFailsTest()
63  +       {
64  +           ticketService.AddTicket(ticketRequest);
65  +
66  +           this.ticketsMock.Setup(m => m.Exists(It.IsAny<int>())).Returns(true);
67  +           this.ticketsMock.Setup(m => m.Add(It.IsAny<TicketEntity>())).Throws(new RepositoryException());
68  +           ticketService = new TicketService(this.ticketsMock.Object, this.testAppSettings, this.mapper);
69  +
70  +           OperationResult result = ticketService.AddTicket(ticketRequest);
71  +
72  +           Assert.IsTrue(result.ResultCode == ResultCode.FAIL);
73  +       }
```

```
107 +       [TestMethod]
108 +       public void UpdateTicketSuccesfullyTest()
109 +       {
110 +           var updateRequest = new UpdateTicketRequest
111 +           {
112 +               Code = ticket.Code,
113 +               Status = ticket.Status
114 +           };
115 +
116 +           this.ticketsMock.Setup(m => m.Update(It.IsAny<TicketEntity>())).Verifiable();
117 +           OperationResult expected = ticketService.UpdateTicket(updateRequest);
118 +
119 +           Assert.IsTrue(expected.ResultCode == ResultCode.SUCCESS);
120 +       }
```

Clase TicketService:

```
29 +         public OperationResult AddTicket(AddTicketRequest model)
30 +         {
31 +             throw new NotImplementedException();
32 +         }
33 +
34 +         public OperationResult DeleteTicket(int id)
35 +         {
36 +             throw new NotImplementedException();
37 +         }
38 +
39 +         public bool ExistsTicketByName(string name)
40 +         {
41 +             throw new NotImplementedException();
42 +         }
43 +
44 +         public Ticket GetTicket(int id)
45 +         {
46 +             throw new NotImplementedException();
47 +         }
48 +
49 +         public IEnumerable<Ticket> GetTickets()
50 +         {
51 +             throw new NotImplementedException();
52 +         }
53 +
54 +         public OperationResult UpdateTicket(UpdateTicketRequest model)
55 +         {
56 +             throw new NotImplementedException();
57 +         }
58 +     }
```

Commit 8fa6670 (Se implementa la clase TicketService):

```
19        public class TicketService : ITicketService
20        {
      -         private readonly ITicketRepository repository;
      -         private readonly IAppSettings appSettings;
21    +         private readonly IServiceFactory serviceFactory;
22              private readonly IMapper mapper;
23    +         public IGenericRepository<TicketEntity> ticketRepository;
24    +         public IGenericRepository<ConcertEntity> concertRepository;
25
      -         public TicketService(ITicketRepository repository, IOptions<IAppSettings> appSettings, IMapper mapper)
26    +         public TicketService(IServiceFactory factory, IMapper mapper)
27              {
28                  this.mapper = mapper;
      -             this.repository = repository;
      -             this.appSettings = appSettings.Value;
29    +             this.serviceFactory = factory;
30    +             ticketRepository = serviceFactory.GetRepository<TicketEntity>() as IGenericRepository<TicketEntity>;
31    +             concertRepository = serviceFactory.GetRepository<ConcertEntity>() as IGenericRepository<ConcertEntity>;
32              }
33
34              public OperationResult AddTicket(AddTicketRequest model)
35              {
      -             throw new NotImplementedException();
      -         }
36    +             try
37    +             {
38    +                 EventEntity newEvent = concertRepository.Get(model.Event);
39
      -         public OperationResult DeleteTicket(int id)
      -         {
      -             throw new NotImplementedException();
```

Commit 14563ee (Se crean tests para utilización de ServiceFactory):

```
+ namespace TicketPal.Factory.Tests.Repository
+ {
+     [TestClass]
+     public class RepositoryDependenciesTest :BaseTestFactoryConfig
+     {
+         private ServiceFactory factory;
+         private ServiceProvider serviceProvider;
+
+         [TestInitialize]
+         public void Init()
+         {
+             // Factory
+             this.factory = new ServiceFactory(this.services);
+
+             this.factory.AddDbContextService("SomeFakeConnectionString");
+             this.factory.RegisterRepositories();
+
+             this.serviceProvider = services.BuildServiceProvider();
+         }
+
+         [TestMethod]
+         public void UserRepositoryDependencyCheck()
+         {
+             var repository = serviceProvider.GetService<IGenericRepository<UserEntity>>();
+             Assert.IsNotNull(repository);
+         }
+     }
+ }
```

Commit 129884f (Se crea clase ServiceFactory):

```
+ namespace TicketPal.Factory
+ {
+     public class ServiceFactory
+     {
+         private readonly IServiceCollection services;
+ 
+         public ServiceFactory(IServiceCollection services)
+                 {
+                         this.services = services;
+                 }
+ 
+         public void AddDbContextService(string connectionString)
+                 {
+                         services.AddDbContext<DbContext, AppDbContext>
+                 (options => options.UseSqlServer(connectionString));
+                 }
+ 
+         public void RegisterRepositories()
+         {
+             services.AddScoped(typeof(IGenericRepository<UserEntity>),typeof(UserRepository));
+         }
+     }
+ }
```