UNIVERSIDAD ORT
Uruguay

Diseño de Aplicaciones 2

Obligatorio I - Grupo N6A-AN

Javier Aramberri - 202373 **Maximiliano Pollinger - 202066**

Tutores: Ignacio Valle, Fernando Olmos

Índice

1 Descripción general

La solución propone una implementación de una *WebAPI* la cual expone funcionalidades para 2 tipos de usuarios, las cuales incluyen: alta, baja y modificación de equipos, deportes, usuarios y encuentros y la alta de comentarios.

Los usuarios en rol de administrador, pueden desarrollar las 3 funcionalidades (alta, baja, modificación), mientras que los usuarios con rol de seguidor, solo pueden seguir a equipos e introducir comentarios. Los comentarios se realizan sobre un encuentro determinado.

Los encuentros se pueden generar manualmente o utilizando un algoritmo de generación de encuentros. Cada encuentro cuenta con 2 equipos y son de un deporte específico. Cada equipo puede contar con una foto, pertenece a un deporte y además puede ser seguido por múltiples usuarios. Cada deporte puede tener múltiples equipos. Cada usuario puede seguir a múltiples equipos. El usuario se identifica en el sistema con su email y una *password*. Cuando se inicia sesión en el sistema, se le asigna un *token*, el cual es necesario para poder realizar las funciones de administrador y así además poder autenticar al usuario.

En palabras generales, el sistema es un administrador y generador de usuarios interesados en deportes que pueden comentar sobre ciertos encuentros de los equipos participantes de estos últimos.

2 Decisiones de diseño

Al principio habíamos implementado una solución del patrón *Unit of Work*, en la que tendríamos un *Singleton* de cada repositorio a usar, por ejemplo los repositorios de deportes o equipos, que nos permitirá usar cualquier repositorio con la instancia del *unit of work*, lo cual solucionaría problemas del tipo de no tener que tener la lógica de una entidad en la lógica de otra entidad diferente. Pero dado que al momento de comenzar con el uso de el *framework* de *mocking* para realizar las pruebas unitarias, nos dimos cuenta que el utilizar el patrón nos estaba dando problemas al momento de hacer el *mock* de la interfaz de la implementación del patrón. Es por esto que decidimos eliminar el patrón implementado de *unit of work* y quedarnos solo con la implementación (y su respectiva *interface*) del patrón Repositorio.
Al realizar esto pudimos evolucionar correctamente con la implementación de pruebas unitarias, a pesar de tener las desventajas ocasionadas por la pérdida del patrón de *unit of work*.

En ocasionales pruebas, principalmente en las que probamos el repositorio, hacemos uso de Assert con llamadas a métodos de lista o de clases de *System*. Esto lo hacemos además del uso del método *Verify* del *mock*, para asegurarnos de que además de que se ejecuten los métodos que hacemos *mock*, también nos aseguramos que no falle en el *assert*. Entendemos que podemos hacer uso de los métodos de *System* porque consideramos que la estabilidad de ese assembly es muy alta y que ya fueron probados por miles de desarrolladores además de ser un assembly que tiene años en uso y utilizado por básicamente todas las aplicaciones que se desarrollan en el ámbito *.NET*.

Para poder tener navegabilidad entre deportes y equipos, pusimos una referencia por ID del deporte en el equipo y una lista de equipos en el deporte. De esta manera podemos tener múltiples equipos con el mismo nombre pero que pertenecen a distintos deportes. Además un deporte puede tener N equipos asociados.
Esto también nos simplifica para cuando debemos obtener todos los equipos de un deporte, solo necesitamos consultar por la lista de equipos de ese deporte.
Si estamos en un equipo, podemos saber a qué deporte pertenece mediante el ID del deporte, yendo a buscar al repositorio de deportes con ese ID.

Dado que *Entity Framework Core* no implementa la configuración automática de relaciones N-a-N sin usar una entidad que las relacione, debimos crear la clase *UsersTeams*, en la cual tenemos una referencia al usuario y una referencia al equipo. Mediante *FluentAPI*, configuramos las restricciones referenciales de claves foráneas para que un equipo pueda ser seguido por N usuarios, y a su vez, un usuario pueda seguir a N equipos.

Decidimos llamar al proyecto en general como *SportFixtures*, dado que es un generador de fixtures para deportes.
Dividimos la solución en varios proyectos. El proyecto principal es el que contiene la *WebAPI* denominado *SportFixtures.Portal* ya que es el portal de entrada a la aplicación. En esta ocasión la API es el único punto de entrada y a su vez la única manera de acceder a las operaciones que se encuentran en la lógica de la aplicación.
Luego separamos en varias capas lo que son las entidades del negocio, las interfaces de la lógica del negocio, las implementaciones de la lógica del negocio y el acceso a datos. Los proyectos fueron denominados *SportFixtures.Data*, *SportFixtures.BusinessLogic*, *SportFixtures.BusinessLogic.Implementations* y *SportFixtures.Data.Access*, respectivamente.
También separamos las excepciones en un proyecto aparte, como indicado en la parte de fundamentación de *Clean Code*.
En otro proyecto se encuentra la implementación del patrón Repositorio. El cual cuenta con una interface que es implementada por una clase llamada *GenericRepository*.
Por último, tenemos el proyecto *SportFixtures.Test*, en el cual se encuentran todas las pruebas unitarias. Si bien podríamos haber tenido las pruebas unitarias en cada uno de los proyectos dependiendo de lo que se estuviera probando, consideramos que tenerlas agrupadas en un proyecto es una mejor manera de separar responsabilidades y además no acoplar otros proyectos a paquetes *NuGet* como por ejemplo *Moq*.
Luego contamos con un proyecto donde definimos una interfaz para la generación de encuentros automática, denominado *SportFixtures.FixtureGenerator*. Esta interfaz define el método para la generación de encuentros para los equipos dados, la fecha de inicio dada y el ID del deporte.
Las implementaciones se encuentran en otro proyecto denominado *SportFixtures.FixtureGenerator.Implementations*. Por el momento contamos con 2 algoritmos de generación de encuentros, *FreeForAll* y *RoundRobin*, todos los equipos contra todos los equipos e ida y vuelta, respectivamente.
Esta separación de proyectos de las interfaces por un lado y las implementaciones por otro, la hicimos para aumentar la cohesión y bajar el acoplamiento de clases que necesiten realizar determinadas funciones, como por ejemplo generar encuentros, no dependan de una implementación específica, sino que se acoplen a interfaces. También hicimos la separación en distintos proyectos de las interfaces de la lógica de negocio de las implementaciones por la misma razón. Esto se puede observar mejor en el diagrama de paquetes que se muestra a continuación.

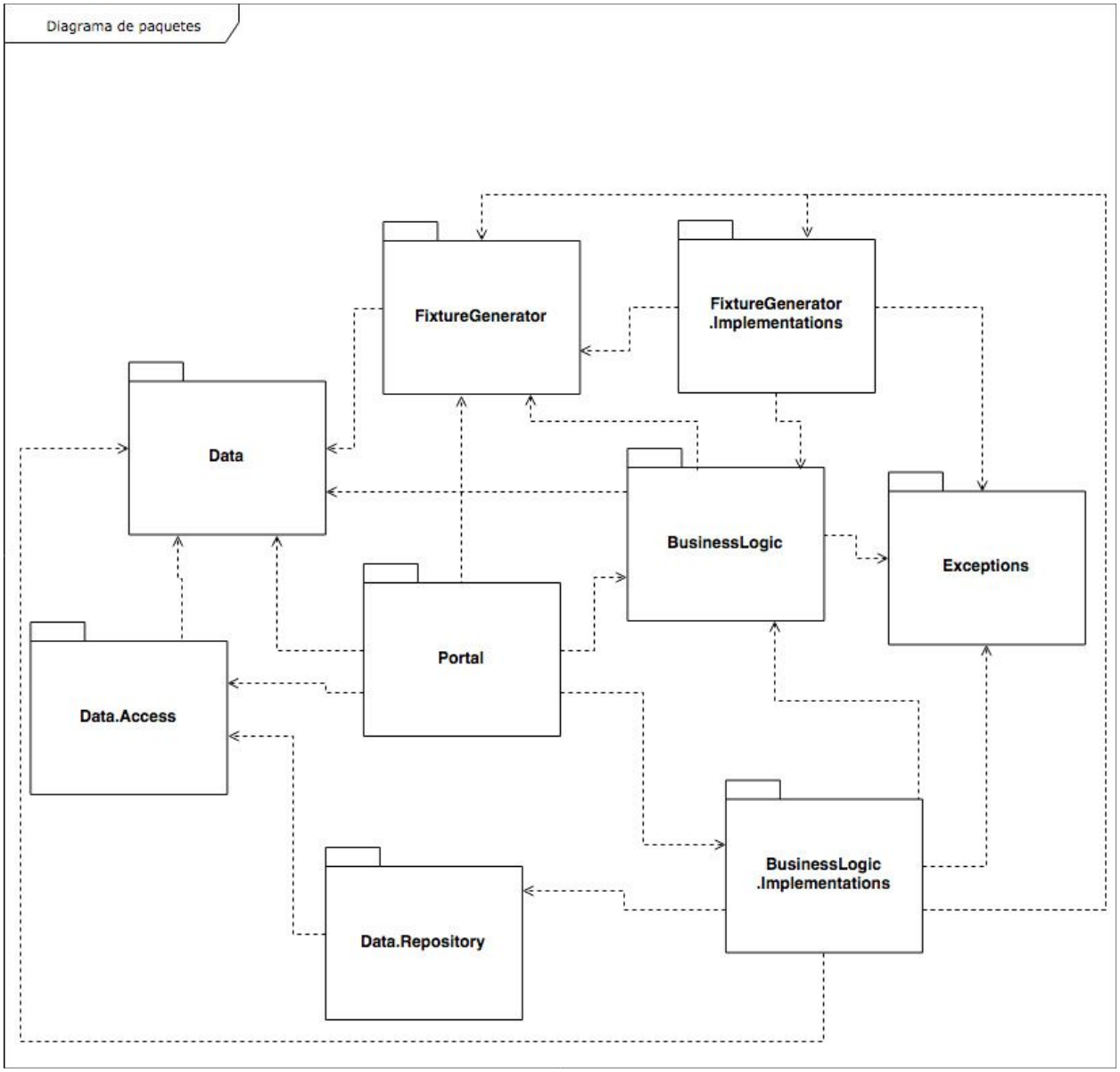


Diagrama de paquetes
FixtureGenerator
FixtureGenerator
.Implementations
Data
BusinessLogic
Exceptions
Portal
Data.Access
BusinessLogic
.Implementations
Data.Repository

En cuanto a la realización de las pruebas unitarias haciendo TDD, al comienzo pensamos en hacer las pruebas unitarias sin utilizar un *framework* de *mocking* y luego hacer un refactor para incorporarlo. Luego de discutirlo, entendimos que el esfuerzo de aprender a usar un framework y empezar a utilizarlo desde el arranque del proyecto no era tanto, entonces decidimos investigar la herramienta y comenzar a utilizarla.
Esto nos llevó a que al empezar a utilizarlo fuimos ganando experiencia y conocimiento, con lo cual logramos mejores pruebas, totalmente unitarias y aprendimos una nueva herramienta la cual ninguno de los dos integrantes conocíamos.

A medida que iban creciendo la cantidad de pruebas y la complejidad, nos dimos cuenta que íbamos repitiendo demasiado código en cada prueba, que podía ser compartido entre ellas. Para eso, empezamos a utilizar el *TestInitialize* para inicializar los objetos necesarios para las pruebas y tenerlos disponibles en cada una de ellas. Esto hizo que el desarrollo de las pruebas fuera más rápido, más ágil, que cada prueba tenga varias menos líneas y se entiendan aún mejor.

Para el desarrollo de la *WebAPI*, consideramos las buenas prácticas descritas en el curso. Los *endpoints* no tienen más de 3 niveles de profundidad. En los mismos, tratamos de seguir las siguientes convenciones:
/api/<nombre del controller>/, para traer todos los datos de ese *controller*. Ejemplo: /api/teams, trae todos los teams.
/api/<nombre del controller>/<id>/, para traer un objeto específico de ese *controller*. Ejemplo: */api/sports/1*, trae el *sport* con ID 1.
/api/<nombre del controller>/<id>/<subaccion>/, para traer de un objeto específico, más objetos que pueda tener este. Ejemplo */api/sports/1/teams*, trae todos los *teams* del *sport* con ID 1.
Otra de las convenciones que usamos fue el que los nombres de los controller sea en plural, ejemplo *TeamsController*, para el *controller* de *Teams*.
En ocasiones particulares, para el caso de los encuentros, en el *controller* de *Encounter*, dada las particularidades de las operaciones a realizar, algunos *endpoints* tienen nombres que no respetan la convención. Para el caso de obtener todos los encuentros de un deporte específico, la forma más clara, sin pasar a 4 niveles de profundidad en la URL, que encontramos fue nombrando al endpoint de la siguiente forma: */api/encounters/<entidad>/<id>*, ejemplo: */api/encounters/sports/1*, para traer todos los encuentros del deporte con ID 1; */api/encounters/bydate/<fecha>*, para traer todos los encuentros de la fecha.
Esto último consideramos que no es lo mejor, que existen mejores soluciones, pero fue la mejor solución que pudimos encontrar que no pase de 3 niveles en la URL.
La solución con 5 niveles que creíamos quizás era más entendible al leerlo era: */api/sports/<id del sport>/team/<id del team>/encounters/*, pero esto alargaba mucho la URL y no se apegaba en nada a las buenas prácticas.
La otra solución era una URL de la siguiente forma: *api/sports/<id del sport>/encounters*, pero esto necesitaba que en la lógica de deportes, tengamos a la lógica de encuentros, lo cual no creíamos que fuera una buena solución. Es por ello que decidimos romper con la convención que siguen todos los demás *endpoints* del sistema.

Dado que el sistema debe contar con 2 roles definidos, optamos por usar un *enum* con los 2 roles. Sabemos que no es la mejor implementación de un sistema de roles, pero dado que el cliente especifica que no van a haber subsecuentes roles agregados al sistema, no consideramos que sea ni necesario ni valioso implementar un sistema de roles más complejo. Estos roles, en específico el rol de administrador, es usado en un filtro de la *WebAPI* en el cual controlamos que, dado el token que recibimos en el *Authorization Header*, el usuario con ese token tenga el rol de administrador. Este filtro lo utilizamos en los *endpoints* que requieran que el usuario identificado tenga el rol de administrador.

Estas validaciones del requerimiento que para ciertas funciones el usuario debe ser administrador solo las realizamos usando el mencionado filtro porque consideramos que para este proyecto, que no va a cambiar la *WebAPI*, no es necesario realizar las validaciones en la lógica si ya restringimos el acceso del usuario si no tiene el rol en la puerta de entrada de la aplicación. Aún así, entendemos que lo mejor seria tener las validaciones del requerimiento en la lógica de negocio, para así poder reutilizar esa lógica de validación del rol y no tener que volver a implementarla si en algún momento se decide cambiar la *WebAPI* por una aplicación *WinForms* por ejemplo. En el contexto de una aplicación que no sabemos con certeza los cambios que pueda tener a futuro, nuestra implementación de la validación mostraría un mal diseño y complejiza el futuro mantenimiento; aún así, para este proyecto consideramos que con las restricciones de tiempo, es la mejor solución.

Dado que básicamente todas las interfaces de la lógica de negocio tienen los mismo métodos básicos de crear, actualizar, borrar y obtener, ser podría tener una clase abstracta con estos métodos para reutilizar código. Las clases que implementan las interfaces de la lógica de negocio también deberían heredar de la clase abstracta, hacer un *override* de los métodos, llamando a las validaciones que sean específicas de la clase y luego al *base()* para que se ejecute el método genérico. Si bien esto nos servirá para reutilizar código y tener código más mantenible, por razones de tiempo no pudimos implementarlo para esta iteración.

Para darle *feedback* al usuario que ejecuta la acción, además de los códigos de status básicos de HTTP, consideramos que es bueno además retornar, por ejemplo cuando se crea un equipo, el equipo creado. De esta manera, además de tener el código de status que puede ser satisfactorio o no, si es satisfactorio el usuario que llama a la acción puede ver un resultado de lo que ocurrió en el servidor. Y así además poder concatenar subsecuentes llamadas a la *WebAPI* dado que ya obtuvo el ID de la entidad que acaba de agregar, por ejemplo.
Esto lo implementamos ya cuando las funcionalidades estaban todas implementadas, por lo que tuvimos que hacer un refactor y además implementar *DTOs* para no devolver las entidades del negocio. Si bien esto consumió un tiempo de desarrollo, consideramos que es una mejor práctica el trabajar con *DTOs* para los datos que entran y salen, e internamente convertimos esos *DTOs* en las entidades del negocio, usando *AutoMapper*.

Esto nos permite agregar o quitar datos para ser usados internamente por la lógica pero que no queremos devolver al usuario por la razón que sea, por ejemplo, cuando se crea un usuario no queremos retornar la *password*; o cuando un usuario hace login, no es necesario que nos pase todos los datos de un usuario o datos incompletos, sino que tenemos un *DTO* que solo recibe la información necesaria para la autenticación.

3 Evidencia de Clean Code

No utilizamos comentarios en el código a excepción de algunos tests en los cuales consideramos que era necesario aclarar por si otro desarrollador mira el código y no entiende por qué determinado test está hecho así.

Lo que si intentamos usar en los métodos de las interfaces y métodos internos de algunas implementaciones fue usar Summary, para introducir documentación dentro del código. De esta manera el desarrollador que vaya a utilizar el método, sabe que es lo que hace con tan solo leer la descripción del método. Además, gracias al IntelliSense, esta descripción se muestra cuando estamos por llamar al método, sin necesidad de acceder al código del mismo.

```
/// <summary>
/// Validates business rules for a user.
/// </summary>
/// <param name="user"></param>
1 reference | Maximiliano Pollinger, 2 days ago | 1 author, 1 change | 0 exceptions
private void ValidateUser(User user)
```

```
/// <summary>
/// Throws exception if LoggedUser is not an admin.
/// </summary>
1 reference | Maximiliano Pollinger, 2 days ago | 1 author, 1 change | 0 exceptions
private void CheckIfLoggedUserIsAdmin()
```

```
9 references | Javier Aramberri, 6 days ago | 2 authors, 15
public interface ITeamBusinessLogic
{
    6 references | 4/4 passing | Javier Aramberri, 14 days
    void Add(Team team);
    4 references | 2/2 passing | Javier Aramberri, 14 days
    void Update(Team team);
    4 references | 2/2 passing | Javier Aramberri, 6 days
    void Delete(int id);
    4 references | Maximiliano Pollinger, 7 days ago | 1 a
    void CheckIfExists(int teamId);
    3 references | 1/1 passing | Javier Aramberri, 8 days
    IEnumerable<Team> GetAll();
    4 references | 2/2 passing | Maximiliano Pollinger, 7
    Team GetById(int teamId);
```

Capítulo 3 - Funciones

Consideramos que respetamos lo dicho en este capítulo dado que las funciones no exceden las 30 líneas, las funciones son pequeñas y unitarias, hacen una sola cosa y la hacen bien.

```
private bool ValidateEmail(string email)
{
    return Regex.IsMatch(email,
        @"^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$");
}
```

```
public void Update(User user)
{
    CheckIfExists(user.Id);
    CheckIfLoggedUserIsAdmin();
    repository.Update(user);
    repository.Save();
}
```

```
public void Add(Sport sport)
{
    ValidateSport(sport);
    repository.Insert(sport);
    repository.Save();
}
```

Capítulo 5 - Formato

Consideramos que respetamos este capítulo dado que en el sentido de formato horizontal, las líneas no exceden los 120 caracteres de largo, lo cual no obliga al programador a tener que desplazarse a la derecha ni está yendo y viniendo de un lado a otro para poder entender la línea.

Usamos líneas en blanco como separadores, ya sea en una misma función o en una clase, para poder separar conceptos que están directamente relacionados con otros que lo están menos.

```
foreach (Team team in teams)
{
    teamList.Remove(team);
    foreach (Team rival in teamList)
    {
        Encounter encounter = new Encounter() { Team1 = t
        while (encounterBL.TeamsHaveEncountersOnTheSameDa
        {
            encounter.Date = encounter.Date.AddDays(1);
        }

        Encounter encounter2 = new Encounter() { Team1 =
        while (encounterBL.TeamsHaveEncountersOnTheSameDa
        {
            encounter2.Date = encounter2.Date.AddDays(1);
        }

        encounters.Add(encounter);
        encounters.Add(encounter2);
    }
}
```

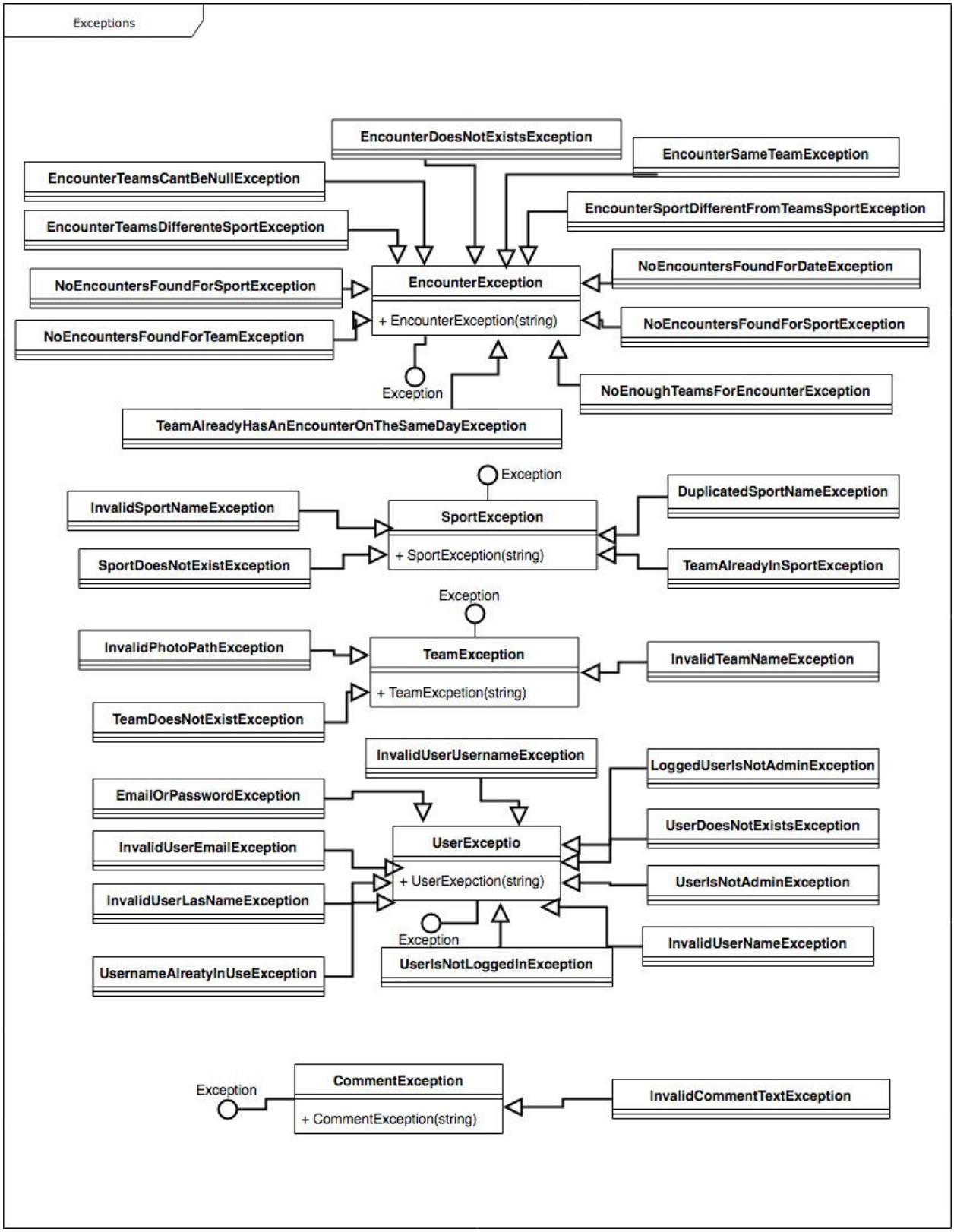
```
if (String.IsNullOrWhiteSpace(user.Name))
{
    throw new InvalidUserNameException();
}

if (String.IsNullOrWhiteSpace(user.Username))
{
    throw new InvalidUserUsernameException();
}

if (String.IsNullOrWhiteSpace(user.Email) || !V
{
    throw new InvalidUserEmailException();
}

if (String.IsNullOrWhiteSpace(user.LastName))
{
    throw new InvalidUserLastNameException();
}
```

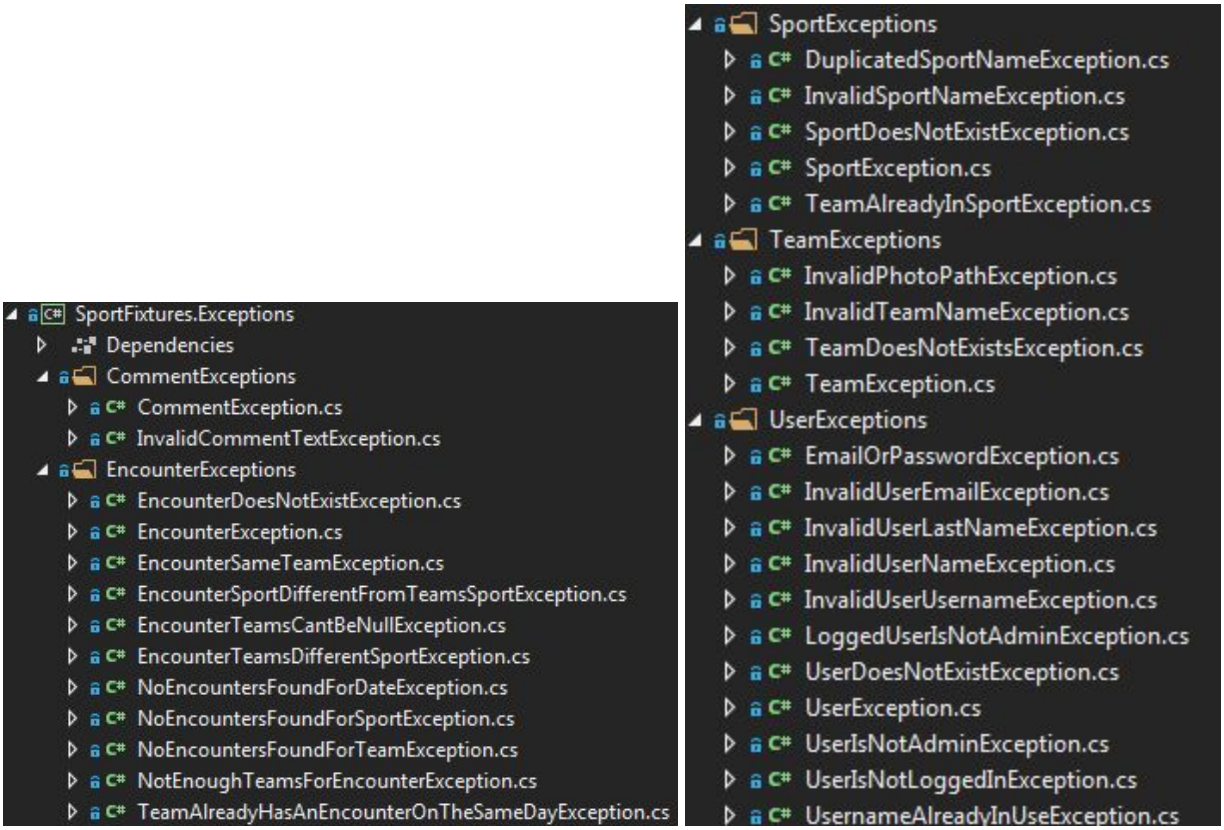
Capítulo 7 - Procesar errores

Utilizamos excepciones en lugar de códigos de error o retornar booleanos. Esto facilita la implementación de métodos, emprolija el código y no tenemos que estar recordando o yendo a buscar referencias de qué significa cada código.

Organizamos las excepciones en un proyecto separado, en el cual además organizamos las excepciones por clase, es decir, las de Sport están separadas de las de Team y de las de User.

En el siguiente diagrama se puede ver a alto nivel como íbamos a implementar las excepciones y luego como quedaron plasmadas a nivel de código en las imágenes subsiguientes al diagrama.


Exceptions
EncounterDoesNotExistsException
EncounterSameTeamException
EncounterTeamsCantBeNullException
EncounterSportDifferentFromTeamsSportException
EncounterTeamsDifferenteSportException
NoEncountersFoundForDateException
NoEncountersFoundForSportException
EncounterException
+ EncounterException(string)
NoEncountersFoundForSportException
NoEncountersFoundForTeamException
Exception
NoEnoughTeamsForEncounterException
TeamAlreadyHasAnEncounterOnTheSameDayException
Exception
InvalidSportNameException
SportException
DuplicatedSportNameException
+ SportException(string)
SportDoesNotExistException
TeamAlreadyInSportException
Exception
InvalidPhotoPathException
TeamException
InvalidTeamNameException
+ TeamExcpetion(string)
TeamDoesNotExistException
InvalidUserUsernameException
LoggedUserIsNotAdminException
EmailOrPasswordException
UserExceptio
UserDoesNotExistsException
InvalidUserEmailException
+ UserExepction(string)
UserIsNotAdminException
InvalidUserLasNameException
Exception
InvalidUserNameException
UserIsNotLoggedInException
UsernameAlreatyInUseException
Exception
CommentException
InvalidCommentTextException
+ CommentException(string)



Además, utilizamos una excepción genérica para cada clase y luego las excepciones específicas heredan de esta excepción, pero introducen un mensaje específico.

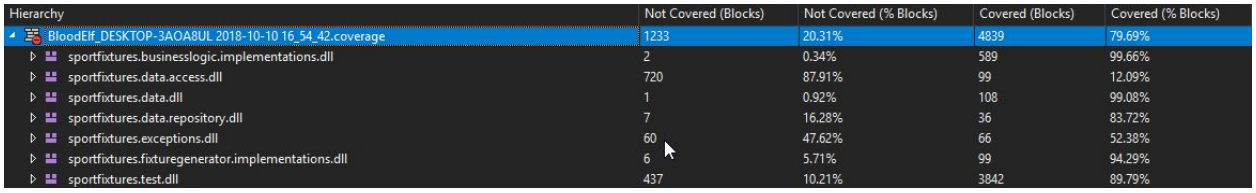
```
namespace SportFixtures.Exceptions.SportExceptions
{
    5 references | Maximiliano Pollinger, 10 days ago | 1 author, 1 change
    public class SportException : Exception
    {
        8 references | Maximiliano Pollinger, 10 days ago | 1 author, 1 change | 0 exceptions
        public SportException(string message) : base(message)
        {
        }
    }
}
```

```
6 references | Maximiliano Pollinger, 4 days ago | 1 author, 1 change
public class SportDoesNotExistException : SportException
{
    2 references | Maximiliano Pollinger, 4 days ago | 1 author, 1 change | 0 exceptions
    public SportDoesNotExistException() : base("Sport does not exist.")
    {
    }

    0 references | Maximiliano Pollinger, 4 days ago | 1 author, 1 change | 0 exceptions
    public SportDoesNotExistException(string message) : base(message)
    {
    }
}
```

Por estas y otras aplicaciones de los capítulos del libro de Clean Code, consideramos que el código de la solución se puede considerar código limpio.

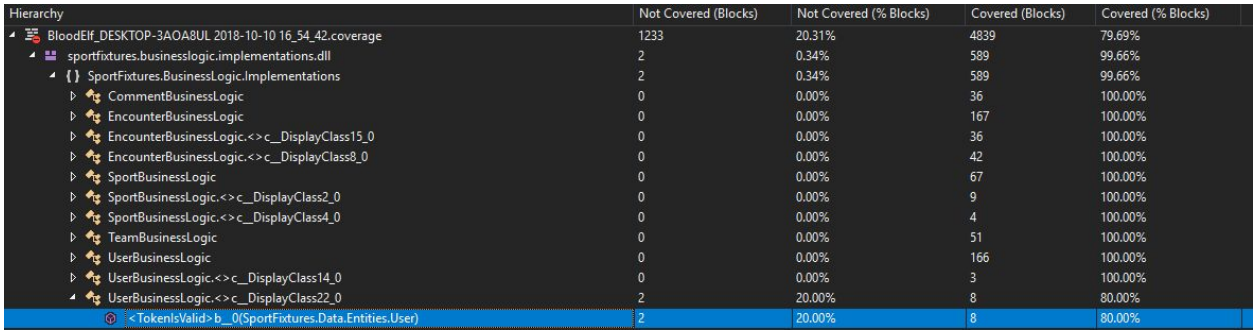
Evidencia de TDD

Para evidenciar que se realizó TDD, podemos mostrar la cobertura de las pruebas unitarias realizadas. Como se muestra en las imágenes, llegamos casi al 100% de cobertura. No llegamos a 100% por la estructura interna de algunos métodos que dificultan su prueba.



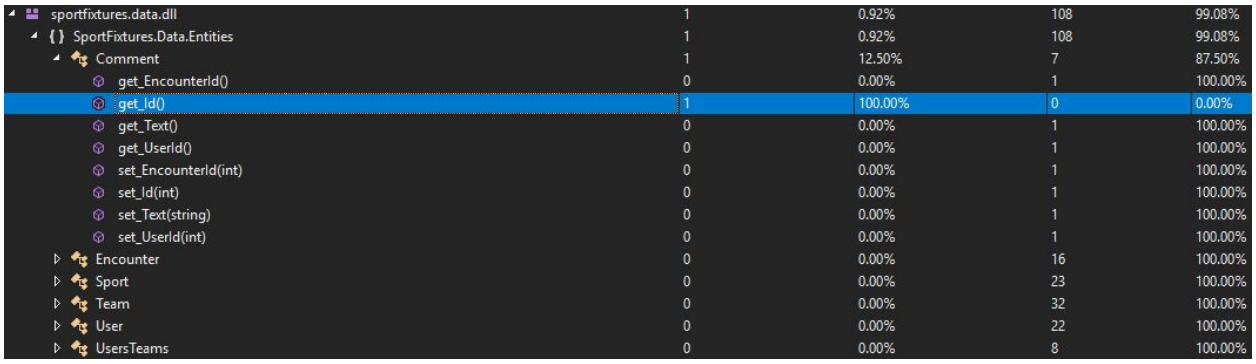
Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
BloodElf_DESKTOP-3AOA8UL 2018-10-10 16_54_42.coverage	1233	20.31%	4839	79.69%
sportfixtures.businesslogic.implementations.dll	2	0.34%	589	99.66%
sportfixtures.data.access.dll	720	87.91%	99	12.09%
sportfixtures.data.dll	1	0.92%	108	99.08%
sportfixtures.data.repository.dll	7	16.28%	36	83.72%
sportfixtures.exceptions.dll	60	47.62%	66	52.38%
sportfixtures.fixturegenerator.implementations.dll	6	5.71%	99	94.29%
sportfixtures.test.dll	437	10.21%	3842	89.79%

Consideramos que los paquetes más importantes que deberíamos tener la mayor cobertura eran el de la lógica de negocio y el de las entidades del negocio.



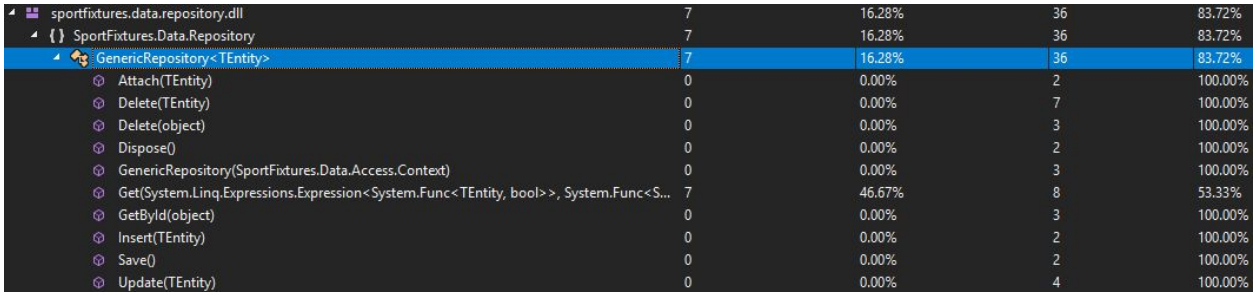
Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
BloodElf_DESKTOP-3AOA8UL 2018-10-10 16_54_42.coverage	1233	20.31%	4839	79.69%
sportfixtures.businesslogic.implementations.dll	2	0.34%	589	99.66%
SportFixtures.BusinessLogic.Implementations	2	0.34%	589	99.66%
CommentBusinessLogic	0	0.00%	36	100.00%
EncounterBusinessLogic	0	0.00%	167	100.00%
EncounterBusinessLogic.<>c__DisplayClass15_0	0	0.00%	36	100.00%
EncounterBusinessLogic.<>c__DisplayClass8_0	0	0.00%	42	100.00%
SportBusinessLogic	0	0.00%	67	100.00%
SportBusinessLogic.<>c__DisplayClass2_0	0	0.00%	9	100.00%
SportBusinessLogic.<>c__DisplayClass4_0	0	0.00%	4	100.00%
TeamBusinessLogic	0	0.00%	51	100.00%
UserBusinessLogic	0	0.00%	166	100.00%
UserBusinessLogic.<>c__DisplayClass14_0	0	0.00%	3	100.00%
UserBusinessLogic.<>c__DisplayClass22_0	2	20.00%	8	80.00%
<TokenIsValid>b__0(SportFixtures.Data.Entities.User)	2	20.00%	8	80.00%

Como se muestra en las imágenes, creemos haber logrado lo dicho de tener la mayor cobertura en estos paquetes.



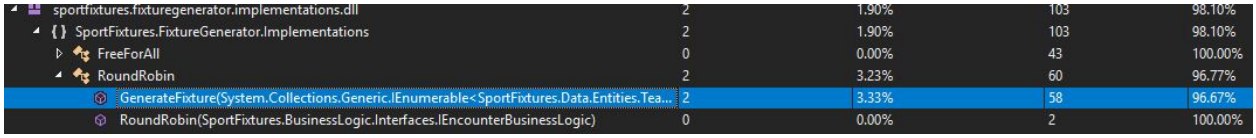
sportfixtures.data.dll	1	0.92%	108	99.08%
SportFixtures.Data.Entities	1	0.92%	108	99.08%
Comment	1	12.50%	7	87.50%
get_EncounterId()	0	0.00%	1	100.00%
get_Id()	1	100.00%	0	0.00%
get_Text()	0	0.00%	1	100.00%
get_UserId()	0	0.00%	1	100.00%
set_EncounterId(int)	0	0.00%	1	100.00%
set_Id(int)	0	0.00%	1	100.00%
set_Text(string)	0	0.00%	1	100.00%
set_UserId(int)	0	0.00%	1	100.00%
Encounter	0	0.00%	16	100.00%
Sport	0	0.00%	23	100.00%
Team	0	0.00%	32	100.00%
User	0	0.00%	22	100.00%
UsersTeams	0	0.00%	8	100.00%

Si bien el patrón repositorio, tanto la implementación como la interfaz usamos la brindada en el ejemplo de Microsoft, también hicimos pruebas unitarias para corroborar el correcto funcionamiento de la implementación, ya que todo el acceso a los datos depende de esta implementación. A su vez, agregamos métodos como el Attach(), Dispose() y Save().



sportfixtures.data.repository.dll	7	16.28%	36	83.72%
SportFixtures.Data.Repository	7	16.28%	36	83.72%
GenericRepository<TEntity>	7	16.28%	36	83.72%
Attach(TEntity)	0	0.00%	2	100.00%
Delete(TEntity)	0	0.00%	7	100.00%
Delete(object)	0	0.00%	3	100.00%
Dispose()	0	0.00%	2	100.00%
GenericRepository(SportFixtures.Data.Access.Context)	0	0.00%	3	100.00%
Get(System.Linq.Expressions.Expression<System.Func<TEntity, bool>>, System.Func<S...	7	46.67%	8	53.33%
GetById(object)	0	0.00%	3	100.00%
Insert(TEntity)	0	0.00%	2	100.00%
Save()	0	0.00%	2	100.00%
Update(TEntity)	0	0.00%	4	100.00%

También realizamos pruebas unitarias para las implementaciones de los algoritmos de generación de encuentros.



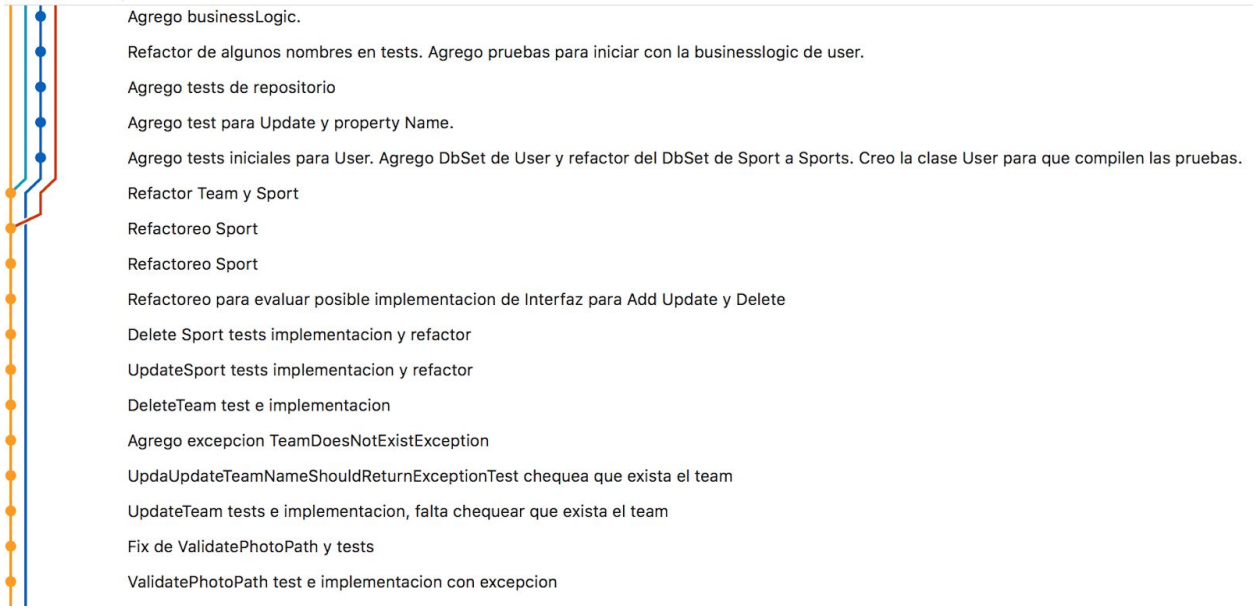
sportfixtures.fixturegenerator.implementations.dll	2	1.90%	103	98.10%
{ } SportFixtures.FixtureGenerator.Implementations	2	1.90%	103	98.10%
FreeForAll	0	0.00%	43	100.00%
RoundRobin	2	3.23%	60	96.77%
GenerateFixture(System.Collections.Generic.IEnumerable<SportFixtures.Data.Entities.Tea...	2	3.33%	58	96.67%
RoundRobin(SportFixtures.BusinessLogic.Interfaces.IEncounterBusinessLogic)	0	0.00%	2	100.00%

En resumen, consideramos que la cobertura de código obtenida gracias a TDD y gracias al resto de las pruebas que hicimos, logramos una cobertura del código satisfactoria.

Para la creación de las entidades y la lógica de negocio utilizamos la metodología TDD, buscando aumentar la calidad del código al obtener feedback inmediato sobre la implementación de cada método y mejorar, en cada refactor, la entendibilidad del mismo.
A la hora de implementar un método, buscamos primero crear una prueba que testee la mínima funcionalidad del mismo y, a medida que lo fuimos implementando aumentamos el alcance de los tests buscando abarcar todos los casos posibles para cada método, siempre buscando testear funcionalidad antes de implementarla.

En cuanto a Repository, no aplicamos TDD ya que utilizamos la implementación brindada por Microsoft. Igualmente, hicimos los tests correspondientes para probar cada uno de los métodos, pero sin aplicar el ciclo planteado por la metodología.

Optamos por no testear la API ya que esta no tiene lógica, por lo tanto sería redundante hacer tests para métodos que lo único que hacen son llamadas a la lógica ya testeada.

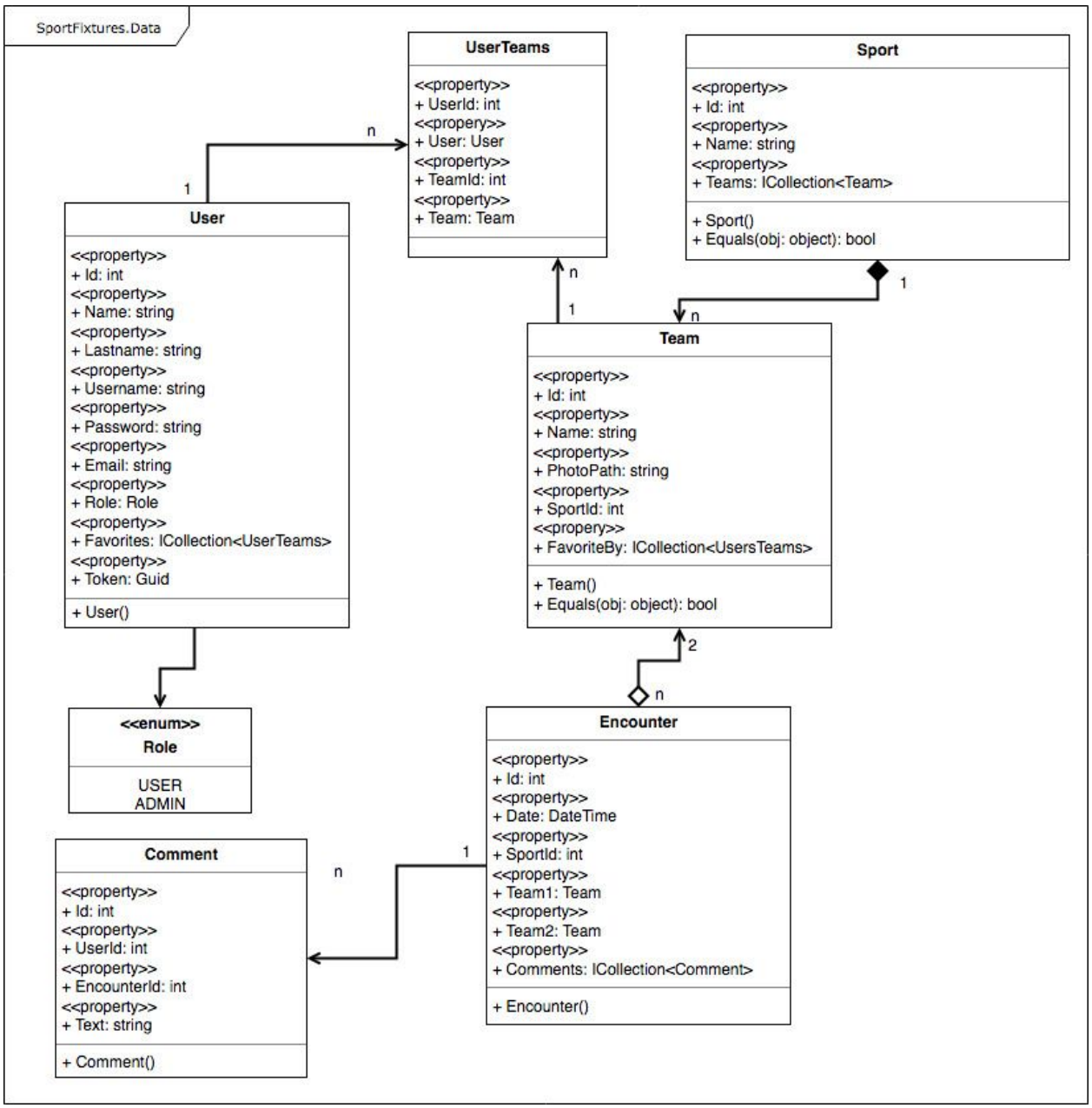


Agrego businessLogic.
Refactor de algunos nombres en tests. Agrego pruebas para iniciar con la businesslogic de user.
Agrego tests de repositorio
Agrego test para Update y property Name.
Agrego tests iniciales para User. Agrego DbSet de User y refactor del DbSet de Sport a Sports. Creo la clase User para que compilen las pruebas.
Refactor Team y Sport
Refactoreo Sport
Refactoreo Sport
Refactoreo para evaluar posible implementacion de Interfaz para Add Update y Delete
Delete Sport tests implementacion y refactor
UpdateSport tests implementacion y refactor
DeleteTeam test e implementacion
Agrego excepcion TeamDoesNotExistException
UpdaUpdateTeamNameShouldReturnExceptionTest chequea que exista el team
UpdateTeam tests e implementacion, falta chequear que exista el team
Fix de ValidatePhotoPath y tests
ValidatePhotoPath test e implementacion con excepcion

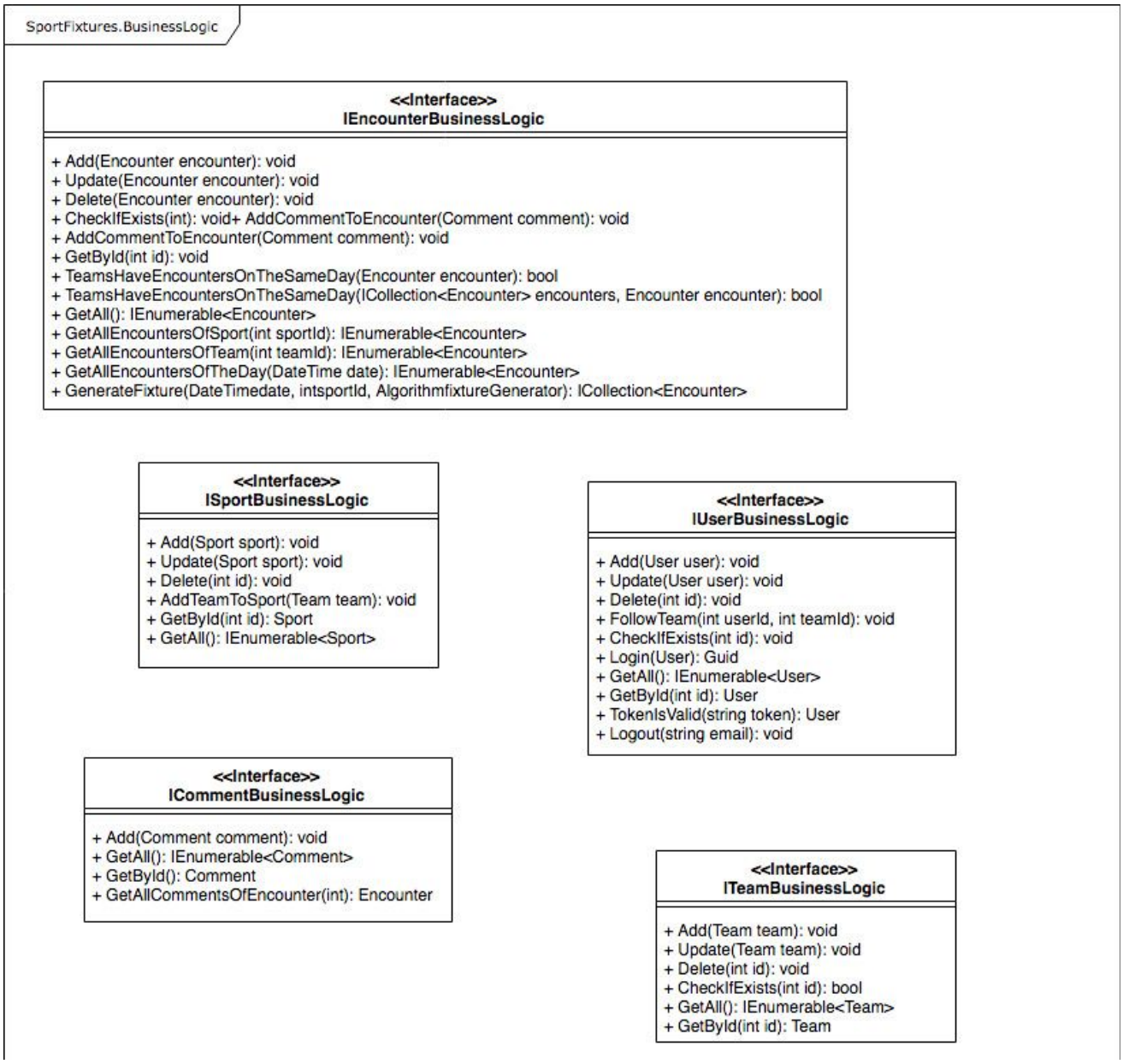
4 Anexo I - Diagramas

4.1 Diagramas de Clases

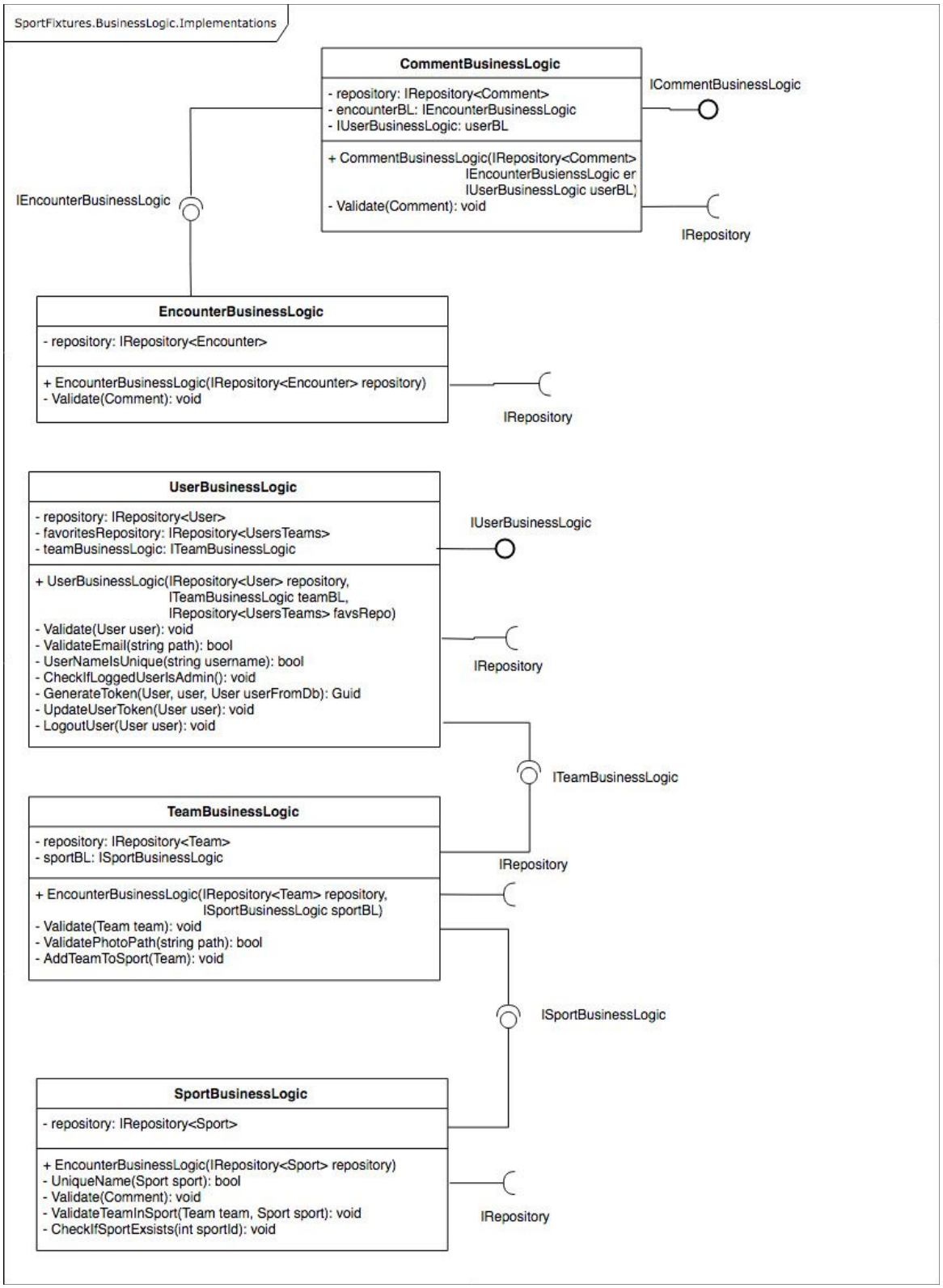
4.1.1 SportFixtures.Data



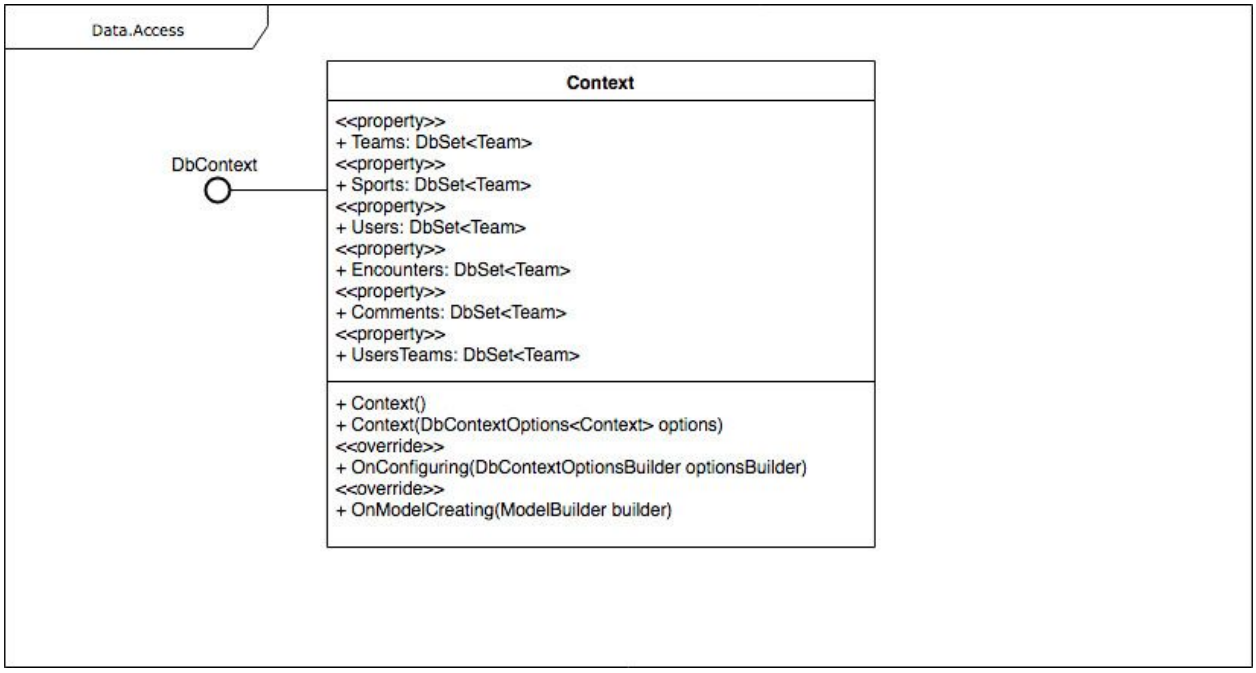
4.1.2 SportFixtures.BusinessLogic



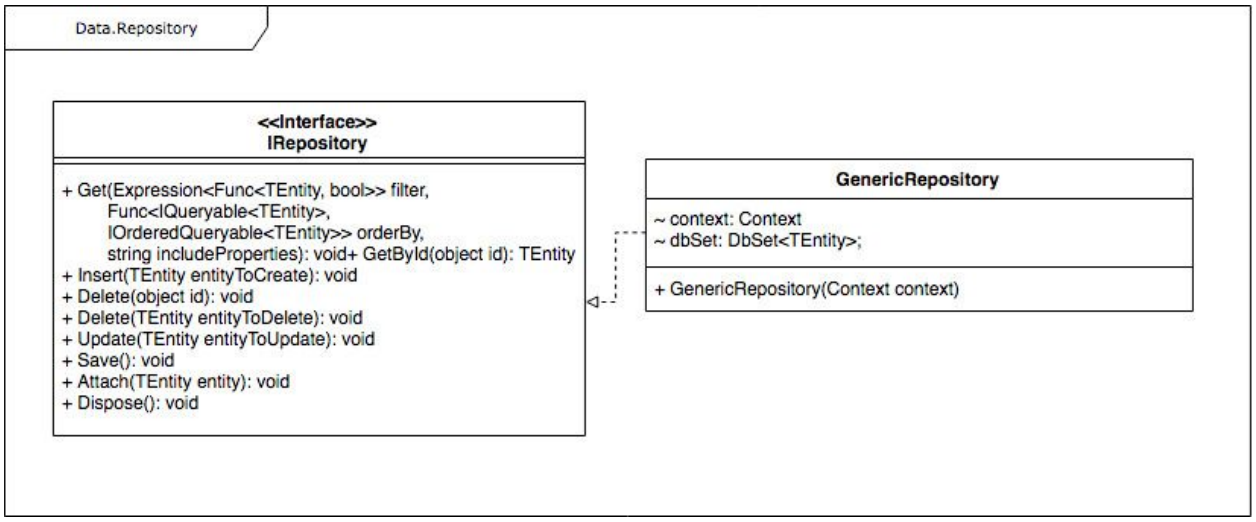
4.3 SportFixtures.BusinessLogic.Implementations



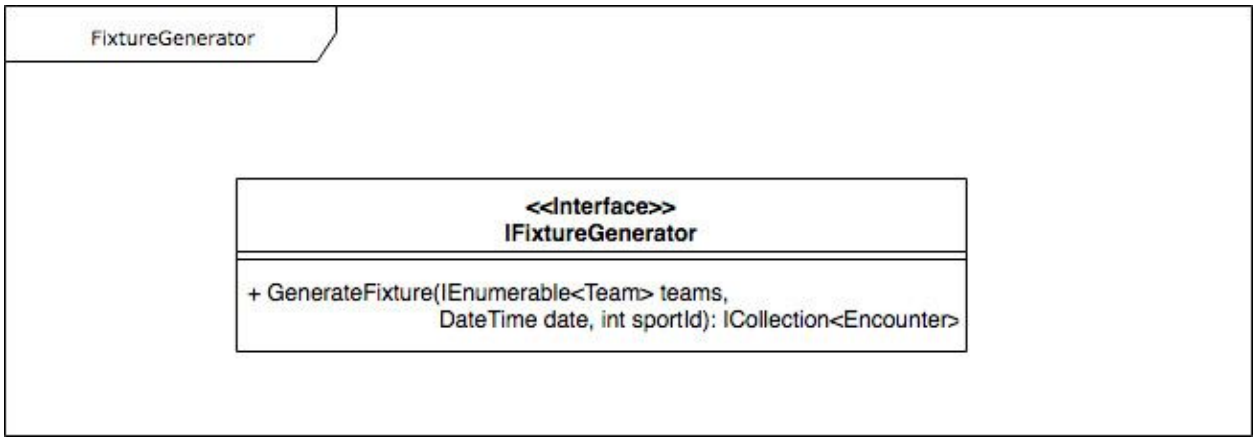
SportFixtures.Data.Access



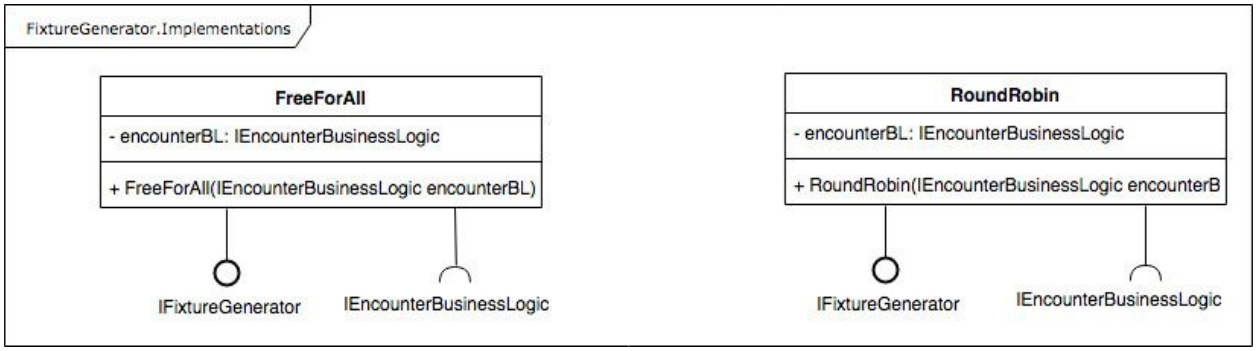
4.1.4 SportFixtures.Repository



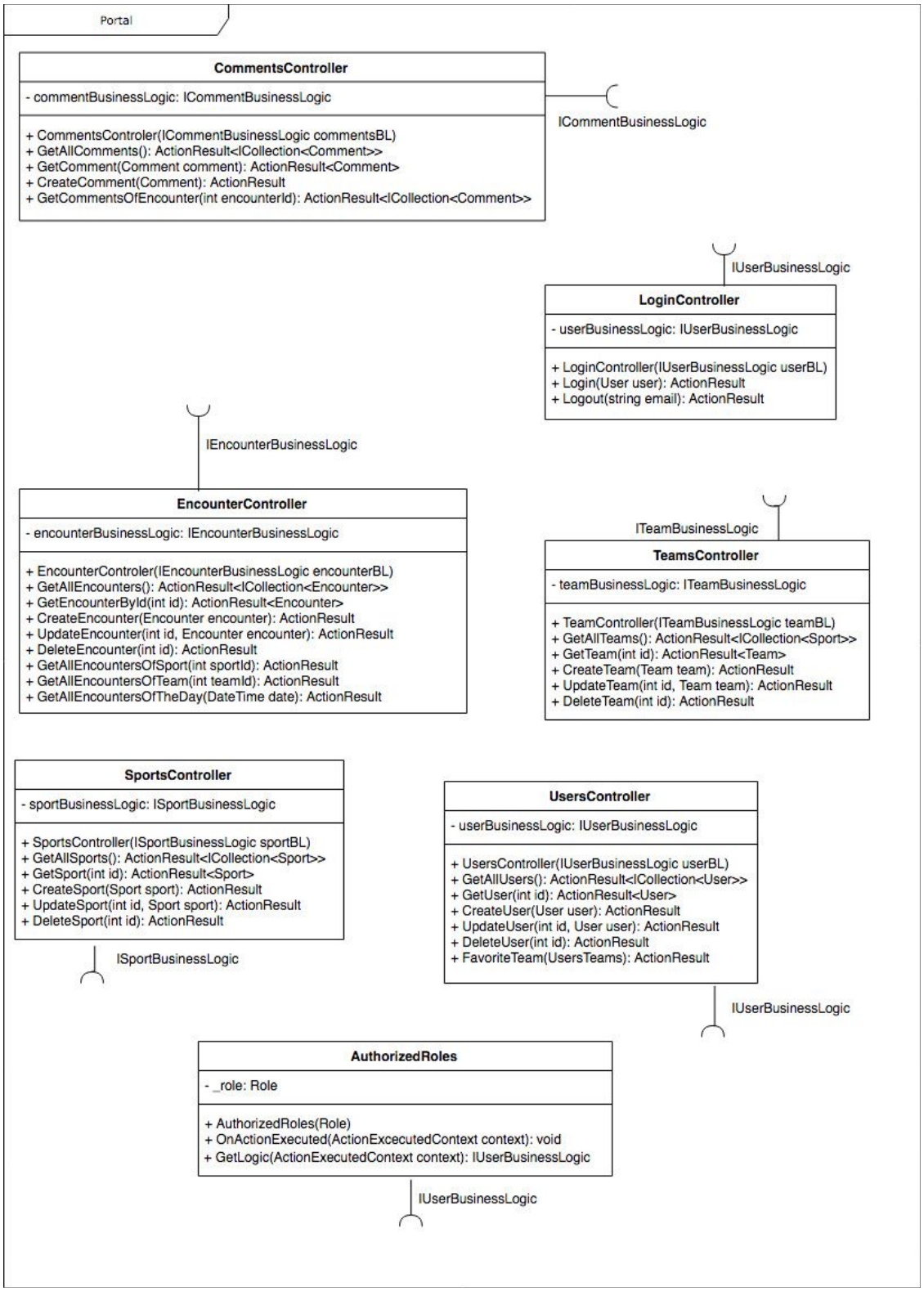
4.1.5 SportFixtures.FixtureGenerator



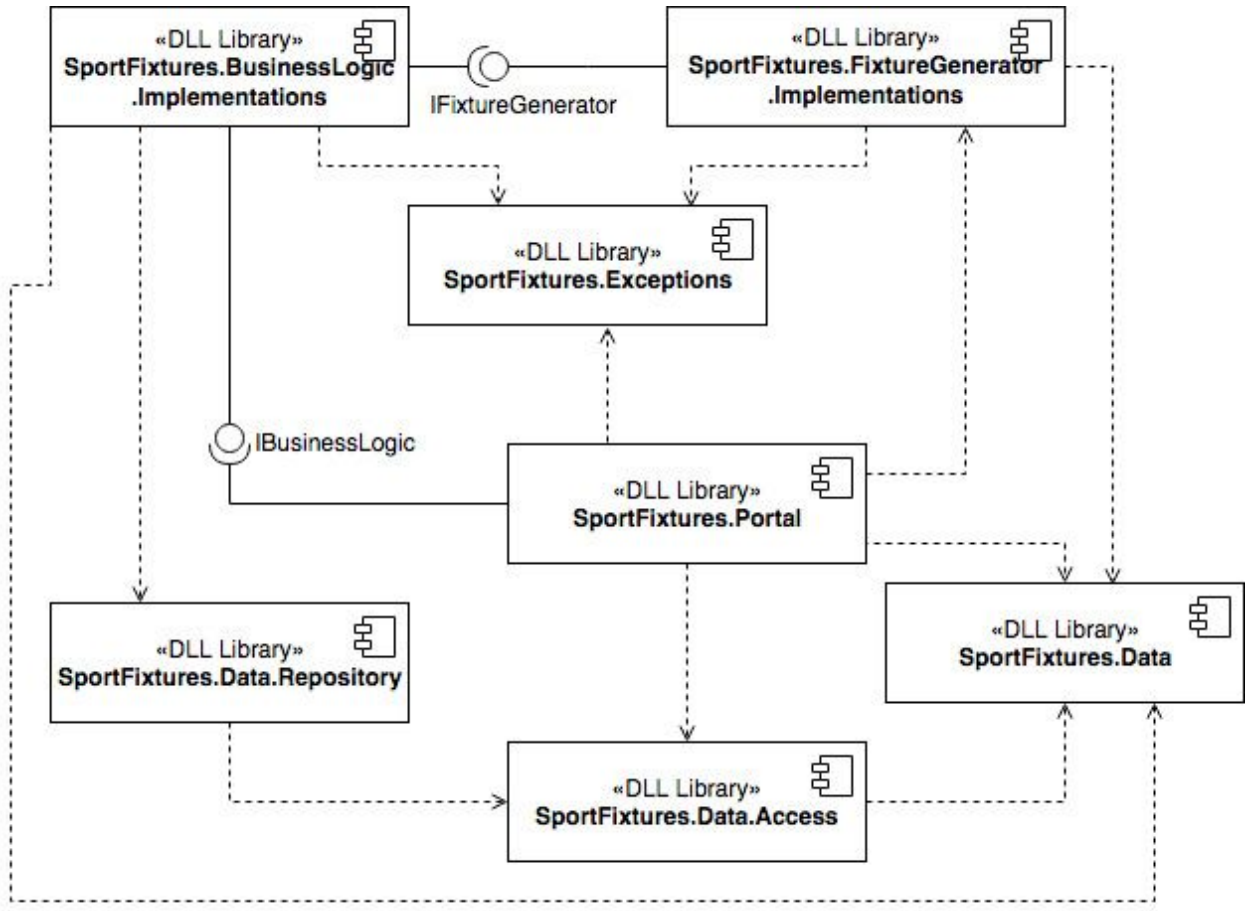
4.1.6 SportFixtures.FixtureGenerator.Implementations



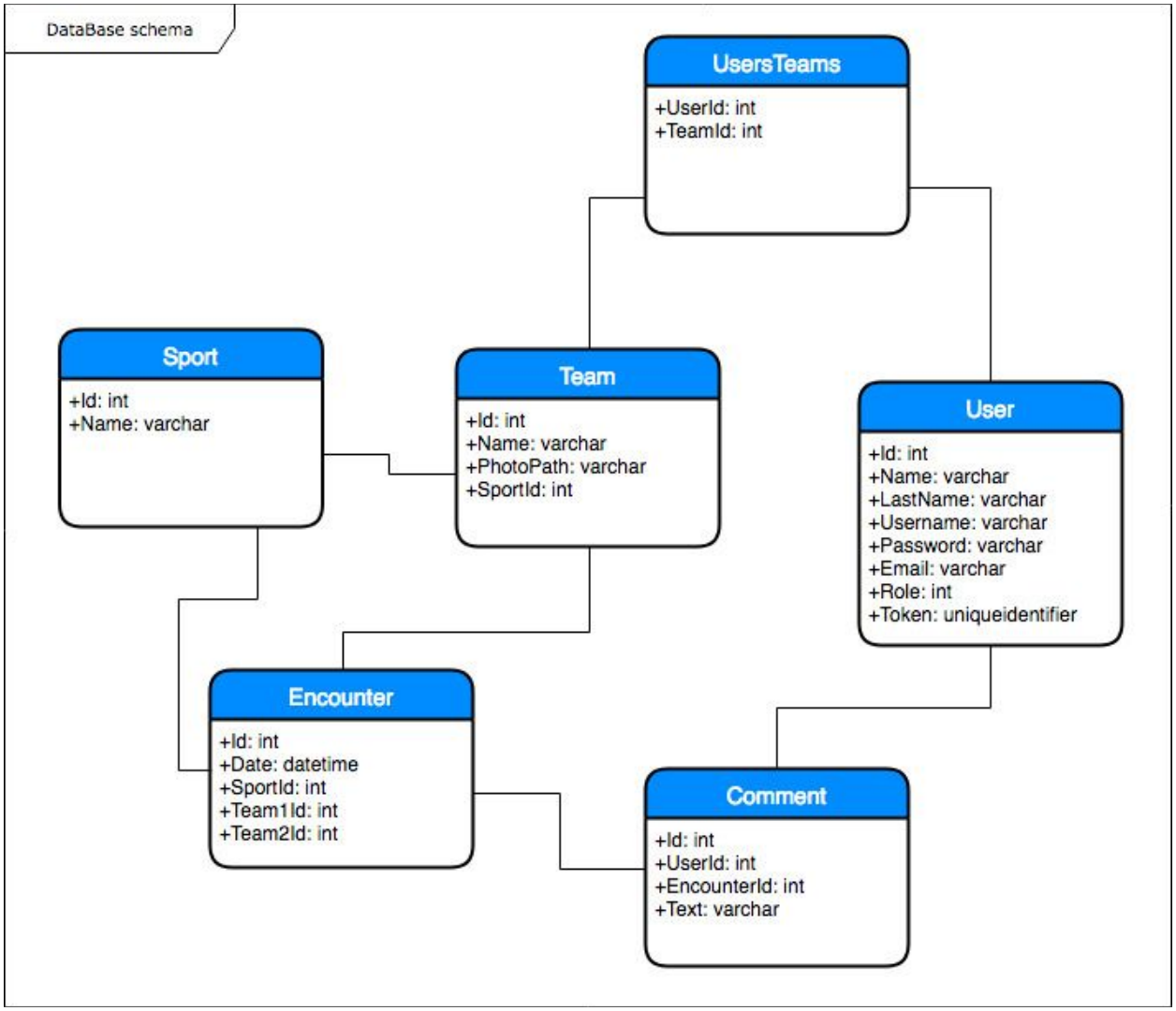
4.1.7 SportFixtures.Portal



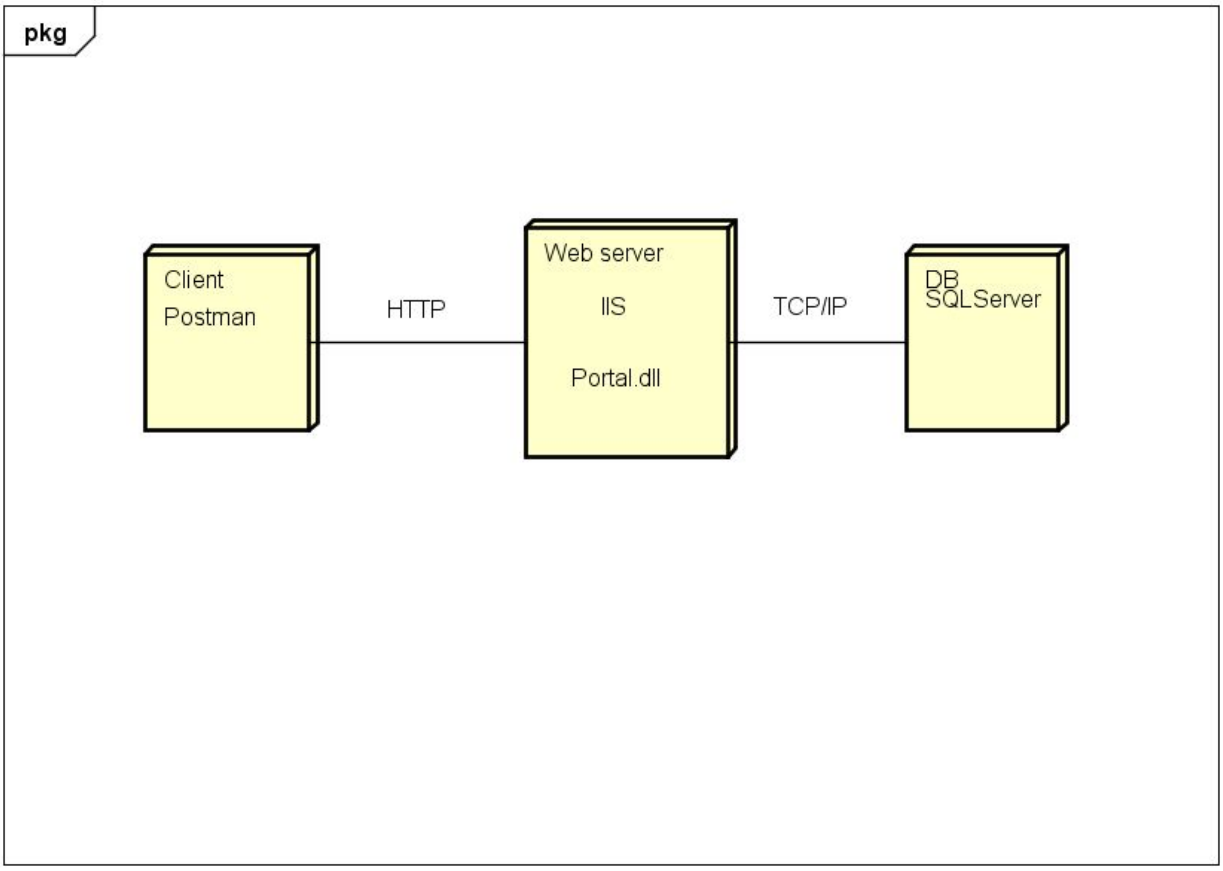
4.2 Diagrama de componentes



4.3 Diagrama base de datos

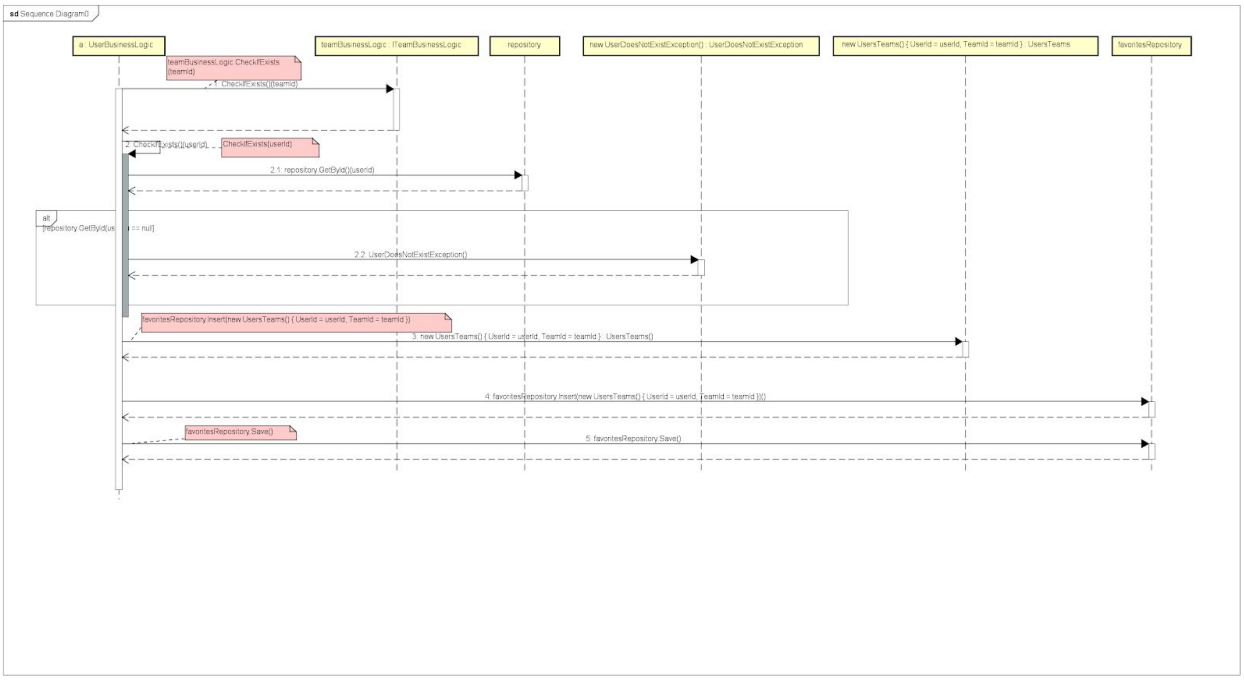


4.4 Diagrama de entrega



4.5 Diagramas de Interacción

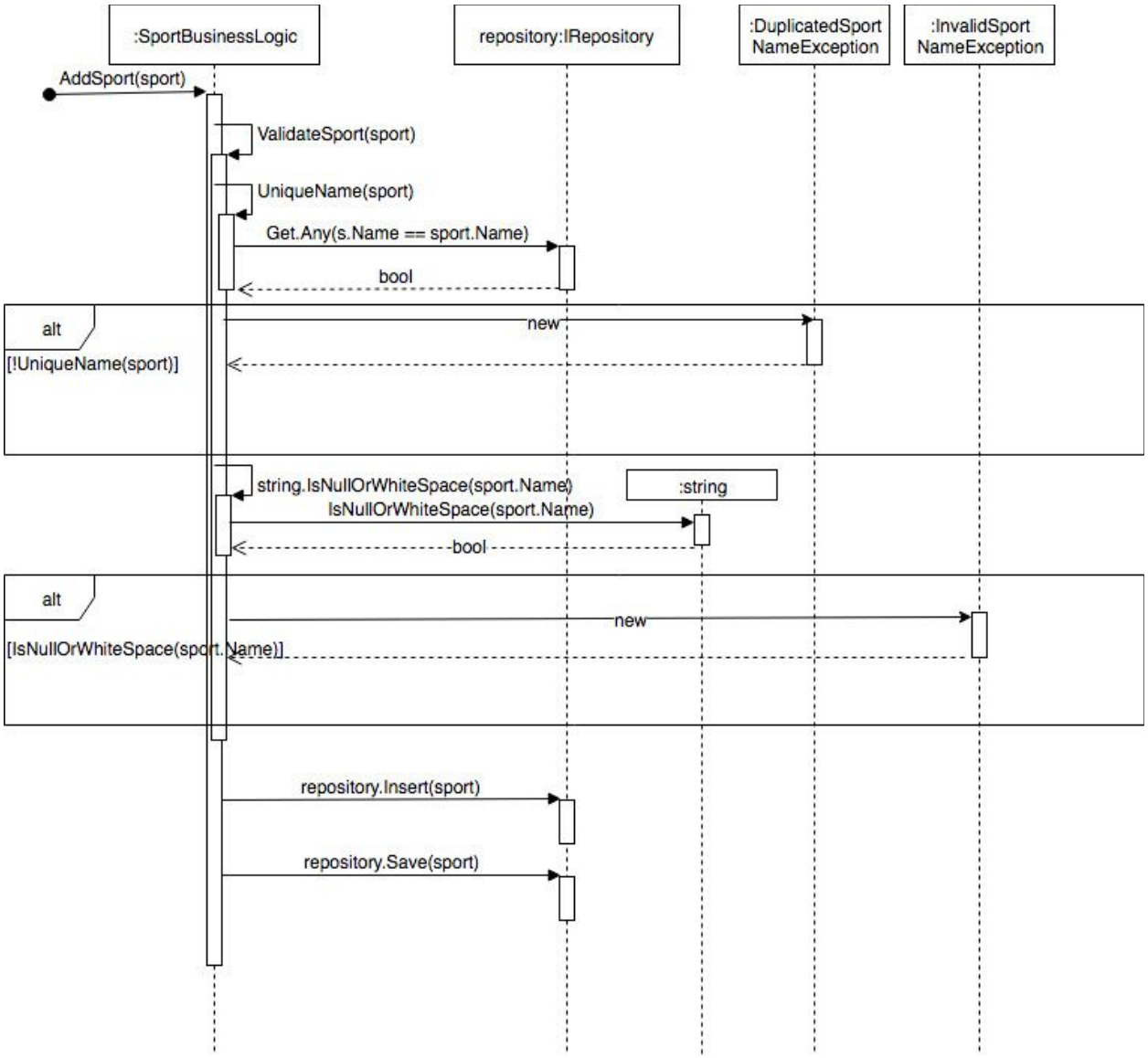
4.5.1 Follow Team



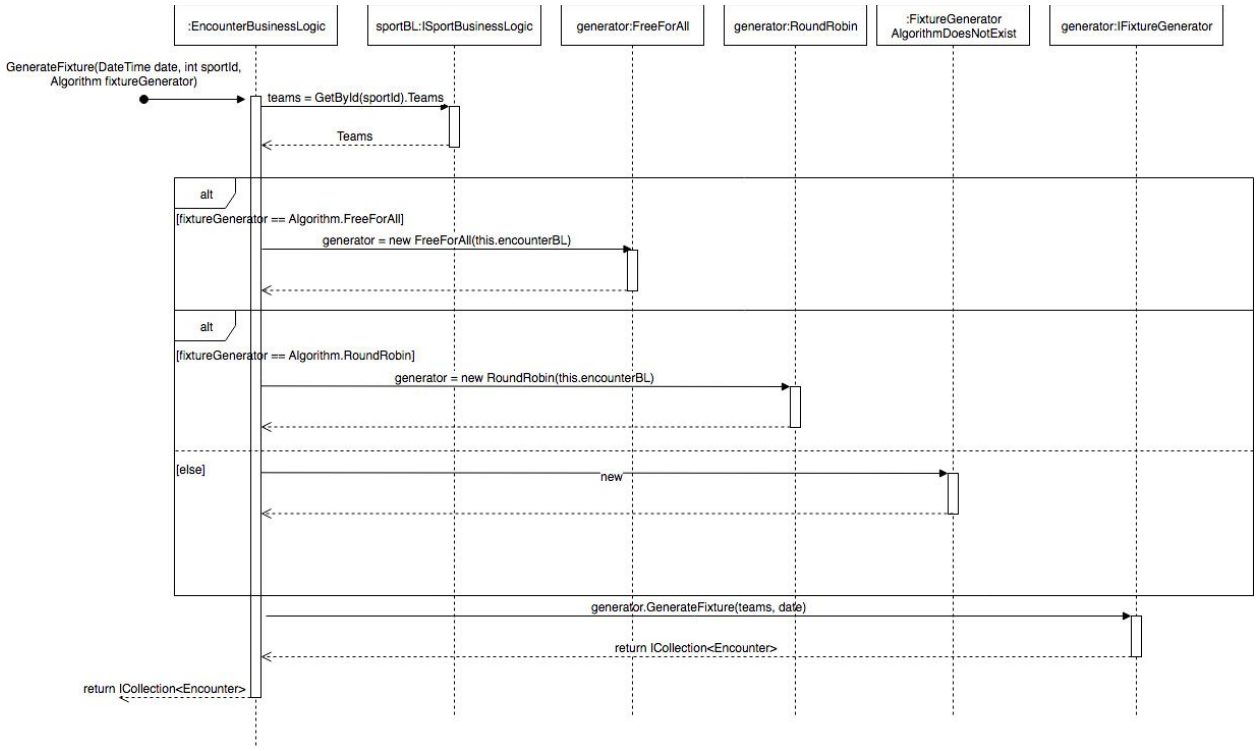
4.5.2 Login


sd Sequence Diagram0
a : UserBusinessLogic
repository
users
new UserDoesNotExistException() : UserDoesNotExistException
user : User
user.Email : string
user.Password : string
new EmailOrPasswordException() : EmailOrPasswordException
userFromDb : User
var users = repository.Get(u => u.Email == user.Email, null, "");
1: repository.Get(u => u.Email == user.Email, null, "")
2: users.Count()
alt
[users.Count() > 0]
throw new UserDoesNotExistException()
3: UserDoesNotExistException()
var userFromDb = users.FirstOrDefault();
4: users.FirstOrDefault()
<<get>>
5: Email()
6: user.Email.Equals(userFromDb.Email)
<<get>>
7: Password()
8: user.Password.Equals(userFromDb.Password)
alt
[!user.Email.Equals(userFromDb.Email) && !user.Password.Equals(userFromDb.Password)]
throw new EmailOrPasswordException()
9: EmailOrPasswordException()
var token = GenerateToken(user, userFromDb);
10: GenerateToken(user, userFromDb)
var token = Guid.NewGuid();
LoggedUser = user
<<set>>
10.1: LoggedUser()
userFromDb.Token = token
<<set>>
10.2: Token()
UpdateUserToken(userFromDb)
10.3: UpdateUserToken(userFromDb)
repository.Update(user)
10.3.1: repository.Update(user)
repository.Save()
10.3.2: repository.Save()
return token
return token

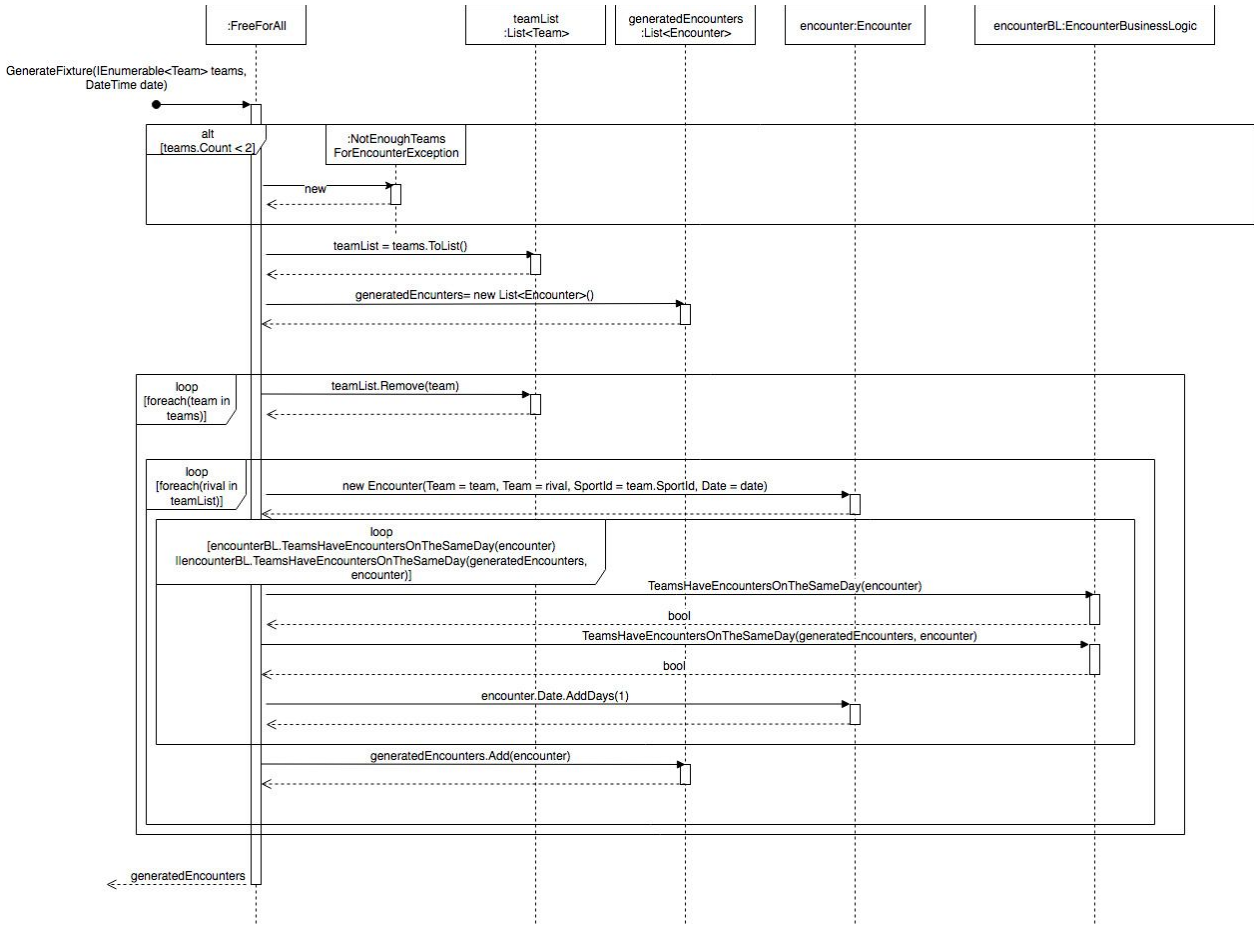
4.5.3 Add Sport



4.5.4 GenerateFixture



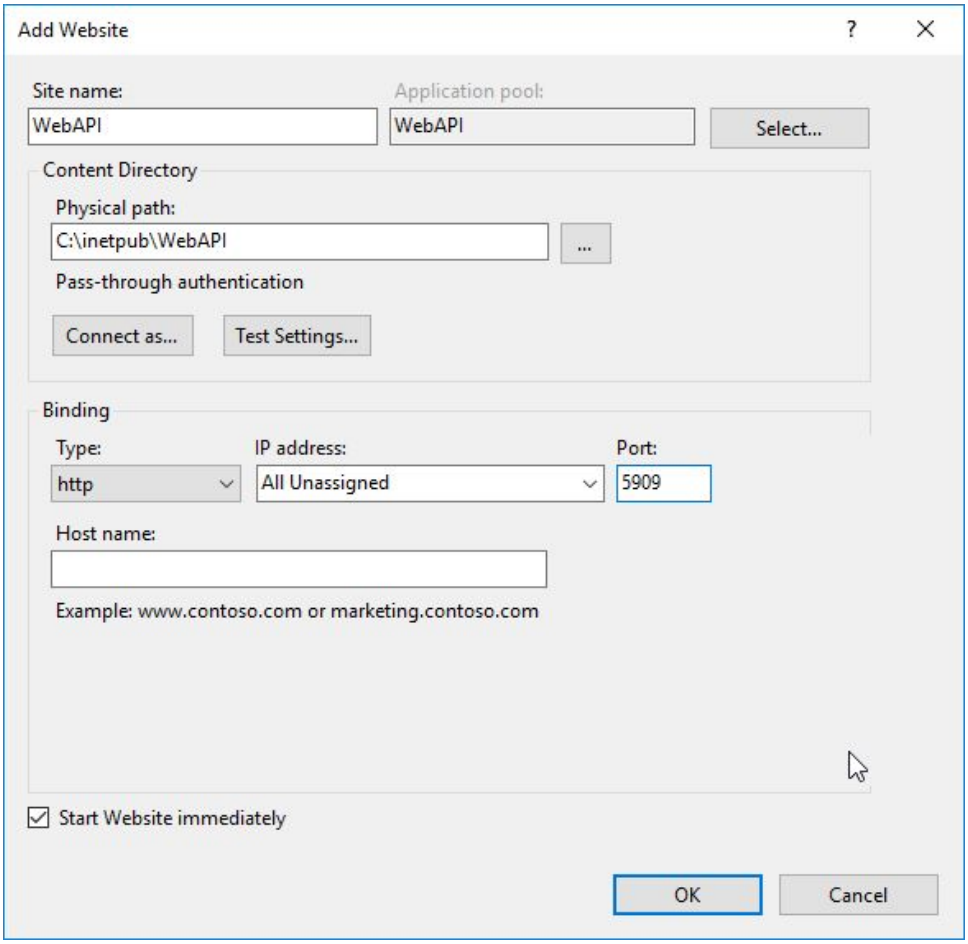
4.5.5 GenerateFixture - Algoritmo Free for all



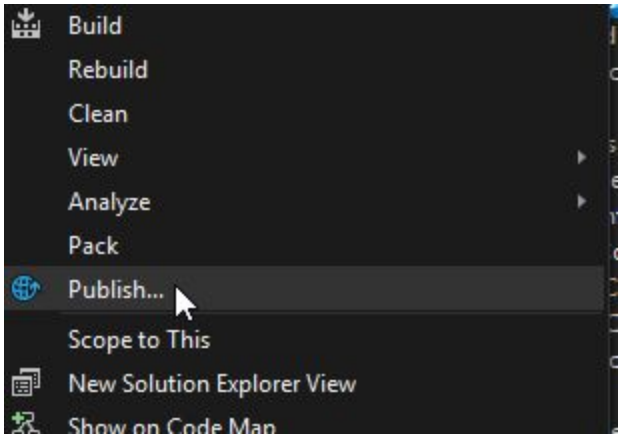
5 Anexo II - Instalación de WebAPI

Para poder hacer uso de la WebAPI se debe contar con un servidor web, en este caso usaremos IIS, que es el servidor de aplicaciones web por defecto de Windows.
Abrimos una instancia de IIS, creamos un sitio

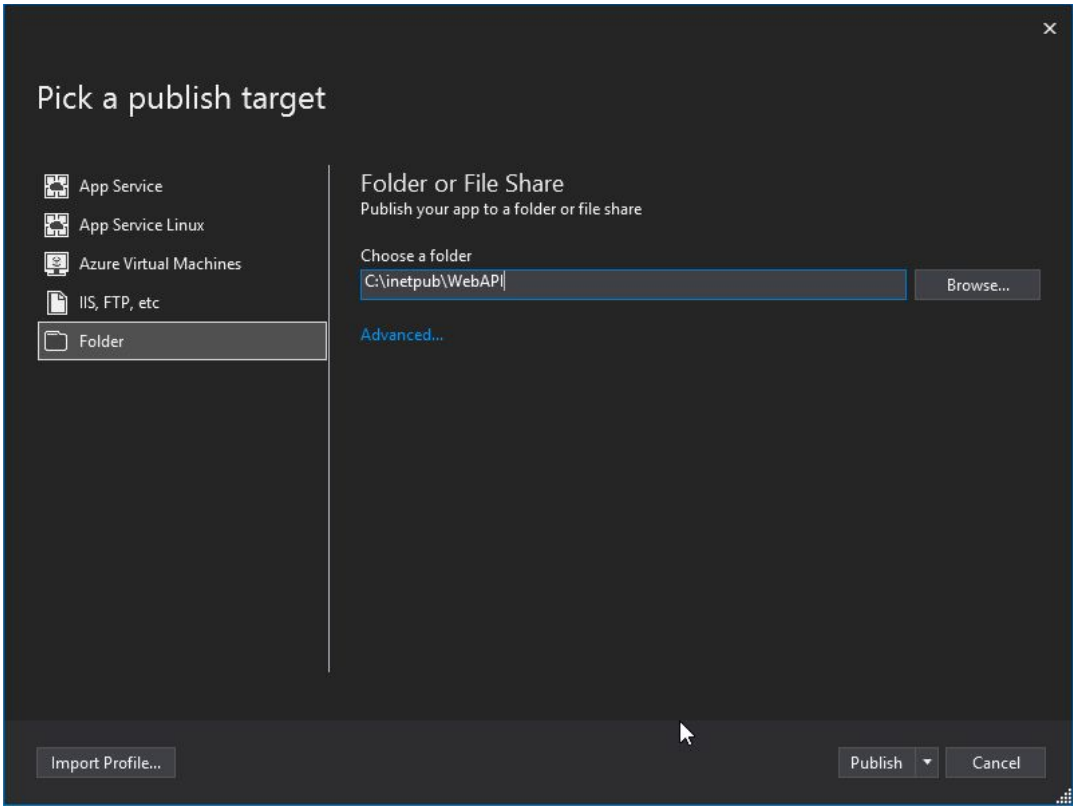
En la siguiente ventana configuramos la siguiente información:



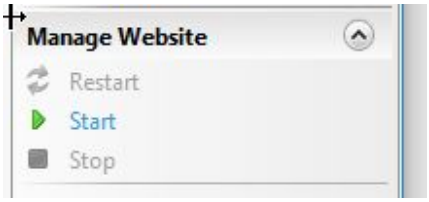
Elegimos un puerto que no esté reservado por el sistema (mayor a 1024) ni por otra aplicación. La ruta física debe ser donde pongamos los archivos generados cuando hacemos dotnet publish por línea de comando o cuando le damos publish en VisualStudio.



En la siguiente ventana podemos configurar donde queremos que se haga el deploy.



También podemos configurar para que se haga el deploy automáticamente en el sitio que ya configuramos en IIS. En este caso vamos a publicarlo en la carpeta automáticamente. Esto lo hacemos porque así no es necesario tener primero configurado el sitio de IIS, o si algo falla, igualmente el deploy en una carpeta lo podemos hacer en cualquier carpeta y luego moverlo a donde quieramos que se ejecute la WebAPI.
Luego de esto, podemos iniciar la aplicación de IIS



En el caso de que no se puedan ejecutar algunos métodos, por ejemplo los de borrado o actualizado, se deben seguir los siguientes pasos:
Parase en el sitio > Modules > WebDavModule > Quitar

En el archivo appsettings.se debe configurar el string de conexión de la base de datos.
"ConnectionString":
"Server=*NOMBRE_DEL_SERVIDOR*;Database=*NOMBRE_DE_BASE*;Trusted_Connection=True;Integrated Security=True"

Incluidos cuando se levanta la aplicación, tiene cargado 2 usuarios, uno de ellos con rol administrador y el otro un usuario seguidor.
Los datos para acceder a ellos son:
Administrador:
- Email: admin@admin.com
- Password: admin

Seguidor:
- Email: user@user.com
- Password: user

6 Bibliografia

Repository Pattern
https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

Moq
https://github.com/Moq/moq4/wiki/Quickstart

https://blogs.encamina.com/piensa-en-software-desarrolla-en-colores/moq-net-introduccion-utilizarlo-ejemplos/

https://github.com/ORT-DA2/AN-N6A_TEC-Clase5

https://stackoverflow.com/questions/21441792/sql-cannot-insert-explicit-value-for-identity-column-in-table-table-when-ident/21441948

https://www.istr.unican.es/asignaturas/is1/is1-t13-trans.pdf

https://stackoverflow.com/questions/38704025/cannot-access-a-disposed-object-in-asp-net-core-when-injecting-dbcontext

https://stackoverflow.com/questions/40122162/entity-framework-core-lazy-loading

http://www.sparxsystems.com.ar/resources/tutorial/uml2_compositediagram.html

http://blog.getpostman.com/2014/01/27/extracting-data-from-responses-and-chaining-requests/

https://github.com/aspnet/DependencyInjection/issues/440

https://www.c-sharpcorner.com/UploadFile/dacca2/unit-test-using-mock-object-in-dependency-injection/

https://www.telerik.com/products/mocking/unit-testing.aspx

https://www.reddit.com/r/dotnet/comments/8f7lh0/entityframeworkcore_many_to_many_relationships/