Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio II

Entregado como requisito para la obtención del crédito de la materia
Diseño de Aplicaciones II

Maximiliano Pollinger – 202066
Javier Aramberri - 202373

Tutores:
Ignacio Valle
Fernando Olmos

2018

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. Estos algoritmos pueden ser seleccionados y cargados de forma dinámica en tiempo de ejecución. Cada encuentro cuenta con dos o más equipos dependiendo del deporte al cual son atribuidos. A su vez, el sistema provee la funcionalidad de ingresar los resultados de dichos encuentros, a partir de los cuales el usuario puede ver la tabla de posiciones de cada equipo. Cada equipo puede contar con una foto, pertenece a un deporte y además puede ser seguido por múltiples usuarios. 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 permite a usuarios interesados en deportes que ver el calendario de encuentros de los equipos a los cuales sigue y hacer comentarios sobre estos encuentros.

Además, la solución cuenta con una aplicación web desarrollada en Angular la cual se comunica con la WebAPI, permitiendo realizar todas las acciones antes mencionadas (entre otras) a partir de una intuitiva interfaz gráfica.

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.

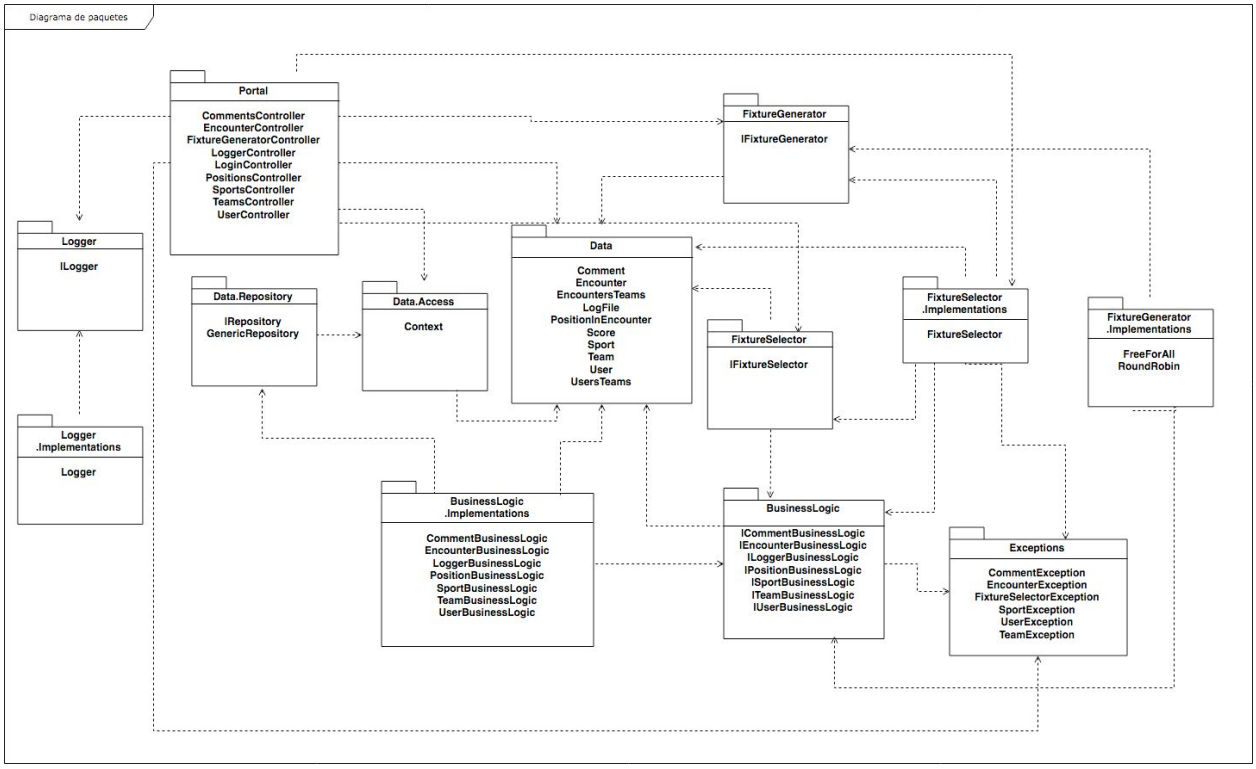


Diagrama de paquetes de la solucion

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.

El diagrama de dicho filtro es el siguiente:

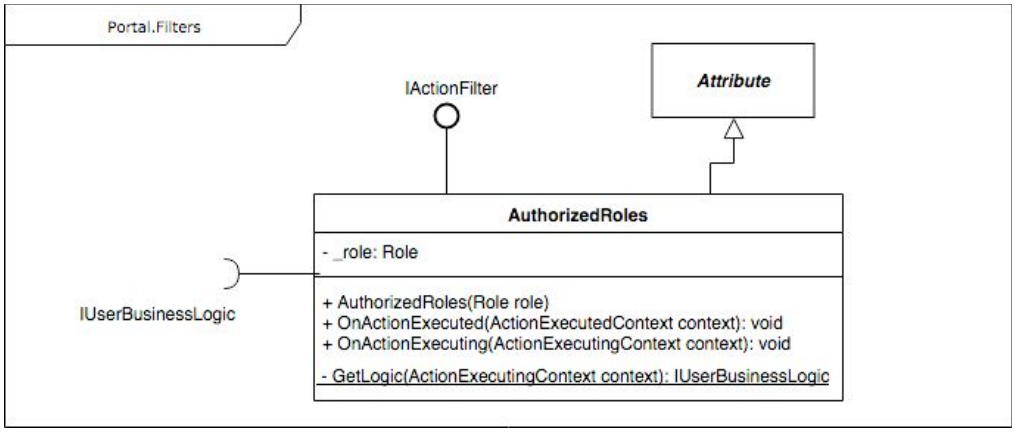


Diagrama clase AuthorizedRoles utilizado por controllers para autorización

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.

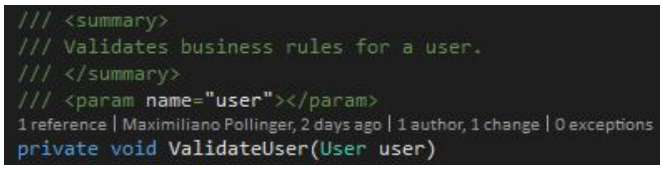
Dado que básicamente todas las interfaces de la lógica de negocio tienen los mismo métodos básicos de crear, actualizar, borrar y obtener, ser podría tener una clase abstracta con estos métodos para reutilizar código. Las clases que implementan las interfaces de la lógica de negocio también deberían 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

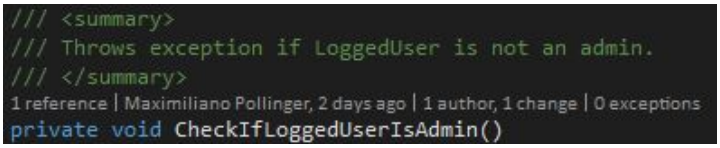
práctica el trabajar con *DTOs* para los datos que entran y salen, e internamente convertimos esos *DTOs* en las entidades del negocio, usando *AutoMapper*.
Esto nos permite agregar o quitar datos para ser usados internamente por la lógica pero que no queremos devolver al usuario por la razón que sea, 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.

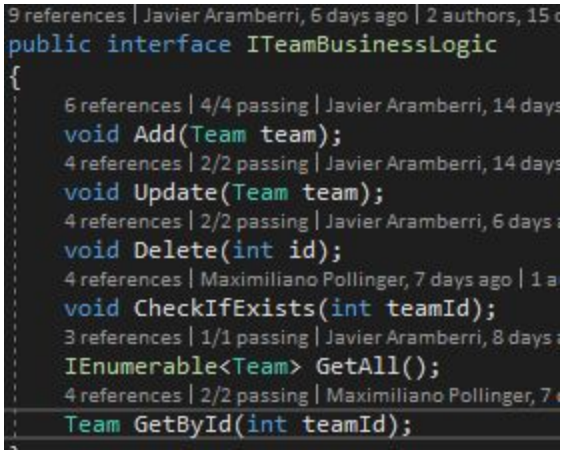
3. Evidencia de Clean Code

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.







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.

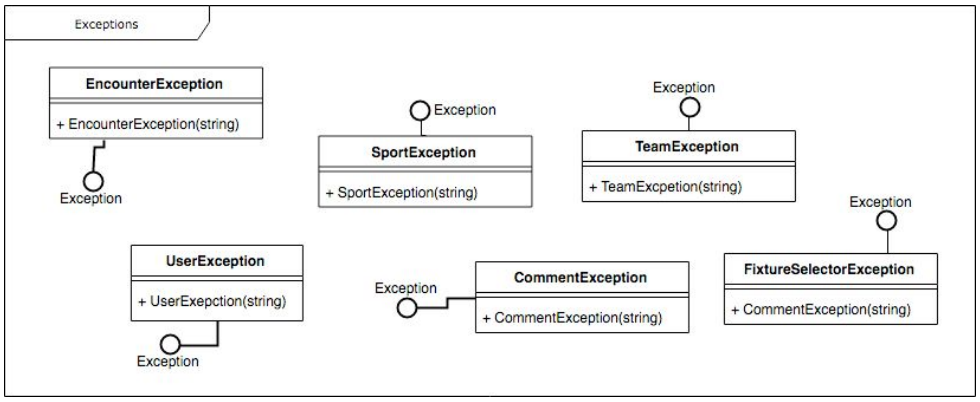
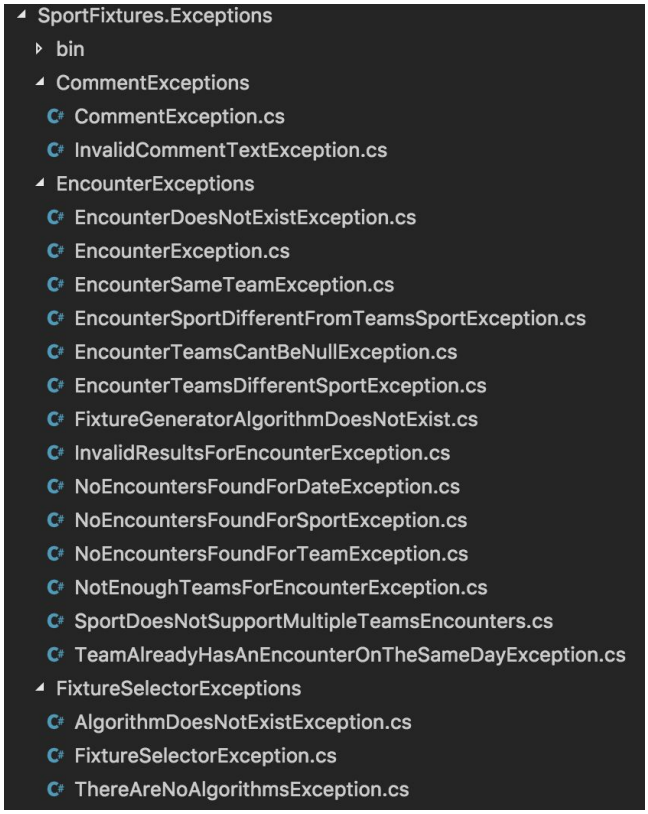
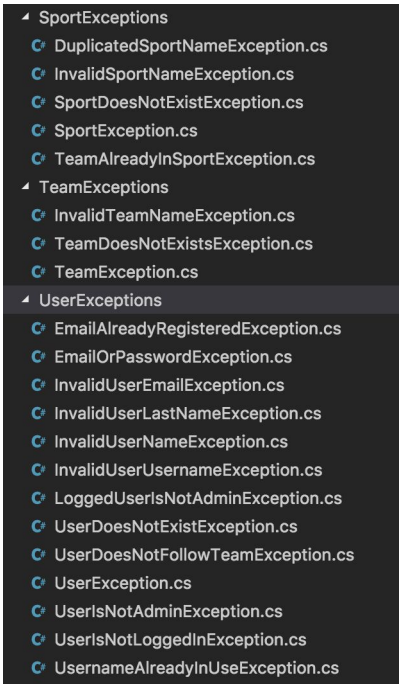
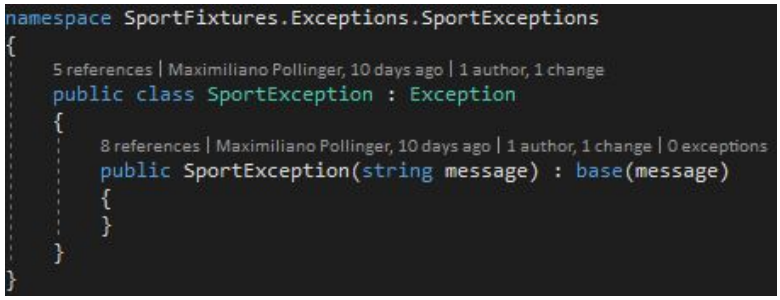


Diagrama de clases paquete Exceptions





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)
        {
        }
    }
}
```

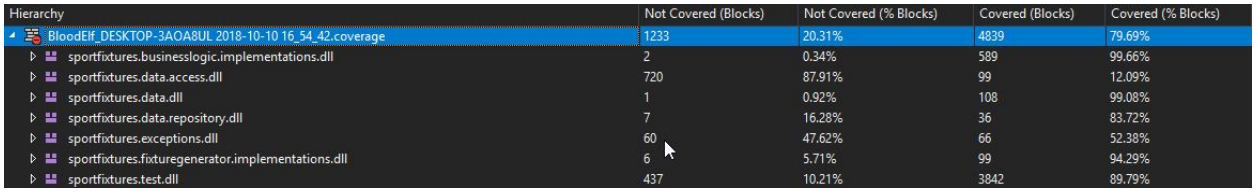
```
6 references | Maximiliano Pollinger, 4 days ago | 1 author, 1 change
public class SportDoesNotExistException : SportException
{
    2 references | Maximiliano Pollinger, 4 days ago | 1 author, 1 change | 0 exceptions
    public SportDoesNotExistException() : base("Sport does not exist.")
    {
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

    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.

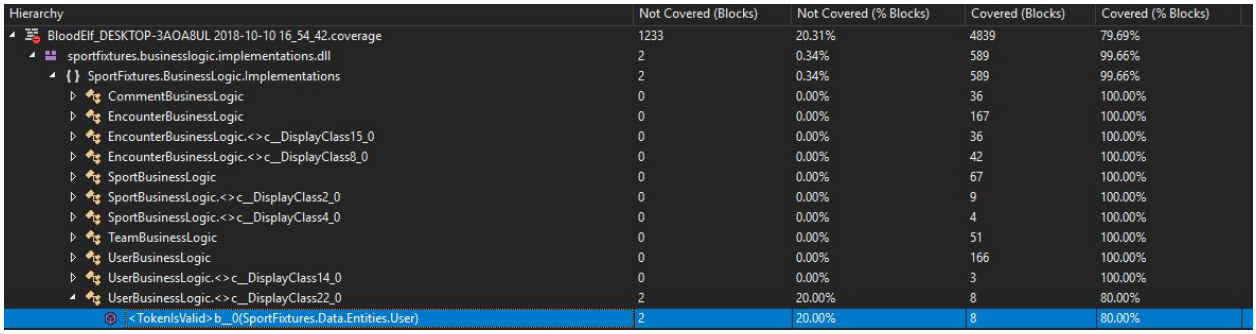
Evidencia de TDD

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.



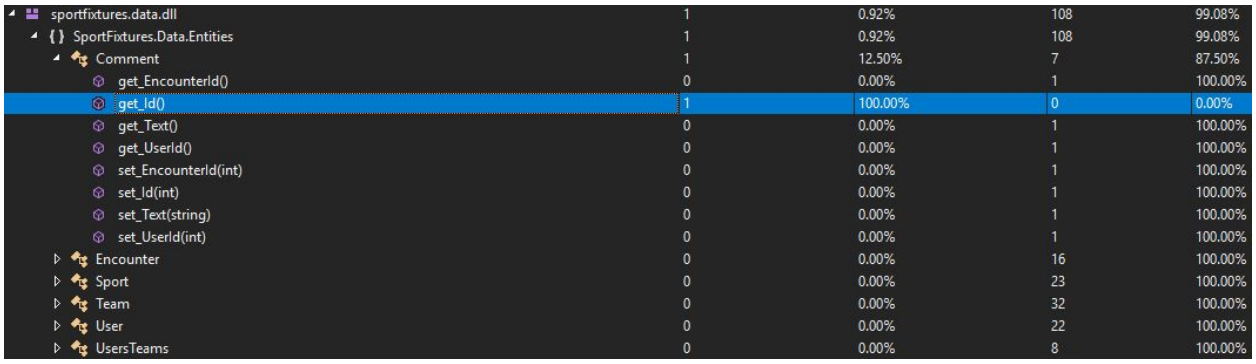
Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
BloodElf_DESKTOP-3AOA8UL 2018-10-10 16_54_42.coverage	1233	20.31%	4839	79.69%
sportfixtures.businesslogic.implementations.dll	2	0.34%	589	99.66%
sportfixtures.data.access.dll	720	87.91%	99	12.09%
sportfixtures.data.dll	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.



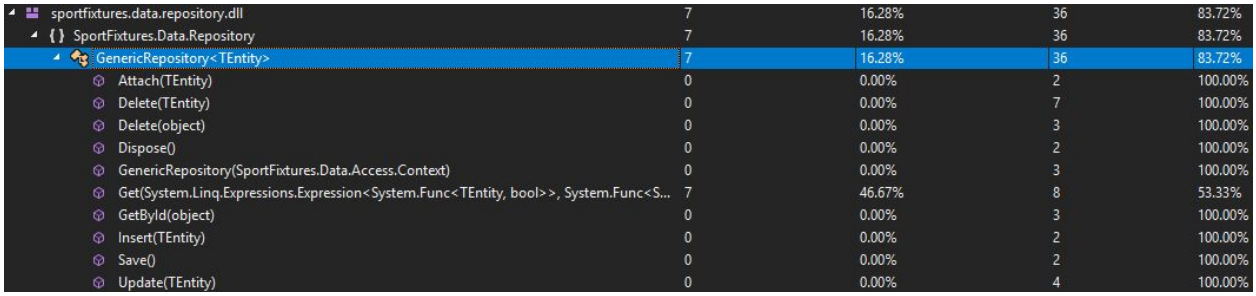
Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
BloodElf_DESKTOP-3AOA8UL 2018-10-10 16_54_42.coverage	1233	20.31%	4839	79.69%
sportfixtures.businesslogic.implementations.dll	2	0.34%	589	99.66%
SportFixtures.BusinessLogic.Implementations	2	0.34%	589	99.66%
CommentBusinessLogic	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.



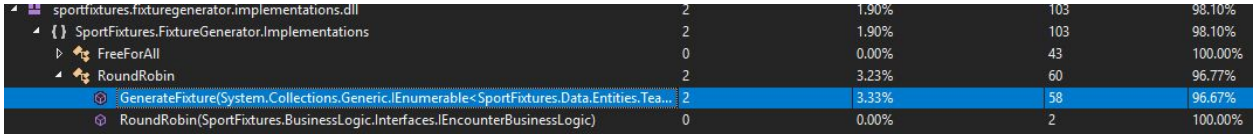
sportfixtures.data.dll	1	0.92%	108	99.08%
SportFixtures.Data.Entities	1	0.92%	108	99.08%
Comment	1	12.50%	7	87.50%
get_EncounterId()	0	0.00%	1	100.00%
get_Id()	1	100.00%	0	0.00%
get_Text()	0	0.00%	1	100.00%
get_UserId()	0	0.00%	1	100.00%
set_EncounterId(int)	0	0.00%	1	100.00%
set_Id(int)	0	0.00%	1	100.00%
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.



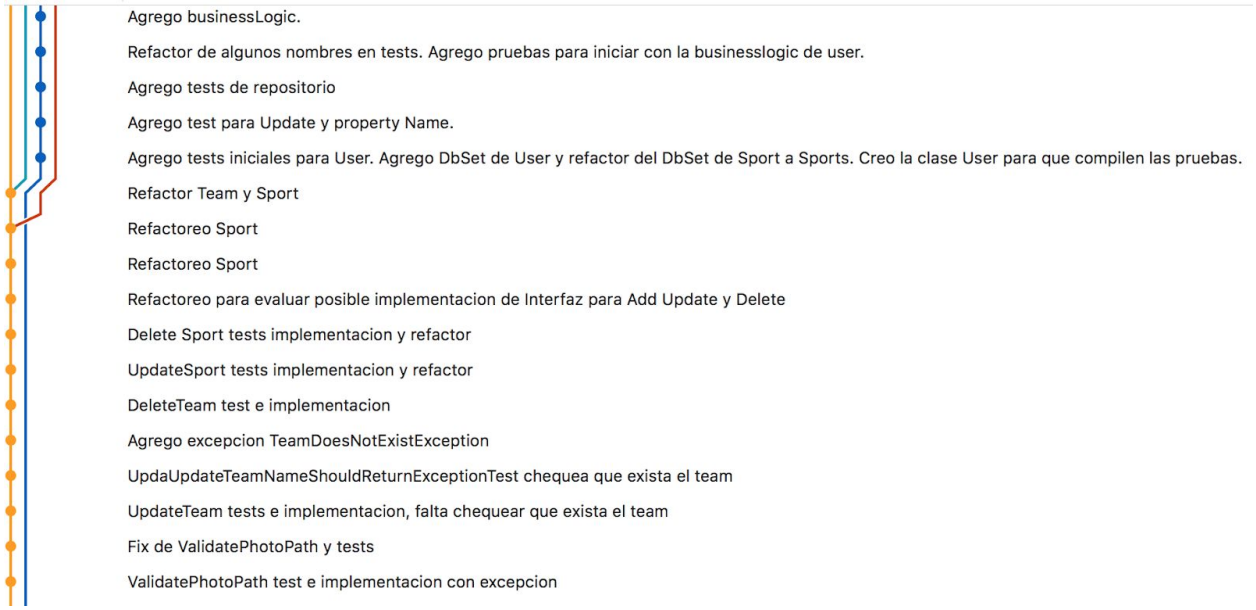
sportfixtures.fixturegenerator.implementations.dll	2	1.90%	103	98.10%
{ } SportFixtures.FixtureGenerator.Implementations	2	1.90%	103	98.10%
FreeForAll	0	0.00%	43	100.00%
RoundRobin	2	3.23%	60	96.77%
GenerateFixture(System.Collections.Generic.IEnumerable<SportFixtures.Data.Entities.Tea...	2	3.33%	58	96.67%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

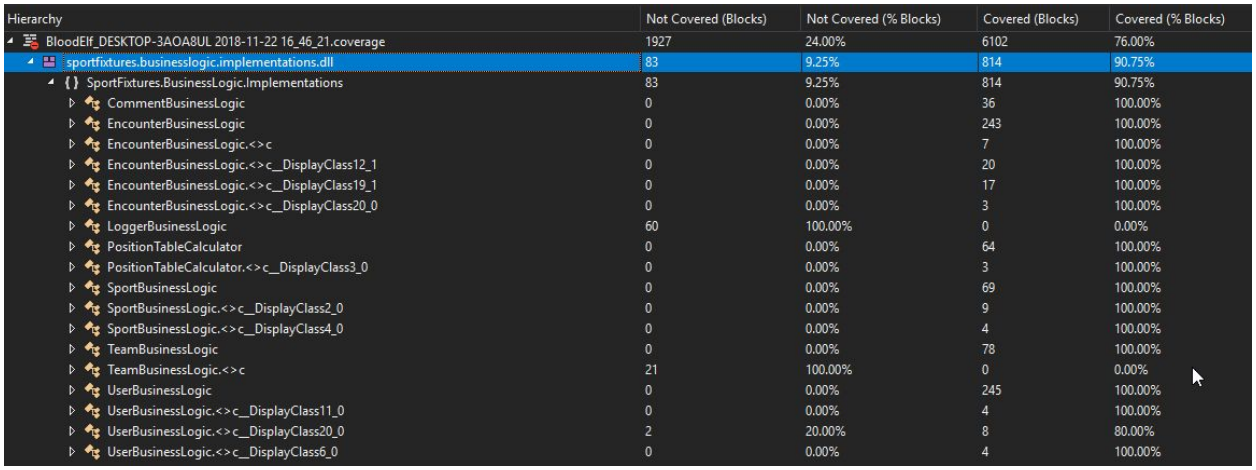
En cuanto a Repository, no aplicamos TDD ya que utilizamos la implementación brindada por Microsoft. Igualmente, hicimos los tests correspondientes para probar cada uno de los métodos, pero sin aplicar el ciclo planteado por la metodología.

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.



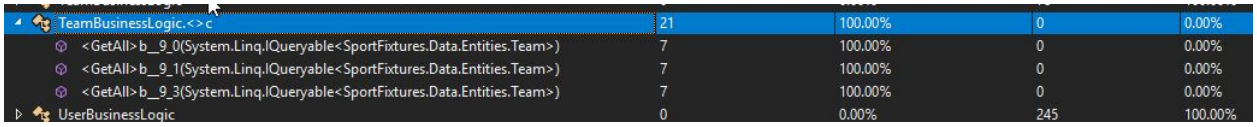
Agrego businessLogic.
Refactor de algunos nombres en tests. Agrego pruebas para iniciar con la businesslogic de user.
Agrego tests de repositorio
Agrego test para Update y property Name.
Agrego tests iniciales para User. Agrego DbSet de User y refactor del DbSet de Sport a Sports. Creo la clase User 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

Para la segunda parte el resultado de las pruebas de cobertura es el siguiente:



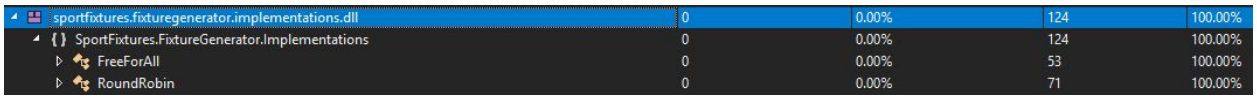
Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
BloodElf_DESKTOP-3AOA8UL 2018-11-22 16_46_21.coverage	1927	24.00%	6102	76.00%
sportfixtures.businesslogic.implementations.dll	83	9.25%	814	90.75%
SportFixtures.BusinessLogic.Implementations	83	9.25%	814	90.75%
CommentBusinessLogic	0	0.00%	36	100.00%
EncounterBusinessLogic	0	0.00%	243	100.00%
EncounterBusinessLogic.<>c	0	0.00%	7	100.00%
EncounterBusinessLogic.<>c__DisplayClass12_1	0	0.00%	20	100.00%
EncounterBusinessLogic.<>c__DisplayClass19_1	0	0.00%	17	100.00%
EncounterBusinessLogic.<>c__DisplayClass20_0	0	0.00%	3	100.00%
LoggerBusinessLogic	60	100.00%	0	0.00%
PositionTableCalculator	0	0.00%	64	100.00%
PositionTableCalculator.<>c__DisplayClass3_0	0	0.00%	3	100.00%
SportBusinessLogic	0	0.00%	69	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%	78	100.00%
TeamBusinessLogic.<>c	21	100.00%	0	0.00%
UserBusinessLogic	0	0.00%	245	100.00%
UserBusinessLogic.<>c__DisplayClass11_0	0	0.00%	4	100.00%
UserBusinessLogic.<>c__DisplayClass20_0	2	20.00%	8	80.00%
UserBusinessLogic.<>c__DisplayClass6_0	0	0.00%	4	100.00%

Como se puede observar en la imagen la cobertura de esta entrega es menor a la anterior, esto se debe a que no realizamos pruebas para la lógica del *logger*. Sin embargo todas las otras clases muestran una cobertura de 100%, a excepción del método *GetAll* de *TeamBusinessLogic* que no tiene *tests*, pero entendemos que los *tests* serían exactamente iguales a los de las otras *business logic*, ya que utilizamos un repositorio genérico.



TeamBusinessLogic.<>c	21	100.00%	0	0.00%
<GetAll>b__9_0(System.Linq.IQueryable<SportFixtures.Data.Entities.Team>)	7	100.00%	0	0.00%
<GetAll>b__9_1(System.Linq.IQueryable<SportFixtures.Data.Entities.Team>)	7	100.00%	0	0.00%
<GetAll>b__9_3(System.Linq.IQueryable<SportFixtures.Data.Entities.Team>)	7	100.00%	0	0.00%
UserBusinessLogic	0	0.00%	245	100.00%

En esta entrega si pudimos obtener el 100% de cobertura en el proyecto de implementaciones de algoritmos.



sportfixtures.fixturegenerator.implementations.dll	0	0.00%	124	100.00%
SportFixtures.FixtureGenerator.Implementations	0	0.00%	124	100.00%
FreeForAll	0	0.00%	53	100.00%
RoundRobin	0	0.00%	71	100.00%

En resumen podemos decir que la cobertura de las pruebas es aceptable y se prueban todos los métodos esenciales del sistema.

4. Cambios introducidos

4.1. Encuentros de múltiples equipos

En Sport vamos a tener un enum que identifique el tipo del deporte, si es de 2 equipos o de múltiples equipos. Elegimos un enum en lugar de un booleano para tener mayor flexibilidad si el día de mañana introducen otro tipo de deporte que no sea ni de 2 equipos ni de múltiples equipos genéricamente. Es decir, si ahora hay un tipo de deporte de fijo 4 equipos, los cambios serían de introducir una variante más al enumerado en lugar de tener que cambiar el booleano por otra cosa.

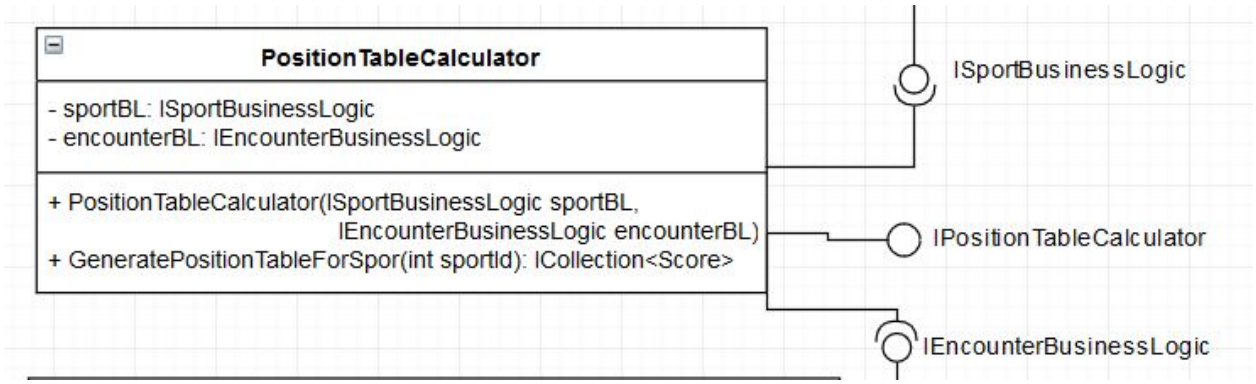
En cuanto al encuentro, antes estaba conformado por dos equipos, lo cual cambiamos a una lista de equipos para poder satisfacer el nuevo requerimiento de que un encuentro puede ser de más de un equipo. Para que esto sea posible tuvimos que crear una relación Many to Many entre encuentro y equipo, lo cual, para que sea mapeado correctamente por Entity Framework, se traduce a crear una nueva entidad con el encuentro, el team y los id de ambos. A esta entidad la llamamos EncountersTeams. Por lo tanto, el encuentro tiene una lista de equipos del tipo EncountersTeams.

4.2. Resultados de encuentros

Cuando decidimos cómo implementar el nuevo requerimiento del ingreso de resultados y el cálculo de posiciones, discutimos de qué forma hacerlo más eficiente. Una primer idea fue crear una nueva entidad, con su propio DataSet, que guardara para cada equipo su puntaje. La idea era hacer que cada vez que se creara un equipo, crear una entidad “Position” que guardase el team y su puntaje correspondiente (inicialmente siendo cero) y, cada vez que se agregase un resultado para un encuentro del cual este equipo formase parte, sumarle el puntaje correspondiente. Esto implicaba tener una nueva lógica de negocio que permitiera agregar la entidad a la base y poder hacer updates, por lo tanto, implicaba crear nuevos tests. Con esta solución evitabamos que cada vez que un usuario realizara la consulta de posiciones se realizaran tantas consultas a la base y evitamos todo el cálculo que se debe hacer para sumar los

puntajes y así obtener la lista de posiciones. Es decir, que cuando un usuario consultara la tabla de posiciones solo haria una consulta a la base pidiendo la lista de puntajes para un deporte sin tener que hacer ningun calculo. El problema de esta solución es que implicaba mucho tiempo de desarrollo y testeo del cual era escaso, por lo cual nos decidimos por una segunda opción.
Esta opción no es la más eficiente a nivel de performance, pero permite, si se desea a futuro, crear varias formas de calcular posiciones de forma sencilla. Lo que hicimos fue crear una Interfaz con un método que genera la tabla de posiciones y retorna una lista con los puntajes de cada equipo. Luego creamos una clase que implementa esa interfaz, que lo que hace es obtener todos los equipos de un deporte y buscar para cada uno los encuentros y calcular el puntaje basándose en los resultados de dichos encuentros. Es por esto que decimos que tiene una baja performance, ya que cada vez que se quiere consultar la tabla de posiciones se realizan todas estas acciones.

En el siguiente diagrama podemos ver que implementamos la Interfaz con el método, y que esta clase depende de *ISportBusinessLogic* y *IEncounterBusinessLogic* para consultar los equipos de los deportes y los encuentros de dichos equipos.



4.3. Logger

Como requerimiento se agregó la integración de un *logger* que registre las acciones de *login* y generación del *fixture* de encuentros. Además, era un requerimiento también que se pueda cambiar de logger en tiempo de compilación, lo que agregó una capa más de complejidad al tema. De todas formas decidimos crear una interfaz la cual expone ciertos métodos para hacer el log. También creamos una primera implementación de esta interfaz que crea los *logs* en archivos separados nombrados por fecha.
De esta forma implementar un nuevo mecanismo de logging, por ejemplo en base de datos o mediante un servicio, es tan simple como implementar la interfaz y desarrollar la implementación. Luego esas implementaciones podrían ser cambiadas en tiempo de compilación para cambiar la forma en la que se *loggea*.
Luego también teníamos que decidir si las llamadas al *log* iban a estar únicamente en los *controller*, o en la lógica de negocio o en ambos lugares. Decidimos que solo se hagan llamadas al *logger* en los *controller*, dado que para poder usarlo desde la lógica de negocio necesitábamos además acceder al usuario que actualmente está logueado en la aplicación, lo que requería la implementación de una clase que maneje la sesión. Entendemos que al solo hacer llamadas al logger desde los controller implica que si en el futuro se cambia la API por un frontend como por ejemplo *WinForms*, se tenían que volver a hacer las llamadas al *logger*, pero si están en la lógica de negocio, no importa cuantas veces cambie el *frontend*, el *logger* no se va a ver afectado. Aún así, creemos que por las particularidades del proyecto, no se justifica el hacer las llamadas desde la lógica de negocio, además de que también creemos que posiblemente se utilicen con distintas finalidades el hacer las llamadas al *logger* desde la lógica de negocio que cuando se realizan en la API o en el *frontend*.
Es por ello que decidimos hacer las llamadas en la API.

El impacto de introducir el logger fue mínimo, ya que tanto la interfaz como la implementación se desarrollaron por separado a lo que es toda la estructura del proyecto y hasta que no se decide utilizar una implementación, el resto del sistema no tiene conocimiento de la existencia de estas nuevas clases. Para usar el logger, agregamos la definición de inyección de dependencias en la clase Startup de la WebAPI y luego inyectamos en el constructor de las clases que vamos a utilizar el logger la interfaz. De esta manera tan solo cambiando la implementación en el Startup podemos cambiar la forma que el sistema loggea.

4.4. Login

Anteriormente el *login* devolvía únicamente el *token* generado para el usuario que se logueaba, ahora el *login* devuelve todo el usuario con todos sus datos incluyendo el *token* generado pero sin la contraseña. De esta manera en el *frontend* podemos guardar y manejar el usuario que está logueado, además de poder validar si su token es válido y usarlo para futuras peticiones a la API.

Este cambio no tuvo impacto ya que solo fue cambiar el retorno de la función y adaptar una interfaz y un *test* unitario.

4.5. Frontend

Para el *frontend* se trataron de seguir las recomendaciones de angular y sus estándares. De esta manera logramos un código mucho más limpio, fácil de entender, fácil de mantener, fácil de extender y cumplimos con un cierto nivel de calidad como para poder mostrar el código a otros desarrolladores Angular y que lo puedan seguir desarrollando.
Incluimos *Feature Modules*, en los casos de las pantallas de encuentros, equipos, usuarios, deportes, inicio, *fixtureGenerator* y todas las páginas que consideramos que tenían que tener un módulo separado. De esta manera podemos tener un *AppModule* menos cargado y también poder hacer *lazy loading* de los módulos, lo que lleva a una ganancia de performance.

Tenemos un index.html en el cual lo único que se hace es llamar al *app-root* y tener una rueda giratoria de carga para dar *feedback* al usuario cuando la página está cargando.
En el *appcomponent*.*html* tenemos la invocación al componente de la barra de navegación, así como también el *router-outlet* y un *toastr-container*. Consideramos que esta arquitectura es la mejor para así poder llevar el contenido específico a componentes separados. Por eso también tenemos la barra de navegación como un componente separado, que se puede reutilizar en otros proyectos también.

A nivel de servicios contamos un con *BaseService* el cual tiene las llamadas genéricas de HTTP, así como algunos métodos auxiliares como el *jwt* para crear los *headers* de

las peticiones. De esta forma, todos los servicios concretos, extienden a este servicio y lo único que hacen es pasar la llamada HTTP al *BaseService*, y retornar la respuesta al componente.
Los servicios son inyectables y la mayoría de ellos son singleton, ya que se encuentran declarados como providers en el *AppModule*. Los que son específicos y solo se usan en un cierto módulo, son declarados como *providers* en esos módulos únicamente. De esta forma también ganamos performance ya que no tenemos instancias de servicios que no se van a utilizar.

A nivel de componentes como ya mencionamos tenemos *Feature Modules*. Elegimos separar los componentes con la siguiente jerarquía:
app/pages/<nombre-componente/<nombre-componente>/<componente>
app/pages/<nombre-componente/<feature-module>
De esta forma es fácil de ubicar los componentes y también facilita el agregar nuevos componentes ya que sabemos dónde tienen que ir.
Otros componentes que consideramos que no pertenecen bajo la jerarquía de "*pages*", están ubicados en *app/*, como por ejemplo el componente de *login*.

Contamos con una clase denominada *AppSettings*, en la cual están definidas todas las rutas de la API, la URL base de la misma, los nombres de las variables a utilizar en el *localStorage* y las URLs para el *router*. De esta forma es mucho más fácil agregar y modificar rutas, ya sean del *RoutingModule* o de la API. También facilita el cambiar la IP/puerto de la API, además de que estas configuraciones son grabadas en el *main.js* luego de transpilar, es fácil modificar una vez transpilado el código también.

Usamos el *localStorage* del navegador para guardar la sesión del usuario que hace login. Por un lado guardamos toda la información del usuario y por otro guardamos el *token* que fue generado por la API para poder identificar al usuario.
Esta información puede ser consultada mediante el *sessionService*, quien se encarga de proveer los métodos de acceso y seteo de estos datos, así como también informar si el usuario logueado tiene rol de administrador o si hay un usuario logueado actualmente.

Utilizamos componentes de terceros como por ejemplo listas, combos, tablas y calendarios. La familia de componentes de terceros más grande que utilizamos proviene de *PrimeNg*, ya que consideramos que son faciles de usar, performantes y cumplen con las funcionalidades que necesitamos. También utilizamos calendarios de *SyncFusion* tanto para elegir una sola fecha como para elegir rangos de fechas. Estos

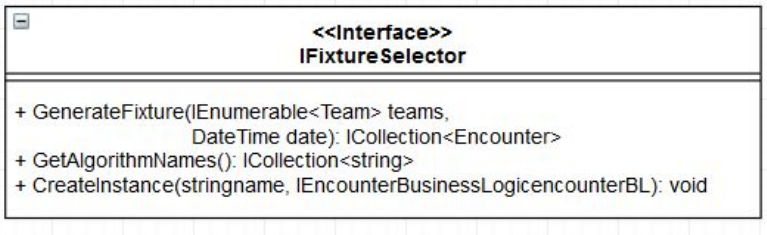
componentes fueron los que mejor se adaptaron a nuestras necesidades, como por ejemplo, para seleccionar el rango de fechas para consultar los logs.
Utilizamos estilos de *bootstrap* para darle un buen aspecto visual a toda la aplicación, intentando además mantener siempre el mismo estilo de botones, tamaños, combinaciones de colores, ubicación de los botones, etc., siempre lo más parecidos posibles, para así poder tener una buena experiencia de usuario al usar la aplicación.

Utilizamos el componente toastr para mostrar mensajes amigables, tanto de éxito, error y advertencia. Estos mensajes son pequeños pero visibles, así como también tienen animación de entrada y salida y se puede modificar el mensaje acorde a la situación. Consideramos que no son invasivos comparados con mostrar alertas que molestan al usuario o modales que deben ser cerrados para continuar la navegación, estos mensajes se muestran en la esquina inferior derecha en todas las pantallas, sea cual sea su severidad.
Intentamos que los mensajes sean lo más amigables e informativos posibles, para que el usuario pueda entender lo que está pasando.

Tenemos dos tipos de guardas, una para autorización y otra para autenticación.
La guarda de autorización es para saber si el usuario logueado es administrador. Esta guarda la usamos para seguridad en las rutas que solo pueden ser accedidas por los administradores. De esta manera aunque un usuario normal conozca las rutas restringidas, no va a poder acceder a ellas, ya que la guarda va a impedir su acceso y lo va a redireccionar a la página principal, además de emitir un mensaje de que no tiene permisos.
La guarda de autenticación es para saber si el usuario que intenta acceder a la ruta está logueado o no. Esta guarda se usa en las rutas que requieren de un usuario logueado, es decir, todas menos las rutas de página no encontrada y login. El resto de las rutas requieren que el usuario se encuentre logueado.

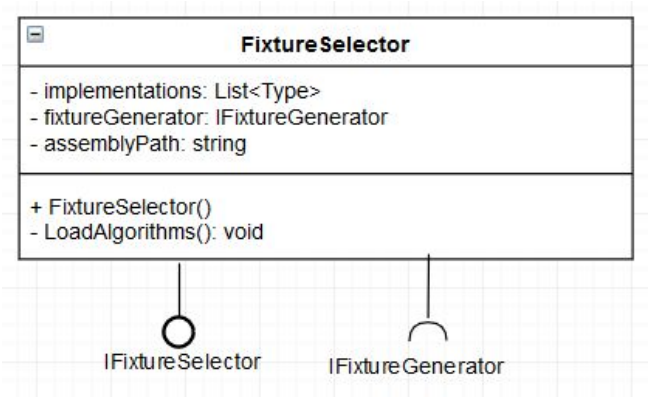
4.6. Carga dinámica de algoritmos

Para resolver la carga dinámica de algoritmos en tiempo de ejecución utilizando reflection, creamos una interfaz *IFixtureSelector*.



Como se puede ver en el diagrama, esta interface provee métodos para obtener los nombres de los algoritmos que se encuentran en el directorio de la aplicación, otro para crear la instancia del algoritmo y otro para generar el fixture.

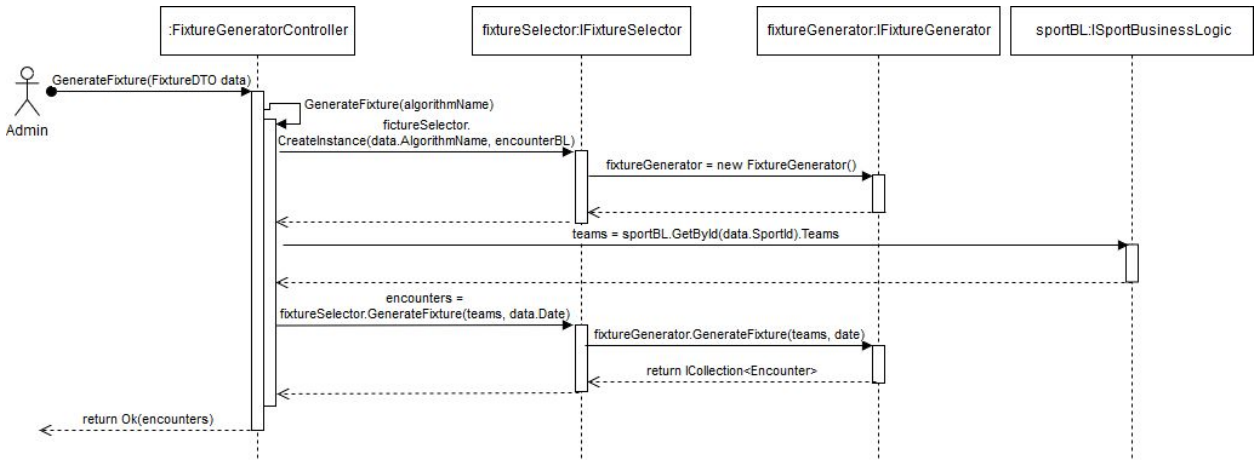
Luego tenemos la clase *FixtureSelector* que implementa dicha interfaz:



Esta clase, contiene una lista de implementaciones que son cargadas por el método *LoadAlgorithms*, el cual carga los algoritmos que se encuentran en el directorio del proyecto. Este método es llamado en el constructor y cada vez que se llama al método *GetAlgorithmNames*.
También guarda la instancia del algoritmo seleccionado para poder hacer el llamado al método encargado de generar el fixture

Para crear una instancia de un algoritmo utilizamos el nombre del mismo, lo cual entendemos no es la mejor opción ya que pueden existir varios algoritmos con el mismo nombre.
Algo que tampoco tomamos en cuenta es que no diferenciamos entre los algoritmos que son para encuentros de múltiples o dos equipos.

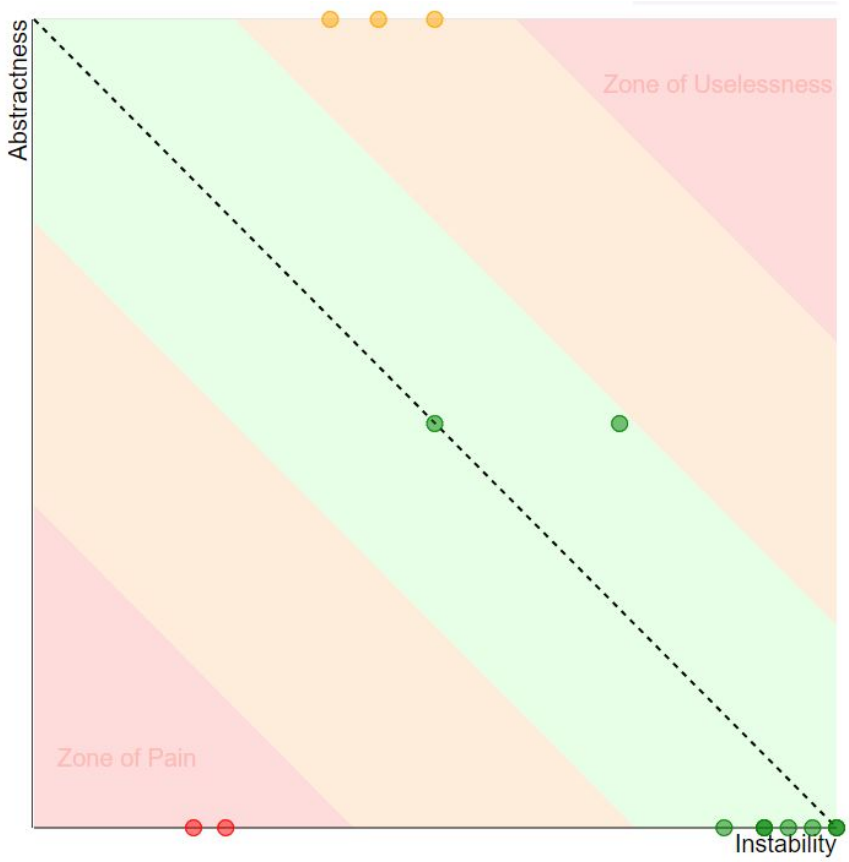
Los administradores pueden generar los fixtures a través del controller *FixtureGeneratorController*, el cual contiene dos endpoints. Uno para obtener la lista de algoritmos y otro para generar el fixture.
Para entender un poco mejor la solución hicimos un diagrama de secuencia que muestra las llamadas a partir de que un usuario administrador hace un request para generar el fixture hasta que obtiene la lista de encuentros.



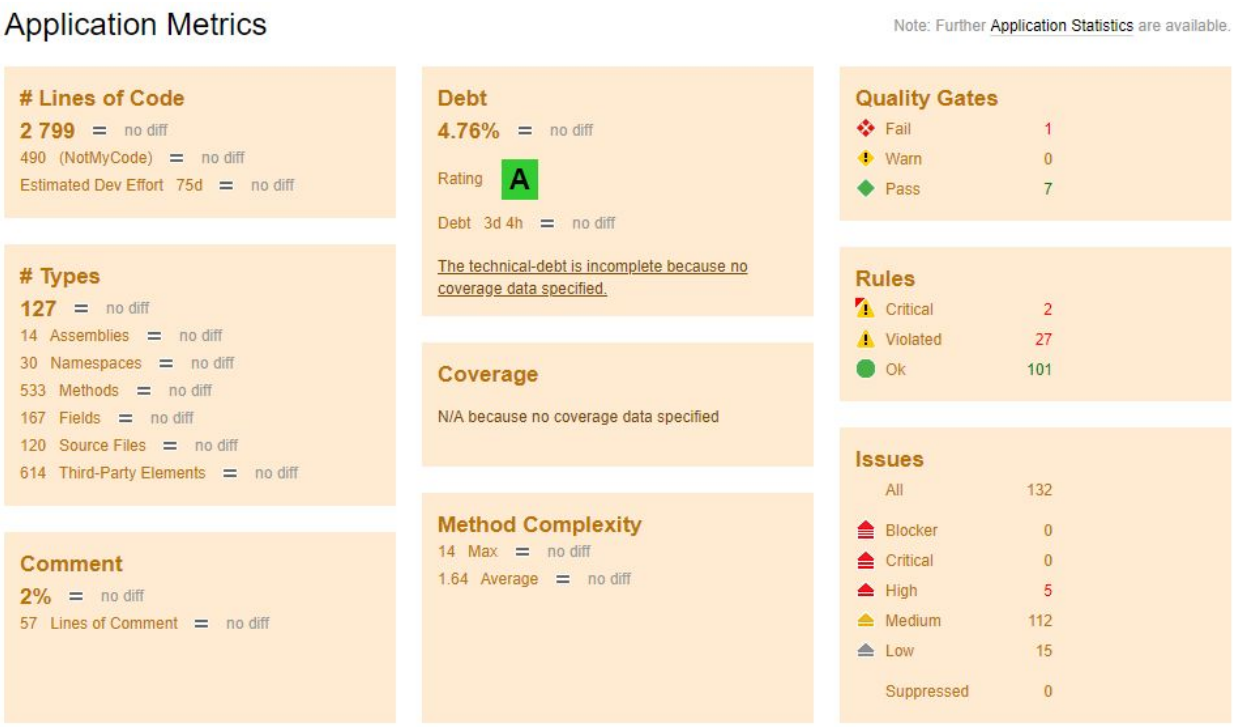
Cuando el usuario envía el nombre del algoritmo, se crea una instancia la cual se guarda en "*fixtureGenerator*" de la clase *FixtureSelector*, y cuando se llama al método *GenerateFixture* lo que hace es llamar al método *GenerateFixture* de la instancia guardada.

Con esta solución podemos resolver el problema de poder seleccionar en tiempo de ejecución el algoritmo que deseamos ejecutar, una vez que el usuario envía el nombre del algoritmo se crea una instancia y se llama al método encargado de generar el fixture.

5. Metricas



En la esquina inferior derecha tenemos los paquetes que corresponden a las implementaciones de la lógica de negocio, por lo que son clases concretas y claramente van a ser inestables porque no hay clases abstractas ni interfaces. Entonces en el contexto que se encuentran, están bien donde están ubicadas.
En la zona de dolor se encuentran los proyectos de *Data* y *Exceptions*, lo cual es esperado ya que son las entidades del negocio y las excepciones. Que se encuentren en esta zona quiere decir que cualquier cambio en ellos va a impactar básicamente en todo el sistema, lo cual es correcto y esperado ya que todo el sistema depende de las entidades del negocio y si las cambiamos es esperable que tengan un gran impacto en el sistema. Lo mismo con las excepciones.
Cercanos a la zona de poca utilidad, se encuentran los paquetes de *Logger*, *FixtureGenerator* y *BusinessLogic*. Es entendible que estos paquetes se encuentren en esta zona, ya que tienen interfaces pero también son usados y usan otras clases e interfaces. Esto es así porque usamos lógicas de negocio dentro de otras lógicas de negocio. Esto podría ser una posible mejora. Pero por la arquitectura que tenemos, deben usarse así.



Se puede observar que tenemos una buena deuda técnica, lo cual significa que solo 4.76% es código no probado. Solo 2 reglas críticas son violadas, las cuales se detallan más abajo. Tenemos 127 tipos distintos en 14 ensamblados, distribuidos en 30 namespaces.

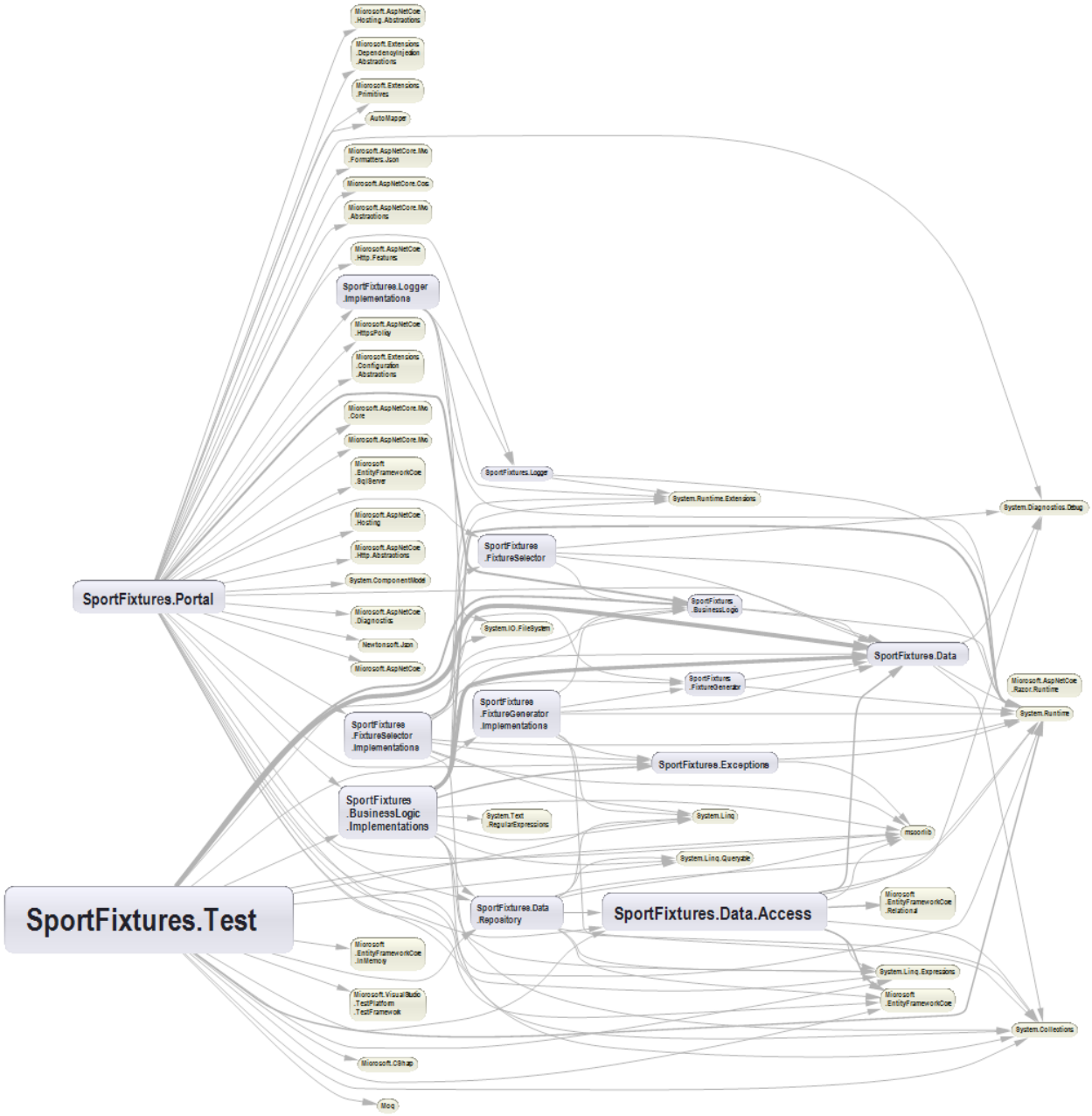
2 rules	issues	Full Name
Avoid types too big	1 issue	Rule
Attribute class name should be suffixed with 'Attribute'	1 issue	Rule

Showing 1 to 2 of 2 entries

Consideramos que las reglas violadas que se muestran no son tan importantes, pero podrían ser consideradas en un futuro de la aplicación.

Name	Trend	Baseline Value	Value	Group
Percentage Coverage			N/A because no coverage data	Project Rules \ Quality Gates
Percentage Coverage on New Code			N/A because no coverage data	Project Rules \ Quality Gates
Percentage Coverage on Refactored Code			N/A because no coverage data	Project Rules \ Quality Gates
Blocker Issues	=	0 issues	0 issues	Project Rules \ Quality Gates
Critical Issues	=	0 issues	0 issues	Project Rules \ Quality Gates
New Blocker / Critical / High Issues			0 issues	Project Rules \ Quality Gates
Critical Rules Violated	=	2 rules	2 rules	Project Rules \ Quality Gates
Percentage Debt	=	4.76 %	4.76 %	Project Rules \ Quality Gates
New Debt since Baseline			0 man-days	Project Rules \ Quality Gates
Debt Rating per Namespace	=	0 namespaces	0 namespaces	Project Rules \ Quality Gates
New Annual Interest since Baseline			0 man-days	Project Rules \ Quality Gates

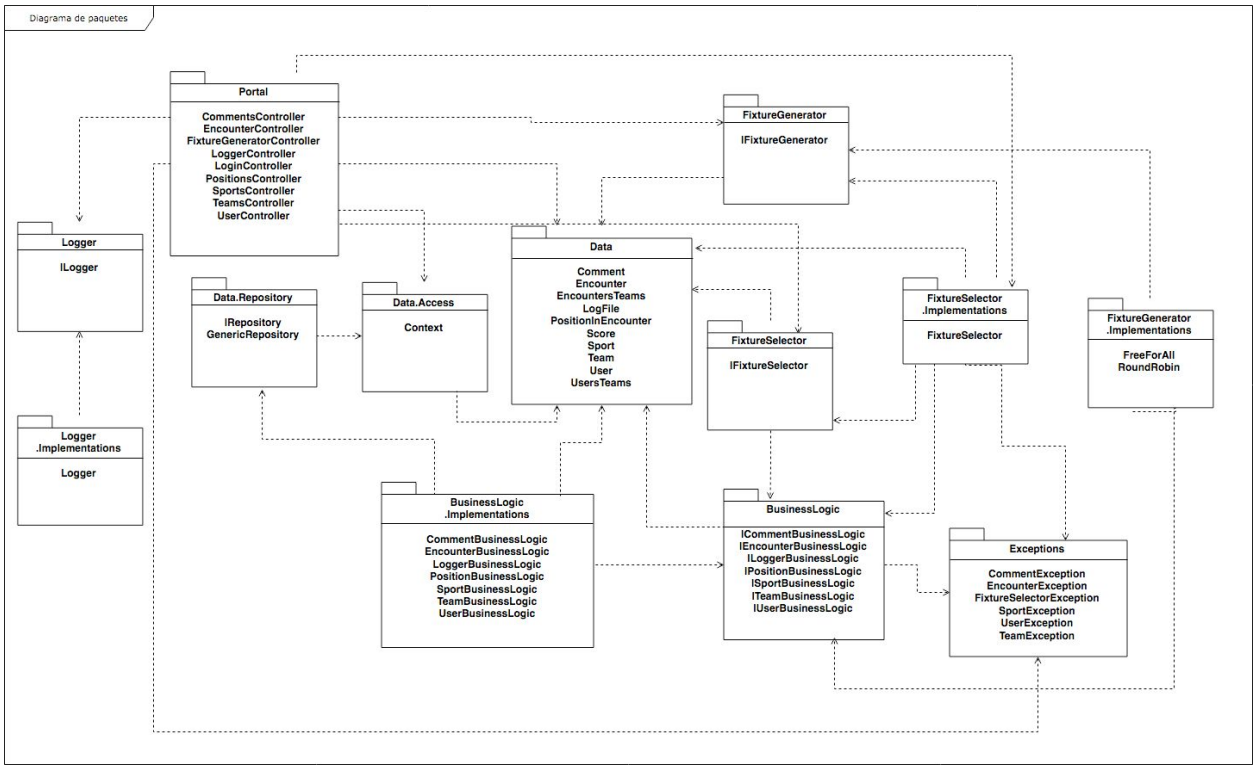
Showing 1 to 11 of 11 entries



Se puede observar como la API usa básicamente todos los paquetes además de paquetes de .NET. Así como también el paquete de Test, por razones de las pruebas unitarias. Y se comprueba lo dicho anteriormente de que el paquete Data (las entidades de negocio) son usadas por todo el sistema, lo que las hace inestables.

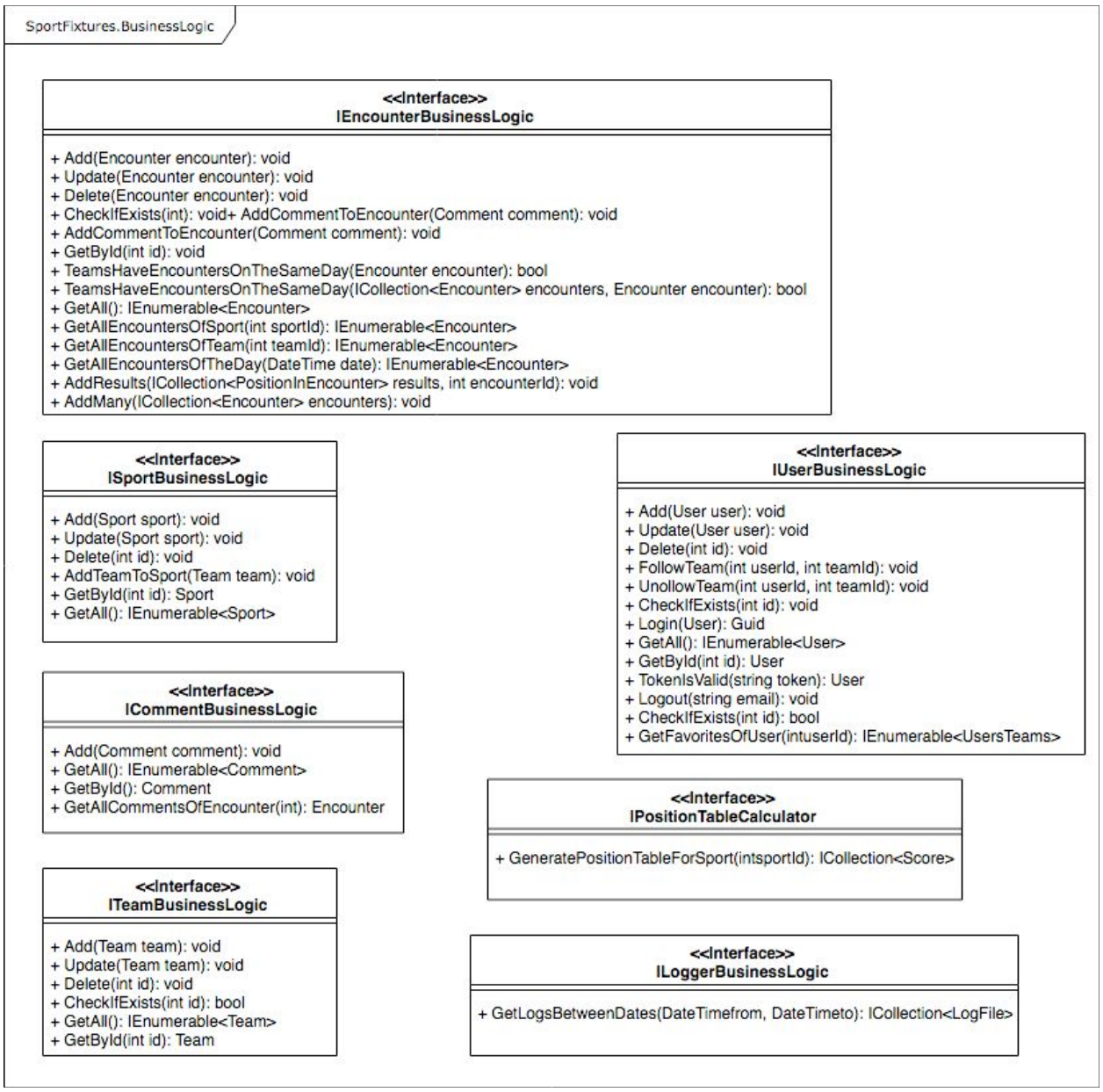
6. Anexo I - Diagramas

6.1. Diagrama de paquetes

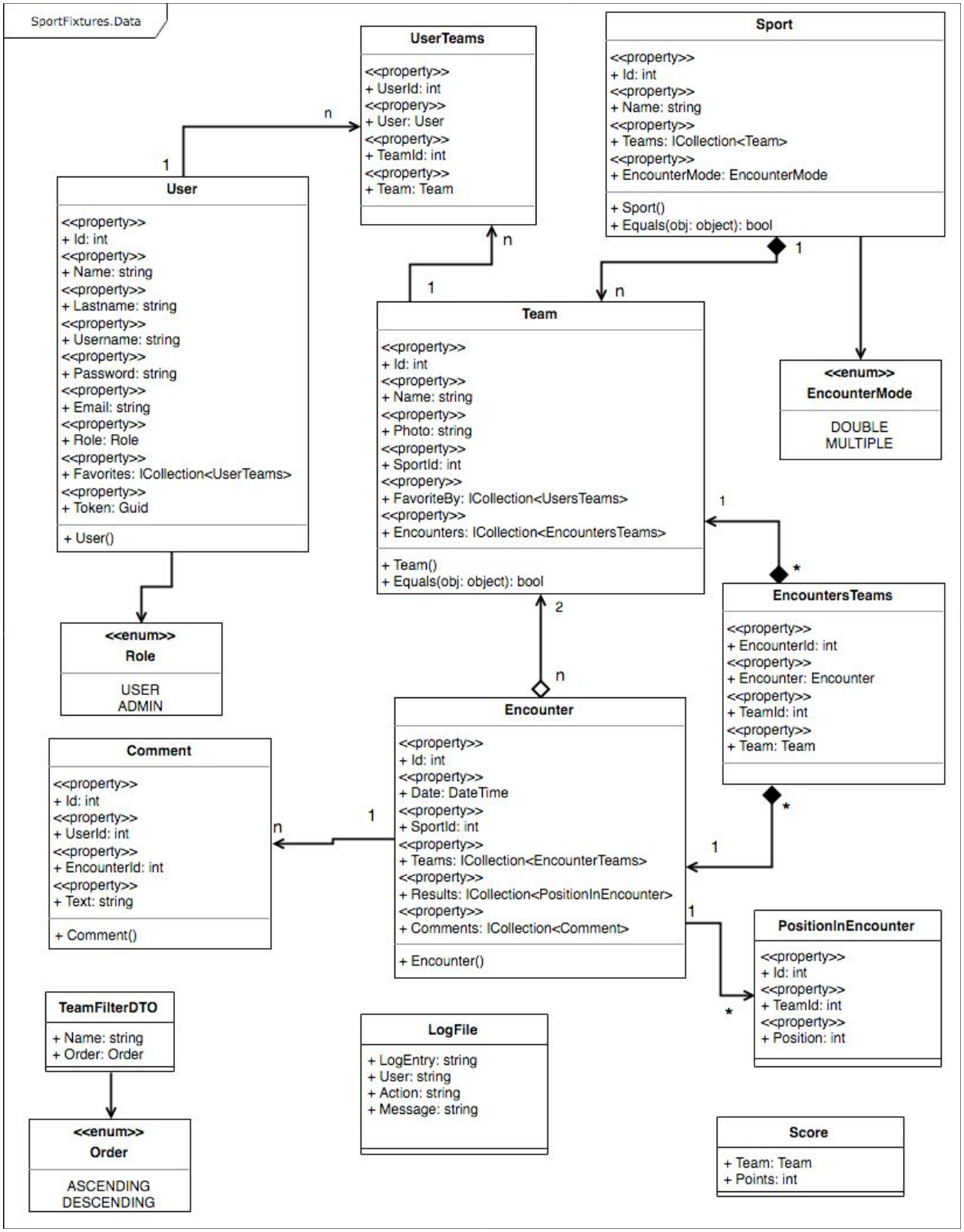


6.2. Diagramas de Clases

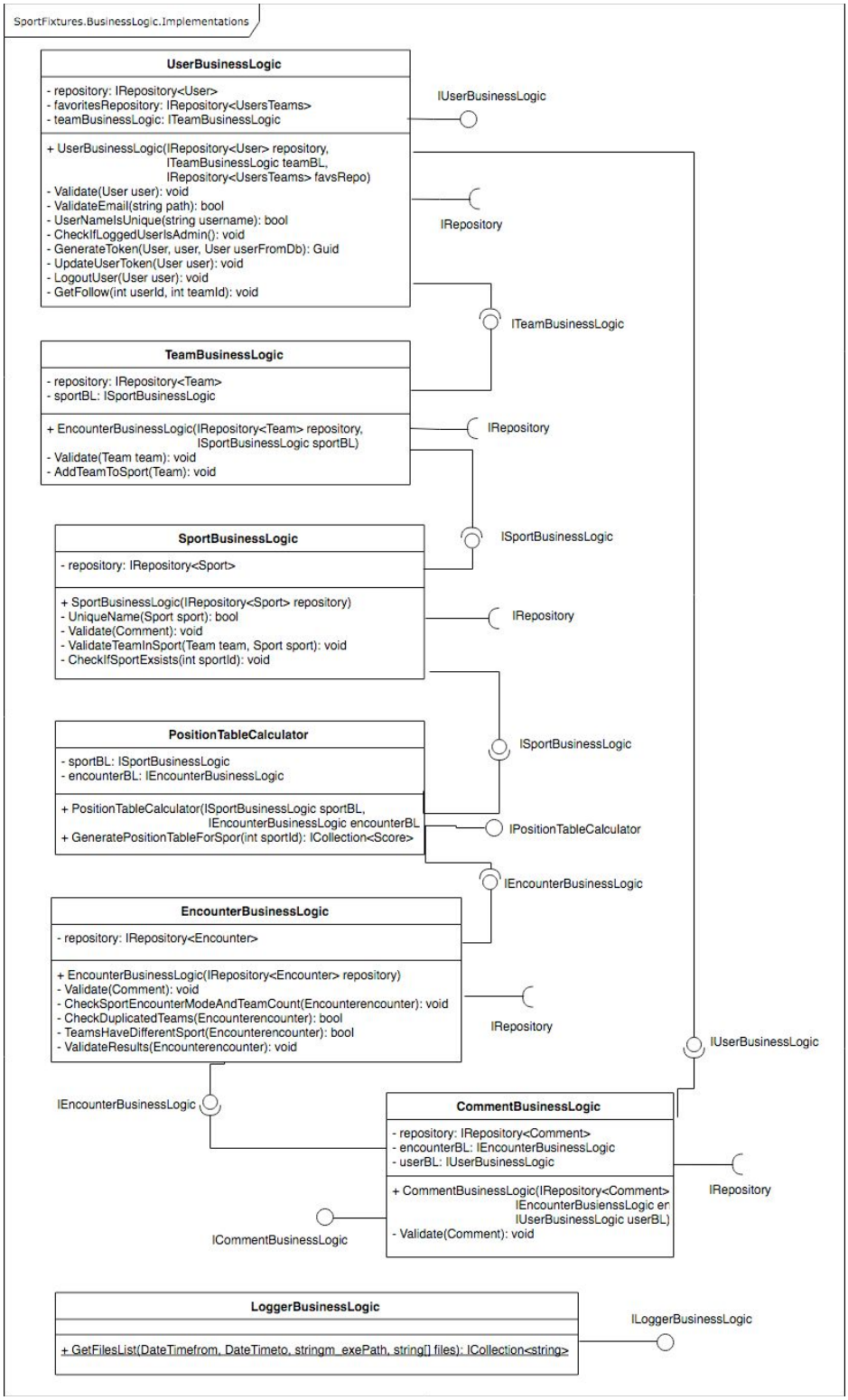
6.2.1. SportFixtures.BusinessLogic



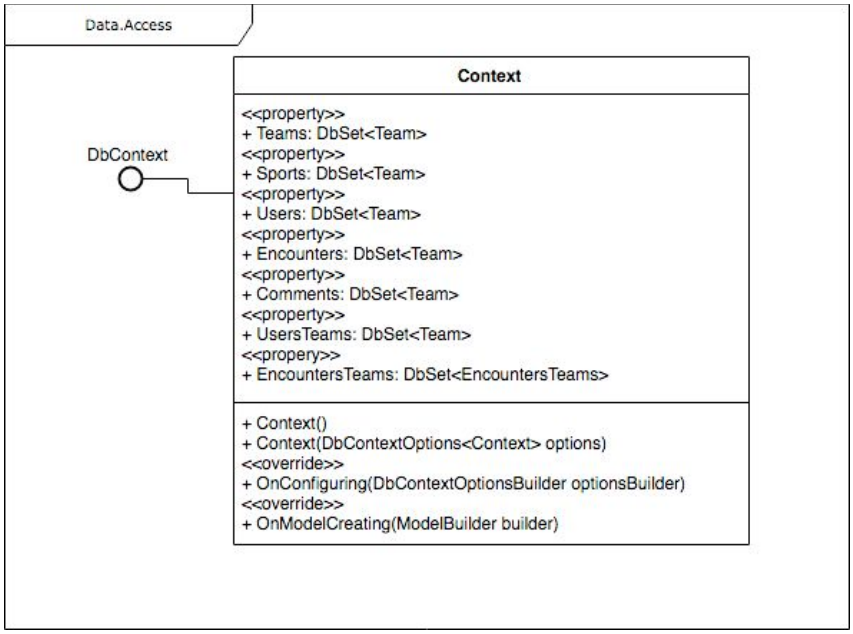
6.2.2. SportFixtures.Data



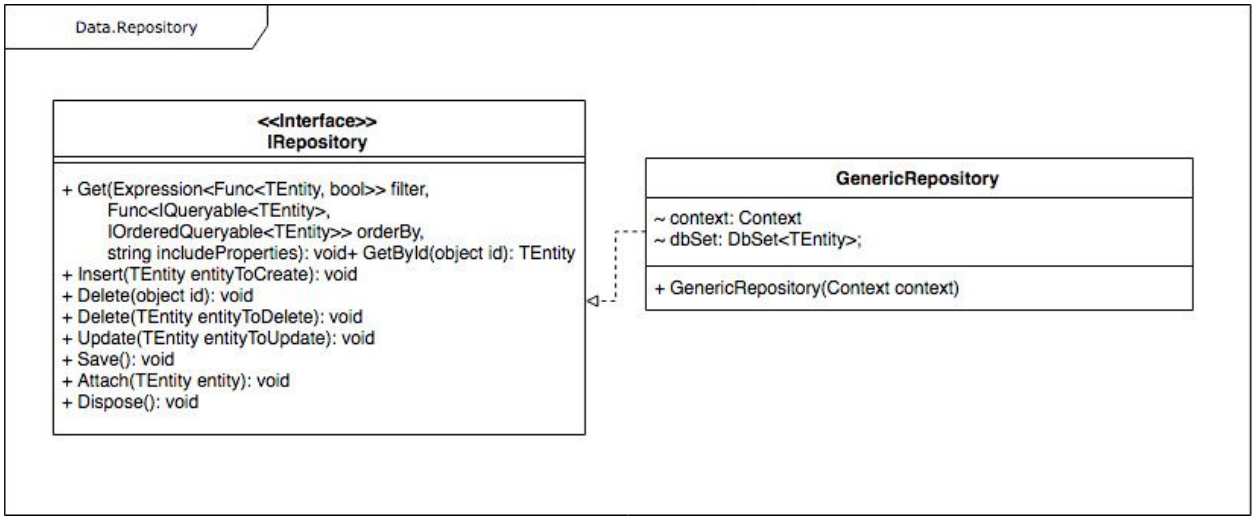
6.2.3. SportFixtures.BusinessLogic.Implementations



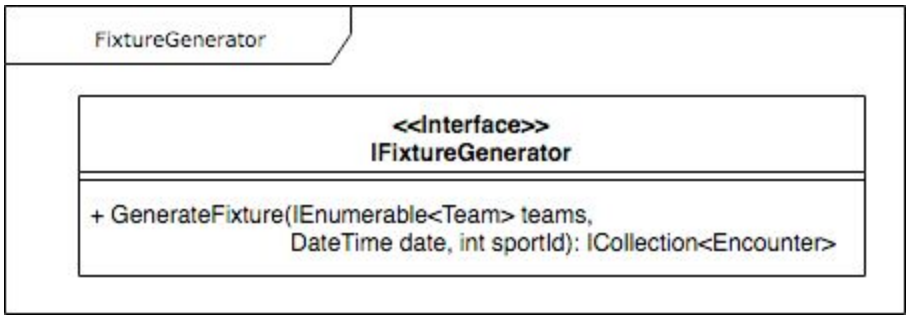
6.2.4. SportFixtures.Data.Access



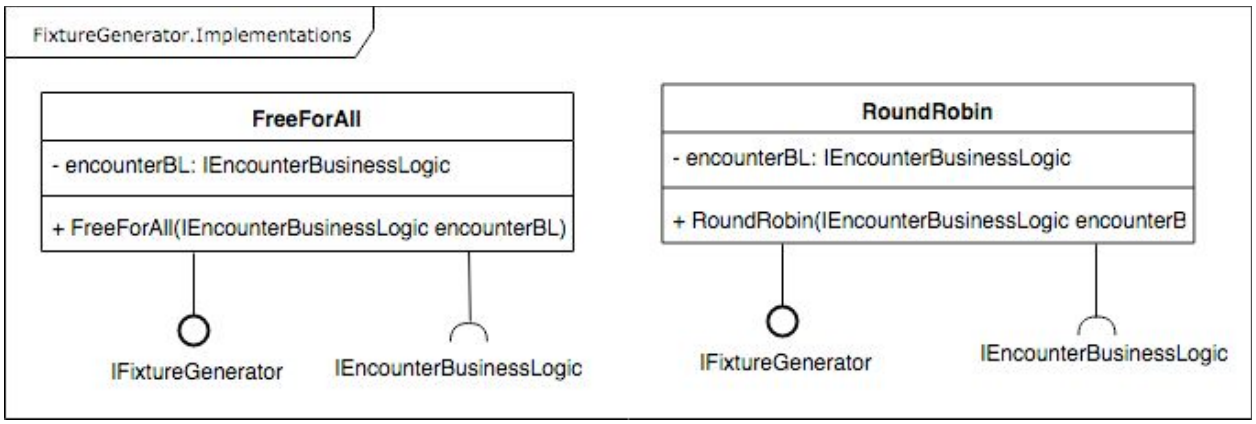
6.2.5. SportFixtures.Repository



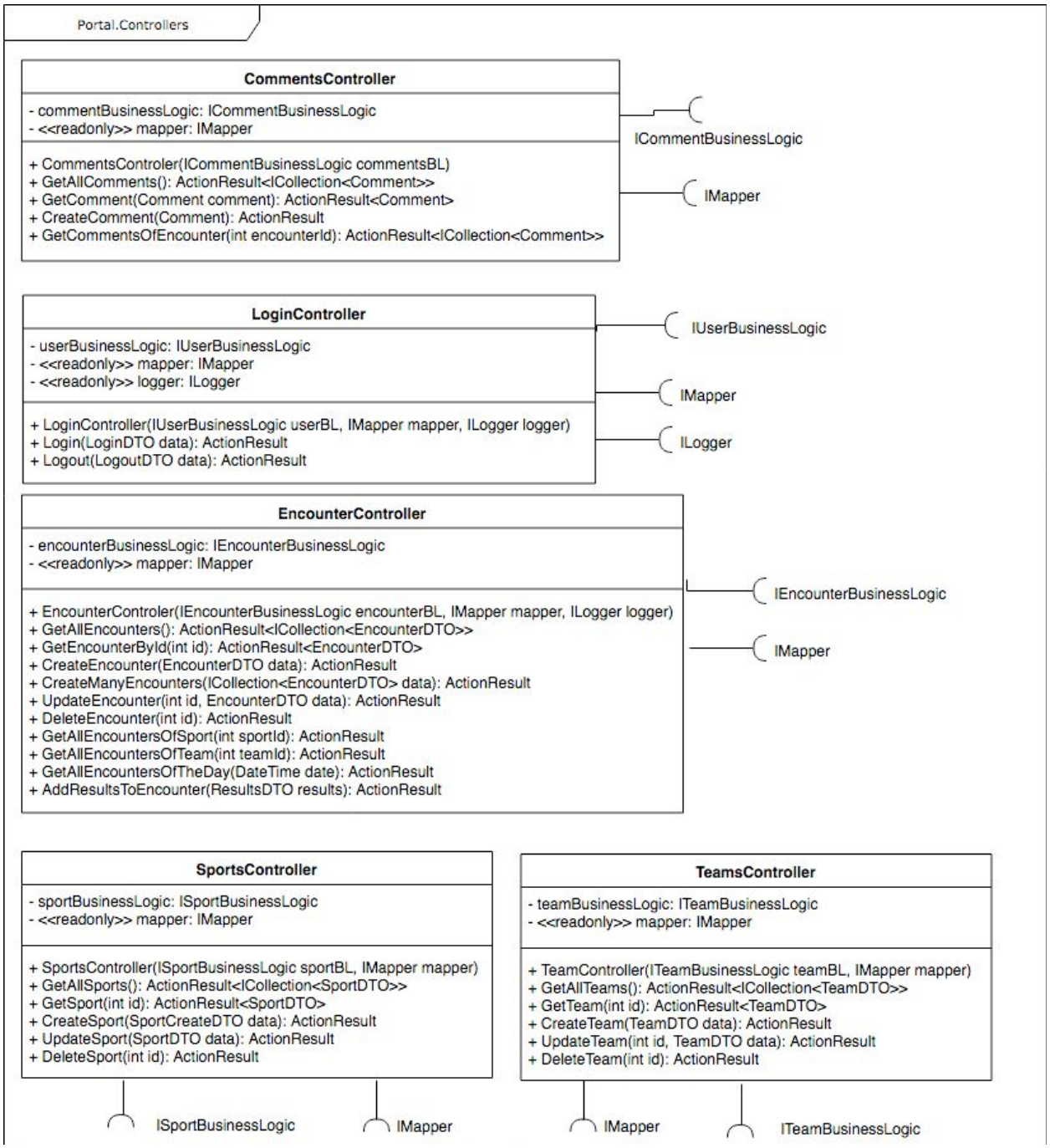
6.2.6. SportFixtures.FixtureGenerator

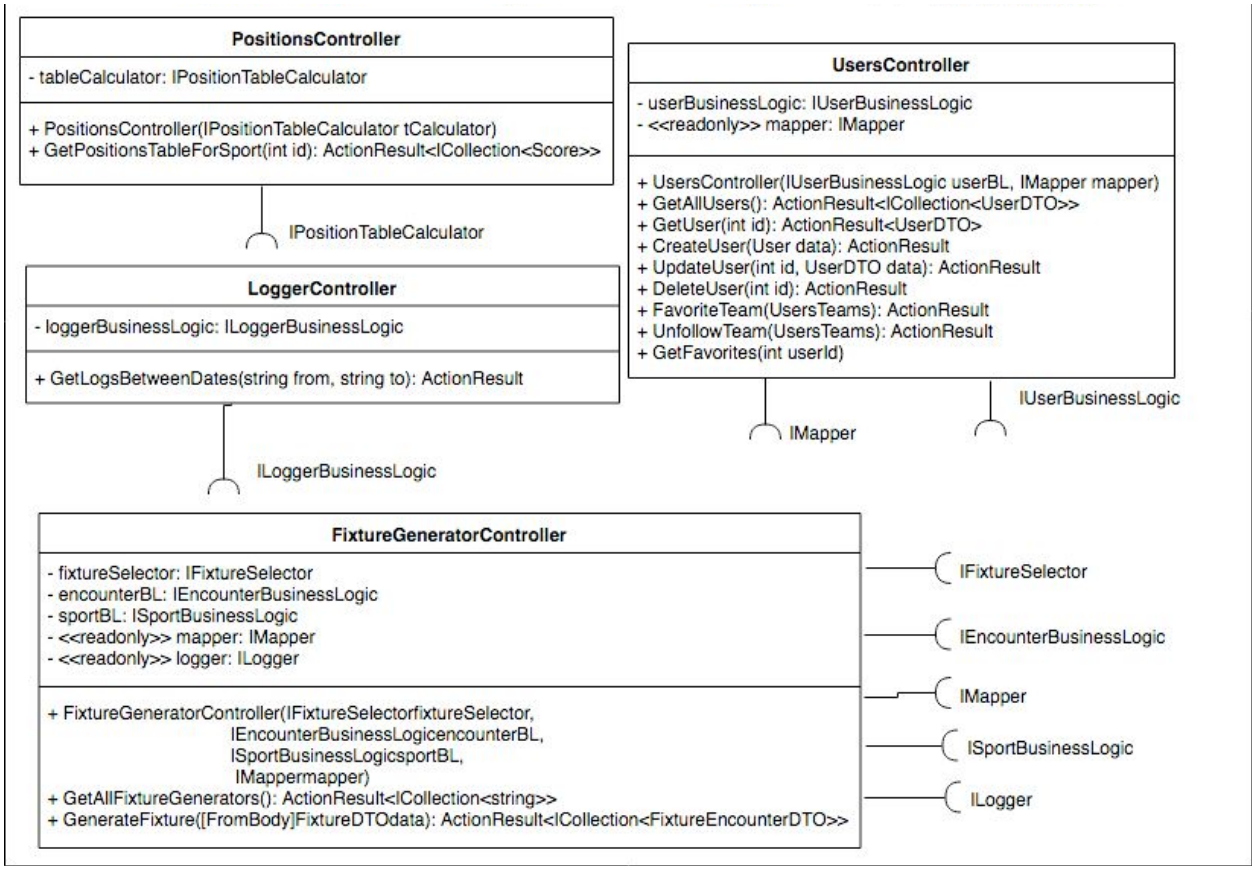


6.2.7. SportFixtures.FixtureGenerator.Implementations

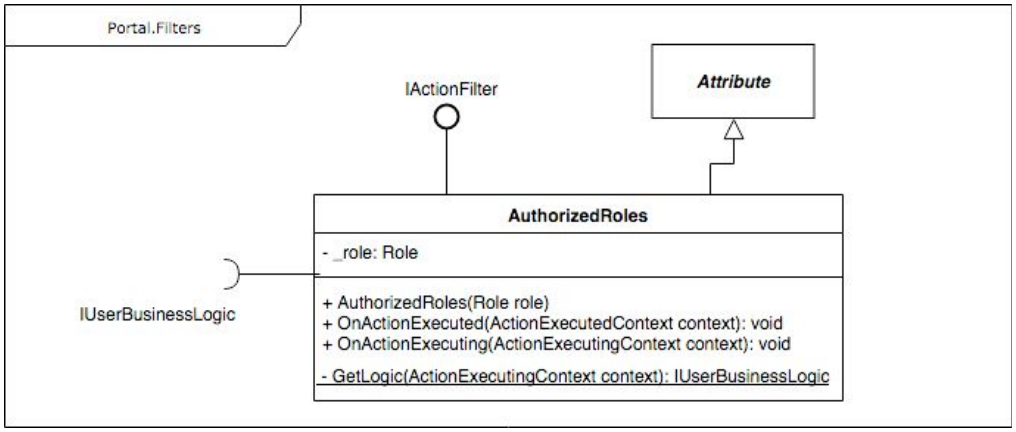


6.2.8. SportFixtures.Portal

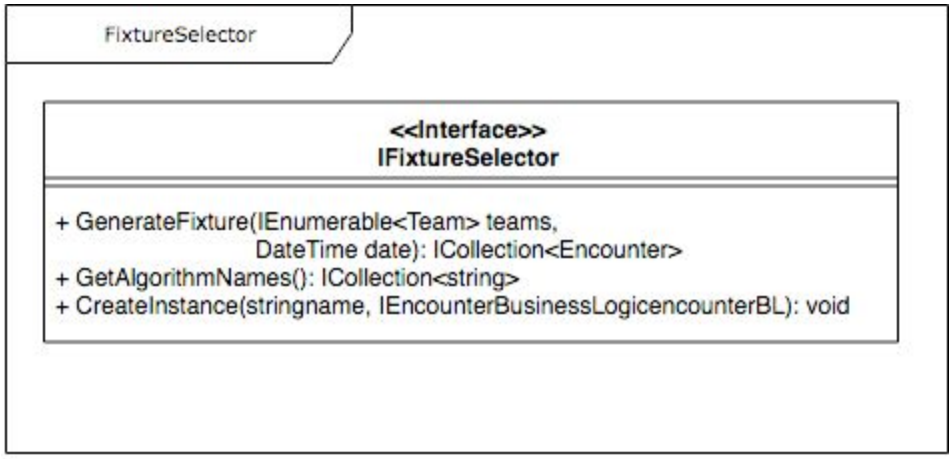




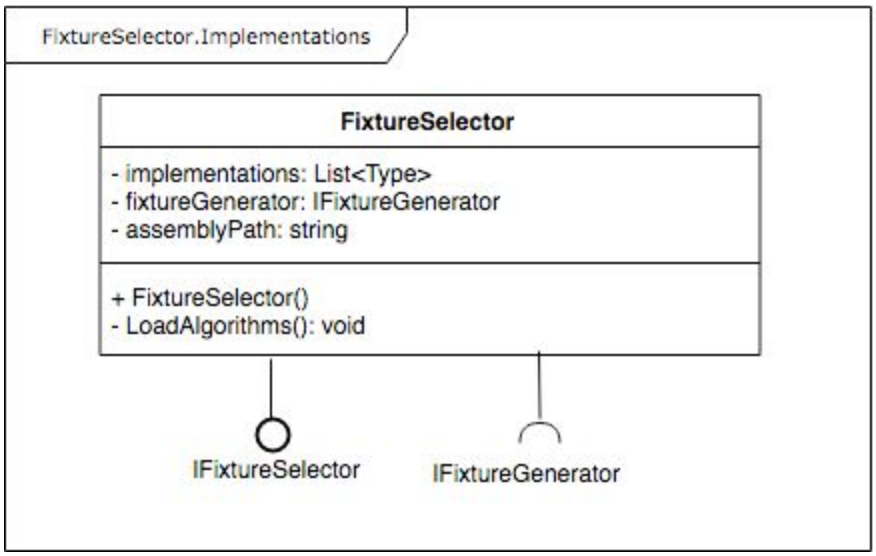
6.2.9. Portal.Filters



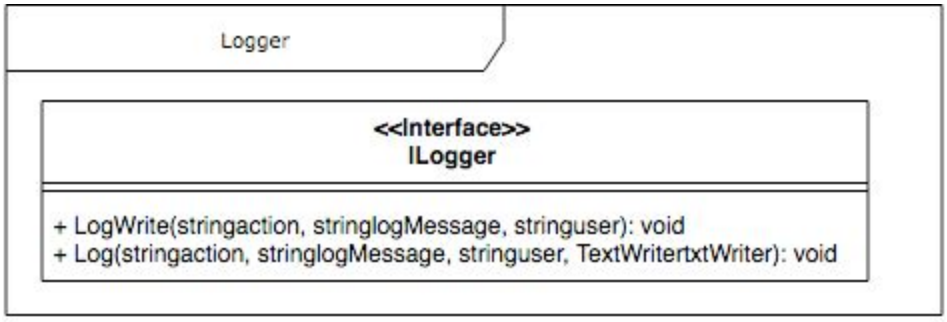
6.2.10. FixtureSelector



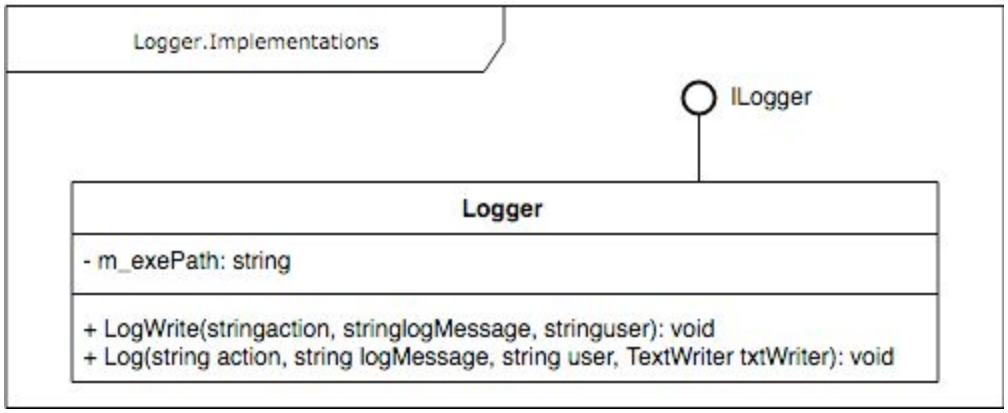
6.2.11. FixtureSelector.Implementations



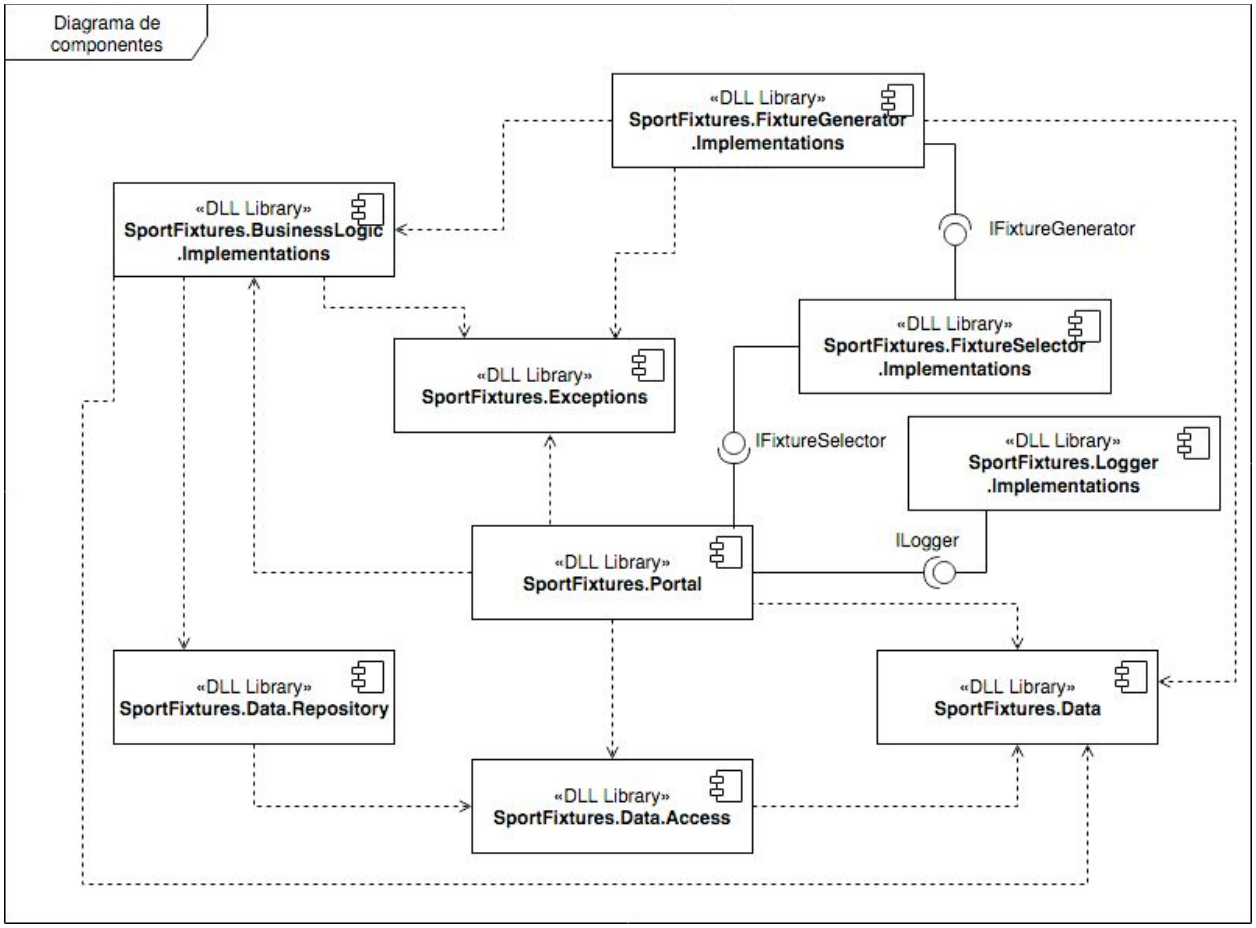
6.2.12. Logger



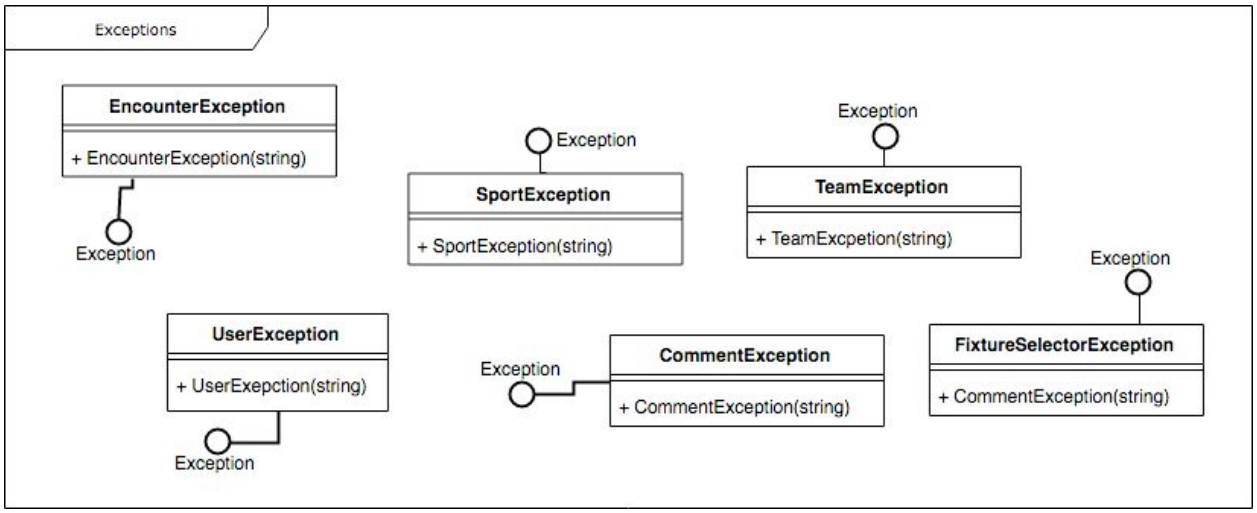
6.2.13. Logger.Implementations



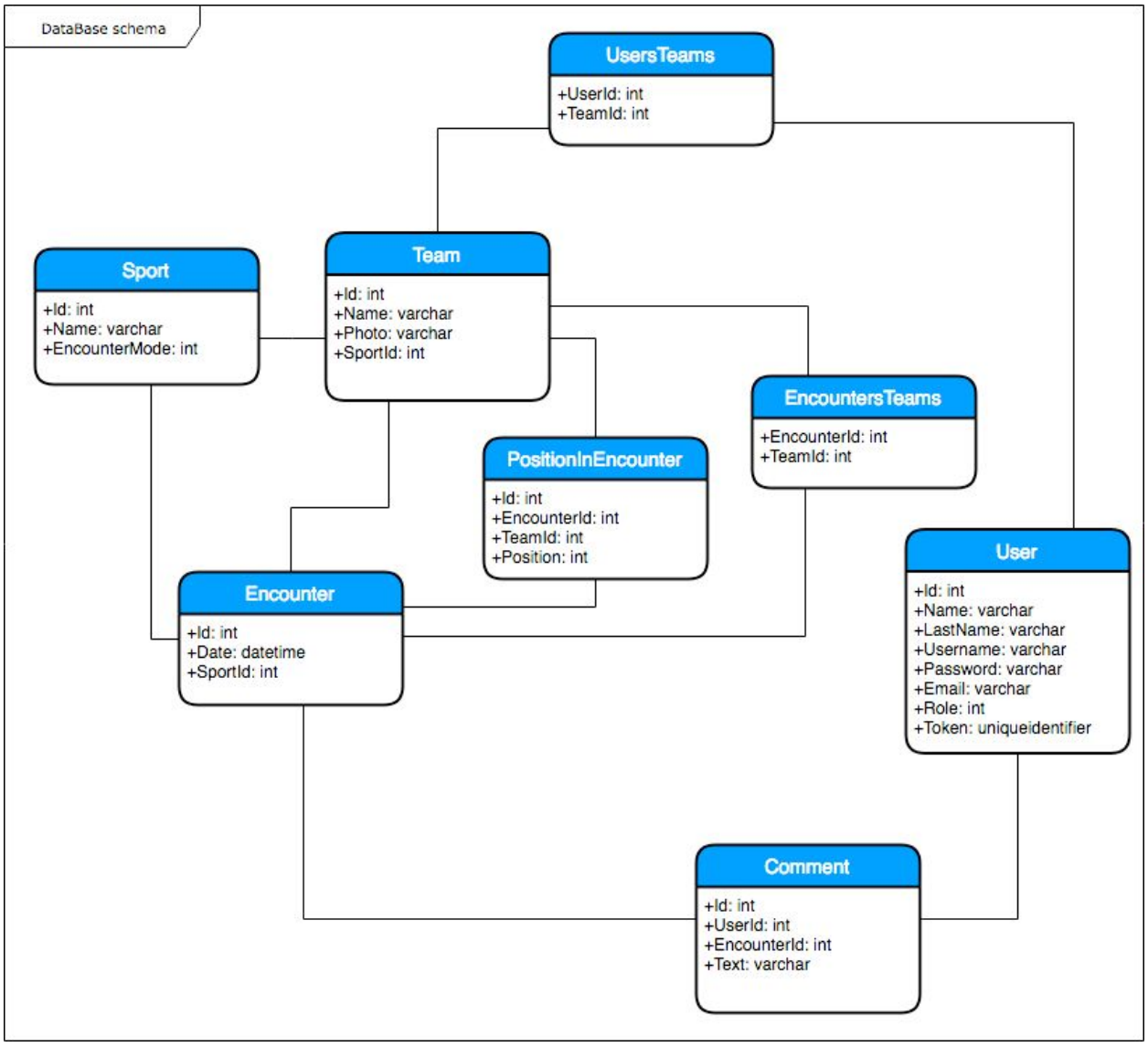
6.3. Diagrama de componentes



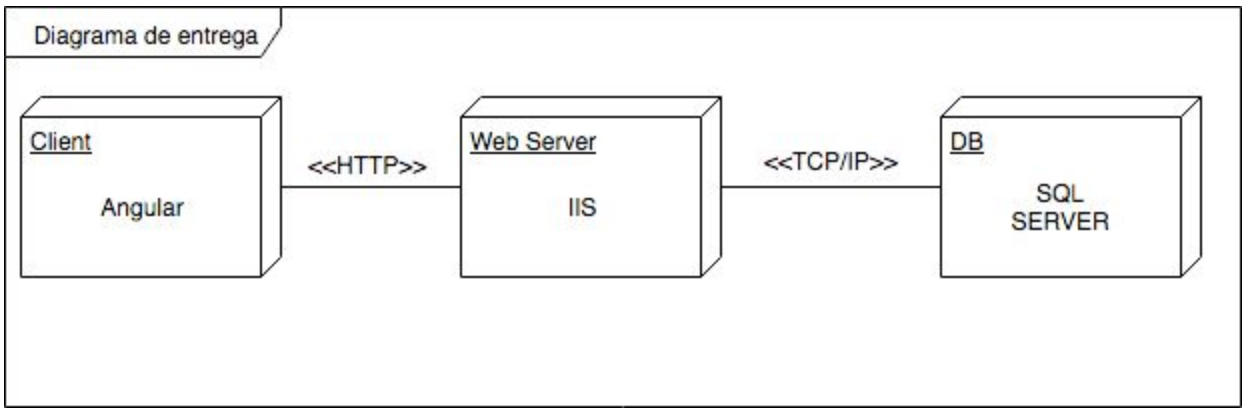
6.3.1. Exceptions



6.4. Diagrama base de datos

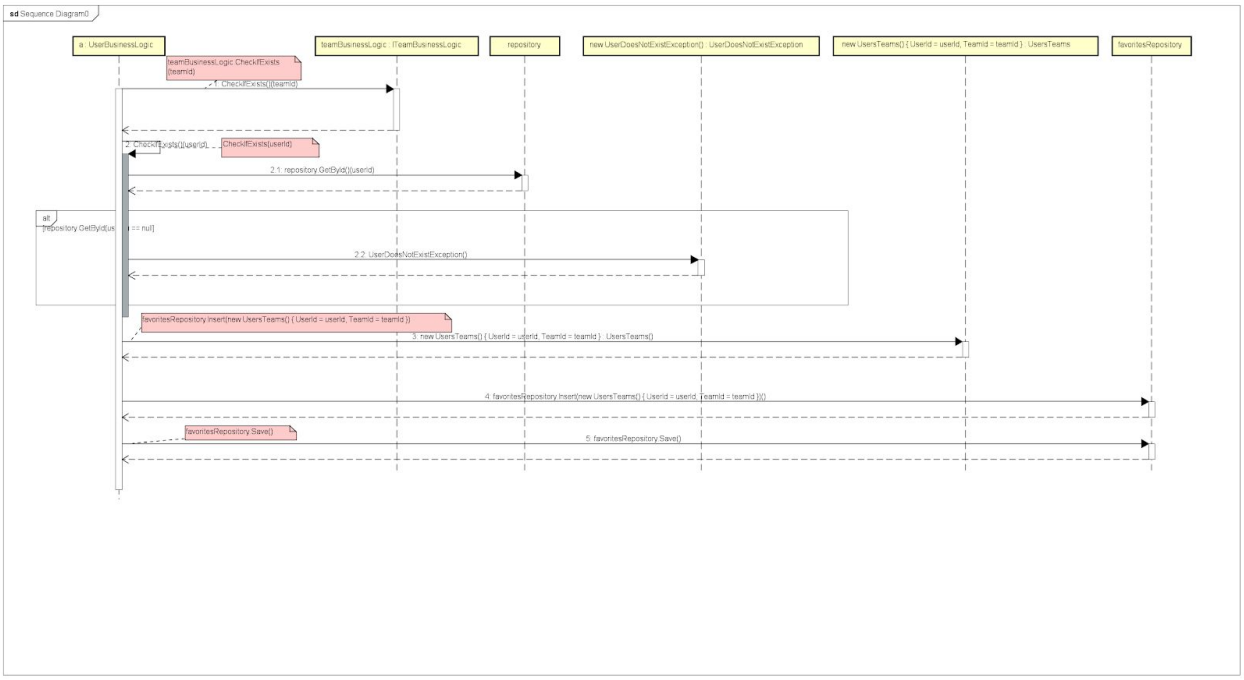


6.5. Diagrama de entrega



6.6. Diagramas de Interacción

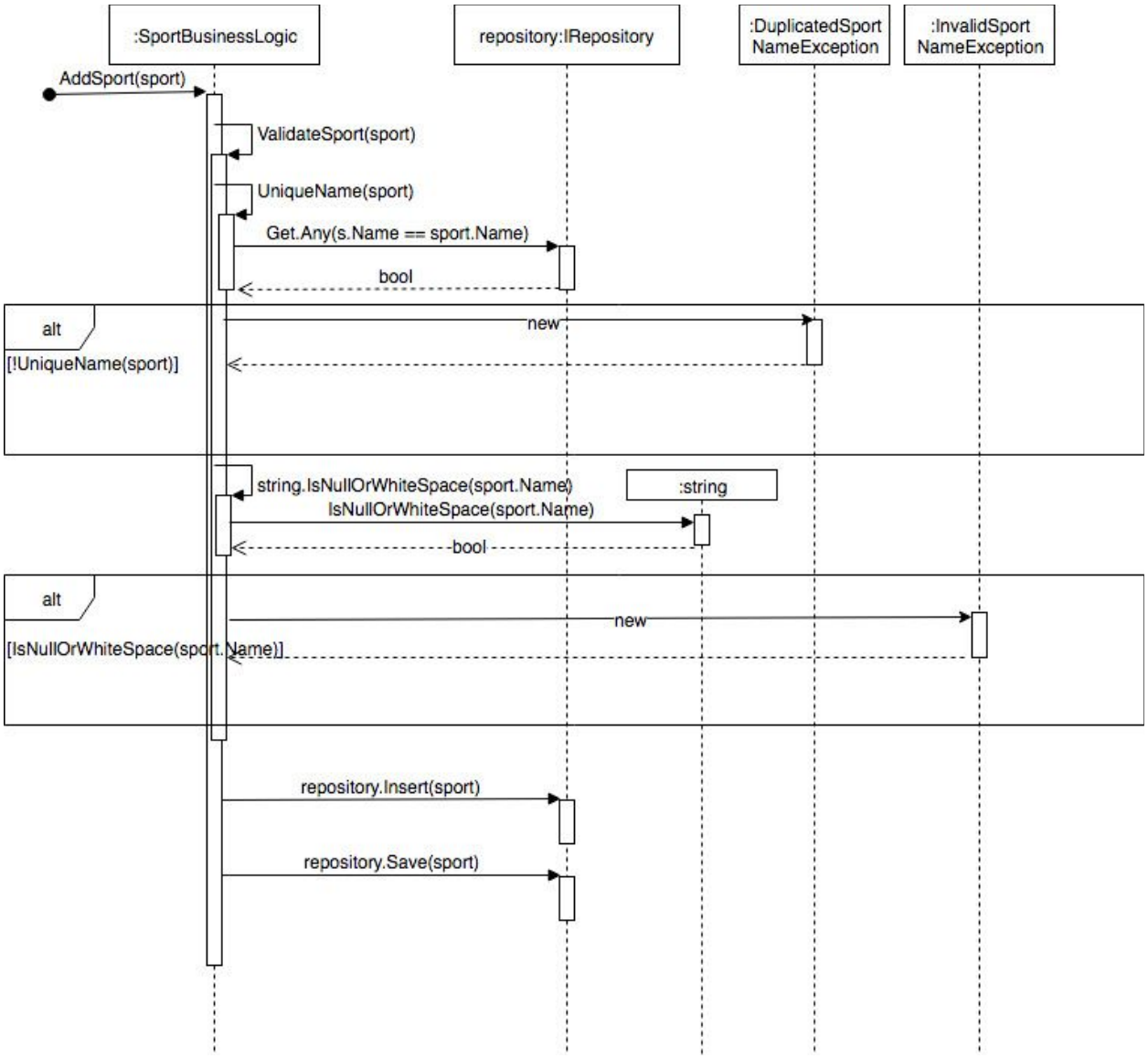
6.6.1. Follow Team



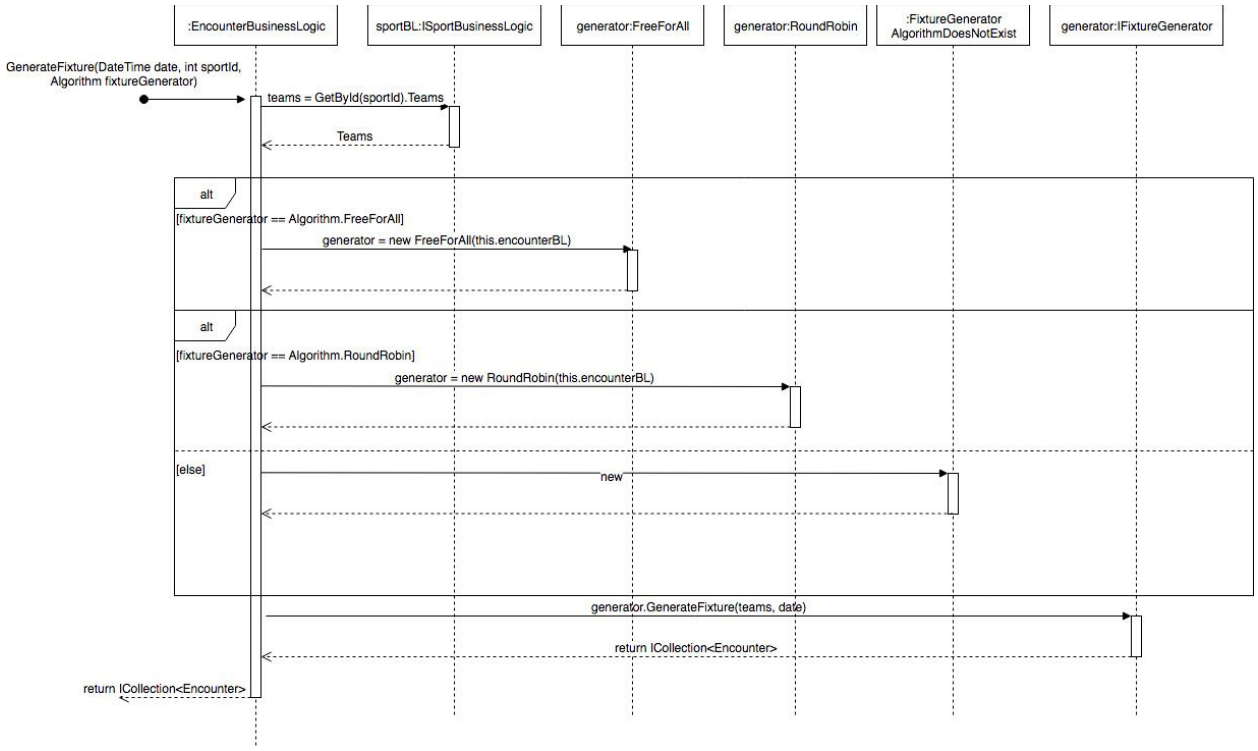
6.6.2. Login


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.GetAll(u => u.Email == user.Email, null, "");
1: repository.GetAll(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

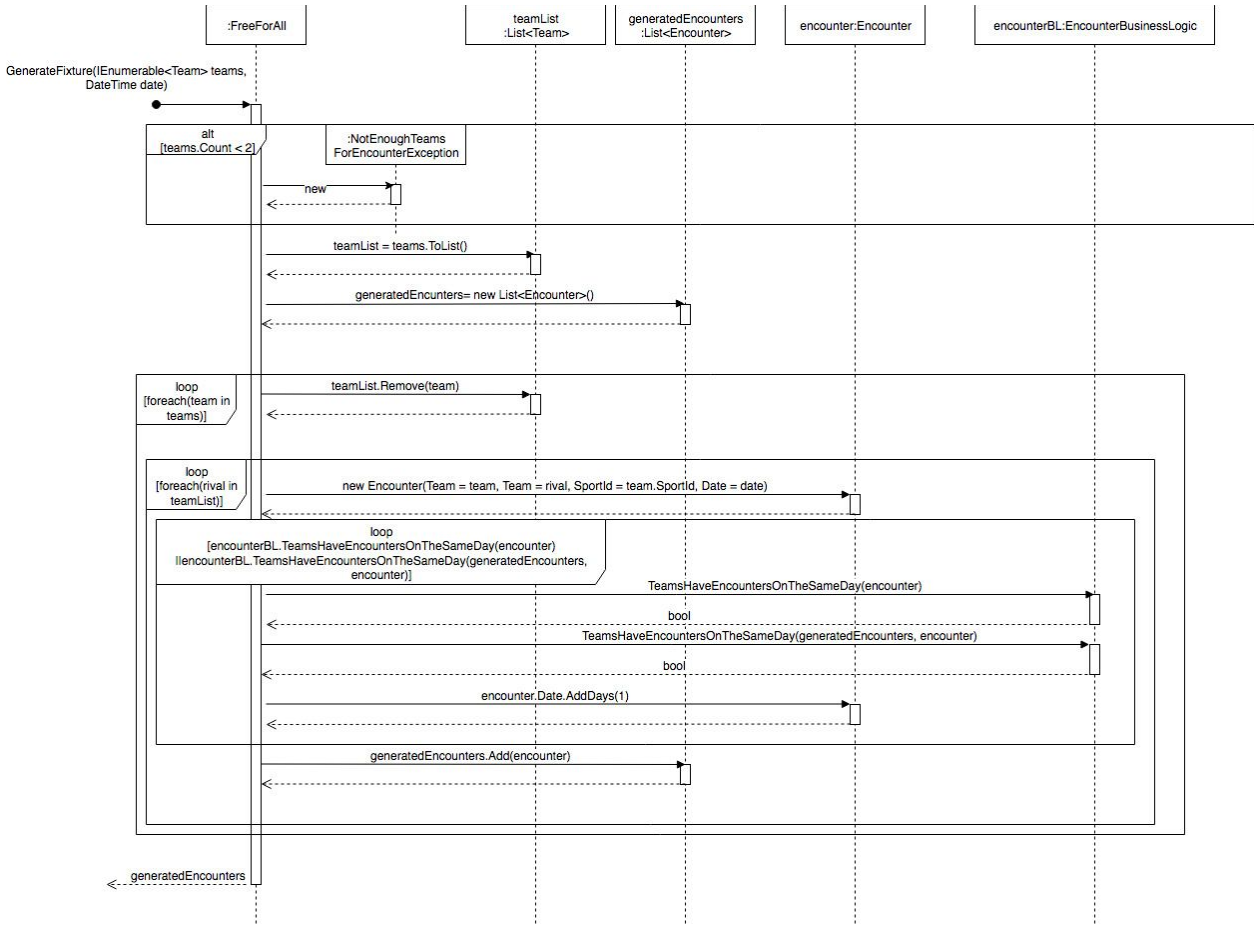
6.6.3. Add Sport



6.6.4. GenerateFixture



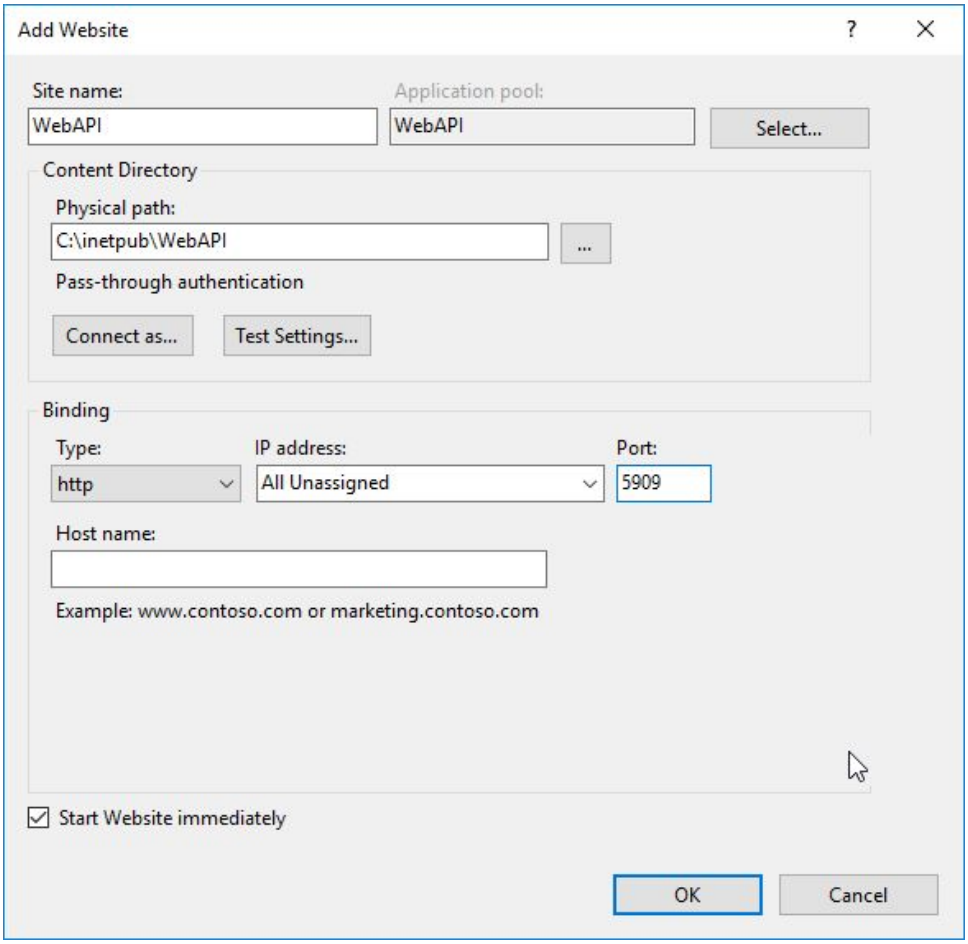
6.6.5. GenerateFixture - Algoritmo Free for all



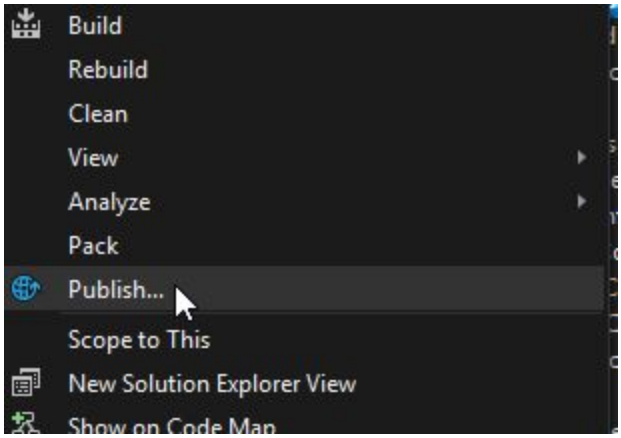
7. Anexo II - Instalación de WebAPI

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

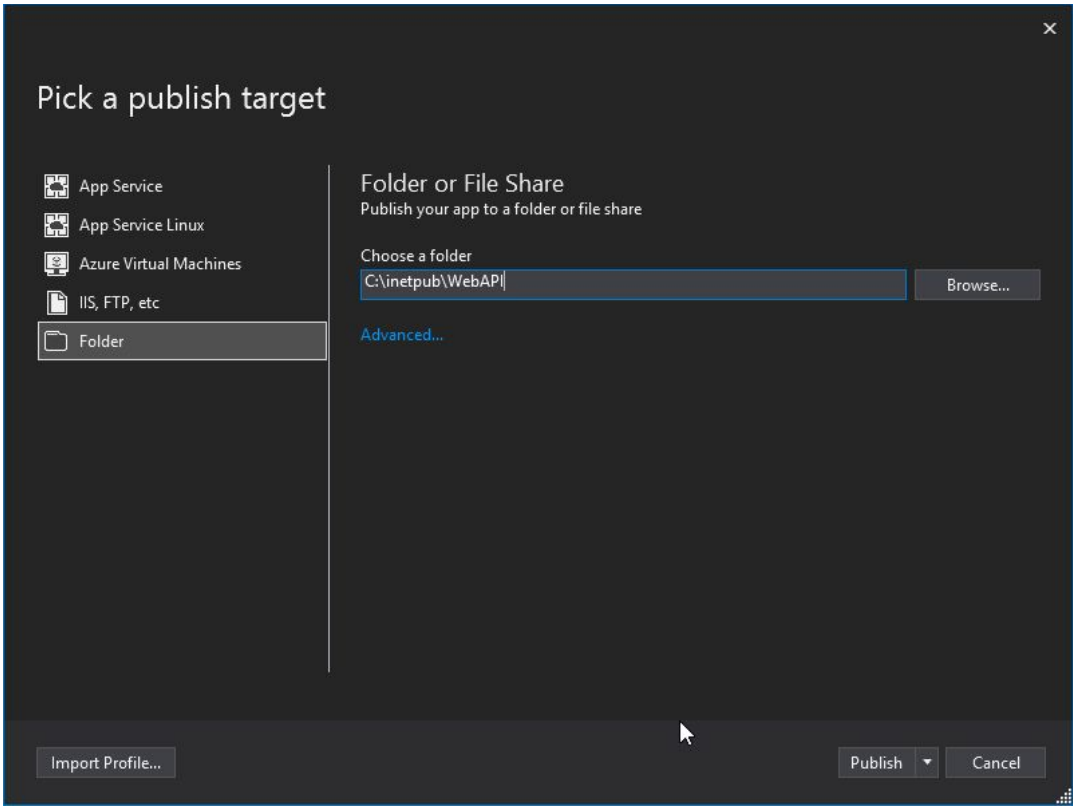
En la siguiente ventana configuramos la siguiente información:



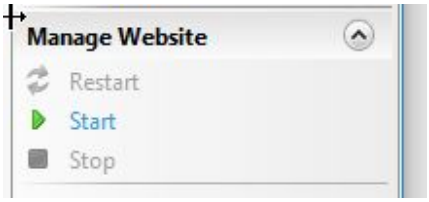
Elegimos un puerto que no esté reservado por el sistema (mayor a 1024) ni por otra aplicación. La ruta física debe ser donde pongamos los archivos generados cuando hacemos dotnet publish por línea de comando o cuando le damos publish en VisualStudio.



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

8. 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/

https://angular.io/

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

https://www.npmjs.com/package/angular2-toaster

https://www.npmjs.com/package/rxjs

https://angular.io/guide/http

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

https://stackoverflow.com/questions/20185015/how-to-write-log-file-in-c

https://stackoverflow.com/questions/9173904/byte-array-to-image-conversion

https://github.com/pollingerMaxi/LayersReflection

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

https://github.com/Sactos/HomeworksApi/blob/master/Clases/Clase%206%20-%20Reflection.md

https://www.npmjs.com/package/angular-calendar

https://www.primefaces.org/

https://fontawesome.com/

https://ej2.syncfusion.com/angular/documentation/daterangepicker/