Universidad ORT de Montevideo
Facultad de Ingeniería.

Obligatorio 1
Diseño de Aplicaciones 2

Facundo Arancet-210696
Marcel Cohen -212426

Proyecto disponible en:
https://github.com/ORT-DA2/Arancet_Cohen

Octubre, 2018

Índice

Descripción general del trabajo

El proyecto llevado a cabo, es un sistema que contiene deportes, equipos que juegan esos deportes y partidos jugados entre equipos. Los usuarios de la aplicación, pueden darse de alta, y seguir equipos y realizar comentarios en partidos. En la aplicación existen usuarios con rol de administrador, que tienen autorización para realizar el alta, baja y modificación de las entidades del sistema mencionadas anteriormente.
Una particularidad de este sistema, es que contienen algoritmos de creación de fixtures, es decir, algoritmos que generan una cantidad de partidos dada una fecha y un conjunto de equipos.

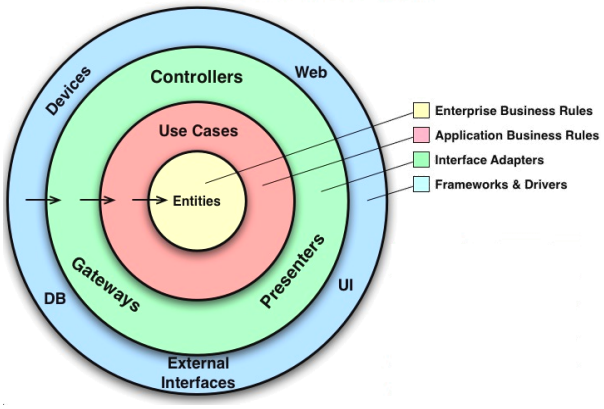
Errores detectados:

Uno de los errores detectados es que si bien pudimos manejar las excepciones de base de datos cuando esta es inaccesible, no se manejan excepciones cuando los datos de la base de datos son alterados y corrompidos.
Tampoco se logró que entity framework pudiera actualizar valores de claves y claves foráneas de entidades, estas acciones solo se pueden ejecutar haciendo baja y luego alta de las entidades. El resto de los campos se pueden modificar sin problemas.
Se procuró que la aplicación web tenga buena usabilidad, por eso, se intentó que los mensajes de respuesta sean lo más descriptivos posibles, sin embargo quedaron algunos mensajes de respuesta que solo tienen código 200 (OK) y ningún mensaje de respuesta.

Arquitectura y diseño

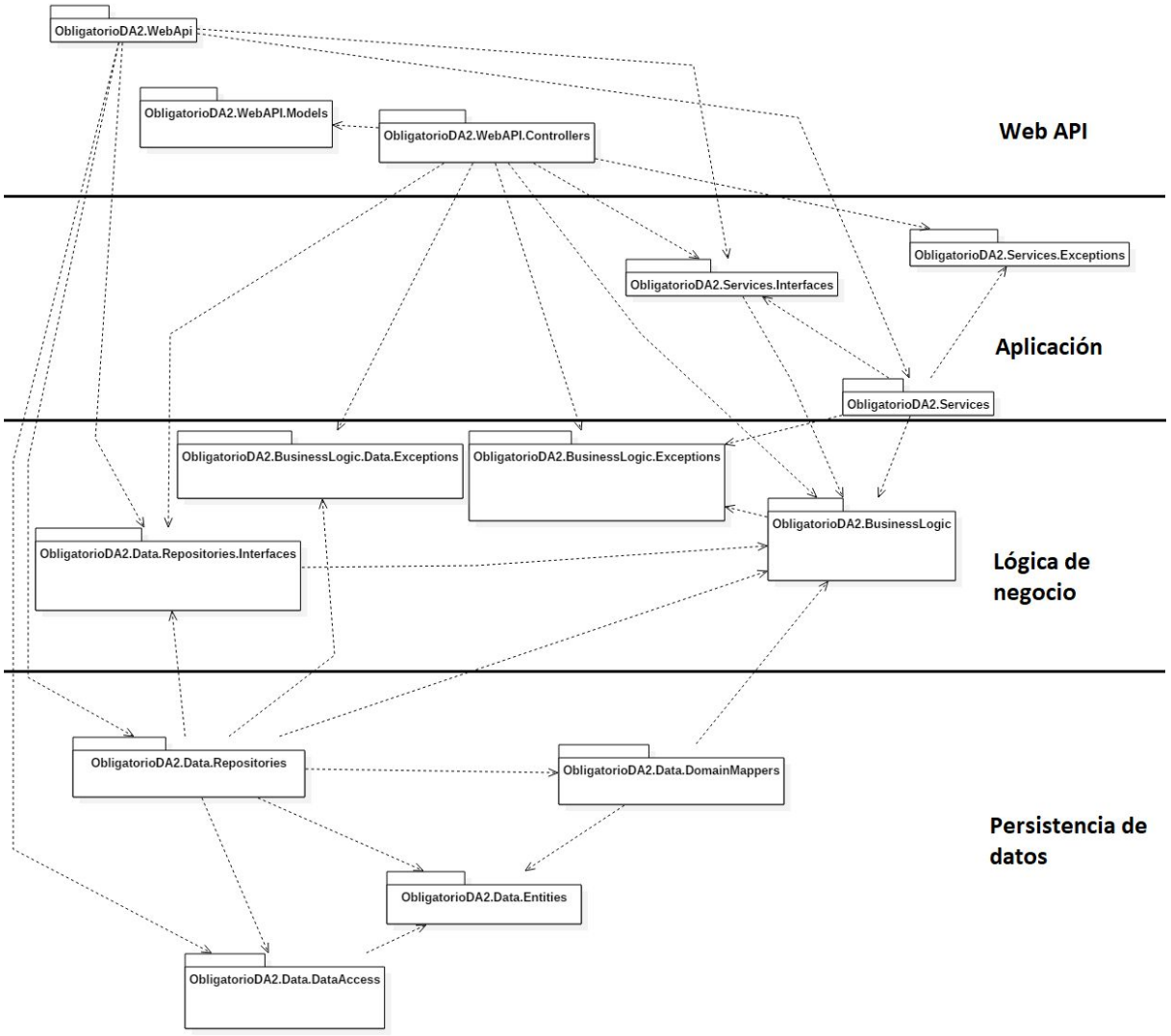
Diagrama general de paquetes

En el diagrama a continuación, se muestra una vista general de los paquetes del sistema y sus dependencias, separados en capas. Su finalidad es demostrar que se siguió la arquitectura vista en clase “Arquitectura clean”.



Para este diagrama, se tomaron todos los paquetes como disjuntos, es decir, se ignoró el hecho de que hayan paquetes dentro de paquetes, ya que hay casos donde un

namespace que contiene lógica de negocio (por ejemplo Data.Repositories.Interfaces), que están, lógicamente, dentro de un namespace que contiene módulos que implementan lógica de bajo nivel (en el ejemplo anterior, sería Data.Repositories), y el de lógica de alto nivel no depende de los módulos de bajo nivel, incluso se encuentran en assemblies separados, como se verá más adelante.

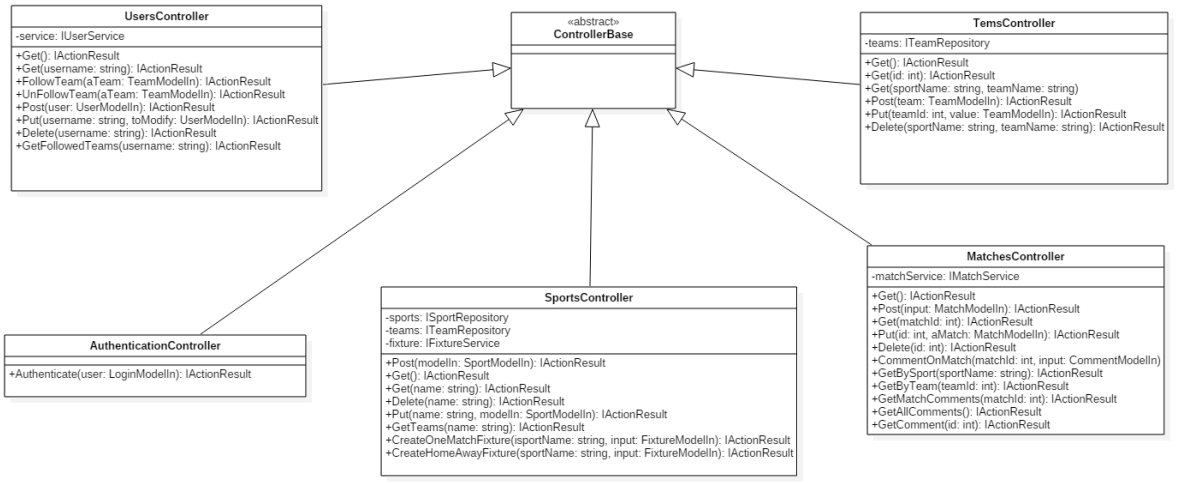


En la capa Web Api, se encuentran los controllers, los modelos de entrada y salida que utilizan, y también están las clases que se encargan de inyectar las dependencias en el sistema, como se puede ver, el paquete WebApi depende de las implementaciones de los repositorios y servicios así como de sus interfaces, también depende de la persistencia de datos, esto es por que dentro de esta esta la clase startup que tiene la responsabilidad de instanciar esas dependencias e inyectárselas a quienes la usen. De esta forma, podemos cambiar las implementaciones de los servicios y de la persistencia, minimizando el impacto de cambio. Nos apoyamos en dos principios GRASP: Polimorfismo, ya que con las interfaces de repositorios y servicios podemos tener muchas implementaciones. El segundo principio es protección de las variaciones, por que el sistema se abstrae de la persistencia, que es una implementación muy cambiante.

También nos basamos en el principio SOLID de inversión de la dependencia, ya que la lógica de negocio y de aplicación no dependen de los detalles de implementación, por todo esto que ya mencionamos.

Diagrama de clases del paquete de Controllers

Este paquete (ObligatorioDA2.WebAPI.Controllers) contiene los controladores, son las clases que se encargan de atender los eventos de el sistema. Los clientes acceden a la aplicación por medio de estas clases.



Al recibir peticiones, los controllers validan que sean correctas y llaman a los servicios o repositorios. Se decidió utilizar servicios sólo para algunas operaciones, las que tienen un flujo de negocio, como la autenticación y el armado del fixture. Existen otras operaciones que son más sencillas, como lo relativo a los equipos y deportes, donde las operaciones son en su mayoría alta baja y modificación, entonces un Service para estos recursos no tendría lógica, solo llamaría a los métodos de los repositorios, lo cual solo agregaría complejidad al diseño. Entonces, se optó por la creación de servicios solo cuando hay lógica de la aplicación, que no debería ser delegada a los repositorios. Por ende algunos controllers utilizan services, y estos usan los repositorios, y otros directamente utilizan los repositorios.

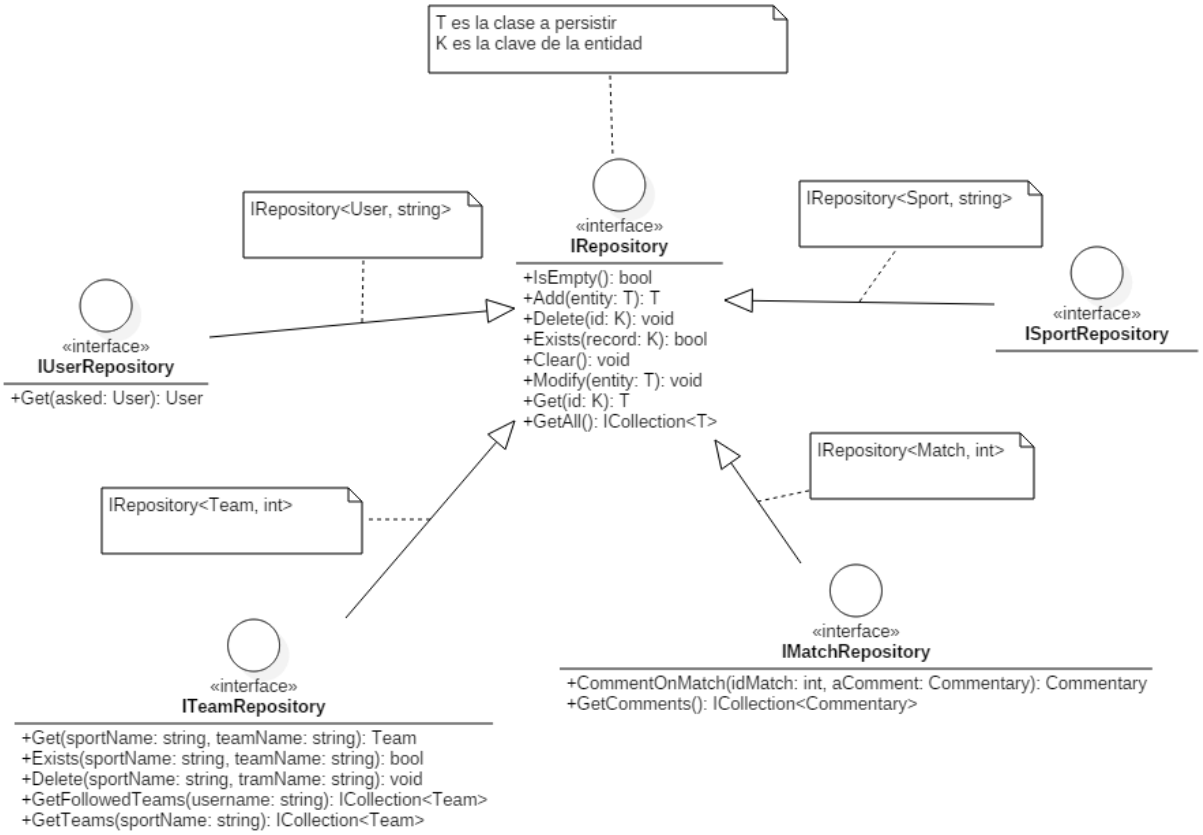
Todos los métodos de los controllers tienen tipo IActionResult, se tomó esta decisión porque el IActionResult permite manejar los StatusCodes de Http con mayor flexibilidad. El equipo procuró ser lo más descriptivo posible con los mensajes de retorno para tener una buena usabilidad. Esto implicó un sacrificio a nivel de diseño ya que se optó que el controller distinguiera entre distintos tipos de excepciones del sistema para poder enviar distintos códigos de error. Ampliando más esta idea, para las excepciones de entidades no encontradas, se enviaría un mensaje de error con código 404, para las excepciones por entidades que ya existen y se quieren agregar, mensajes con código 400 y para otras excepciones como las de datos inaccessibles (que en esta

versión son por que no se puede conectar a la base de datos), mensajes de error 500, ya que es un problema del servidor.
Esta decisión implicó que los controllers atraparan muchas excepciones para enviar los distintos mensajes de error, en lugar de una excepción más genérica, y esto hace el código más complejo.

Por último se usan models, que son objetos de transferencia de datos que usan los controllers para recibir y devolver datos. Representan entidades del dominio, pero en general muestran menos información, por ejemplo, el UserModelOut no lleva la contraseña, por seguridad.

Repositorios

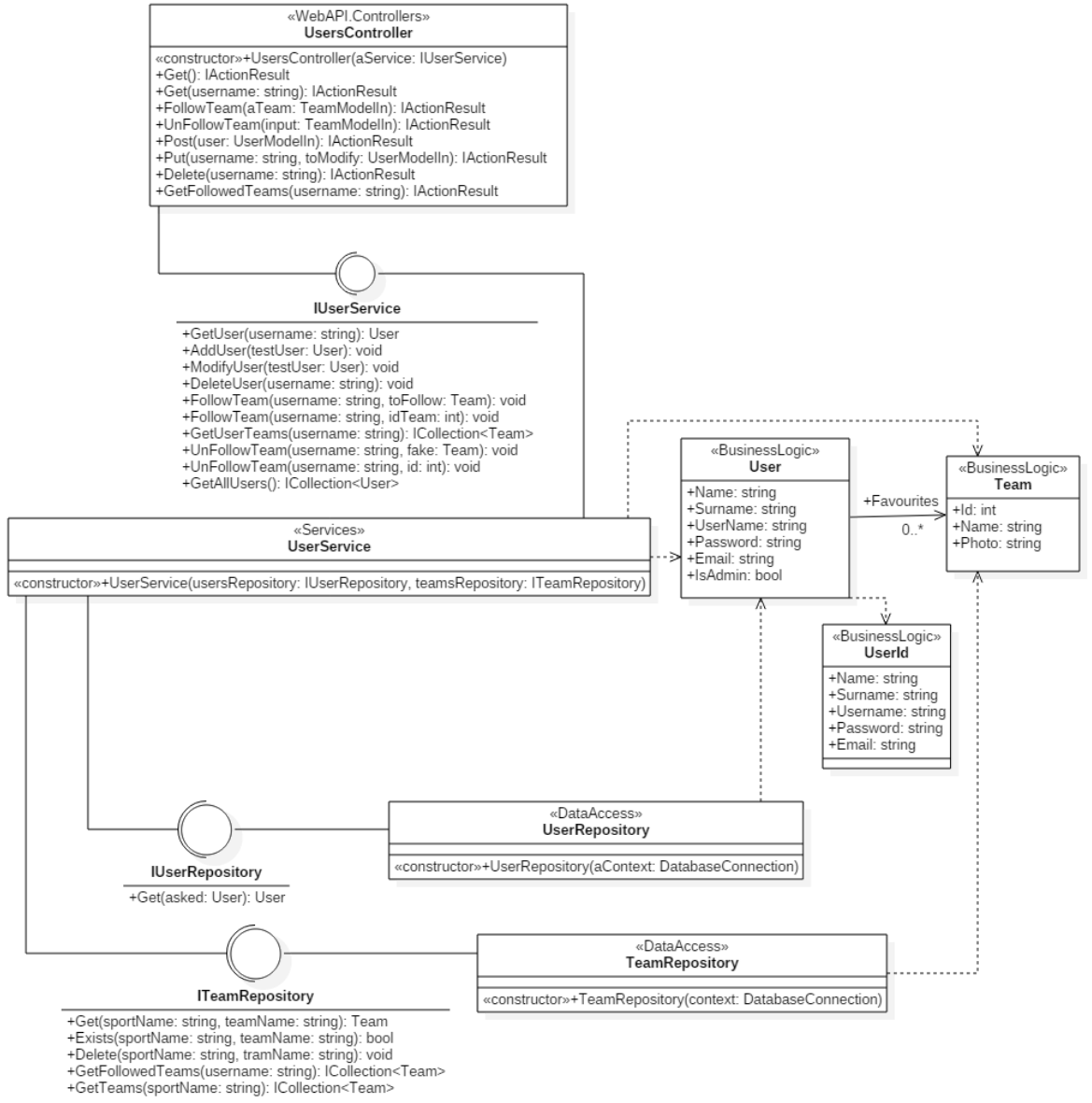
El acceso a datos se realiza utilizando diferentes interfaces que implementan una interfaz genérica IRepository.



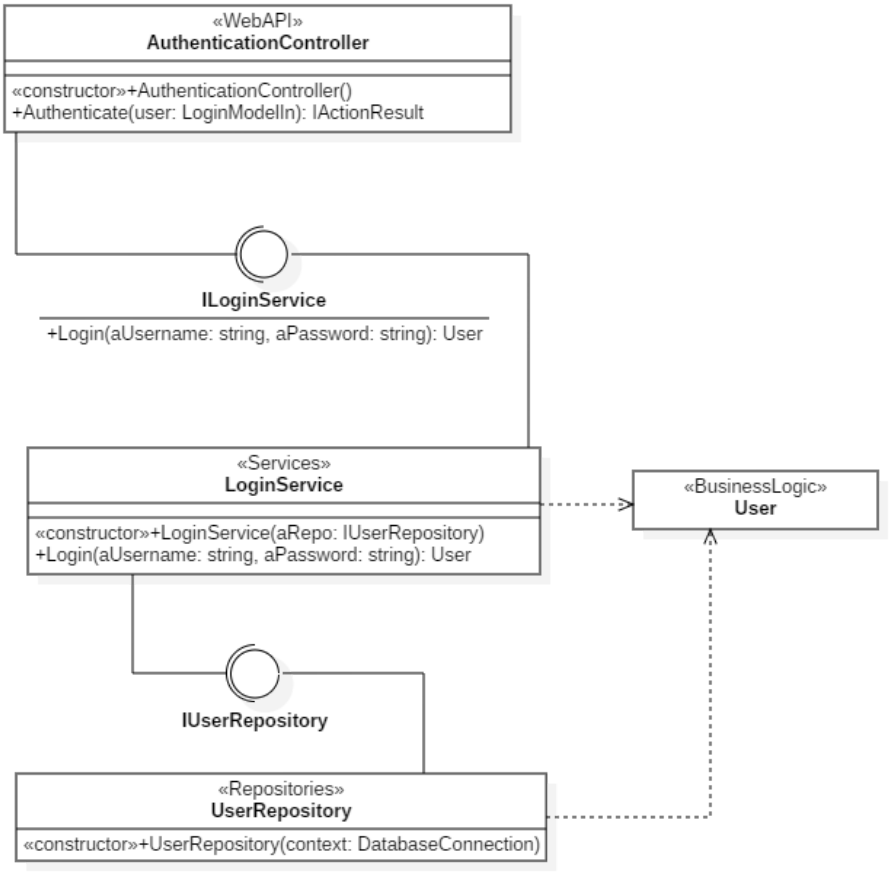
Cada una de estas interfaces implementa IRepository con sus tipos correspondientes. Además, definen operaciones concretas de cada repositorio.
Por ejemplo, ITeamRepository define Get, Exists y Delete con el nombre del equipo y el nombre del deporte, ya que estas conforman una clave alterna de la entidad. Hacer uso de esa clave alterna y no de una clave numérica, permite crear end points en la API Rest más amigables para el usuario (ej: “teams/football/nacional” en lugar de “teams/673”).
Aclaración: Si bien “teams/football/nacional” es más amigable para el usuario que “teams/673”, lo mejor hubiera sido que el recurso fuera sports/football/teams/nacional. La mejora se detectó el día de la entrega por lo que no se implementó para evitar la introducción de bugs.

Usuarios

El sistema requiere almacenar información de los usuarios y administrar credenciales para el inicio de sesión y uso de ciertas features. Dado que no se exigía una solución compleja de la administración de usuarios, se optó por tener una clase concreta que se encargue de modelar la solución.
Los usuarios pueden seguir a sus equipos favoritos y consultar por sus próximos encuentros.
Un usuario puede ser administrador o no serlo. En caso de serlo, puede ejecutar funcionalidades del sistema que los demás usuarios no (como la creación de nuevos encuentros).
El siguiente diagrama muestra cómo se hace uso de esta funcionalidad y cuáles son las interfaces utilizadas:



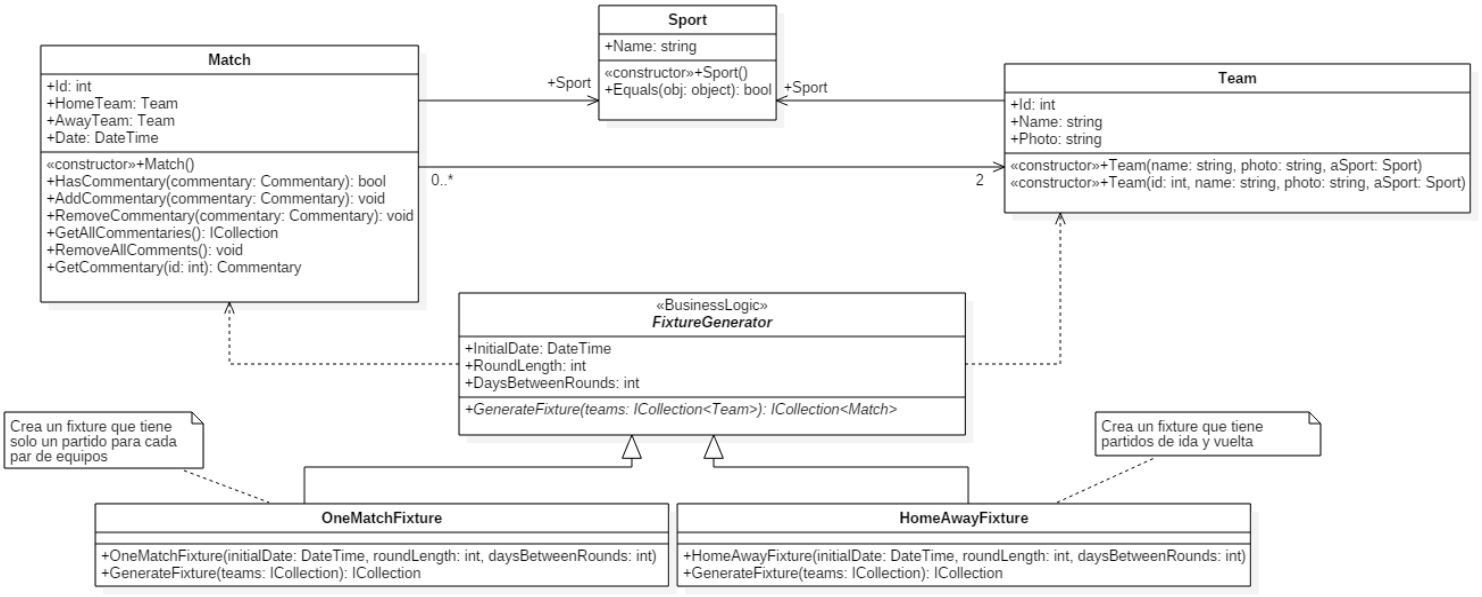
Como se mencionó anteriormente, un usuario puede ejecutar ciertas funciones del sistema si tiene permisos para ellos. Es por esto que se requiere autenticación.



Un usuario se autentica utilizando su nombre de usuario y contaseña.
Cuando un usuario se autentica, se le entrega un token generado por el sistema. El usuario debe enviar el token cada vez que desee utilizar funcionalidades que requieran permisos de administrador. El token expira en 30 minutos.

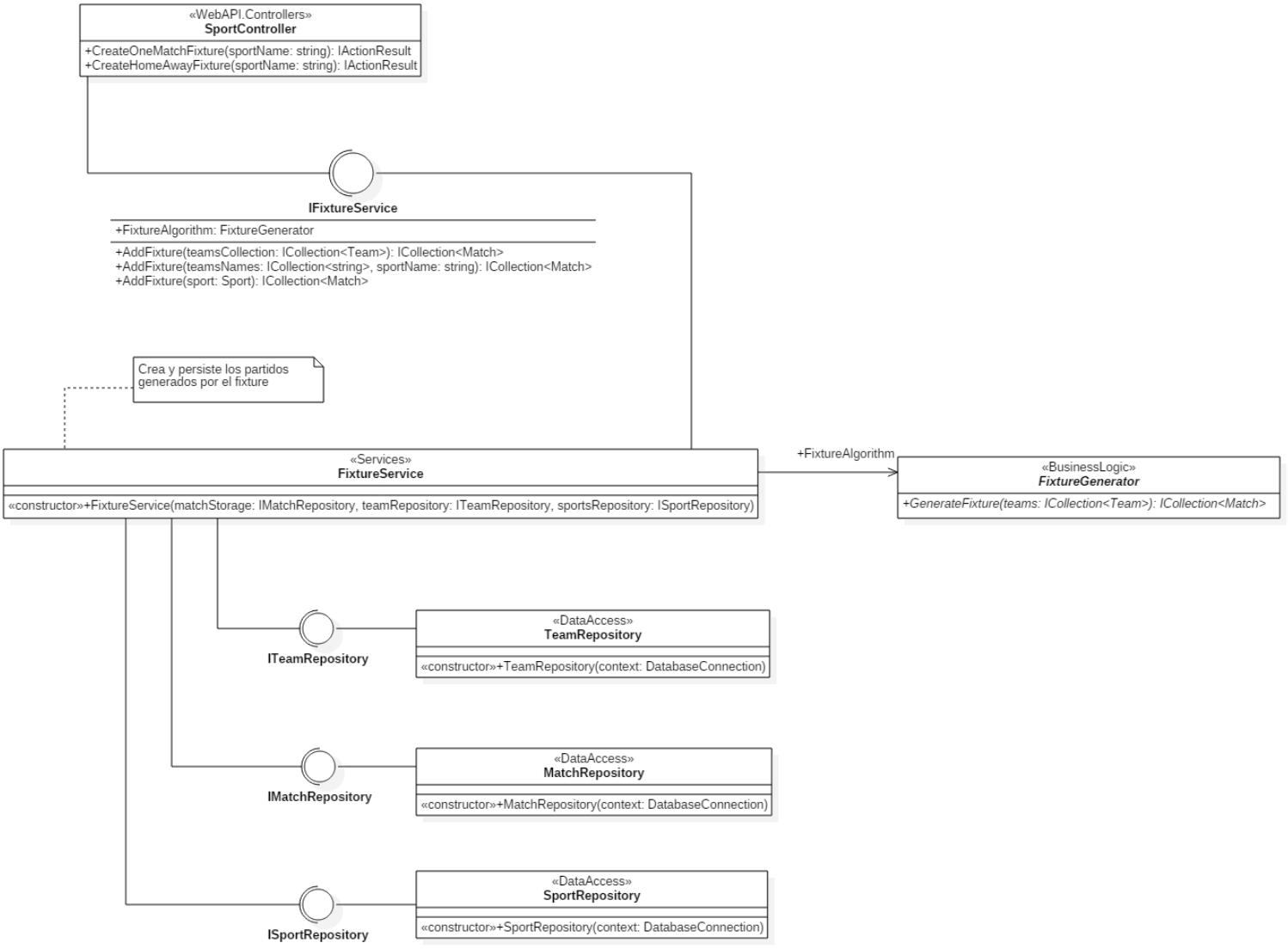
Lógica del Fixture

Para desarrollar este requerimiento funcional, se aplicó el patrón Strategy. La clase FixtureGenerator posee un método abstracto para generar un fixture, que sus clases hijas implementan, permitiendo la incorporación de nuevos algoritmos de armado diferentes sin impacto en el código de los usuarios de esta clase.
Se implementaron dos variantes para el algoritmo de armado del fixture, OneMatchFixture y HomeAwayFixture. El primero crea una lista de partidos tal que todos los equipos juegan contra todos los demás, y en el segundo se hace lo mismo que el primero pero agregando además partidos de vuelta.



La clase FixtureService hace uso de la lógica descrita anteriormente. La misma se encarga de elegir el algoritmo y también verifica que los equipos ya no tengan otros partidos en la misma fecha que alguno de los partidos que serán agregados.

En el siguiente diagrama se aprecia claramente la separación de capas y la utilización de interfaces para cumplir con el principio de inversión de dependencias:



La lógica del fixture no depende de la API Rest, ni del acceso a datos. Sí requiere de interfaces para los repositorios de equipos, partidos y deportes.

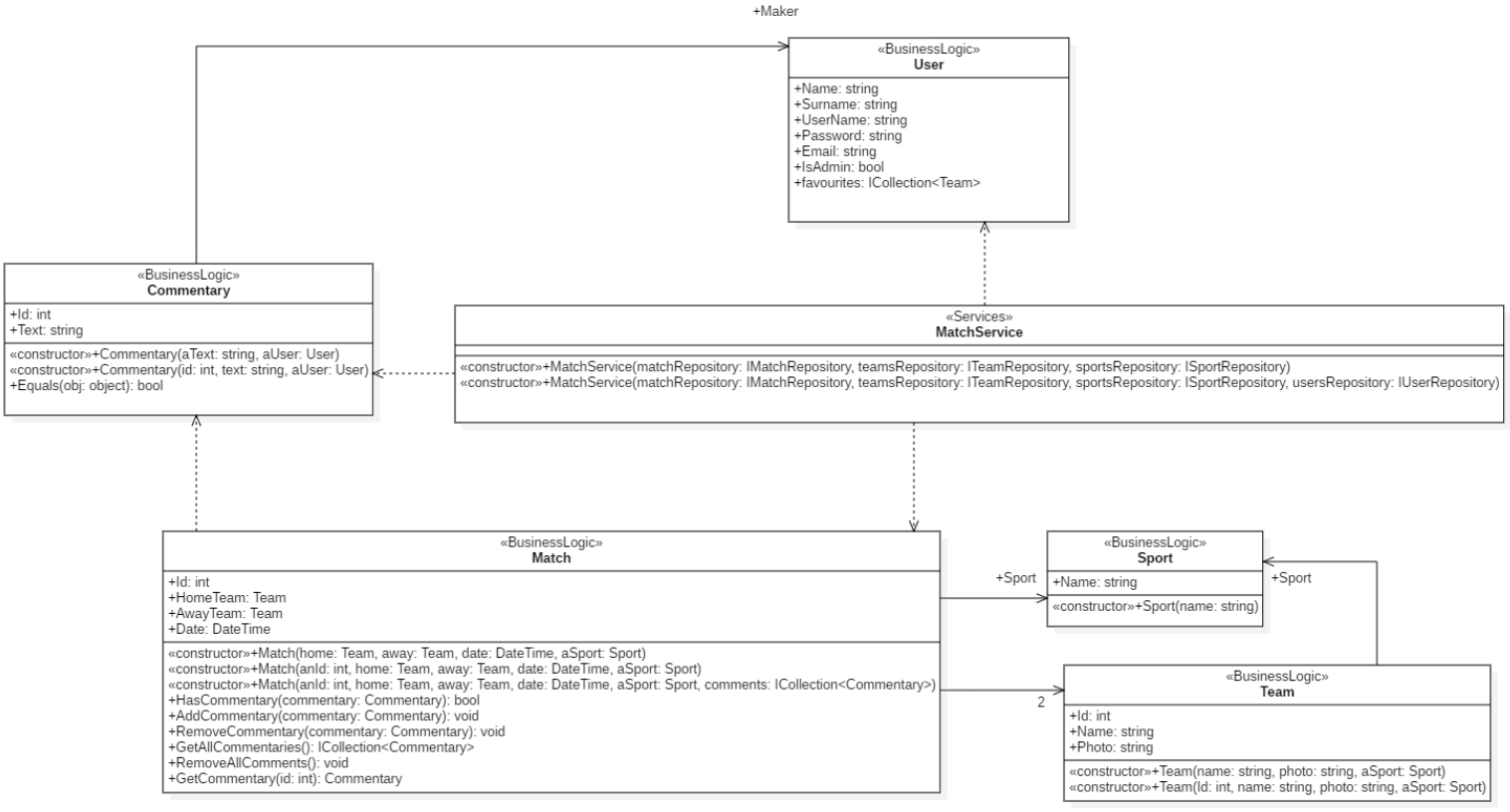
Supongamos que queremos agregar un nuevo algoritmo para la creación del fixture. Lo que debemos hacer es: programar el algoritmo (heredando de la clase FixtureGenerator) y creando una nueva función en SportController que instancie FixtureService con el nuevo algoritmo creado. El impacto del cambio en el código de otras clases sería nulo.

Esta separación en capas también permite que el sistema sea portable. Esto quiere decir que, por ejemplo, si se desea migrar el sistema a una aplicación de windows forms, bastaría sólo con programar la nueva UI haciendo uso de la interfaz IFixtureService, provista por FixtureService.

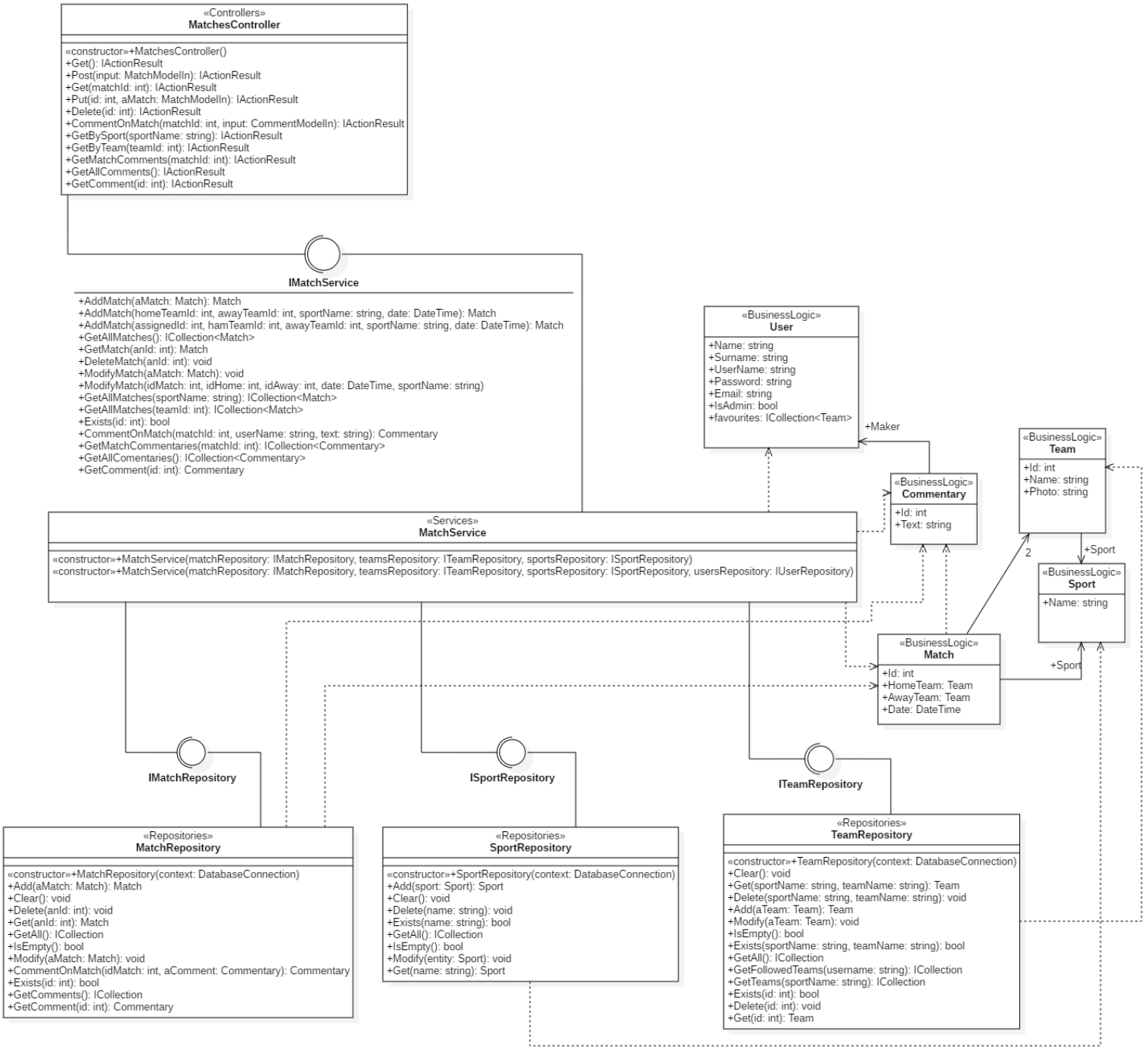
Además, el uso de interfaces para el acceso a datos soporta el cambio de implementación de persistencia de datos, sin impactar en el código de la lógica del fixture.

Encuentros

Un encuentro está dado por dos equipos, una fecha, un deporte y una lista de comentarios realizado por usuarios. Un encuentro tiene un equipo local (home) y otro visitante (away). La clase Match se encarga de modelar los encuentros.
La clase MatchService se encarga de la comunicación con los repositorios requeridos para la persistencia de encuentros. Es en esta clase que un equipo no tenga dos encuentros el mismo día.
No consideramos necesaria la creación de un service exclusivo para comentarios dado que son algo propio de los encuentros, MatchService puede encargarse de esta tarea.



El siguiente diagrama muestra la interacción entre controller, service y repositorios en capas:



Para los comentarios de los encuentros, se decidió que la interfaz IMatchRepository tenga una función que agregue un comentario a un encuentro. Esto significa que esta lógica de negocio quedaría del lado de DataAccess, en la implementación de IMatchRepository, lo cual no es conveniente porque al cambiar la implementación de base de datos se debe reescribir este código. Sin embargo, al hacer esto, se logra una mayor eficiencia para esta función, dado que se realiza una sola llamada a la base de datos. Suponemos que la funcionalidad de comentar un encuentro será usada con mayor frecuencia que las demás, por muchos usuarios, por lo que priorizar la eficiencia en este punto puede ser favorable.

Equipos y Deportes

Para las operaciones con encuentros y deportes decidimos que no sería necesario tener services debido a que no existe flujo de negocio en ellas, son sencillas y directas . Por esto, dejamos que el controller llame directamente al repositorio. En caso de que cualquier regla de negocio surja referente a encuentros y deportes, será necesario crear services para ellos, de lo contrario quedaría lógica en los controllers o en los repositorios concretos.

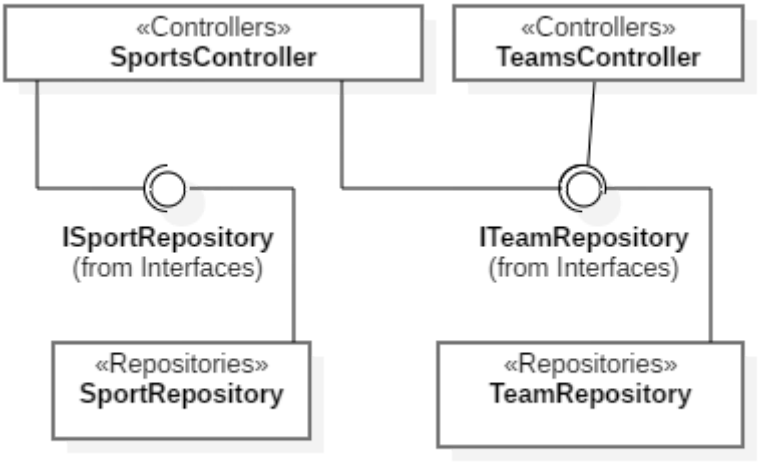
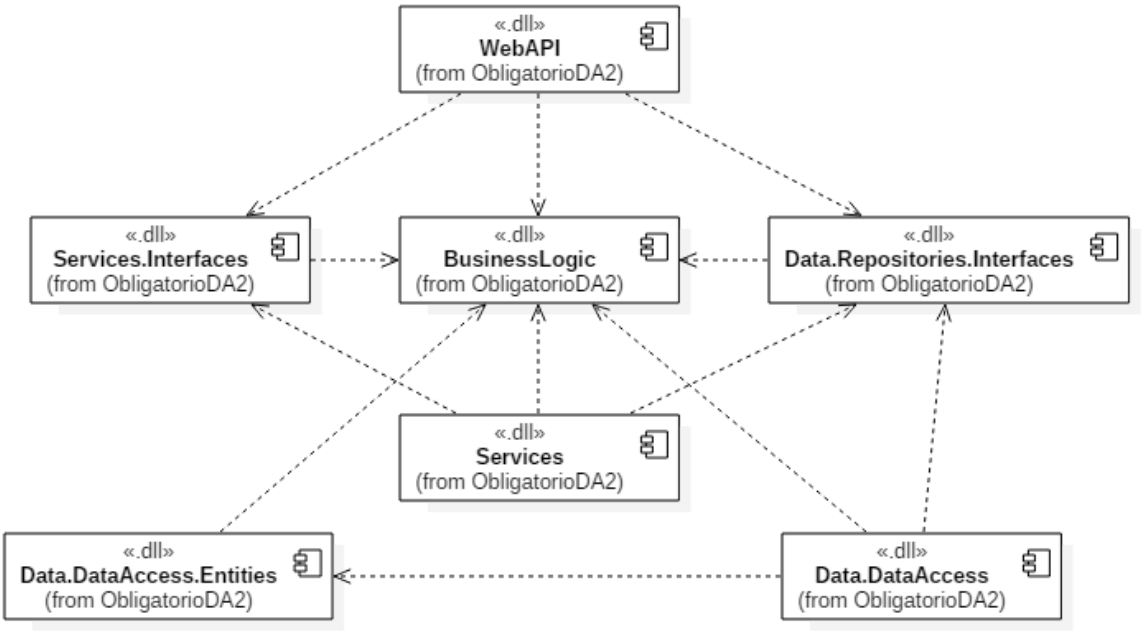


Diagrama de componentes

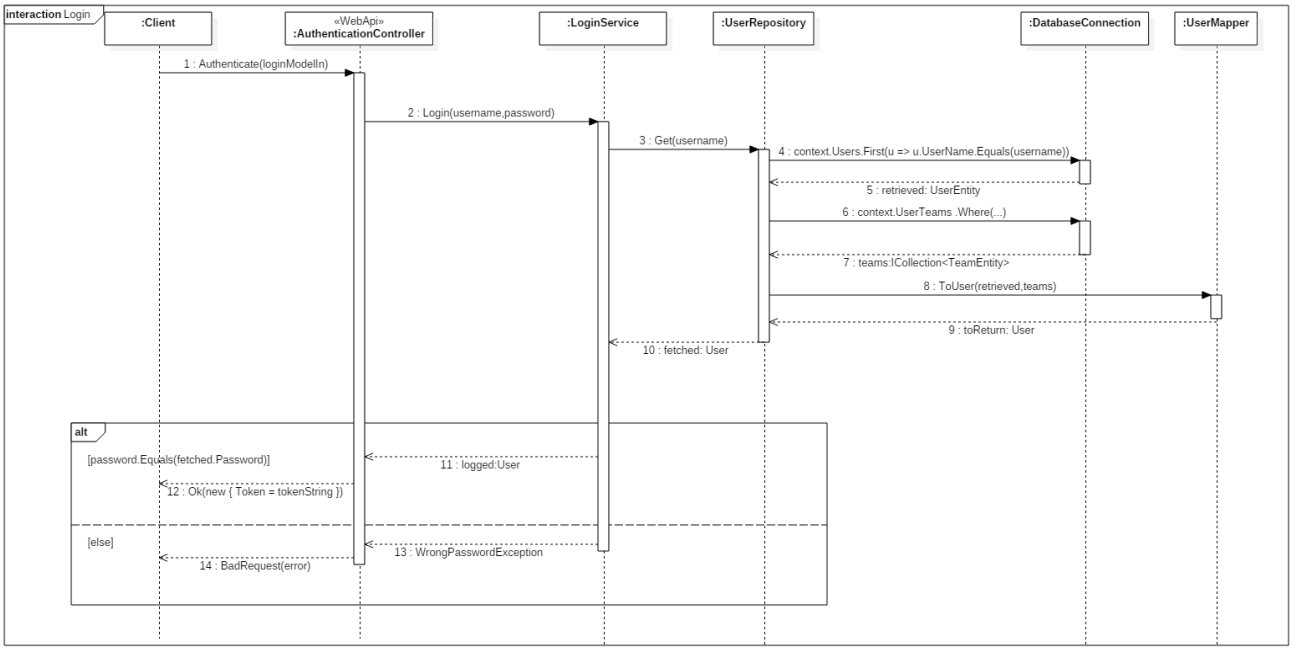


En el diagrama anterior, se puede observar que la lógica de negocio no depende de ningún otro componente y que los servicios no dependen de componentes de bajo nivel, como lo son el acceso a datos o la api rest.
Decidimos utilizar estos componentes porque son los que consideramos que le otorgan mayor portabilidad al sistema.
Como se puede apreciar en el diagrama, las interfaces están separadas de su implementación. Esto permite realizar cambios en la implementaciones sin que haya impacto en los otros componentes.
Por ejemplo, supongamos que se quiere cambiar el motor de base de datos, el cambio impactaría en el componente Data.DataAccess.

Diagramas de secuencia

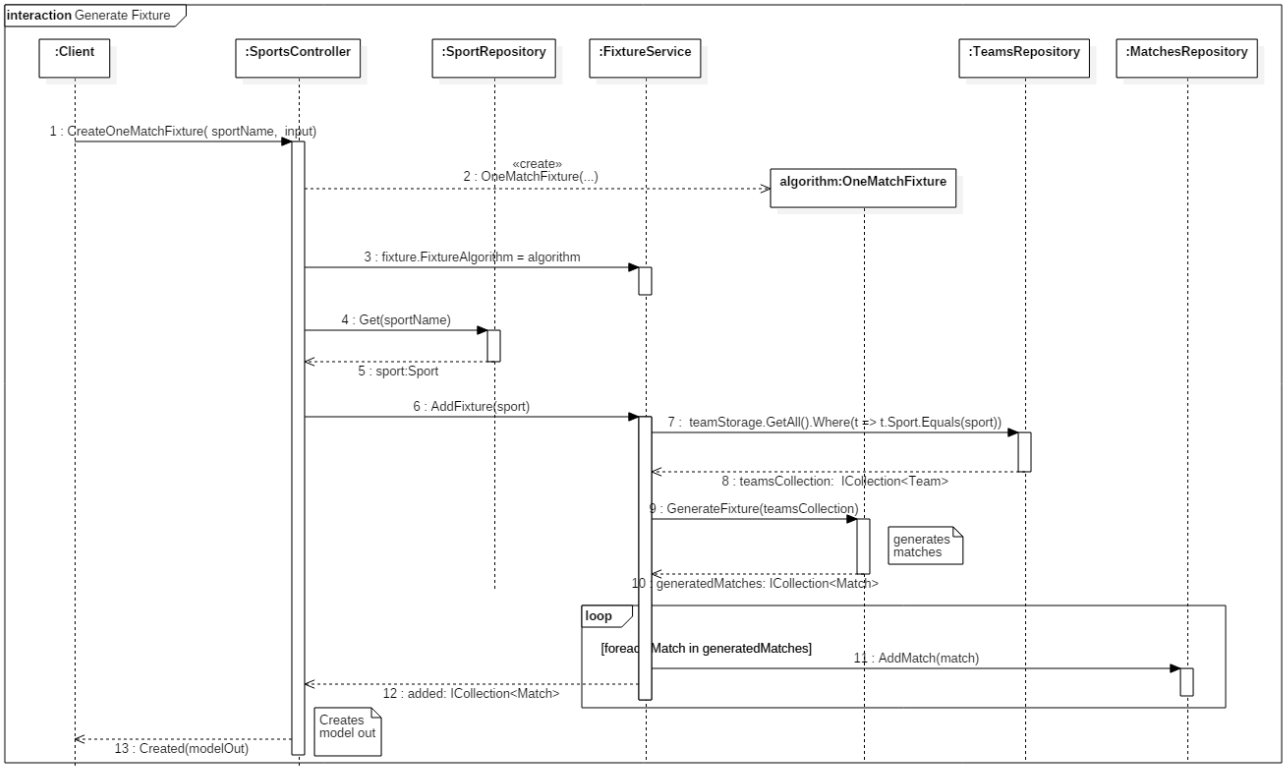
Los siguientes diagramas ilustran el flujo de algunos procesos del sistema. Estos son ejemplos de procesos donde existe cierto flujo de negocio y no son simplemente operaciones CRUD. En el proyecto se intentó conservar esta lógica dentro de la capa de aplicación, y no dejar que migre hacia la capa de persistencia o la capa de los controladores de WebAPI, ya que, como mencionado anteriormente, estas capas contienen los detalles de la implementación, que son muy cambiantes. Por lo tanto, el tener lógica de la aplicación en estas partes del sistema implica que en el momento de sustituir estas implementaciones por otras, se tendrá que reescribir toda la lógica de la aplicación que estas contenían. Sin embargo, el equipo no logró cumplir este objetivo en su totalidad, ya que hay detalles de la lógica que quedaron retenidos en la Web API, como la validación de permisos (si el usuario es administrador). Se reconoce este error, pero no se pudo agregar la validación del lado de los servicios por la cercanía a la fecha de entrega. Por otro lado se tiene la ventaja de eficiencia, ya que se ahorra la consulta de permisos del usuario cada vez que los controllers llaman a los servicios, de todas formas se puede optimizar. También es poco probable que se cambie la implementación a una aplicación que no utilice una WebAPI.

Interacción Log in - curso normal

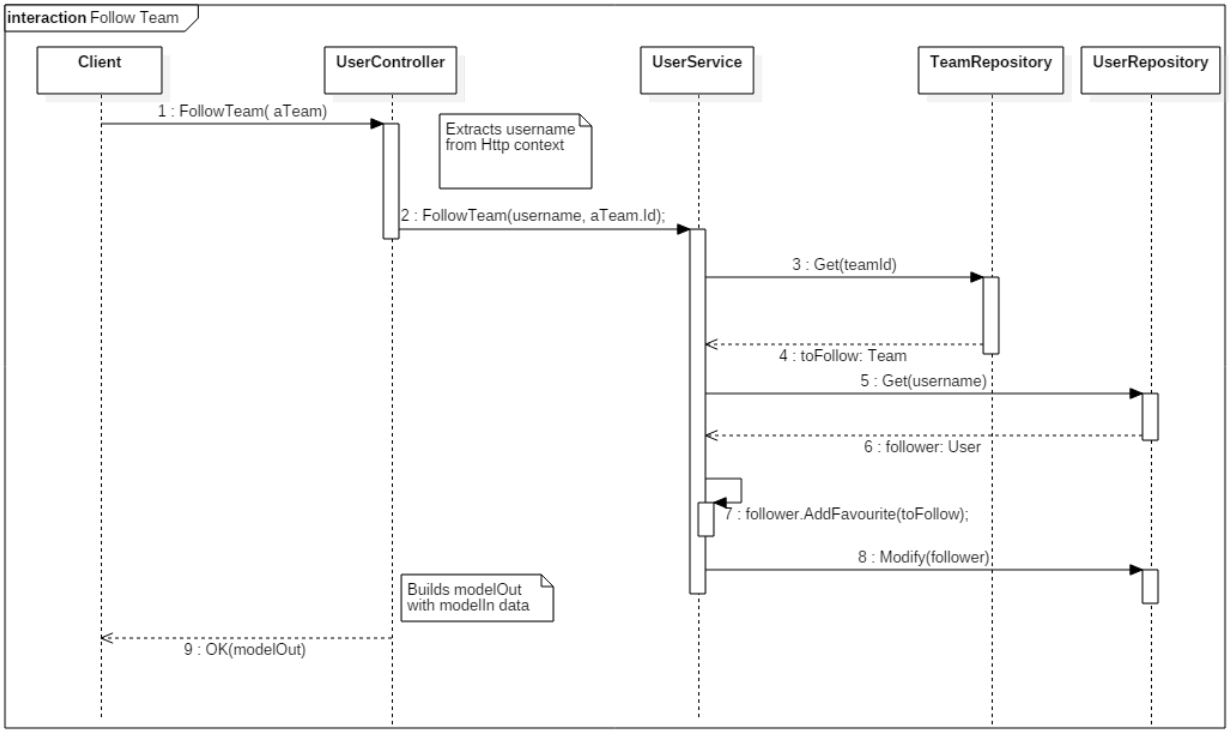


En este diagrama se puede observar la participación de uno de los mappers de Data.DomainMappers.Mappers. Su función es realizar la conversión de entities persistidas en base de datos a objetos del dominio de la aplicación. Más adelante se justificará la elección de este diseño.

Interacción Generar Fixture - curso normal



Interacción Seguir equipo - curso normal



Persistencia de datos

La versión actual del sistema persiste en base de datos, el framework utilizado para llevar a cabo la persistencia fue Entity Framework core, y el servicio e base de datos asociado a este fue Sql Server. Se utilizó el modo Code-First de este framework.
Para diseñar la persistencia de datos se consideraron dos posibles modelos:

El primero era persistir las entidades del dominio en la base de datos directamente, es decir, Entity Framework realiza un esquema de base de datos basado en las entidades del dominio y sus relaciones, y el mismo se encarga de mapearlos a registros en estas tablas al ser almacenados. Este modelo tiene la ventaja de la simplicidad, ya que no se tiene que el esquema se genera solo a partir del dominio y solo basta con que el contexto almacene los objetos para que sean persistidos. Por otro lado, el equipo cree que hay restricciones que deben tomar los elementos del dominio para ser persistidos correctamente por Entity Framework, como el tener todas las properties públicas, también puede suceder que se quieran usar DataAnnotations en los objetos del dominio, para indicar características sobre ciertos campos. Entonces se considera que en muchos casos se termina configurando el dominio en función de Entity Framework y las restricciones que este impone.

Lo mencionado anteriormente, llevó a que se optara por la segunda opción: tener estructuras de datos que representen a los objetos del dominio y que sean estos los que persistan en base de datos. El equipo considera que con este diseño se gana total independencia entre el dominio y la persistencia, no se tiene que diseñar el dominio pensando en su persistencia. En proyectos anteriores con Entity Framework, se aprendió que existen relaciones entre entidades del dominio que no son buenas para ser persistidas. Con este diseño, las relaciones entre las entidades a persistir pueden estar configuradas de una forma distinta para agilizar su almacenamiento y recuperación.
En este proyecto, en particular, se observó la ventaja de este diseño, ya que Entity Framework Core no soporta las relaciones “muchos a muchos” entre entidades, y en el dominio se tienen objetos que se relacionan de tal forma. Con este modelo, existen entidades a persistir que sirven de “intermediarias” en las relaciones n a n. De esta forma, se pudo persistir en base de datos sin ningún problema, y sobre todo, sin cambiar el diseño del dominio condicionados por la implementación de la persistencia. Sin embargo, este diseño tiene la desventaja de agregar complejidad al diseño, ya que al cambiar un objeto del dominio, puede impactar en su entidad asociada y en el código que se encarga de la traducción entre estas dos clases.

Diagrama de entidades de persistencia

Con el siguiente diagrama se ilustra lo mencionado en el punto anterior, la representación en base de datos es distinta de la del dominio.

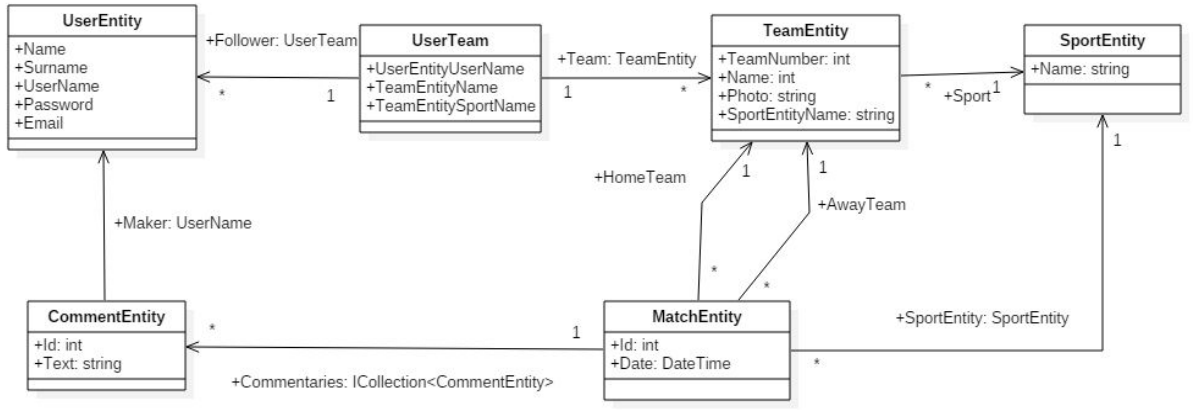
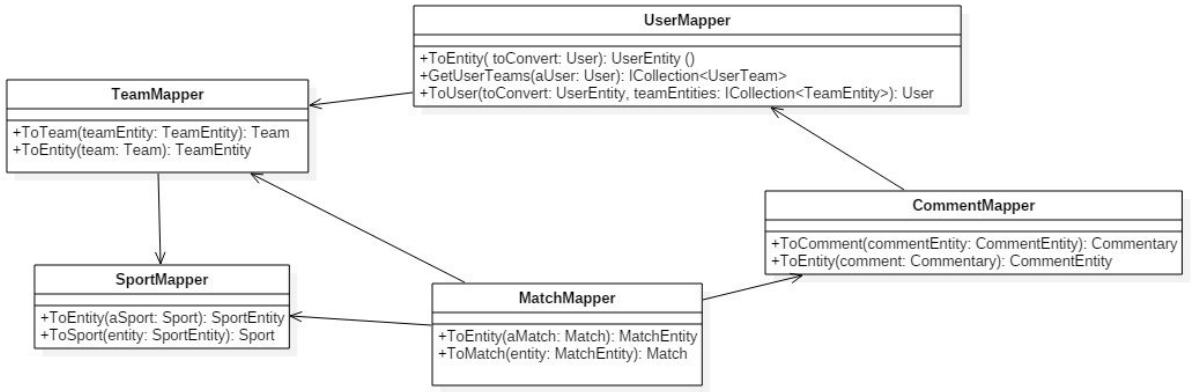


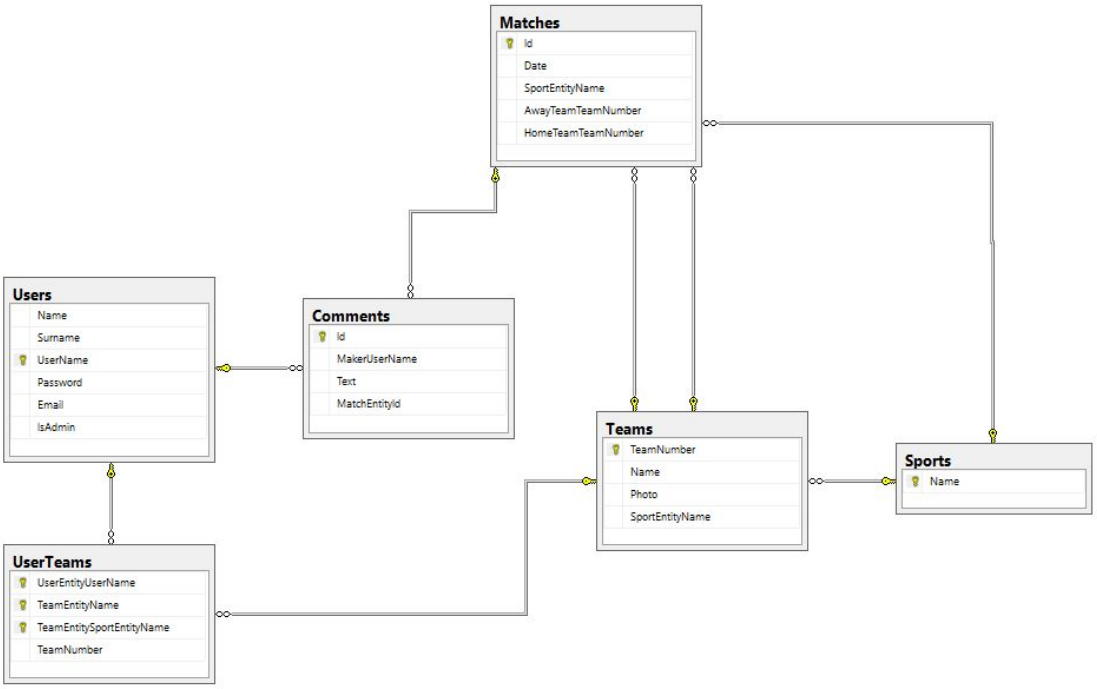
Diagrama de Mappers de entidades de persistencia

Los mappers tienen la función de la traducción de entidades del dominio a entidades de persistencia y viceversa. Los mappers se componen entre ellos para realizar la traducción de las entidades asociadas a la que este traduce, por ejemplo, el mapper de partidos, tiene un mapper de equipos y uno de comentarios, para realizar la traducción de sus partes recursivamente. Estas clases son utilizadas por las implementaciones de los repositorios que deben recibir y devolver entidades de dominio, pero deben guardar y recuperar entidades de persistencia.



Modelo de tablas de la base de datos

A continuación se muestra el modelo de tablas de base de datos que Entity Framework genera a partir de las entidades de persistencia (no las del dominio).

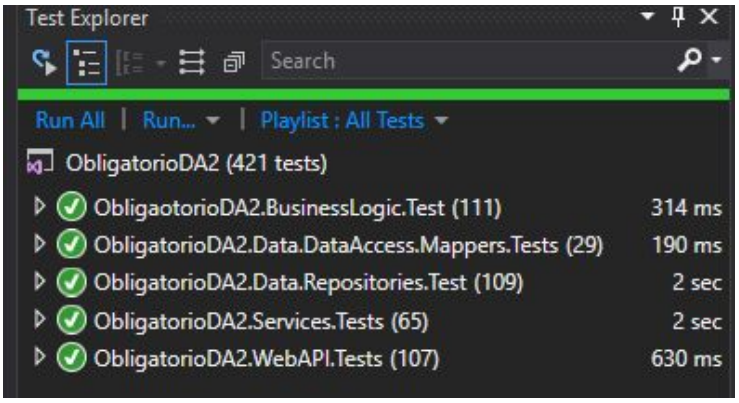


TDD y cobertura de pruebas unitarias

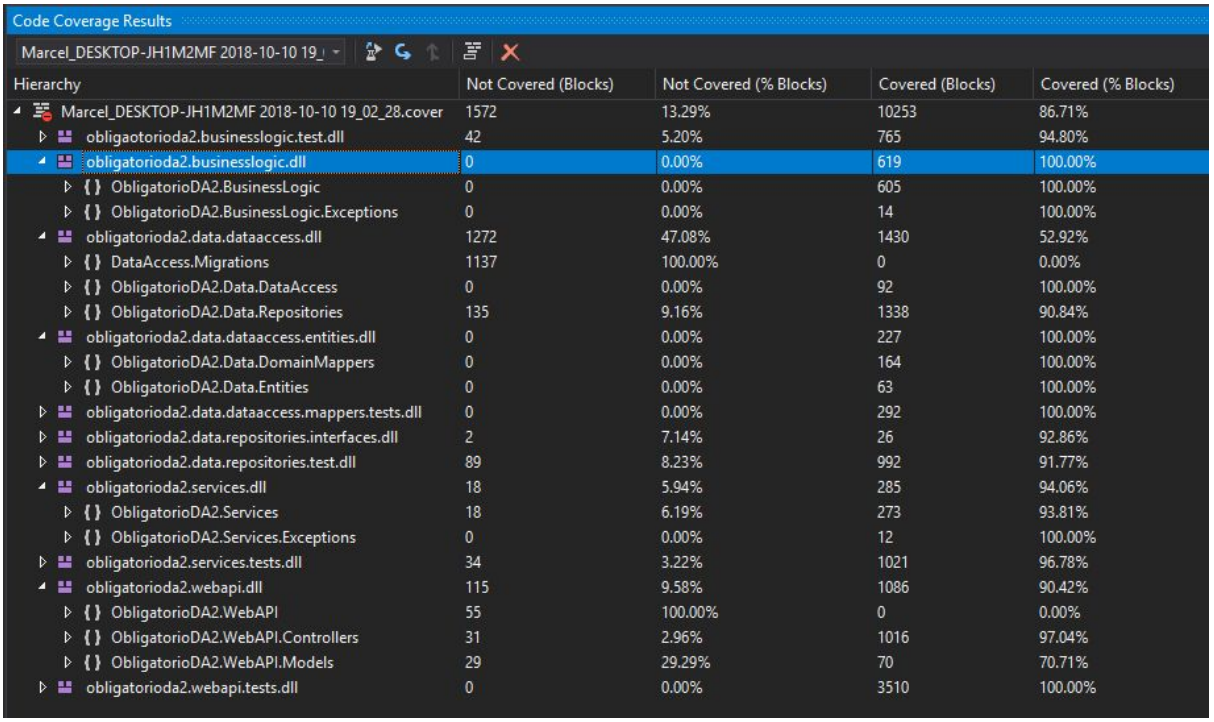
Existen 5 proyectos de test en la solucion:

- ObligaotorioDA2.BusinessLogic.Test
- ObligatorioDA2.Data.DataAccess.Mappers.Tests
- ObligatorioDA2.Data.Repositories.Test
- ObligatorioDA2.Services.Tests
- ObligatorioDA2.WebAPI.Tests

Hay un total de 421 pruebas unitarias, de las cuales todas pasan.



A continuación se muestra el análisis de la cobertura de las pruebas unitarias desglosadas por paquete.

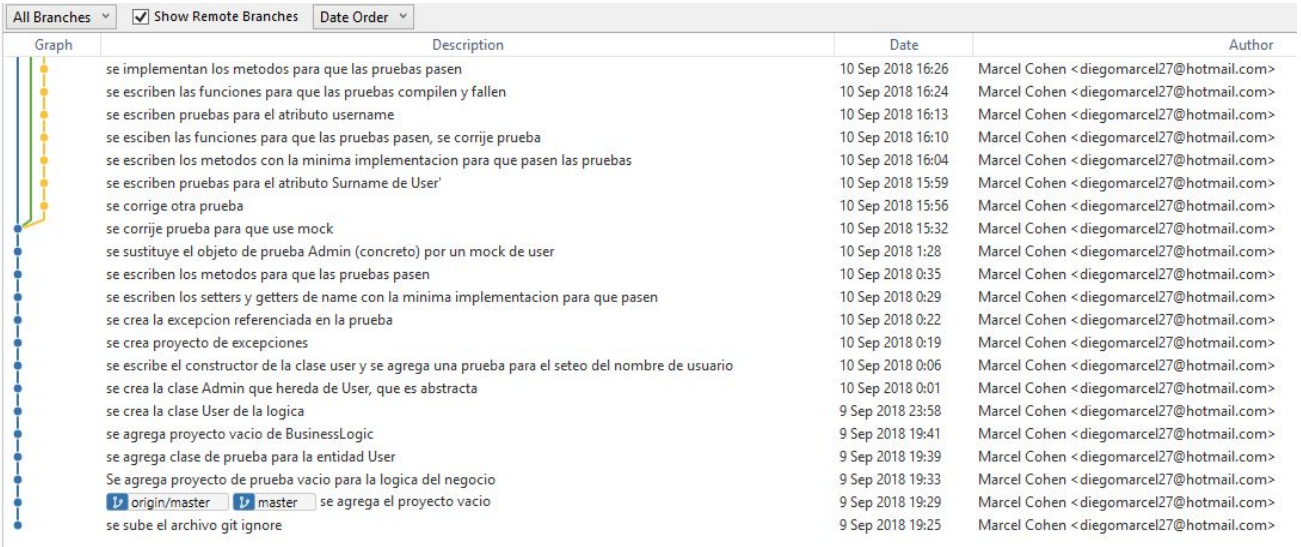


Los paquetes donde se realizo desarrollo guiado por pruebas son los paquetes donde hay lógica de negocio, lógica de aplicación y otras implementaciones como los controllers y los repositories.

Estos namespaces serían:

- ObligatorioDA2.BusinessLogic
- ObligatorioDA2.Data.Repositories
- ObligatorioDA2.Data.DomainMappers
- ObligatorioDA2.Services
- ObligatorioDA2.WebAPI.Controllers

El resto de los namespaces no se tuvieron en consideración por ser paquetes de test, por componerse de estructuras de datos sin comportamiento, ser paquetes de excepciones o interfaces.
BusinessLogic tiene cobertura del 100%, esto es consecuencia del desarrollo guiado por pruebas, cada funcionalidad escrita en ese paquete fue testeada previamente. También, fue el paquete que se mantuvo más estable a lo largo del desarrollo, el resto, tuvieron varias modificaciones, lo que hizo que su cobertura bajara. Otra razón por la cual no se logró la cobertura total en el paquete de repositorios, es que hay código que no pudo ser testeado, ya que solo se ejecuta cuando la base de datos es SQL Server (cuando se quiere ingresar una entidad con un id provisto, como en el PUT, hay que desactivar el identity insert de la base de datos, con un script sql), y como la base de datos es en memoria, no se ejecutan esas partes del código. Pero siendo las coberturas mayores o iguales a 92%, demuestra que se realizó TDD.
De todas formas, tener una cobertura casi total de pruebas unitarias, no es evidencia absoluta de que se hizo desarrollo guiado por pruebas a lo largo del proyecto, bien pudieron haber sido escritas al final, o después de escribir las funcionalidades.
A continuación se muestran commits del repositorio en etapas tempranas del proyecto, con fecha y autor , como evidencia de que efectivamente se realizó TDD, y que el código evolucionó a partir de pruebas.


All Branches | Show Remote Branches | Date Order

Graph	Description	Date	Author
	se implementan los metodos para que las pruebas pasen	10 Sep 2018 16:26	Marcel Cohen <diegomarcel27@hotmail.com>
	se escriben las funciones para que las pruebas compilen y fallen	10 Sep 2018 16:24	Marcel Cohen <diegomarcel27@hotmail.com>
	se escriben pruebas para el atributo username	10 Sep 2018 16:13	Marcel Cohen <diegomarcel27@hotmail.com>
	se esciben las funciones para que las pruebas pasen, se corrije prueba	10 Sep 2018 16:10	Marcel Cohen <diegomarcel27@hotmail.com>
	se escriben los metodos con la minima implementacion para que pasen las pruebas	10 Sep 2018 16:04	Marcel Cohen <diegomarcel27@hotmail.com>
	se escriben pruebas para el atributo Surname de User'	10 Sep 2018 15:59	Marcel Cohen <diegomarcel27@hotmail.com>
	se corrige otra prueba	10 Sep 2018 15:56	Marcel Cohen <diegomarcel27@hotmail.com>
	se corrije prueba para que use mock	10 Sep 2018 15:32	Marcel Cohen <diegomarcel27@hotmail.com>
	se sustituye el objeto de prueba Admin (concreto) por un mock de user	10 Sep 2018 1:28	Marcel Cohen <diegomarcel27@hotmail.com>
	se escriben los metodos para que las pruebas pasen	10 Sep 2018 0:35	Marcel Cohen <diegomarcel27@hotmail.com>
	se escriben los setters y getters de name con la minima implementacion para que pasen	10 Sep 2018 0:29	Marcel Cohen <diegomarcel27@hotmail.com>
	se crea la excepcion referenciada en la prueba	10 Sep 2018 0:22	Marcel Cohen <diegomarcel27@hotmail.com>
	se crea proyecto de excepciones	10 Sep 2018 0:19	Marcel Cohen <diegomarcel27@hotmail.com>
	se escribe el constructor de la clase user y se agrega una prueba para el seteo del nombre de usuario	10 Sep 2018 0:06	Marcel Cohen <diegomarcel27@hotmail.com>
	se crea la clase Admin que hereda de User, que es abstracta	10 Sep 2018 0:01	Marcel Cohen <diegomarcel27@hotmail.com>
	se crea la clase User de la logica	9 Sep 2018 23:58	Marcel Cohen <diegomarcel27@hotmail.com>
	se agrega proyecto vacio de BusinessLogic	9 Sep 2018 19:41	Marcel Cohen <diegomarcel27@hotmail.com>
	se agrega clase de prueba para la entidad User	9 Sep 2018 19:39	Marcel Cohen <diegomarcel27@hotmail.com>
	Se agrega proyecto de prueba vacio para la logica del negocio	9 Sep 2018 19:33	Marcel Cohen <diegomarcel27@hotmail.com>
	origin/master master se agrega el proyecto vacio	9 Sep 2018 19:29	Marcel Cohen <diegomarcel27@hotmail.com>
	se sube el archivo git ignore	9 Sep 2018 19:25	Marcel Cohen <diegomarcel27@hotmail.com>

Justificación de Clean Code

Para el desarrollo de este obligatorio se intentó utilizar las buenas prácticas recomendadas por el libro Clean Code, de Robert C. Martin.

• Nombres mnemotécnicos, representativos. Se puede ver esto en los métodos: CommentOnMatch, GetMatchComments, entre otros. En las clases y los paquetes también podemos ver esta decisión reflejada, ya que todos hacen referencia a lo que representan. Por otro lado, en las distintas variables también se ve reflejado, ya que se utilizan nombres descriptivos como “followedTeams”, “matchService”, entre otros.

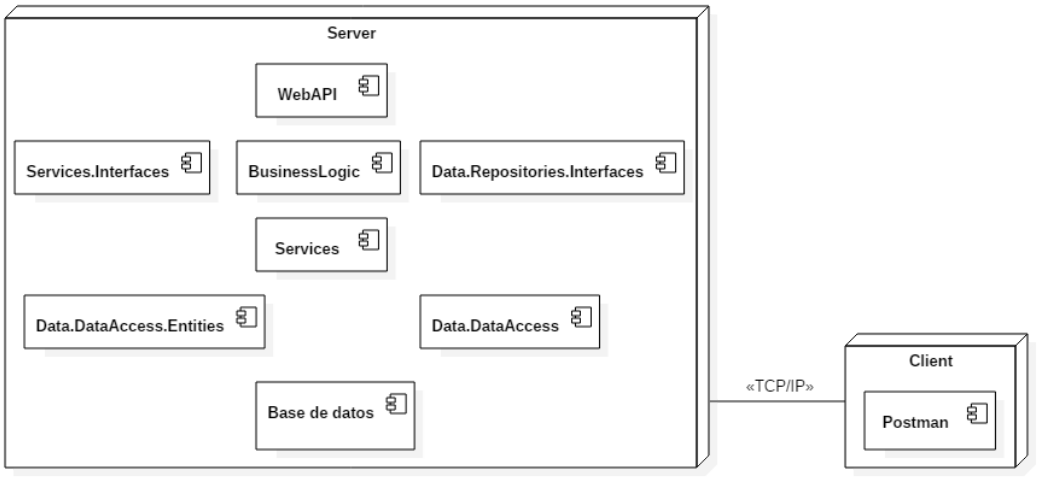
• La forma de escritura de los distintos elementos fue acorde a las buenas prácticas de clean code. Las variables con minúscula y los nombre de las clases, paquetes y métodos con mayúscula.
• Se usó la menor cantidad de comentarios posibles, solo cuando realmente fue necesario.
• Extracción de métodos. Ejemplo: las condiciones de los if son métodos que describen perfectamente la condición representada. Por lo tanto, los métodos se entienden por sí solos perfectamente.
• Las distintas funciones reciben una cantidad de parámetros razonable. Un ejemplo de esto es el constructor de User, al recibir más de 5 parámetros, se utilizó una estructura de datos que agrupa varios parámetros.
• Se crearon excepciones propias con mensajes propios para indicar los distintos errores ocurridos. También se utilizó una excepción padre de la cual heredan excepciones específicas. Se tomó esta decisión por dos razones, la primera es que cuando más específicas las excepciones lanzadas, más explicativo es el código,también se pueden configurar los mensajes de la excepción dentro de la creación de esta y así evitar y repetir el uso de literales cada vez que se lanza. La segunda es que en el futuro se pueden agregar nuevas excepciones que hereden de las genéricas, y no hay necesidad de cambiar el código de quien las atrapa, esto favorece el OCP (open-close principle)
• Se evitó la construcción de métodos extensos.
• Los distintos métodos cumplen que son procedimientos o funciones. Es decir, los métodos que realizan cambios sobre los datos son void, mientras que los que devuelven cierto dato no realizan cambios sobre los datos del sistema.
• Se procuró no utilizar literales en el código, esto llevó a que se crearan variables constantes, que se referencian donde se necesite el dato. Con esto se logra un código mas descriptivo, por que el nombre de la constante describe el dato, y más mantenible, ya que para sustituir el dato, basta con cambiar el valor de la constante, en lugar de actualizar todas sus ocurrencias. Un ejemplo de esto es la clase AuthenticationConstants, en el paquete ObligatorioDA2.WebApi.Controllers, que contiene los nombres de los roles y claims que se utilizan en la autenticación, y tienen formato string.

• Por último, se mantuvieron las clases pequeñas. Dejando las clases de test de lado, las clases más grandes, que son pocas, tienen aproximadamente 350-360 líneas de largo.

Anexo

Deploy de la aplicación

Diagrama de deploy



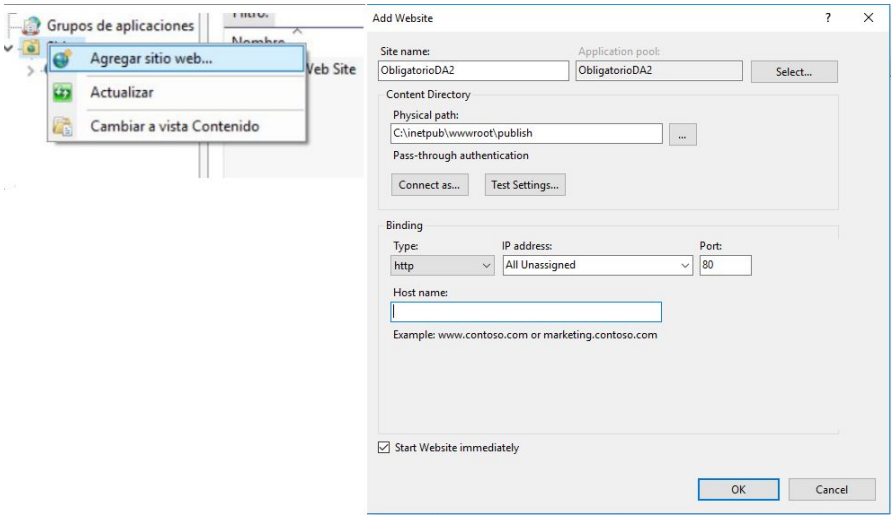
Manual de deploy

Maquina Servidor:

1 - Verificar que las computadoras a utilizar tienen instalado el IIS (Internet information services), y que las demás tecnologías que usa la aplicación estén instaladas, SQL Server 2014 y el Runtime de .NET Core.

2- En la computadora servidor, copiar la carpeta “publish” de la aplicación a la ruta “C:\inetpub\wwwroot”.

3- Ir a ISS y crear sitio nuevo, utilizar la configuración indicada a continuación, completando con la IP del servidor.



4- Detener el “Default Website” y activar el sitio agregado.
5- Quitar el CLR por defecto de ISS.
6- Restaurar la base de datos en la máquina servidor.
7-Otorgar permisos al usuario del sitio de IIS sobre la base.

a) Ir a la carpeta de Security y crear un nuevo inicio de sesión.

b) El Login Name corresponde a “IIS APPPOOL\ObligatorioDA2.

c)A este nuevo usuario debemos asignarle el ROL de deseado, en nuestro casos sysadmin.

Máquina Cliente:

1- Obtener la ip de la máquina servidor, utilizando el comando ipconfig de CMD.
2- En la máquina cliente, abrir el programa Postman (cliente HTTP) y cargar la colección de requests de prueba. Configurar las variables globales, se ingresa la IP de la máquina servidor y el puerto donde corre el sitio en esa máquina.
3- Se debe bajar el proxy de la máquina cliente para que pueda conectarse a la máquina servidor.

Manual de la API

Todas las operaciones requieren que haya un usuario logueado, excepto la operación de login.
Las operaciones que tienen el símbolo *, significa que requieren permisos de administrador para poder ser ejecutadas.

Sports

El recurso Sports representa al conjunto de deportes en el sistema.

Lista de deportes

Recurso: GET /api/sports
Descripción: Obtiene la lista de deportes ingresados en el sistema.

Agregar deporte *

Recurso: POST /api/sports
Descripción: Agrega un nuevo deporte al sistema con nombre Name.
Body:

Propiedad	*Descripción*
Name	Nombre único del deporte

Información de un deporte

Recurso: GET /api/sports/{name}
Descripción: Obtiene la información del deporte con nombre “name”.

Eliminar un deporte *

Recurso: DELETE /api/sports/{name}
Descripción: Elimina el deporte con nombre “name”. El borrado es en cascada, también se eliminan los equipos del deporte y toda la información asociada a ellos

Modificar deporte *

Recurso: PUT /api/sports/{name}.
Descripción: Modifica el deporte con nombre “name”. Si no existe, lo crea.
Body:

Propiedad	*Descripción*
Name	Nombre único del deporte

Lista de equipos de un deporte

Recurso: GET api/sports/{name}/teams
Descripción: Obtiene la lista de equipos que tiene el deporte con nombre “name”.

Fixture

El sistema permite generar fixtures para los equipos de un deporte a partir de una fecha dada. Los partidos se Existen dos algoritmos de armado de fixture disponibles.

Fixture todos contra todos *

Recurso: POST /api/sports/{name}/OneMatchFixture
Descripción: Crea una lista de encuentros con todos los equipos del deporte con nombre “name”, usando el algoritmo todos contra todos, y los agrega al sistema.
Body:

Propiedad	*Descripción*
Day	Día de inicio
Month	Mes de inicio
Year	Año de inicio

Fixture todos contra todos, ida y vuelta *

Recurso: POST/api/sports/{name}/HomeAwayFixture
Descripción: Crea una lista de encuentros con todos los equipos del deporte con nombre “name”, usando el algoritmo de todos contra todos con ida y vuelta, y los agrega al sistema.
Body:

Propiedad	*Descripción*
Day	Día de inicio
Month	Mes de inicio
Year	Año de inicio

Teams

Este recurso representa los equipos ingresados en el sistema.

Lista de equipos

Recurso: GET /api/teams/
Descripción: Retorno la lista de todos los equipos ingresados en el sistema.

Información de un equipo

Recurso: GET /api/teams/{id}
Descripción: Retorna la información del equipo con identificador “id”.

Recurso: GET /api/teams/{sportName}/{teamName}
Descripción: Retorna la información del equipo con nombre “teamName” y deporte “sportName”.

Agregar equipo *

Recurso: POST /api/teams/
Descripción: Agrega un nuevo equipo al sistema.
Body:

Propiedad	*Descripción*
Name	Nombre del equipo
SportName	Nombre del deporte
Photo (opcional)	Foto codificada en Base64

Editar equipo *

Recurso: PUT /api/teams/{teamId}
Descripción: Edita el equipo con identificador “teamId”. Si no existe, lo crea.
Body:

Propiedad	*Descripción*
Name	Nombre del equipo
SportName	Nombre del deporte
Photo (opcional)	Foto codificada en Base64

Eliminar un equipo *

Recurso: DELETE /api/teams/{id}
Descripción: Elimina el equipo con identificador “id”. El borrado es en cascada.

Recurso: DELETE /api/teams/{sportname}/{teamname}
Descripción: Elimina el equipo con nombre teamname

Users

Este recurso representa los usuarios ingresados en el sistema

Autenticación

Recurso: POST /api/authentication/
Descripción: Hace el login de un usuario si su usuario y clave son correctas.
Body:

Propiedad	*Descripción*
Username	Nombre de usuario
Password	Clave del usuario

Lista de usuarios

Recurso: GET /api/users/
Descripción: Retorna la lista de usuarios ingresados en el sistema.

Información de un usuario

Recurso: GET /api/users/{username}
Descripción: Obtiene la información del usuario con identificador “username”

Seguir equipo

Recurso: POST /api/users/followed-teams
Descripción: Agrega un equipo a la lista de equipos seguidos del usuario logueado
Body:

Propiedad	*Descripción*
Id	Id del equipo
Name	Nombre del equipo
SportName	Nombre del deporte

Dejar de seguir equipo

Recurso: DELETE api/users/followed-teams
Descripción: Remueve el equipo de los equipos seguidos por el usuarios logueado.
Body:

Propiedad	*Descripción*

Id	Id del equipo
Name	Nombre del equipo
SportName	Nombre del deporte

Agregar usuario *

Recurso: POST /api/users/
Descripción: Agrega el usuario a la lista de usuarios del sistema.
Body:

Propiedad	*Descripción*
Name	Nombre del usuario
Surname	Apellido del usuario
Username	Nombre de usuario (único)
Password	Clave del usuario
Email	Email del usuario
IsAdmin	Booleano que determina permisos del usuario

Obtener equipos seguidos

Recurso: GET /api/users/{username}/followed-teams
Descripción: Obtiene la lista de equipos seguidos del usuario logueado.

Matches

Los encuentros entre equipos se encuentran en el recurso Matches.

Lista de encuentros

Recurso: GET /api/matches/
Descripción: Obtiene la lista de encuentros de todos los equipos en el sistema.

Agregar encuentro *

Recurso: POST /api/matches/
Descripción: Agrega un encuentro entre dos equipos.
Body:

Propiedad	*Descripción*
SportName	Nombre del deporte

HomeTeamId	Id del equipo local
AwayTeamId	Id del equipo visitante
Date	Fecha del encuentro

Información de un encuentro

Recruso: GET /api/matches/{matchid}
Descripción: Obtiene la información del encuentro con identificador “matchid”.

Editar un encuentro *

Recurso: PUT /api/matches/{id}
Descripción: Edita la información del encuentro con identificador “id”.
Body:

Propiedad	*Descripción*
SportName	Nombre del deporte
HomeTeamId	Id del equipo local
AwayTeamId	Id del equipo visitante
Date	Fecha del encuentro

Eliminar encuentro *

Recurso: DELETE /api/matches/{id}
Descripción: Elimina el encuentro con identificador “id”.

Comentar encuentro

Recurso: POST /api/matches/{matchId}/comments
Descripción: Crea un nuevo comentario en el encuentro con identificador “matchid”, cuyo creador es el usuario logueado.
Body:

Propiedad	*Descripción*
Text	Texto del comentario

Encuentros de un deporte

Recurso: GET /api/matches/sport/{sportname}
Descripción: Obtiene la lista de todos los encuentros del deporte con nombre “sportname”

Encuentros de un equipo

Recurso: GET /api/matches/team/{teamid}
Descripción: Obtiene la lista de todos los encuentros que tiene el equipo con identificador “teamid”.

Comentarios de un encuentro

Recurso: GET /api/matches/{matchid}/comments
Descripción: Obtiene la lista de comentarios del encuentro con identificador “matchid”.

Comentarios de todos los encuentros

Recurso: GET /api/matches/comments
Descripción: Obtiene la lista de comentarios de todos los encuentros del sistema.

Obtener comentario

Recurso: GET /api/matches/comments/{id}
Descripción: Obtiene el comentario con identificador “id”.