Universidad ORT de Montevideo
Facultad de Ingeniería.

Obligatorio 2
Diseño de Aplicaciones 2

Facundo Arancet-210696
Marcel Cohen -212426

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

Noviembre, 2018

Índice

Declaraciones de auditoría

Nosotros, Marcel Cohen y Facundo Arancet, declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos Diseño de Aplicaciones 2.
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad.
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra.
- En la obra, hemos acusado recibo de las ayudas recibidas.
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros.
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

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, 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. El administrador también puede asignar y modificar los resultados de los encuentros.
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. Por último, el sistema lleva un log donde se registran todos los accesos a la aplicación y las creaciones de fixture, este registro solo es visible por el administrador.

Errores detectados:

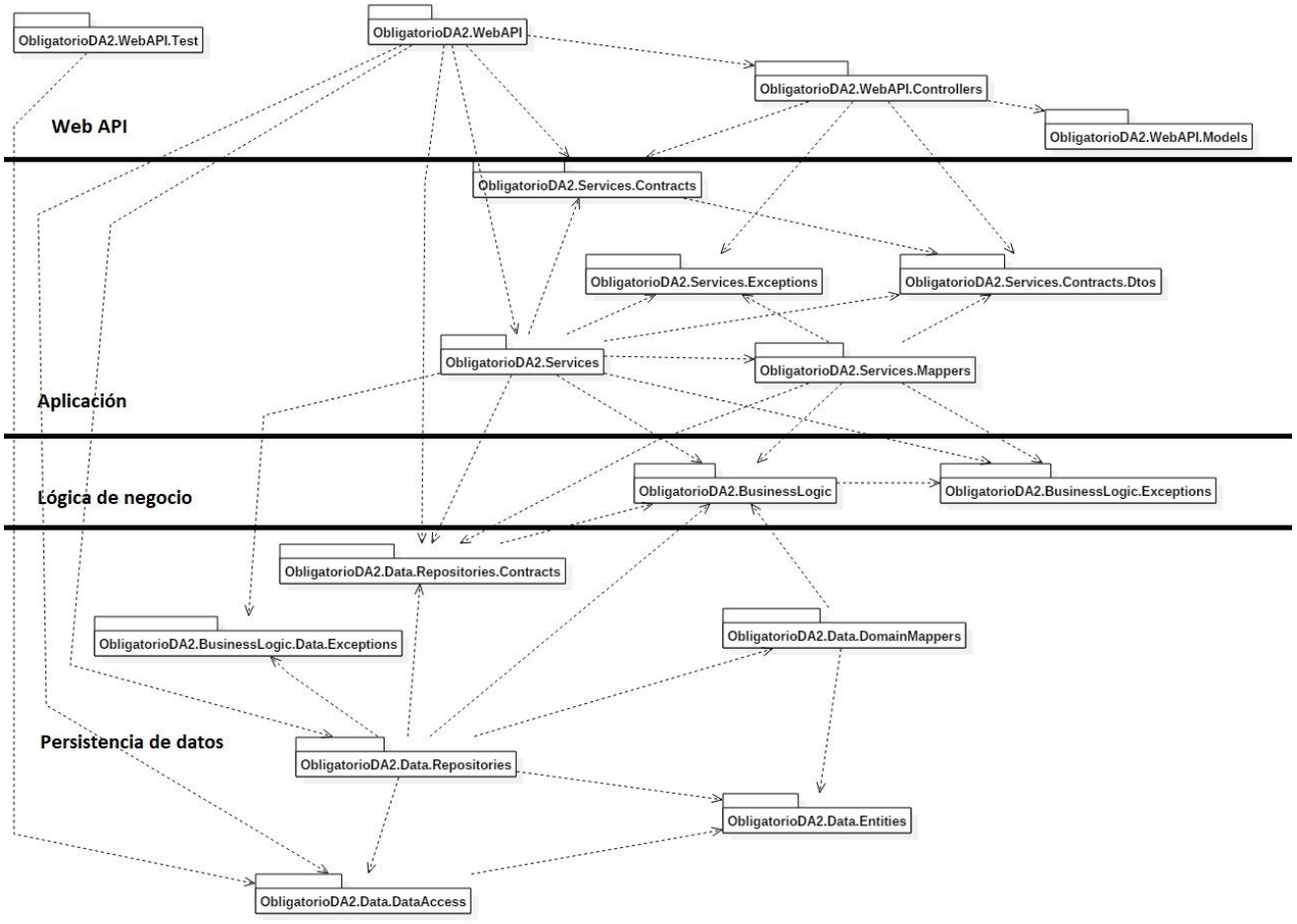
Uno de los errores detectados es que si bien pudimos manejar las excepciones de base de datos cuando esta es inaccesible, no se manejan excepciones cuando los datos de la base de datos son alterados y corrompidos.
Tampoco se logró que entity framework pudiera actualizar valores de claves primarias y 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.
No se pudo implementar el “log out” desde el backend. Es decir, al desloguearse el usuario, el token sigue por un período de tiempo después. Se intentó utilizar un caché

distribuido para guardar los tokens inválidos, pero por temas de tiempo y desconocimiento de la tecnología, no se llegó a implementar completamente y se tuvo que descartar.Por ultimo, por mas que el concepto “Match” pasó a ser un tipo concreto y encounter se refiere a los encuentros en general, el endpoint de encuentros sigue siendo matches.

Arquitectura y diseño

Uno de los objetivos principales propuestos por el equipo para esta entrega, además de cumplir con todo lo requerido, fue desacoplar los controladores de la lógica de la capa de negocio.
En el diagrama a continuación, se muestra una vista general de los paquetes de la nueva versión del sistema y sus dependencias, separados en capas. Su finalidad es demostrar que se mantuvo 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, por que se consideró más claro de expresar.



Se renombraron los namespaces “Interfaces” a “Contracts” con respecto a la versión anterior.

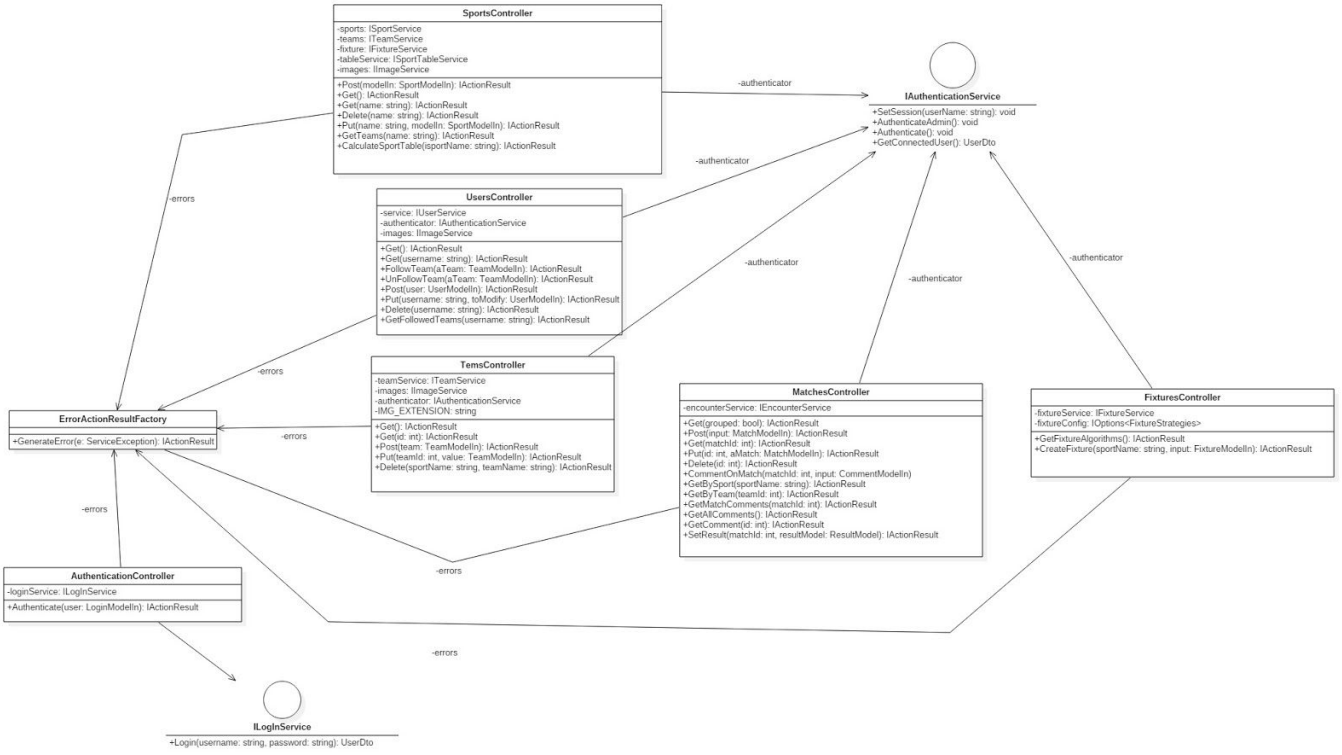
En esta versión, se agregaron tres nuevos namespaces:

- ObligatorioDA2.WebAPI.Test: Este paquete contiene un controller que reinicia la base de datos e inserta datos de prueba, se creó con el objetivo de poder automatizar las pruebas funcionales en Postman. De esto se hablará más adelante.
- ObligatorioDA2.Services.Contracts.Dtos: Contiene dtos que representan los objetos del dominio. Uno de los objetivos que el equipo de desarrollo planteó para esta versión del sistema fue desacoplar los controllers de las entidades del negocio, para que no se de el caso de "salteo de capas" visto en el curso. Con el uso de los dtos, la capa de WebAPI es independiente de la capa de lógica de negocio, y así se reduce el impacto de cambio.
- ObligatorioDA2.Services.Mappers: Contiene las clases que se encargan de transformar los dtos en objetos del dominio, se basan en el patrón Factory.

Diseño de los Controllers

Diagrama de clases del paquete de controllers

En el diagrama a continuación se muestra el nuevo diseño del namespace ObligatorioDA2.WebAPI.Controllers.

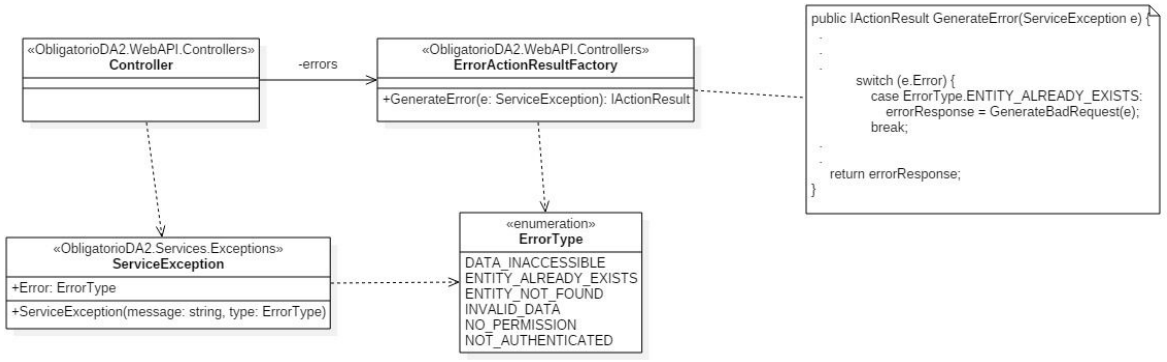


Al igual que la versión anterior del sistema, los controllers reciben las peticiones y validan que estas sean correctas. En la versión anterior, algunos controladores hacían llamadas a servicios y otros hacían llamadas directamente a los repositorios, cuando no existía un flujo de negocio en esos procedimientos, porque eran funciones sencillas. En

esta versión, se migró toda la lógica de aplicación que había quedado contenida en los controladores y repositorios a los servicios, de esta forma, todas las operaciones adquirieron más lógica, por lo que se crearon más servicios, y ahora todos los controladores se comunican con los servicios, y no se saltean a la capa de los repositorios como antes. En apartados posteriores se explicará en detalle la migración de la lógica realizada.

Manejo de mensajes de error

Todos los métodos de los controllers tienen tipo de retorno IActionResult, se tomó esta decisión porque el IActionResult permite manejar los Status Codes de Http con mayor flexibilidad. El equipo procuró ser lo más descriptivo posible con los mensajes de retorno para tener una buena usabilidad. En la entrega pasada, 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 había implicado que los controllers atraparan muchas excepciones para enviar los distintos mensajes de error, en lugar de una excepción más genérica. Esto no solo hacía el código más complejo, sino que le agregaba demasiada lógica a los controladores y también hacía que éstos dependieran de las excepciones de los repositorios, de los servicios y de la lógica del negocio, lo cual producía un gran acoplamiento, el cual se propuso reducir en esta versión.



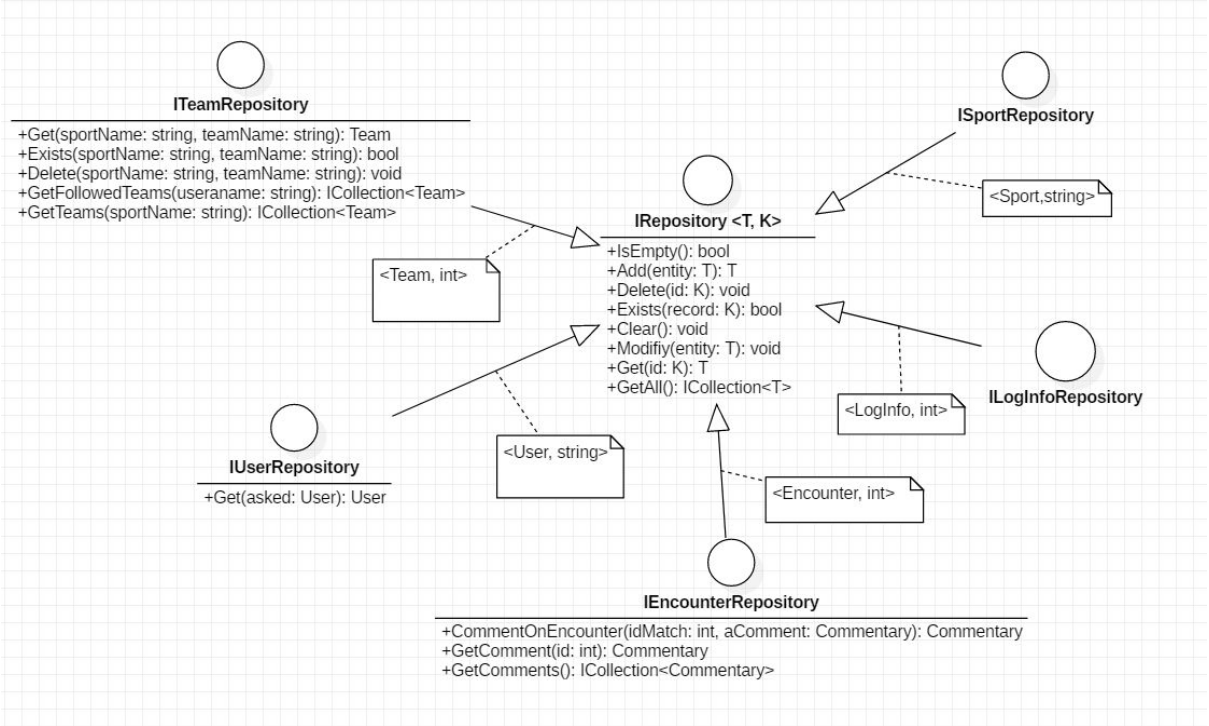
La decisión tomada para resolver este problema, fue la creación de una excepción genérica de todo servicio, ServiceException, que es la única excepción que arrojan los servicios, y la única que atrapan los controllers. Con el fin de poder distinguir entre los distintos tipos de errores del sistema, para poder mantener la descriptividad de los Status Codes en las respuestas http, se le agregó a las excepciones un tipo de error (ErrorType), además del mensaje de error. Luego, del lado de los controllers, se agregó a todas las clases una factory de IActionResults (ErrorActionResultFactory) para los

mensajes de error, que recibe la excepción y construye un IActionResult con el mensaje de la excepción y el Status Code correspondiente al tipo de error. La desventaja de esta decisión de diseño, es que la factory hace RTTI (Runtime Type Identification), al identificar el tipo de error de la excepción. Sin embargo, esta identificación de tipo, está encapsulada solo en esta fábrica, y adicionalmente, el hecho de que se identifica el tipo de error que contiene la excepción y no el tipo de la excepción en sí, hace que el RTTI sea más flexible, ya que se hace la identificación sobre el tipo de error dentro de la excepción (que puede darse en muchas partes del sistema), en lugar de hacerla sobre el tipo de la excepción en sí.

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.

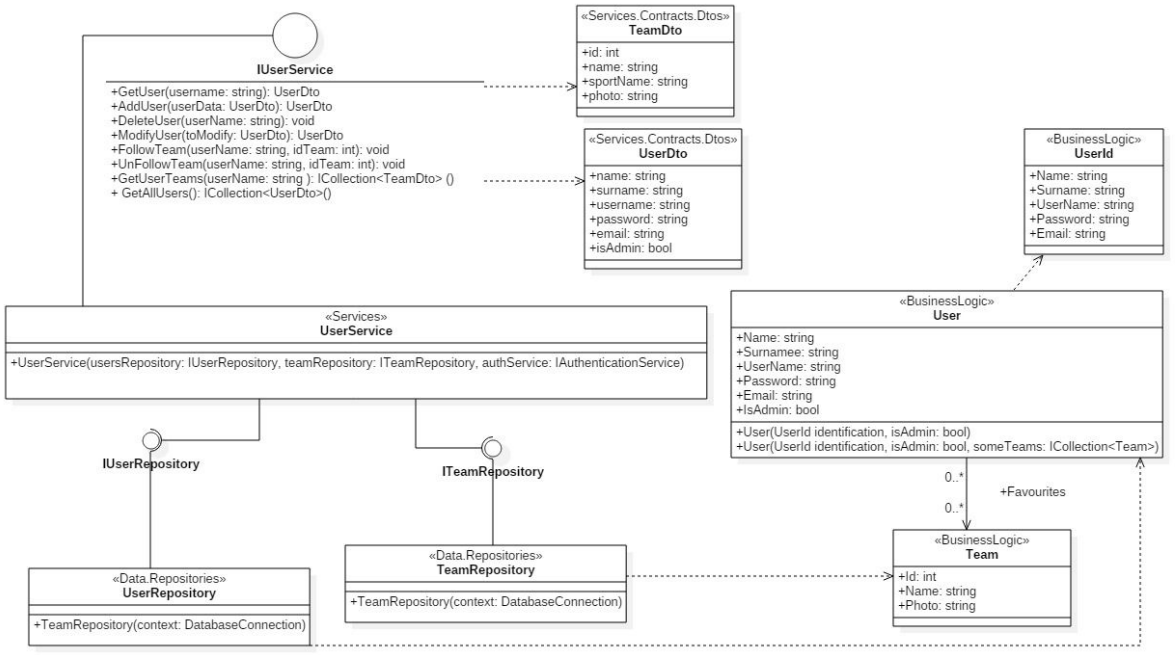


Las implementaciones de los repositorios son inyectados desde el startup de la WebAPI

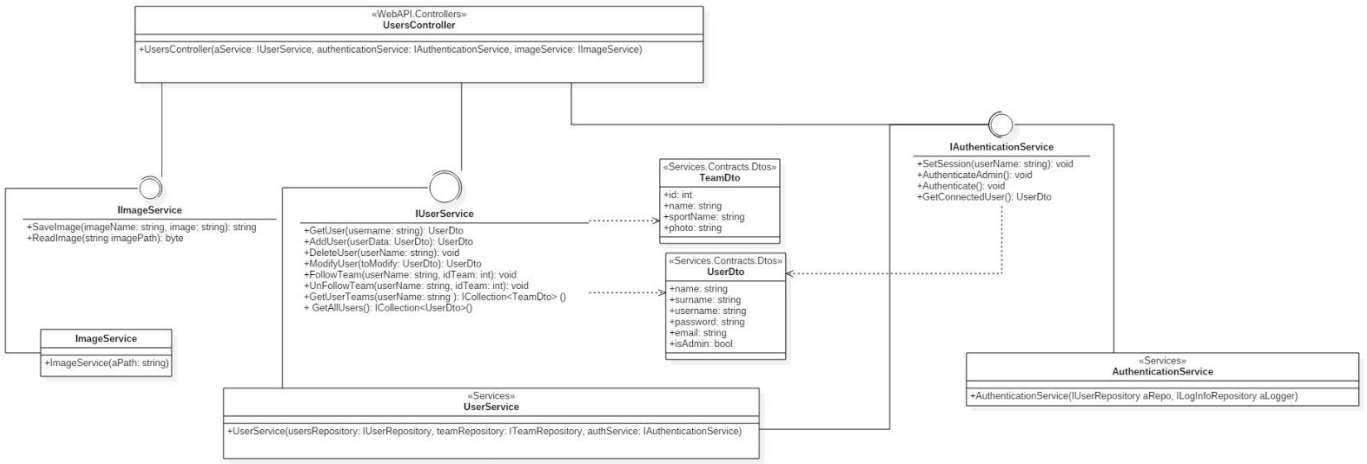
Usuarios y autenticación

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.
En el diagrama a continuación se muestra el diseño de las entidades usuario y equipo, junto con los repositorios y servicios que las administran.

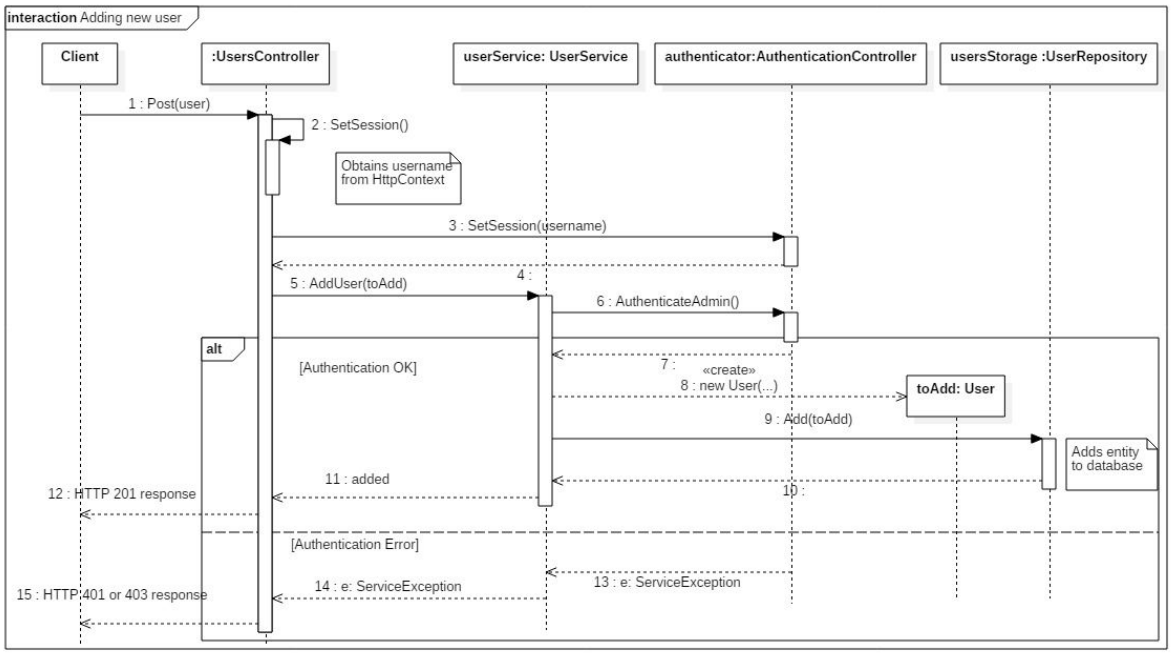


Como se puede ver, y se ha mencionado anteriormente, las interfaces de los servicios sólo manejan dtos de los objetos del dominio. Las implementaciones de los servicios son responsables de crear los objetos del dominio e interactuar con los repositorios.
Un usuario puede ser administrador o no serlo. En caso de serlo, puede ejecutar funcionalidades del sistema que los demás usuarios no (como la creación de nuevos encuentros).
En la versión anterior del sistema, la lógica de autenticación era responsabilidad de los controllers, se utilizaban tokens de JWT y los controladores verificaban que la token sea válida antes de llamar al método del controller. El problema con esto era que los controladores son un detalle de implementación, es decir, son un módulo de bajo nivel, que es muy susceptible a cambiar, y si en el futuro se quisiera realizar una implementación distinta sin los controllers (Ej. una aplicación WinForms), toda esta lógica tendría que ser reescrita. Esta nueva versión mantiene la autenticación del lado de los controllers, por que se considera que es sencilla y performante, y se agregó un sistema de autenticación del lado de los servicios, con el IAuthenticationService.
El diagrama a continuación muestra, con un ejemplo, cómo se relacionan los controladores y los otros servicios con el servicio de autenticación.



Cuando llegan las peticiones, los controllers extraen de la token el nombre de usuario y se lo proveen al servicio de autenticación, simula una sesión. Luego, cuando los controladores hacen llamadas a las funciones de los servicios, estos consultan al servicio de autenticación si el usuario actual, si es que hay uno ingresado, tiene los permisos para realizar esa operación. Si se produce un error de autenticación, se lanza una ServiceException, que el controller termina traduciendo a un error 401 o 403, con la factory mencionada. En general, los errores de autenticación son detectados por los controles, pero esta nueva lógica permitió cubrir un caso borde que no se había pensado en la entrega anterior: si un administrador se elimina a sí mismo, su token sigue siendo válido y el controller no lo detecta. Es el servicio quien identifica este error.

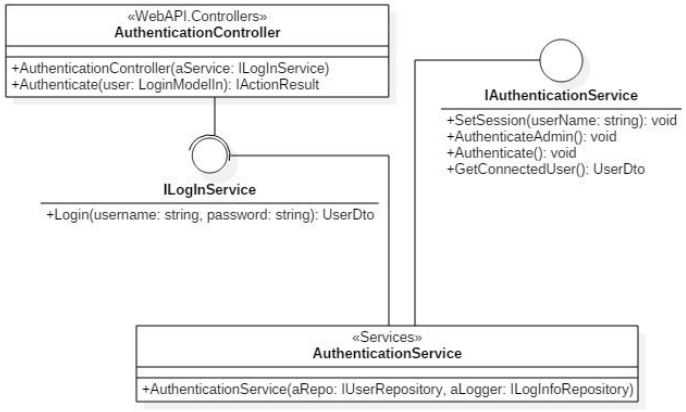
Para ilustrar lo recién explicado, en el siguiente diagrama de secuencia, se muestra el flujo del sistema para el registro de usuario, en este se puede observar la autenticación del lado del servicio. Para simplificar el diagrama, se considera que los datos de ingreso son correctos, que la persistencia de datos funciona y no existe un usuario con ese username, para poder enfocarse en la autenticación.



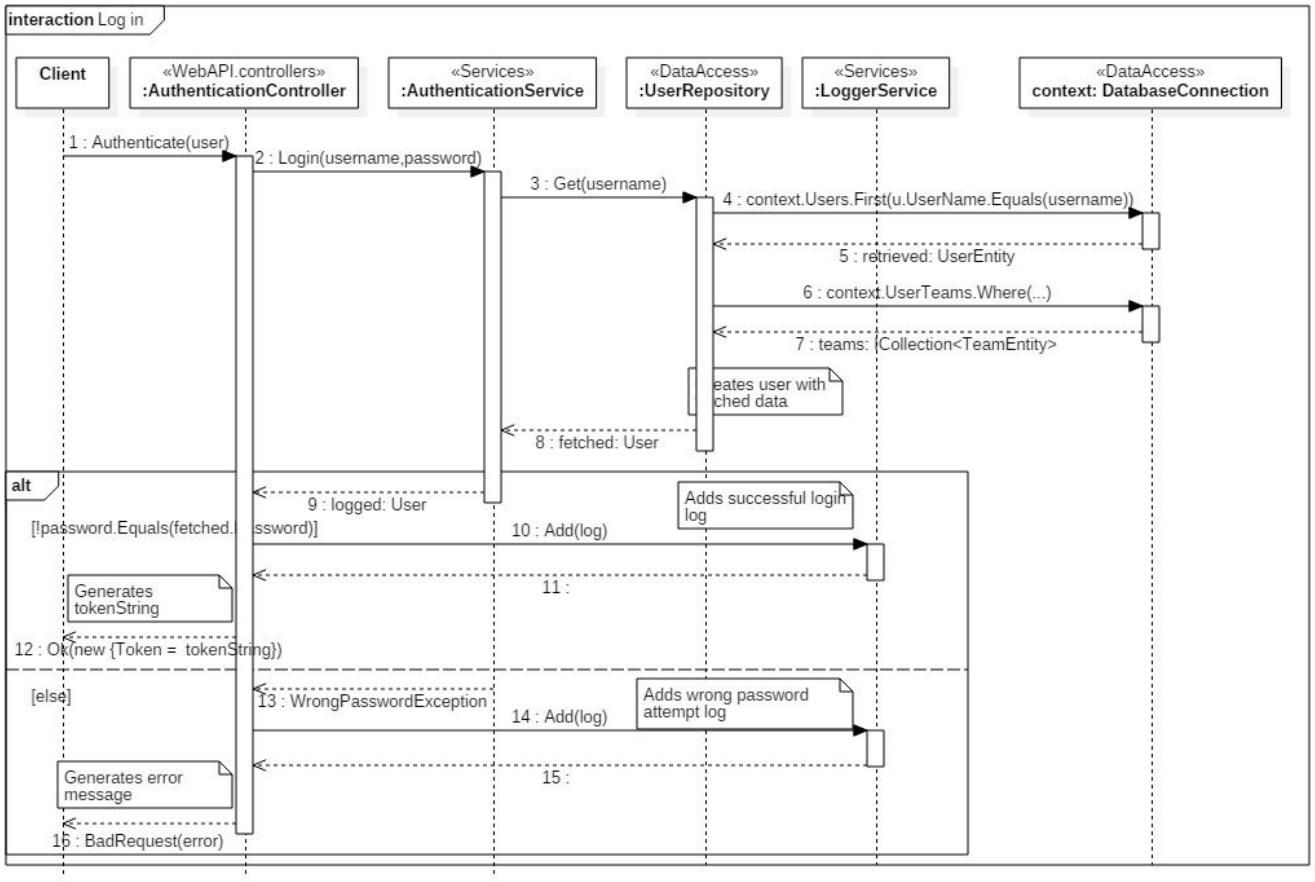
Como se puede ver en el diagrama de clases, hay un servicio de imágenes, se incorporó en esta versión porque se consideró que guardar archivos binarios en una base de datos es ineficiente, porque resulta en tuplas muy grandes, y esto aumenta considerablemente los costos de las operaciones y la materialización de resultados en la base de datos. Entonces, las imágenes se guardan en el file system, y en el appsettings.json se configura la ruta de las imágenes.

Login - ingreso a la aplicación

Como se mencionó anteriormente, un usuario puede ejecutar ciertas funciones del sistema si tiene permisos para ellos. Los mecanismos descritos en la parte anterior, hacen la autenticación consultando los datos del usuario conectado. En cada transacción, el usuario conectado se restablece extrayendo la información de la token que viene en el header de la consulta. Ahora se hablará del login, el proceso para obtener un token, que permitirá seguir realizando transacciones como el usuario logueado.

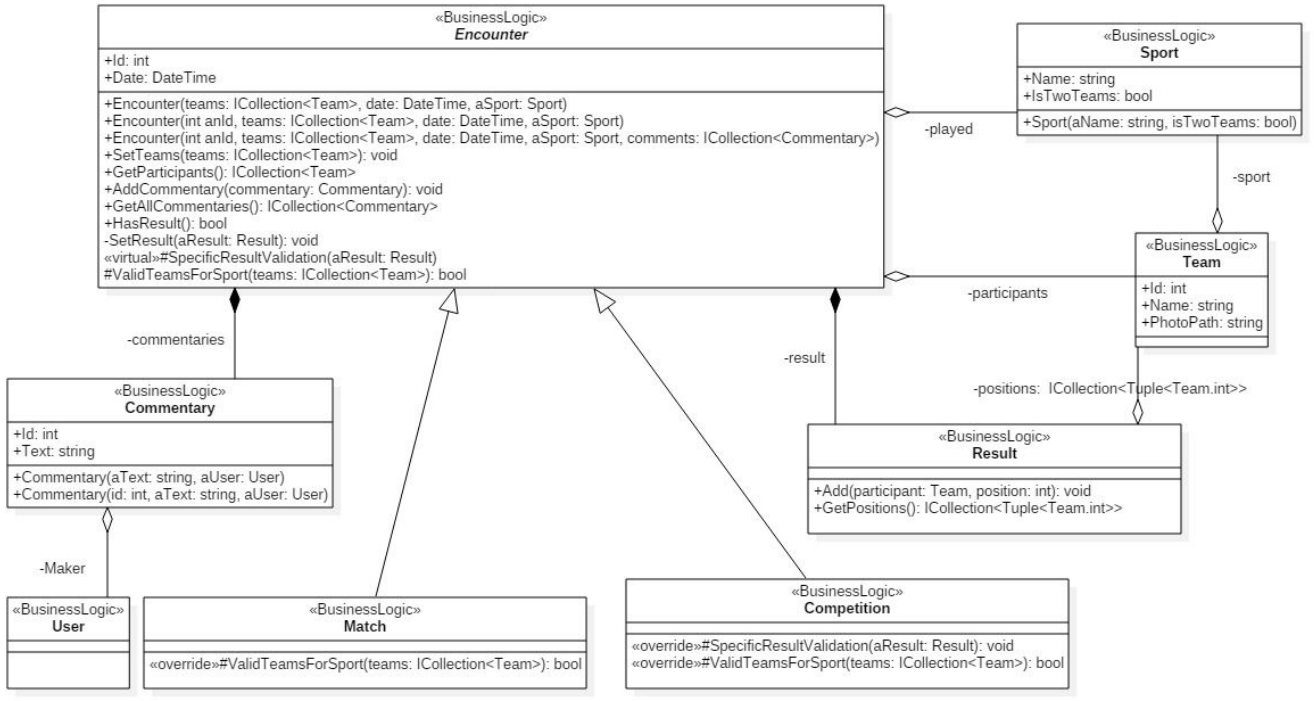


Un usuario se autentica usando su nombre de usuario y contraseña. Cuando un usuario se autentica, se le entrega un token generado por el sistema. El usuario debe enviar el token cada vez que desee utilizar funcionalidades del sistema, de este token, se extrae información que utiliza el servicio de autenticación para controlar los permisos. El token expira en 30 minutos.
AuthenticationService implementa todos los servicios relacionados con el login y la autenticación. Se tuvo en consideración que por seguridad, los controllers pueden acceder a algunas funciones de autenticación sólo cuando hay un usuario logueado en el sistema, como SetSession(). Entonces se aplicó el principio de segregación de interfaces y se crearon dos interfaces distintas: una para las funciones internas de autenticación (IAuthenticationService), y otra para el login, donde no se requiere que el usuario ya esté ingresado (ILogInService). De esta forma el controller de login solo ve la función de login, y los demás controllers solo pueden acceder a las funcionalidades internas de autenticación.
En el diagrama de secuencia a continuación, se muestra el flujo del sistema durante la funcionalidad de login. Para el curso descrito, se asume que el usuario con el que se quiere ingresar, existe en el sistema.



Diseño de encuentros

Anteriormente, solo existía un tipo de partido, el cual tenía solo dos equipos, local y visitante, y no se modelaba el resultado que este pudiera tener. En esta versión, se solicitó que no sólo hubieran deportes que se jugaran de a dos equipos, sino que también deportes que se puedan jugar con dos o más equipos. Adicionalmente, los encuentros debían poder tener un resultado. Al principio se consideró la opción de crear un nuevo tipo de encuentro distinto del que ya existía, y tratarlo como una entidad distinta, de esta forma no habría que cambiar lo que se había hecho la primera entrega. Esta idea se descartó, ya que esa decisión implicaría una nueva tabla de base de datos, repositorio y servicio entre otras cosas, es mucho código nuevo que complejizaría el diseño y dificultaría la mantención a futuro. Entonces se optó por crear una clase abstracta de encuentros, del cual heredarían los dos tipos de encuentros: Match, de dos equipos y Competition, de dos o más equipos.
En el siguiente diagrama se muestra el nuevo diseño de los encuentros.



Esta herencia permitió incluir validaciones comunes a todos los tipos de encuentros en la superclase, y que los tipos concretos solo tuvieran sus reglas del negocio específicas.

Restricción de cantidad de equipos por encuentro según el deporte.

Implementar esta nueva regla del negocio implicó realizar cambios en el dominio (el paquete BusinessLogic), que era el paquete más estable del sistema, del que dependían clases de todas las capas, incluso la más externa, los controladores. Por lo tanto, el cambio repercutió en todas las capas, por que todas las referencias a los encuentros eran a la clase concreta Match, y en esta versión surgió otro tipo concreto.

Se decidió asignar la responsabilidad de creación de encuentros a una Factory, que recibiría los datos del encuentro y construiría alguno de los dos tipos de encuentro basándose en el deporte. Dicha clase es EncounterFactory del namespace BusinessLogic, resultó muy util, ya que cuando los servicios van a agregar un nuevo encuentro, o los repositorios traen de base de datos un encuentro, necesitan construir el encuentro y necesitan identificar qué tipo de deporte es para saber que tipo de encuentro es, esa identificación del tipo de deporte es encapsulada en la Factory, y no se repite en el resto del sistema.
Este cambio en el diseño de los encuentros también afectó el esquema de base de datos, ya que el encuentro pasó de tener dos equipos, a tener dos o más equipos, entonces dejó de ser una relación 2 a N y pasó a ser una relación N a N. Esto se discutirá más adelante en la sección de persistencia de datos.

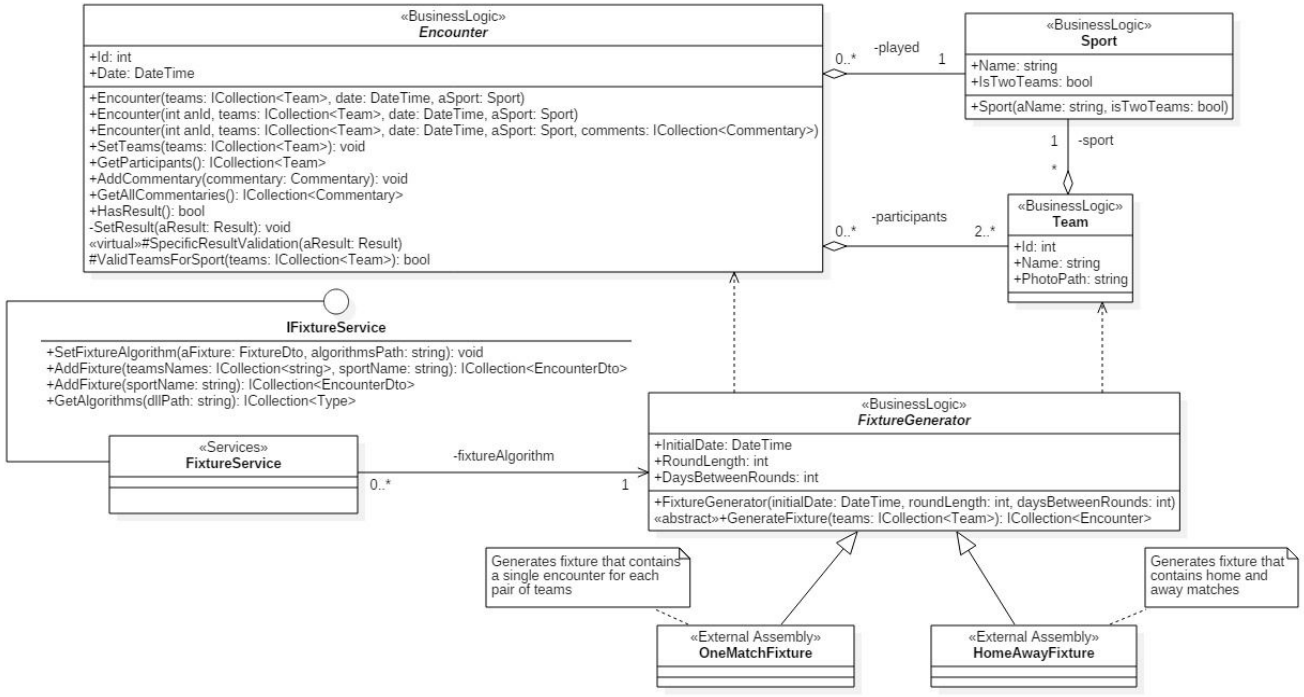
Incorporación de resultado en encuentros

Este nuevo requerimiento no implicó un gran cambio a nivel de la lógica de negocio como el cambio en los encuentros mencionado en la parte anterior. Más que un cambio, fue una adición de una nueva funcionalidad. Para incorporar esta funcionalidad se modeló el objeto resultado, que contiene un conjunto de tuplas equipo y posición. Para hacer más simple el diseño, se modeló el resultado de un empate en un partido de dos equipos como un resultado donde los dos equipos salieron primeros, y una victoria como el equipo ganador en primer puesto, y el perdedor en segundo puesto.
 La clase Result contiene reglas de negocio básicas de cualquier resultado (No tener posiciones repetidas, no tener el mismo equipo en posiciones distintas, etc.). La clase encuentro tiene un resultado. El método de asignación del resultado está en la clase abstracta (SetResult), y contiene las validaciones del resultado que todo encuentro debe hacer (Ej. Que todos los equipos del encuentro esten en una posición), también este método llama a un método virtual, SepecificResultValidation, que es reescrito por los encuentros que tengan validaciones específicas del resultado, como la competencia de dos o más equipos, que no puede tener equipos que compartan la misma posición. De esta forma, no se repite el código de las validaciones en la clases concretas. La desventaja es que se rompe el encapsulamiento.
Esta nueva funcionalidad también significó la adición de un nuevo método en el servicio de encuentros y un nuevo endpoint en el controlador de encuentros para permitir la asignación de un resultado a los encuentros. También, el modelOut de encuentro tuvo que incorporar el resultado. En cuanto a la representación del resultado en base de datos, la adición fue sencilla, se discutirá más adelante en la sección de persistencia de datos.

Lógica de creación de fixtures

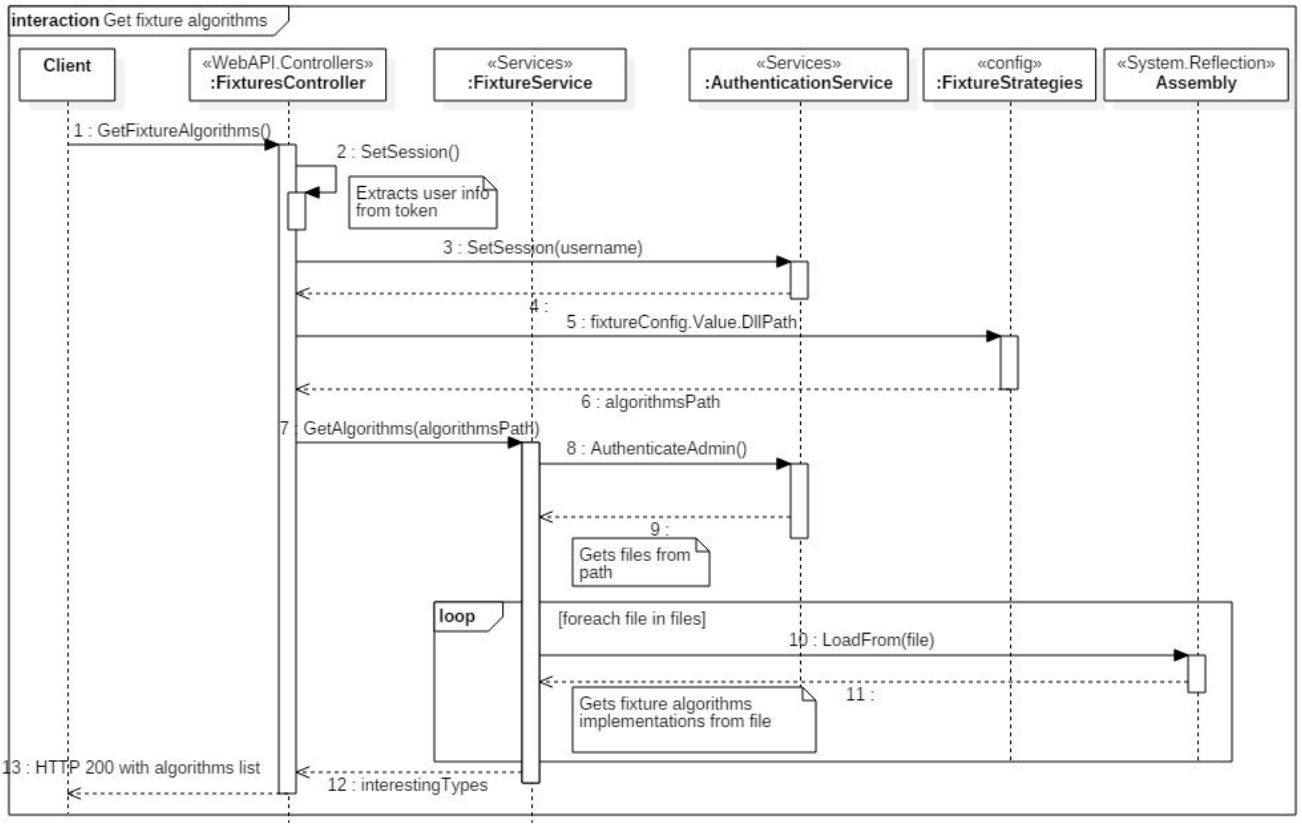
Para desarrollar este requerimiento funcional, se aplicó el patrón Strategy. La clase FixtureGenerator posee un método abstracto para generar un fixture, que sus clases

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. Uno de los requerimientos del sistema era que los algoritmos fueran extensibles en tiempo de ejecución, es decir, que se pudieran agregar nuevos algoritmos de fixture sin tener que recompilar ningún componente del sistema. Para lograr este requerimiento se utilizó reflection.



Obtención de algoritmos de fixture con reflection

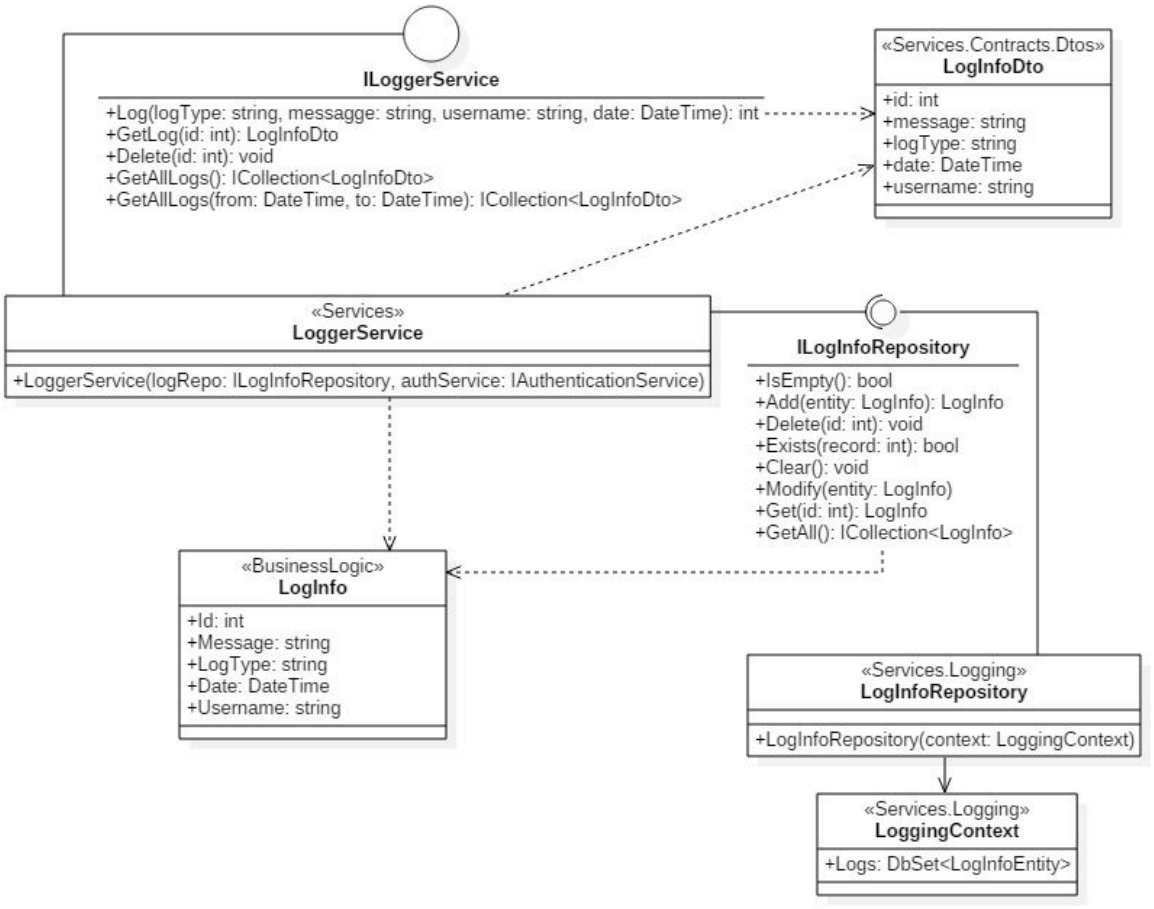
Para poder implementar la funcionalidad usando reflection, se agregó una ruta al archivo de configuración que determina la ruta de los .dll's de los algoritmos. El controlador de fixtures (FixturesController), expone dos endpoints: uno que devuelve los nombres de los algoritmos disponibles en la ruta configurada en appsettings.json, que sirve para mostrar los diferentes fixtures que se pueden ejecutar. El otro, ejecuta un algoritmo de fixture pasando como argumentos el nombre del fixture, el deporte, la fecha y otros parámetros opcionales.
En el diagrama a continuación, se muestra el flujo de la aplicación en el proceso de obtención de los algoritmos de fixture. El diagrama muestra el curso normal, en el cual el usuario está autenticado, tiene permisos de administrador, y la ruta ingresada en el archivo de configuración json (que es representado en el sistema como FixtureStrategies) es válida.



Un par de detalles a aclarar:
La ruta a los dll's provista en json tiene que estar en formato escaped. La ruta se puede formatear fácilmente en este sitio web:
https://www.freeformatter.com/json-escape.html#ad-output
En la versión release, el archivo de configuración ya viene con una ruta a una carpeta de la solución que contiene un dll con los dos algoritmos de la versión anterior del sistema. Sigue existiendo un componente de algoritmos de fixture en la solución, pero no es considerada parte de la solución, se dejó en la solución para poder realizar pruebas unitarias sobre ella, ya que son elaborados por el equipo.

Lógica de logging

Uno de los requerimientos de esta versión, fue incluir un log de las acciones de ingreso al sistema (log in) y la creación de fixtures de los usuarios. Se esperaba que el sistema tuviera independencia del sistema de log teniendo en cuenta que en un futuro puede surgir una nueva forma de mantener el log. Se optó por implementar el sistema en base de datos, utilizando EntityFrameworkCore. La justificación fue que el equipo ya conocía bien la tecnología, y que se consideró que una base de datos es una forma de almacenaje más eficiente y ordenada que otras formas de almacenaje como un archivo, por ejemplo. A continuación se muestra un diagrama de clases que representa el sistema de logging.



Adicionalmente, el sistema debía ser independiente del sistema de logging, por lo que se tomaron las siguientes decisiones:

Se creó un componente nuevo, ObligatorioDA2.Services.Logging, en el cual está contenida la implementación de la persistencia del log.

Se utilizaron interfaces para el service de logging y el repositorio de logs, como mecanismo para poder cambiar la implementación del almacenaje de logs sin afectar al resto de la aplicación. Estas interfaces se agregaron a los namespaces de contratos de servicios y repositorios, por que se aplicó el principio de reuso común, y estas interfaces son utilizadas junto con otras interfaces de servicios o repositorios.

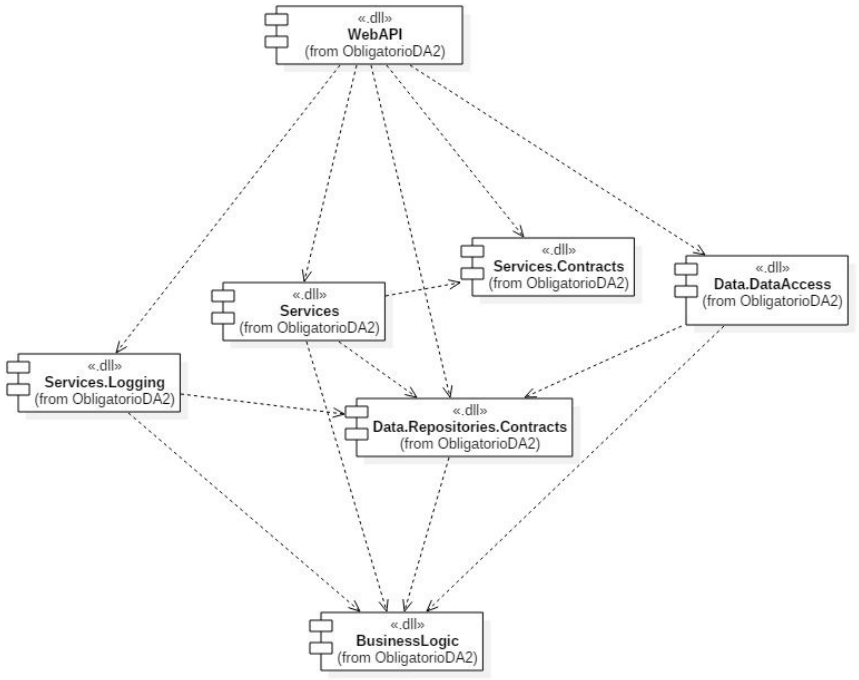
La implementación del servicio de logging, se colocó en el mismo namespace que las demás implementaciones de servicios, también por el mismo principio, y además por que dicha implementación se acopla a la interfaz del repositorio de logs, por lo que no conoce nada de la implementación de la mantención de los logs.

Se consideró que para lograr un cien por ciento de independencia del sistema de logs, estos no podían persistir en la misma base de datos que el sistema utiliza, por que cuando se decida cambiar la forma de mantener el log, se va a tener que cambiar el esquema de la base de datos del sistema para que no guarde más los logs, y el equipo no considera que con esto se haya logrado la independencia de los sistemas. Por lo tanto,

se utiliza otra base de datos separada para los logs, cuyo contexto está contenido dentro del assembly con la implementación del logging.

De la letra del obligatorio, el quipo interpretó que sea cual sea la implementación del sistema de logs, todos deben cumplir con cierta estructura que está indicada en esta misma (fecha, acción, usuario, etc). Se consideró que estas son reglas del negocio que deben estar incluidas en la lógica, por lo que se modeló el objeto log como una entidad del dominio. De esta forma, sea cual sea la implementación futura de los logs, tendrán que seguir las reglas del negocio y tener la estructura mencionada en la letra.

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.
Como se puede observar, el assembly WebAPI no depende de BusinessLogic, cosa que si hacía en la versión pasada, esta es una ventaja respecto de la versión anterior, los controladores están desacoplados de la capa de lógica de negocio.

El assembly WebAPI depende tanto de las interfaces de los servicios y repositorios como de sus implementaciones, esto es porque ahí está contenida la clase StartUp, que es la clase encargada de la inyección de dependencias. Gracias a esta inyección, las implementaciones de los servicios no dependen de las implementaciones de los repositorios, sino de las interfaces.
Adicionalmente, en Services.Logging se encuentra la implementación del sistema de logging, de forma que para cambiar la implementación de este sistema, solo basta cambiar el dll de Services.Logging, y recompilar el assembly WebAPI, que es quien conoce esta implementación.

Análisis de métricas de diseño

Para realizar el análisis basado en métricas, se utilizó la herramienta NDepend, y se consideraron tres métricas: Inestabilidad, Abstracción y Cohesión relacional.

Nombre de assembly	Cohesión relacional	Inestabilidad	Abstracción	Distancia
ObligatorioDA2.BusinessLogic	1.96	0.15	0.13	0.51
ObligatorioDA2.Data.Repositories.Contracts	1.38	0.23	0.33	0.31
ObligatorioDA2.Services.Contracts	1.11	0.11	0.37	0.37
ObligatorioDA2.Services	0.88	0.99	0.06	0.03
ObligatorioDA2.Data.DataAccess	2.28	0.98	0	0.01
ObligatorioDA2.Services.Logging	1	0.96	0	0.03
ObligatorioDA2.WebAPI	1.37	1	0.02	0.02

Cohesión relacional

El equipo considera los assemblies de la solución son poco cohesivos, ya que el rango de cohesión relacional buena esta entre 1.5 y 4.0, y la mayoría de los assemblies están por debajo de este valor.
Esto es en parte por que se utilizaron pocos assemblies en la solución, con el objetivo de hacer la solución lo más sencilla posible. Para lograr esto se agruparon namespaces por assembly, basándose en el principio de reuso común, es decir, que las clases que se usaran juntas estuvieran en el mismo assembly.
Esto llevó a que tengamos los dos assemblies “contracts”, que contienen las interfaces de los repositorios/servicios, los dtos que estos deben manipular, en el caso de los servicios, y las excepciones que estos deben arrojar. Se creyó conveniente, porque estos assemblies, con sus interfaces, excepciones y dtos, definen un “contrato” de como se

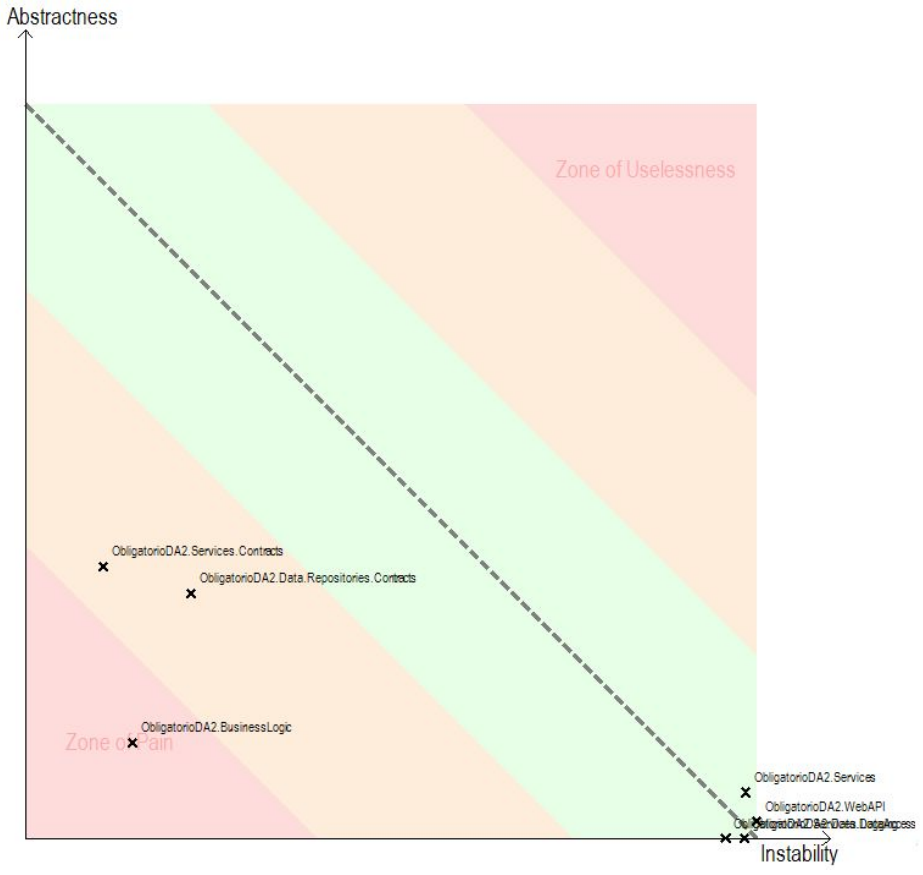
debe comportar las implementaciones. El problema con este tipo de assemblies es que sus componentes no se relacionan entre sí, y por lo tanto tienen baja cohesión.
Por otro lado, hubieron assemblies que se organizaron bajo el principio de clausura común, como BusinessLogic y DataAccess, donde se agruparon las entidades de esa forma porque están fuertemente relacionadas y cambian por un mismo motivo, un cambio en la reglas del negocio o en la forma de persistir los datos. Esos assemblies son los que tienen un buen valor de cohesión relacional.

Principio de dependencias estables

Se considera que este principio se cumplió correctamente, ya que como se puede ver en la tabla de esta sección, y el diagrama de componentes de la sección anterior, las dependencias son en el sentido de la estabilidad. En algunos casos se cumple apenas el principio, como WebAPI que tiene 1 de inestabilidad, y depende de services que tiene 0.99, pero se cumple para todos los casos.

Principio de abstracciones estables

Este principio establece que los paquetes más estables, deben tender a ser abstractos. Para analizar este principio se tomó la medida de distancia de abstracción-estabilidad. A continuación se muestra en una gráfica, los valores de esta medida para cada assembly.



Cuando un paquete se ubica en la “Zone of Pain”, significa que es muy estable y poco abstracto Este es el caso de el assembly de BusinessLogic, que contiene la lógica de negocio, y por lo tanto muchos componentes dependen de él. Se intentó aumentar el nivel de abstracción de este componente, poniendo las implementaciones de los encuentros en un componente separado y dejando la clase abstracta de encuentro en BusinessLogic, con el fin de subir el nivel de abstracción, pero no hizo un cambio significativo. Luego de realizar un análisis, se observó que en el componente existe una gran cantidad de excepciones particulares para distintos errores del sistem. Las excepciones son clases concretas, y el nivel de abstracción se calcula como cantidad de clases abstractas e interfaces dividido el total de clases en el componente, esto significa que ese gran número de excepciones particulares baja el nivel de abstracción, y agranda la diferencia de estabilidad-abstracción. Esto pudo haber sido evitado habiendo usado pocas excepciones más genéricas, la desventaja de esto es que hace el código menos claro, pero mejoraría las métricas. Este análisis sirve como reflexión para próximos proyectos.

Principio de dependencias acíclicas

Se observa que este principio se cumple, tanto a nivel de componentes, como se puede ver en el diagrama de componentes de la sección anterior; como a nivel de paquetes, como se puede ver en el diagrama de paquetes al principio de la sección Arquitectura y diseño.

Persistencia de datos

La nueva versión del sistema también 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 la modalidad Code-First de este framework.
Para diseñar la persistencia de datos se consideraron dos posibles modelos:

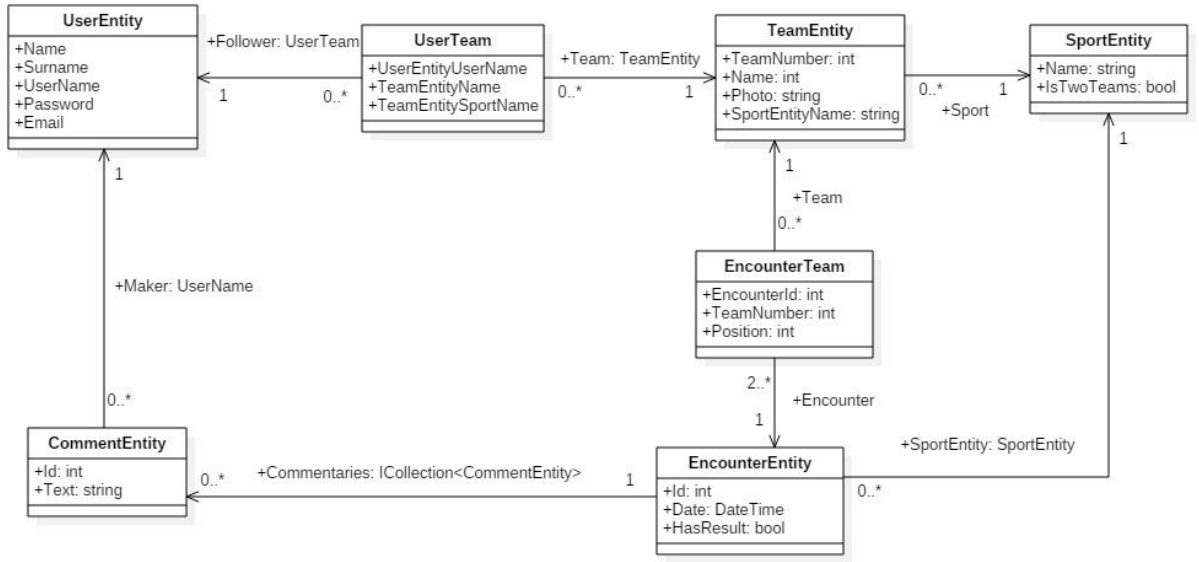
El primero era persistir las entidades del dominio en la base de datos directamente, es decir, Entity Framework realiza un esquema de base de datos basado en las entidades del dominio y sus relaciones, y el mismo se encarga de mapearlos a registros en estas tablas al ser almacenados. Este modelo tiene 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.

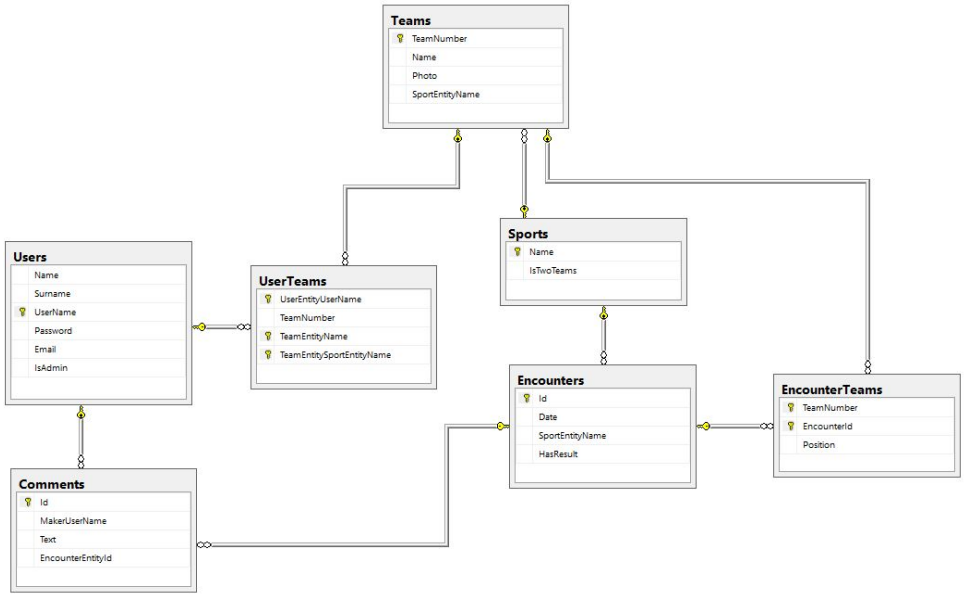


Cómo fue mencionado anteriormente, en la sección de diseño de encuentros, el cambio realizado en los encuentros por el nuevo tipo de encuentro incorporado impactó en todas las capas de la aplicación, incluyendo la capa de persistencia. Al haber dos o mas equipos por encuentro, en lugar de solo dos, ya no se puede representar como dos relaciones “uno a muchos”, se requirió una representación “muchos a muchos”. Como EF Core no soporta este tipo de relaciones, se tuvo que crear otra entidad intermedia, como UserTeam, que representa la relación seguidores-equipos, que también es n a n. EncounterTeam es la entidad intermedia para esta nueva relación, representa la participación de un equipo en un encuentro.

También, en esa sección se discutió sobre la incorporación de el resultado a los partidos. Con esta incorporación, surgió la necesidad de persistir el resultado en la base de datos. Esta incorporación fue fácil, ya que se aprovechó la nueva entidad intermedia EncounterTeam, a esta se le agrego el resultado position, que es la posición que tuvo el equipo en el encuentro, si hay un resultado. Cuando no hay resultado, este atributo tiene el valor por defecto, que es 0. A la entidad del encuentro se le agrego un atributo booleano que indica si tiene resultado o no, esto indica cuándo los atributos de posiciones tienen valor.

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).



Frontend

Descripción

La aplicación fue desarrollada utilizando Angular 6. Se utilizó el patrón modelo vista controlador. Existen servicios que actúan de modelo para la aplicación y se encargan de obtener la información por medio de la web api. Luego están los componentes y templates, que son controladores y vistas. Cada componente tiene asociado un template con el cual intercambia información. Por último, tenemos las clases, las cuales se encargan de modelar los objetos que utilizan los servicios y componentes. Los servicios utilizan las clases para comunicarse con la web api, mientras que los componentes las utilizan para mostrar información o recibirla desde los templates.
Se utilizó el paquete material design para el desarrollo de componentes.
La aplicación no tiene dependencias a servicios webs hosteados en otros servidores.

Componentes

Se crearon los siguientes agrupaciones de componentes, siguiendo cada uno el Style Guide sugerido por Angular:

- home: Modela la pantalla principal, desde donde se muestran todos los demás componentes. En la misma, hay un side-nav, que permite la navegación del usuario por la aplicación.
- calendar: Modela el calendario requerido por el cliente.
- comments: Muestra los encuentros de los equipos que sigue un usuario y permite agregar comentarios a los mismo.
- confirmation-dialog: Muestra un diálogo genérico, al cual se le pasa un mensaje y espera una respuesta positiva o negativa.
- encounters: Modela el CRUD de encuentros que utiliza el usuario administrador.
- login: Modela el login de los usuarios (administrador y seguidor).
- logs: Modela la tabla de los que el administrador puede consultar.
- not-found: Componente utilizado cuando el usuario intenta acceder a una ruta no definida.
- sport-table: Modela la tabla de posiciones de los equipos de un deporte que puede consultar un usuario seguidor.
- sports: Modela el CRUD de deportes que utiliza el usuario administrador.
- teams: Modela el CRUD de equipos que utiliza el usuario administrador.
- teams-follower: Modela la tabla de equipos que puede consultar un usuario seguidor. Desde la misma se permite seguir y dejar de seguir equipos para el usuario loggueado.
- users: Modela el CRUD de equipos que utiliza el usuario administrador.
- welcome: Pantalla de inicio para todos los usuarios.

Servicios

Se creó un servicio para cada controlador que existe en el backend. Estos servicios devuelven errores utilizando una clase definida en la aplicación. Gracias a esto, el manejo de errores en toda la aplicación es uniforme.
Los servicios realizan las consultas al backend de forma asincrónica, por lo que devuelven un observable de la respuesta. Estos observables son utilizados por los controladores para mostrar información.

Ingreso de información

Para el ingreso de información, se utilizaron los formularios de material. Los mismos, ofrecen una implementación del manejo de validaciones la cual fue utilizada. Se logró crear validaciones en tiempo real (por ejemplo, el checkeo de nombre de usuario no repetido en la creación de nuevos usuarios), esto favorece la usabilidad del sistema. Además, se muestra un mensaje de error descriptivo y no invasivo en el campo que generó el error.

Surgió el problema de que el validador de material design no era fácil de implementar cuando las validaciones son personalizadas, y no existía documentación exacta de cómo llevar a cabo esta implementación. Por esto, se crearon banderas booleanas que activan y desactivan las validaciones manualmente cuando estos existen.

Obtención de información

Como ya se mencionó, los servicios se encargan de realizar las consultas al backend. Los controladores solicitan esta información cada 4 segundos, de forma que siempre se mantiene la información mostrada actualizada, en tiempo real, sin necesidad de presionar un botón de refresh.
Se utilizan progress spinners que indican cuando la información se está cargando por primera vez, favoreciendo la usabilidad.

UI

Se logró crear una interfaz de usuario responsiva. Para esto se utilizó la metodología "mobile-first", en la cual se desarrolla primero la vista para dispositivos móviles y luego se adapta para pantallas más grandes (como monitores de escritorios). Se definió como pantalla mínima la del dispositivo IPhone 5. Para pantallas más pequeñas que la del IPhone 5, no se asegura un display correcto de los componentes.
Para lograr mejor usabilidad, la ubicación de los elementos varía dependiendo del tamaño de la pantalla utilizada. Por ejemplo: el botón de agregar usuario está en la parte superior central de la pantalla para dispositivos grandes, mientras que para dispositivos más pequeños se ubica en la parte inferior derecha.
Se siguieron los lineamientos de material design para mejorar la experiencia de usuario. Se utilizaron Cards con sombras para diferenciar la posición de los elementos visuales. Por ejemplo, la barra principal superior es el elemento que se encuentra más arriba siempre (indicando lo de "principal").
Se utilizan iconos de material que mejoran la estética de la aplicación.
El calendario es un componente creado por el equipo (utilizando elementos de material design) debido a que los calendarios que existen en la web no son responsivos.
Las rutas están protegidas, esto evita que un usuario seguidor acceda a las vistas de un usuario administrador, siendo redirigido si esto sucede.
En caso de que el token utilizado actualmente expire, la aplicación lo detecta y solicita uno nuevo, sin que el usuario se entere de esto.

Defectos de UI

No se señala cuál es la opción del side-nav seleccionada.
No todos los componentes actualizan la información en tiempo real (no se implementa por cuestiones de tiempo). Sí se hace en los que se consideran más importantes, como son los comentarios de un encuentros.
Cuando se actualiza la información de una tabla en tiempo real, en dispositivos móviles, aparece y desaparece la barra de desplazamiento horizontal.

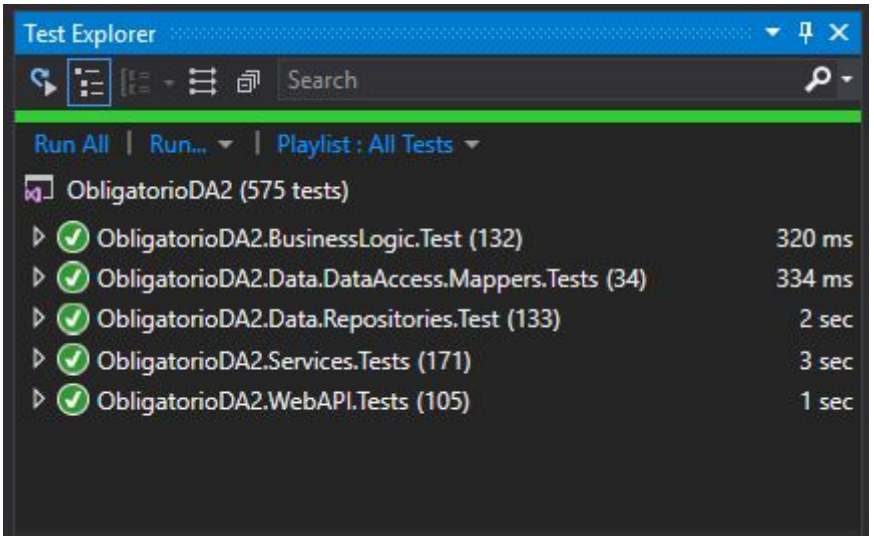
El deporte de un nuevo equipo es ingresado a mano, no en una lista desplegable con los deportes ingresados en el sistema (como sí se hace en el ingreso de encuentros).
La aplicación muestra la opción de ingresar un empate cuando el encuentro es entre dos equipos de un deporte que admite encuentros con dos o más equipos. Se devuelve un mensaje “bad request” poco descriptivo.

TDD y cobertura de pruebas unitarias

Existen 5 proyectos de test en la solución:

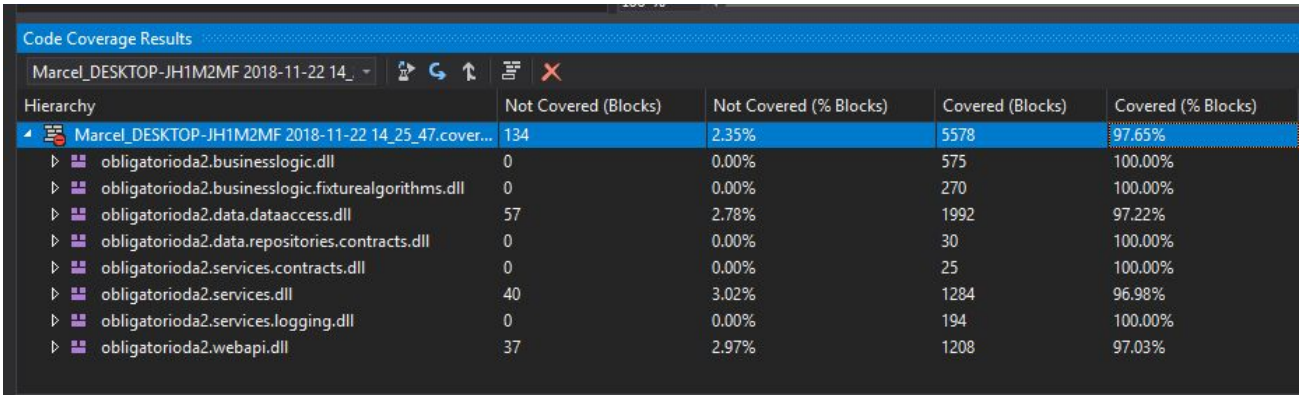
- ObligaotorioDA2.BusinessLogic.Test
- ObligatorioDA2.Data.DataAccess.Mappers.Tests
- ObligatorioDA2.Data.Repositories.Test
- ObligatorioDA2.Services.Tests
- ObligatorioDA2.WebAPI.Tests

Hay un total de 575 pruebas unitarias, de las cuales todas pasan.



Se considera que hubo una gran cantidad de pruebas unitarias, más que en otros proyectos llevados a cabo por el equipo, esto se debe a que hay muchas pruebas que son aisladas, es decir, se utilizan test doubles de las clases relacionadas a la clase a testear. Esto permite aislar las pruebas sin testear otros componentes. La desventaja, es que involucró un número mayor de pruebas, de las cuales tuvieron que ser cambiadas cuando se realizaron los cambios en el sistema, y está mantención resultó tediosa.

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. En el análisis de cobertura de esta entrega, se logró ignorar los assemblies y clases donde no se hicieron pruebas unitarias, como los paquetes de test mismos y las clases de migraciones de base de datos autogeneradas. De esta forma se obtiene un resultado más real y significativo del análisis de cobertura.

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,por ejemplo: El 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 de prueba es en memoria, no se ejecutan esas partes del código. También, otras partes del código, como las que levantan archivos y atrapan excepciones de IO y Reflection. Pero las coberturas son todas mayores o iguales a 96%, que son buenos valores. 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.

Anexo

Justificación de Clean Code

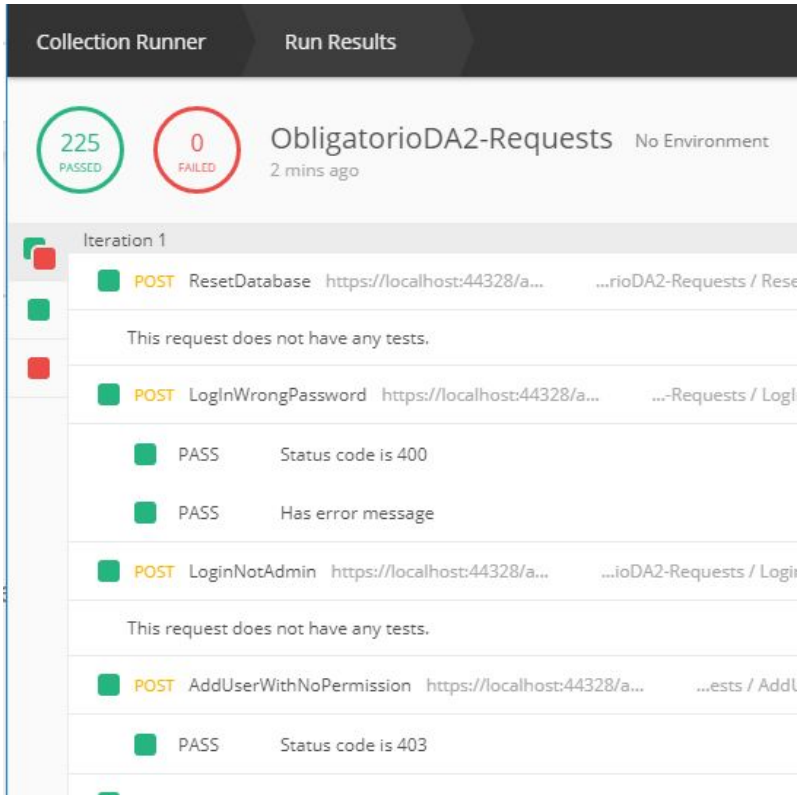
Para el desarrollo de este obligatorio se intentó utilizar las buenas prácticas recomendadas por el libro Clean Code, de Robert C. Martin.

- 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

Pruebas funcionales

Para asegurar el correcto funcionamiento de la aplicación, se elaboraron pruebas automatizadas en postman. Estas pruebas ejecutan requests al backend en secuencia y hacen pruebas sobre los resultados de estas. Para lograr la automatización, se creó un

controller que reinicia las dos bases de datos con los datos de prueba. El controller es TestController, del namespace WebAPI.Test.



durante cada integración de los cambios, no solo se ejecutaban todas las pruebas unitarias, sino que también todas las pruebas funcionales para corroborar el correcto funcionamiento luego de los cambios.
Se deja la colección de pruebas de postman en la carpeta de la entrega.