Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2
Obligatorio 1

Eusebio Durán 202741
Nicolás Eirin 200111

Repositorio en línea: https://github.com/ORT-DA2/Obligatorio-Eirin-Duran/

Entregado como requisito de la materia Diseño de Aplicaciones 2

Diseño de Aplicaciones 2
Obligatorio 1

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

Índice

Diseño de Aplicaciones 2
Obligatorio 1

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

Descripción General

Para la construcción del Software entregado se analizaron los requerimientos definidos en la letra del Obligatorio 1 de Diseño de Aplicaciones 2 y los requerimientos planteados y discutidos a través del foro de Aulas abierto para este fin.

El sistema desarrollado para esta primer instancia es una API REST que permite manejar deportes, equipos y crear encuentros entre otras funcionalidades.

Funcionamiento del Software

Dado que el software actualmente no cuenta con front-end la forma de interactuar con el mismo es a través de endpoints y verbos.

A continuación se muestra una tabla con todos los verbos y endpoints para crear leer modificar y borrar datos en el sistema.

Endpoint	Verbo	Acción	Permisos de Administrador
api/auth	POST	Devuelve un token de autorización para que el usuario pueda identificarse	No
api/users	GET	Retorna todos los usuarios registrados el aplicación	No
api/users/{userName}	GET	Retorna un usuario en particular	No
api/users/	POST	Permite agregar un nuevo usuario	Si
api/users/{userName}	PUT	Modificar un usuario existente	Si
api/users/{userName}	DELETE	Borra a un usuario	Si
api/users/{userName}/commentaries	GET	Muestra los comentarios de los equipos que el usuario logueado sigue	No

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

api/sports	GET	Retorna todos los deportes registrados el aplicación	No
api/sports/{sportName}	GET	Retorna a un deporte	No
api/sports/{sportName}/encounters	GET	Retorna todos los encuentros que existen para un deporte	No
api/sports	POST	Permite agregar un nuevo deporte	Si
api/sports/{sportName}	DELETE	Borra un deporte con todos sus equipos y encuentros	Si
api/teams	GET	Retorna todos los equipos registrados en la aplicación	No
api/teams/{teamName_sportName}	GET	Retorna un equipo	No
api/teams/{teamName_sportName}/encounters	GET	Retorna todos los encuentros para un equipo	No
api/teams	POST	Agrega un nuevo equipo	Si
api/teams/{teamName_sportName}	PUT	Permite editar los datos de un equipo	Si
api/teams/{teamName_sportName}	DELETE	Borra un equipo y todos sus encuentros	Si
api/teams/{teamName_sportName}/follower	POST	Agrega un equipo a la lista de equipos seguidos por el usuario logueado	No
api/encounters?{fechaDesde}?{fechaDesde}	GET	Retorna todos los encuentros entre dos fechas	No
api/encounters/{encounterId}	GET	Retorna un encuentro	No
api/encounters	POST	Permite crear un encuentro	Si
api/encounters/{encounterId}	PUT	Permite editar un encuentro	Si
api/encounters/{encounterId}	DELETE	Borra un encuentro	Si
api/encounters/{encounterId}/commentaries	GET	Retorna todos los comentarios de un encuentro	No
api/encounters/{encounterId}/commentaries	POST	Permite agregar un comentario a un encuentro a nombre del usuario que está logueado	No
api/encounters/fixture	GET	Retorna una lista con todos los fixtures implementados	No

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

api/encounters/fixture	POST	Permite crear un fixture para un deporte	Si

Contenido de los cuerpos para los requests POST

- api/auth

```
{
  "UserName" : "NombreDeUsuario",
  "Password" : "Contraseña"
}
```

- api/users/

```
{
  "UserName" : "NombreDeUsuario",
  "Name" : "Nombre",
  "Surname" : "Apellido",
  "Password" : "Contraseña",
  "Mail" : "direccion@dominio.com"
}
```

- api/sports

```
{
  "SportName" : "NombreDelDeporte"
}
```

- api/teams

```
{
   "Name" : "NombreDelEquipo",
   "SportName" : "DeporteAlQuePertenece"
}
```

- api/encounters

```
{
    "HomeTeamName" : "Equipo1",
    "AwayTeamName" : "Equipo2",
    "SportName" : "NombreDelDeporte",
    "DateTime" : "AAA/MM/DD"
}
```

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

Arquitectura del Sistema

El de diseño de la arquitectura del sistema fue basado en los principios establecidos en el Domain-Driven Design by Evans que establece que una división en las siguientes cuatro capas:

- **Domain Layer:** Contiene toda la lógica y reglas de negocio, se debe encontrar totalmente aislada del resto del sistema.

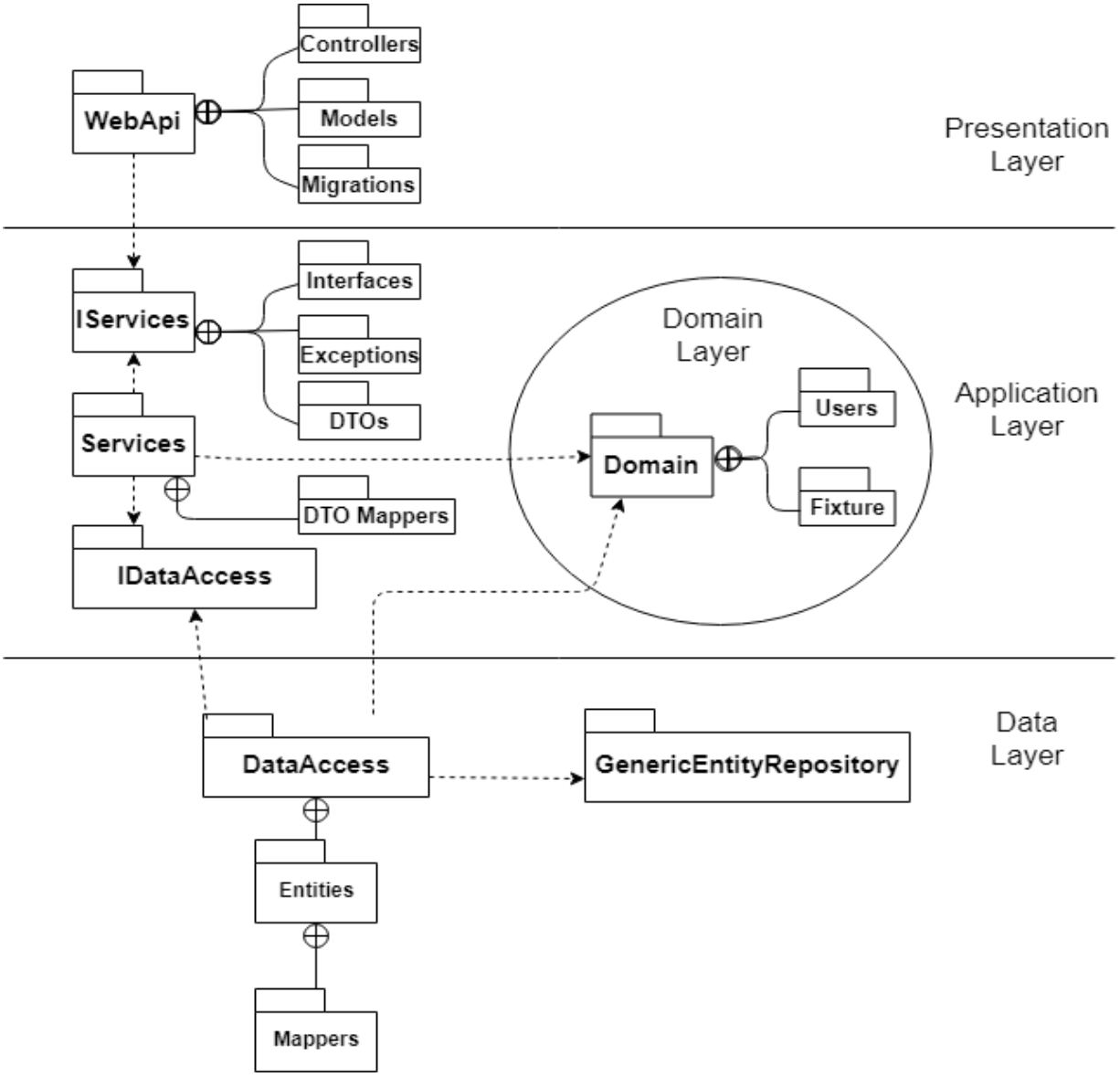
- **Application Layer:** Expone una serie de funciones que puede realizar el sistema y la realiza coordinando entre el dominio, el acceso a datos y posiblemente otros elementos externos.

- **Data Layer:** Esta capa se encarga de guardar los datos. Será donde se gestione todo lo relativo a la base de datos y a la creación, edición y borrado de datos de ésta.

- **Presentation Layer:** Contiene toda la lógica de interacción con el usuario.

A continuación se muestra un diagrama de paquetes, que expone los paquetes y las relaciones de dependencia entre los mismos, quedando fuera los paquetes de tests.

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)



Vista de alto nivel de los paquetes y capas del sistema.

Responsabilidades de cada paquete:

- **Domain:** Contiene las entidades de dominio y la lógica para la generación del fixture. Cada entidad de dominio es responsable de saber validarse a sí misma.

- **Services:** Coordinan los eventos entre la capa de dominio, el acceso a datos (DataAccess) y la WebApi

- **IServices:** Define un API que permite la comunicación entre la capa de presentación y la capa de aplicación mediante un conjunto de interfaces, DTOs y Excepciones. Tiene el propósito de desacoplar las dos capas anteriormente modificadas y permitir la inyección de dependencias.

- **WebApi:** Aparte de definir las configuraciones del framework tiene definidos los Controllers con sus modelos de entrada y salida por los cuales se interactúa mediante los verbos y endpoints con la API REST. A los controllers se les inyecta uno o más servicios. Los métodos definidos en los controllers llaman a métodos definidos en los servicios y si tienen que intercambiar datos lo hacen a través de una estructura de transferencia datos *(Data Object Transfer)* que los servicios mapean a entidades de dominio. Esta es una forma de desacoplar y lograr que los cambios en el dominio, no afecten a la WebApi.

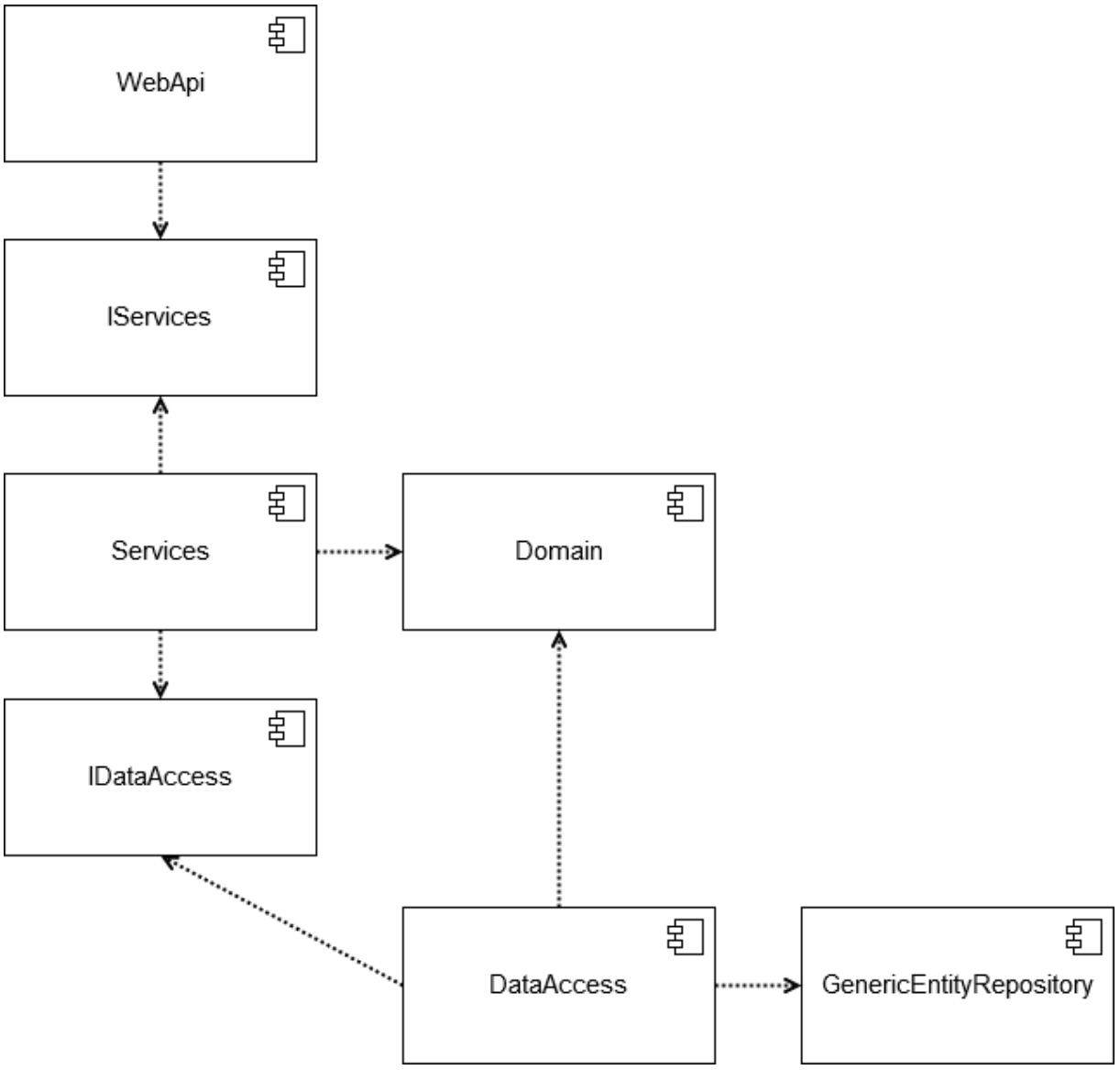
- **DataAccess:** Define las entidades de persistencia con sus respectivas traducciones a modelos de dominio y definite repositorios para estas entidades “Wrapeando” el repositorio de Entidades generico del paquete GenericEntityRepository**.**

- **IDataAccess:** Similar a IServices, define un conjunto de interfaces para que los servicios puedan comunicarse sin que exista una dependencia de Services a DataAccess.

- **GenericEntityRepository:** Define un repositorio de entidades genéricas en un entorno desconectado, que soluciona todas las operaciones de grafos que surgen al trabajar con Entity Framework en modo desconectado.

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

Desde la perspectiva de diagrama de componentes:



Vista de alto nivel de los componentes y sus relaciones.

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

Descripción y Justificación de Diseño

Se buscó un diseño para la aplicación que cumpla lo siguiente:

1. Nadie debe depender del dominio de la aplicación.

2. Todos los paquetes deben tener una única responsabilidad por la cual cambiar.

3. Las clases con mayor acoplamiento se encuentren en paquetes separados.

4. Exista la mínima cantidad de dependencias necesarias y el impacto de cambio de un paquete debe afectar lo mínimo posible a los paquetes dependientes de éste.

Dominio

La capa de dominio es la que se encarga de la generación del fixture y de validar la consistencia de los datos de las entidades.

El dominio consta de dos subpaquetes de los cuales uno tiene todo lo referido a los usuarios en el sistema *(Users)*, y el otro todo lo referido a fixture *(Fixture)*.

Para cumplir con el requerimiento de que debe poderse incorporar nuevos algoritmos de fixture se utilizó el patrón estrategia, por lo que para la incorporación de un nuevo algoritmo solo hay que proveer la interfaz IFixtureGenerator.

Por otro lado y como ya fue mencionado se tomó la decisión que las entidades de dominio sean las responsables de saber validarse, lo bueno de esto es que no importa desde donde de se creen, no pueden ser creadas con datos incorrectos, además de que no se sobrecarga a otra clase con la responsabilidad de validar a la entidades.

Por otra parte el dominio no conoce la implementación, ni accede a los datos persistidos, es responsabilidad de los servicios coordinar estos eventos.

A continuación se muestra el diagrama de clases del paquete fixture, subpaquete de dominio:

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

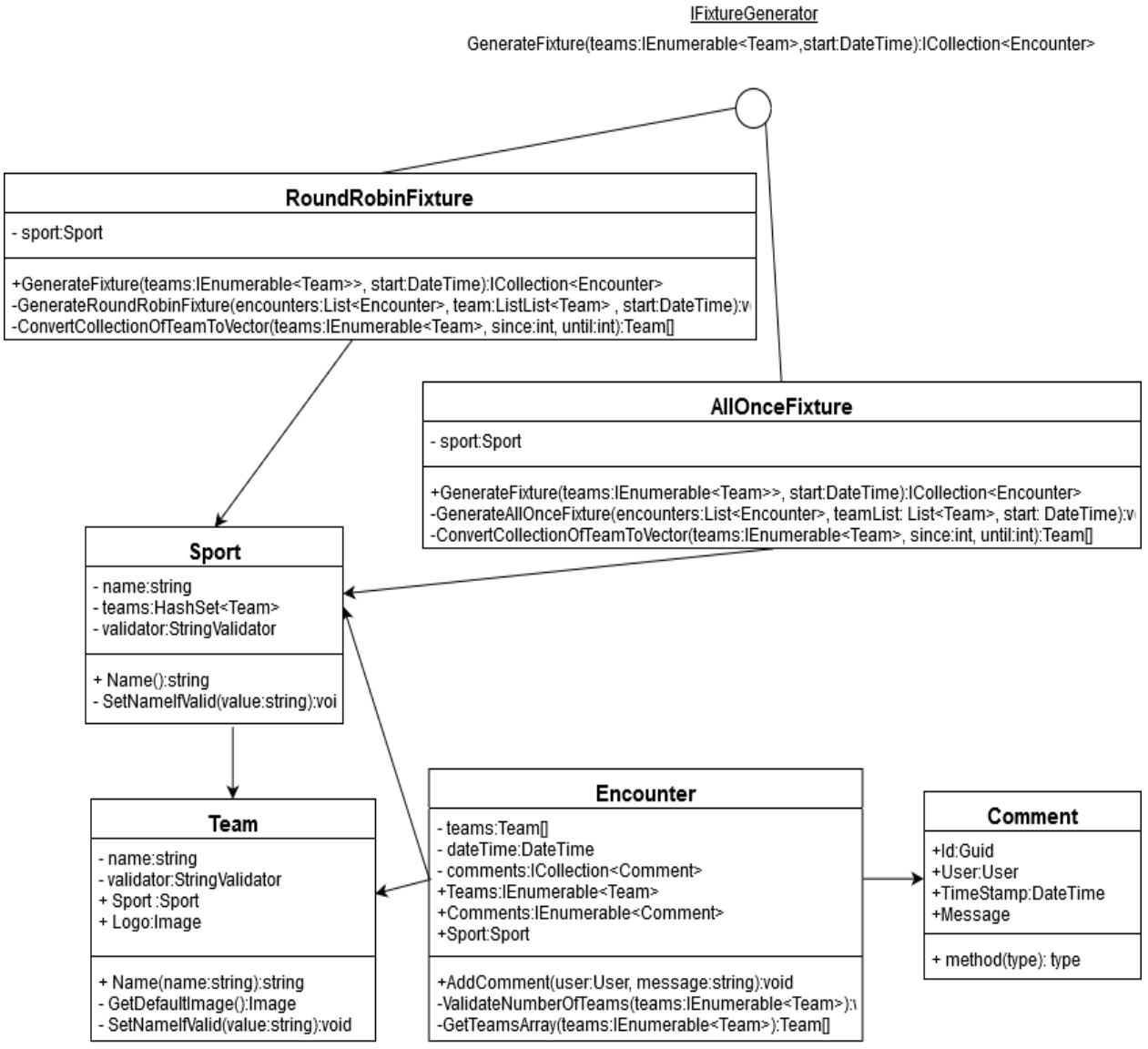


Diagrama de clases del subpaquete de dominio Fixture

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

Controllers

Dentro del proyecto de WebApi existe un subpaquete donde se encuentran los controllers, éstos son cinco, más uno adicional de test.

Fueron diseñados de acuerdo a las especificaciones, estándares y buenas prácticas sugeridas en páginas de documentación de Microsoft para esta tecnología.

Para realizar un request el usuario siempre debe estar autenticado, para esto la aplicación cuenta con un sistema de login basado en tokens (JWT) que tiene determinados beneficios respecto a otras tecnologías. Para entender cuales son los beneficios pasaremos a explicar cómo se realiza el login.

Para que un usuario pueda loguearse, deben enviarse las credenciales del mismo por el cuerpo de un POST a la ruta del AuthController. El mismo a través de los servicios (se explicará más adelante) obtiene los datos del usuario y devuelve un token con los datos necesarios para que el usuario pueda autenticarse la próxima vez que desee hacer un request. Este token debe ser mantenido en el header authentication.

A través de este sistema la API Rest puede saber quien es el usuario que hace el request y sus permisos dentro del sistema. Sin embargo, consideramos el login y el manejo de roles, no debe ser una responsabilidad de los controllers, o por lo menos no deberían ser únicamente los controllers quienes lo manejen. Por lo que en cada request, se obtienen las credenciales del usuario desde el token y se crea un servicio de login.

Esto ofrece algunas ventajas:

- Si se descarta el sistema de API Rest y se crea un Form para la aplicación, el sistema de login seguiría funcionando, ya que la capa de servicios que lo controla permanecería.

- Si se corrompe la base de datos de la aplicación o se cambian los roles o datos que el usuario tiene sería detectado a tiempo.

El proceso de login es mostrado en el siguiente diagrama de secuencia:

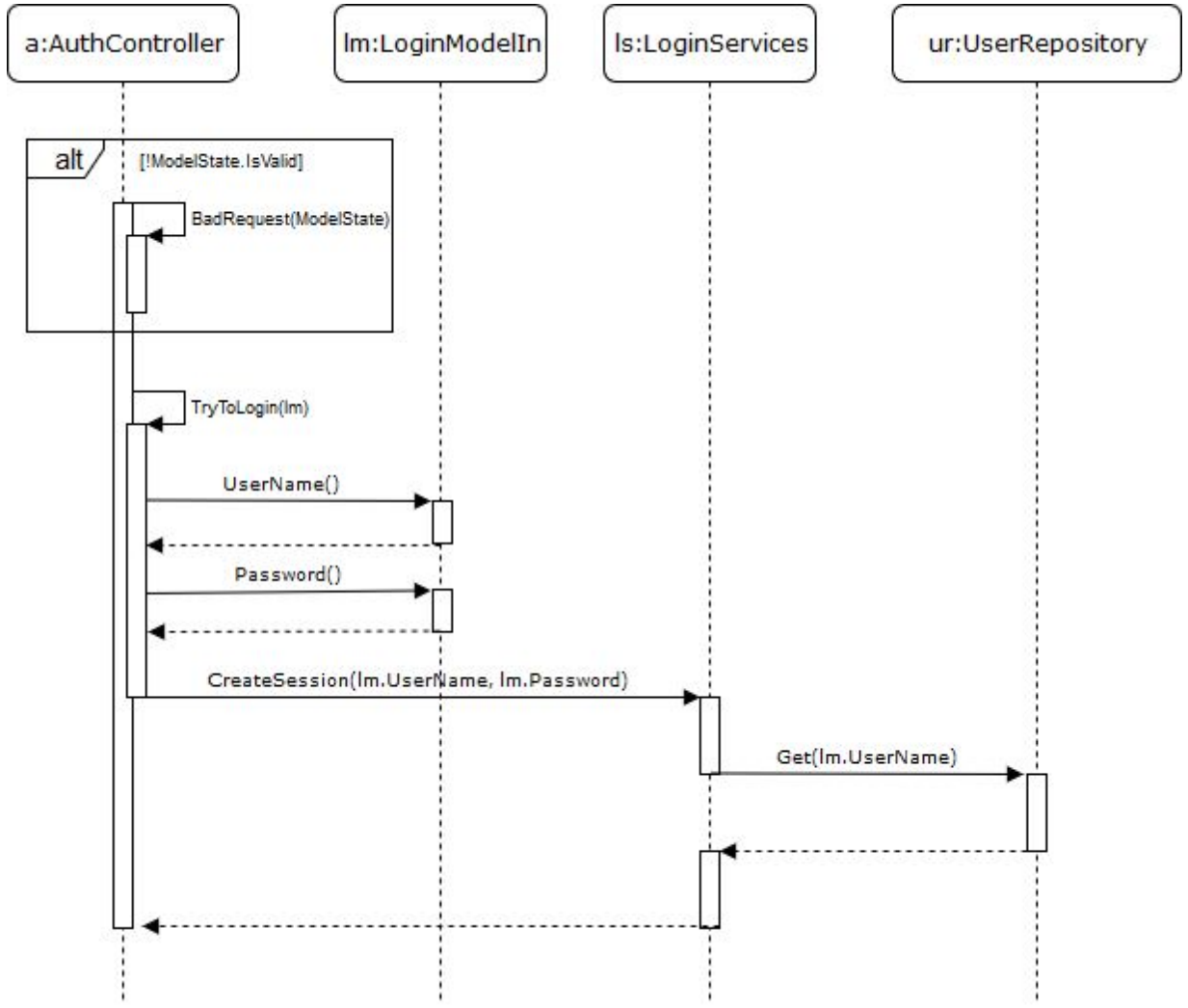


Diagrama de secuencia del proceso de autenticación.

Los demás controllers se ocupan de operaciones que la API puede hacer, los controllers están aislados del resto del sistema y no tienen lógica del negocio o servicios dentro de los mismos. Si se descarta completamente esta capa de la aplicación no se perdería ninguna funcionalidad del sistema.

Los controllers reciben modelos de entrada y retornan modelos de salida, esto ofrece la ventaja de que la implementación de las entidades de dominio y los objetos de transferencia es desconocida por quien usa la aplicación, pudiendo hacerse modificaciones en las entidades del negocio que no afecten la información que se sale o entra a través de los controllers.

La responsabilidad de los controllers es entonces, traducir los modelos de entrada que reciben a objetos de transferencia de datos (DTOs), llamar al método correspondiente en el servicio y capturar y traducir las excepciones lanzadas por los servicios a códigos de status, así como traducir los DTOs que retornan los servicios a modelos de salida.

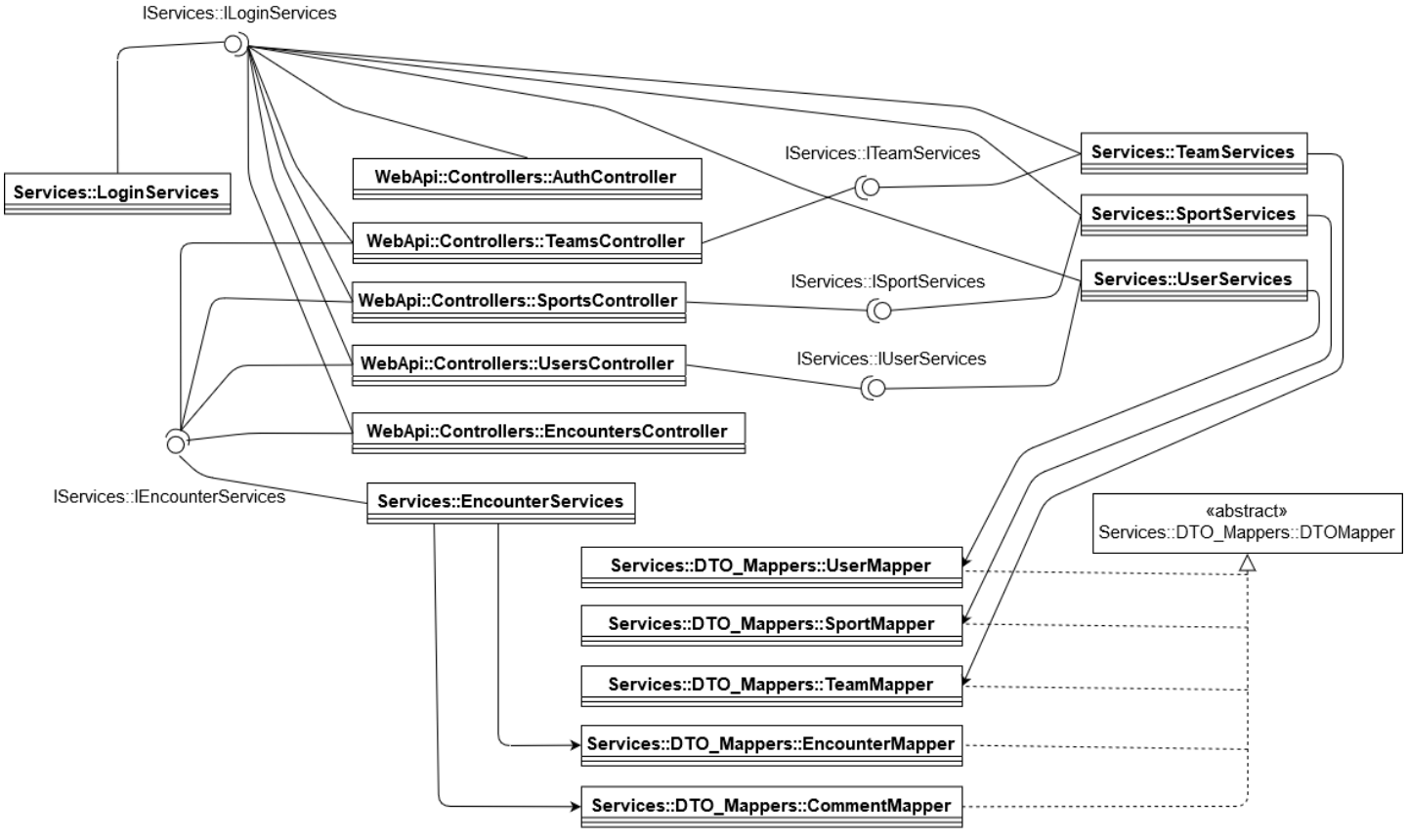


Diagrama UML que muestra las relaciones entre los paquetes WebApi, IServices y Services

El diagrama de clases de arriba muestra la interacción entre los controllers y los servicios (la capa más próxima a los controllers). Como se observa en el mismo la comunicación se hace a través de interfaces para que las dependencias queden de manera correcta. La localización de estas interfaces en otro paquete, podría ser discutible, y en un

proyecto de pequeñas dimensiones como este podrían haber sido colocadas como un subpaquete dentro del dominio. La decisión de colocarlas como un paquete independiente surge para dar una presentación más prolija a la organización lógica del código.

Por otro lado la transferencia de la información entre las capas a través de DTOs podría también ser discutida para un proyecto de estas dimensiones ya que los DTOs son similares a las entidades de dominio. La justificación a esta decisión es similar a la de los modelos de entrada y salida de los controllers, si se cambia la implementación de las entidades de dominio, el impacto de esto no se propagaría a capas superiores a los servicios.

Services

Los services sirven como una fachada entre la capa de presentación y el resto del sistema. La fachada está definida por el paquete IServices que define una API para comunicar la capa de presentación con el resto del sistema utilizando interfaces, DTOs y excepciones del IServices. Esta fachada define una serie acciones o servicios que el sistema puede realizar y que son utilizadas por los elementos de la capa de presentación como el WebApi.
Los servicios en sí mismos con contienen mucha lógica sino que solo de encargan de utilizar el dominio y el data access para realizar estas acciones.

Instanciación del algoritmo de armado de fixture

Como el usuario puede elegir distintos tipos de algoritmos de armado de fixtures (que utilizan el patrón de estrategy) surgió el problema de cómo instanciar la implementación de el IFixtureGenerator. La solución más simple sería utilizar un diccionario o un switch que vincule el input del usuario con la implementación del algoritmo pero esto al ser RTTI exigiría modificar esa parte del código cada vez que se agregara o borrarase uno de estos algoritmos, lo que cerraría el sistema a la expansión.
Solucionamos este problema buscando las implementaciones del IFixtureGenerator en el assembly correspondiente y instanciando la que corresponda al input del usuario, de estas manera el sistema queda abierto a la modificación ya que por ejemplo si se quisiese crear un nuevo algoritmo de creación de fixture lo unico que se debería hacer es crear la clase que implementa la interfaz IFixtureGenerator y no haría falta hacer cambios en ninguna otra clase.

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

DataAccess

Independización de la base de datos

Para el acceso a datos se utilizó el ORM (object relationship mapper) Entity Framework Core, de manera de abstraernos de la implementación específica de las base de datos y de manera de generar y interactuar con la base de datos en un nivel de abstracción más alto evitando tener que usar SQL. Trabajar de esta manera resulta mucho mas facil, y rapido comparado con hablar directamente con una Base de datos, además nos permite cambiar la tecnología de base de datos sin tener que cambiar nuestro sistema.

Independización del ORM

Además de esto aplicamos el patrón de diseño de repositorios, utilizando una interfaz intermediaria de manera de abstraernos del ORM y para encapsular toda la lógica del acceso a datos. De esta manera el resto de nuestro sistema es totalmente independiente al acceso a datos lo que quiere decir que se podría cambiar todo el acceso a datos sin tener que realizar cambios en el resto del sistema.

Modelo de Entity Framework

Decidimos utilizar un modelo desconectado de Entity Framework ya que utiliza menos recursos y que no mantiene la conexión con la base de datos abierta. Sus desventajas son que es menos eficiente que el conectado y que se necesita mas logica manejar todo el grafo de estado de las entidades manualmente. Aunque el modo conectado hubiese sido apropiado para un sistema transaccional como es el de la webapi preferimos utilizar el desconectado ya que la responsabilidad de manejar el contexto se mantiene en el DataAccess y no se delega a otras partes del sistema como webapi.

Entidades de persistencia

En vez de persistir los objetos de dominio directamente optamos por persistir entidades de persistencia que luego son mapeadas a objetos de dominio. De esta manera podemos independizar los objetos de dominio con su persistencia en el ORM. Esto es útil ya que puede llegar a ser conveniente tener una representación diferente en la persistencia que la de dominio y además porque la tecnología de acceso a datos puede tener restricciones que no deberían afectar al dominio. Este es el caso con las relación many to many de los usuarios con sus equipos que siguen ya que EF Core no soporta nativamente estas relaciones, por lo tanto tuvimos que crear una entidad para la relación entre estas dos clases. Si no hubiésemos

elegido tener entidades de persistencia diferentes a la de dominio los objetos de dominio debería haber sido cambiados por la tecnología de ORM, lo que viola el patrón de inversión de dependencias.

Generic Entity Repository

Realizar el alta baja y modificación de entidades y todas las entidades que se relacionan a ellas implica ,especialmente cuando se utiliza un modo desconectado, crear código específico a las entidades que se quieren persistir para manejar los grafos de estados de las entidades. Lo que significa que cada vez que se que se agrega un entidad nueva se tiene que crear o modificar el repositorio específico de esa entidad, lo que dificulta mucho el cambio y expansión del sistema. Para evitar todo esto creamos el paquete GenericEntityRepository que se encarga de solucionar los grafos de estado de las entidades para cualquier entidad genérica, gracias a esto si se cambia una entidad no hace falta cambiar la implementación de los repositorios.

Data Access

Tiene la responsabilidad de definir lo específico lo específico a este sistema de acceso a datos, que son las entidades de acceso a datos y el contexto de EF utilizado. Además “Wrapea” el Generic Entity Repository de manera de simplificar la creación de repositorios específicos.

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

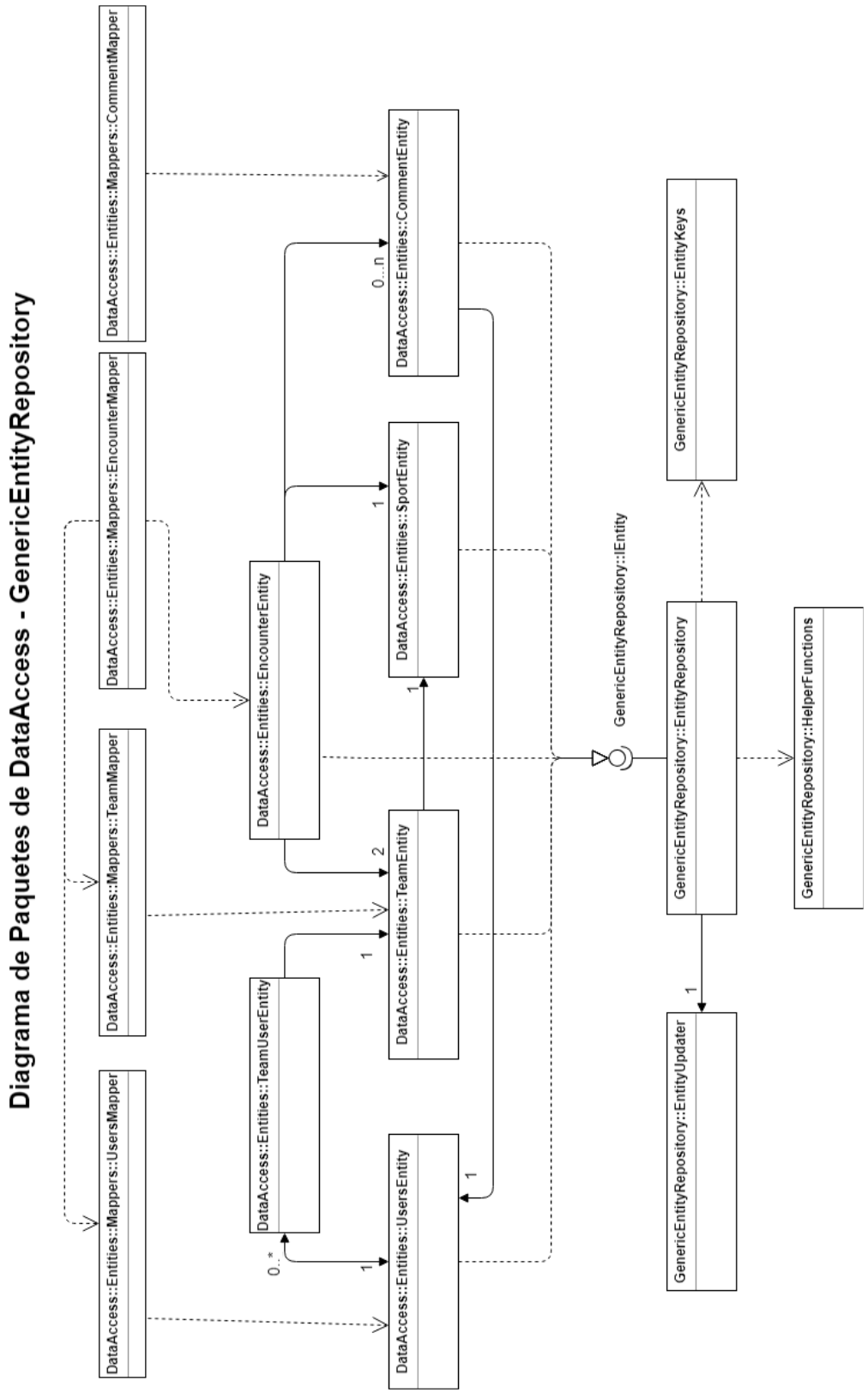


Diagrama de secuencia Seguir Equipo

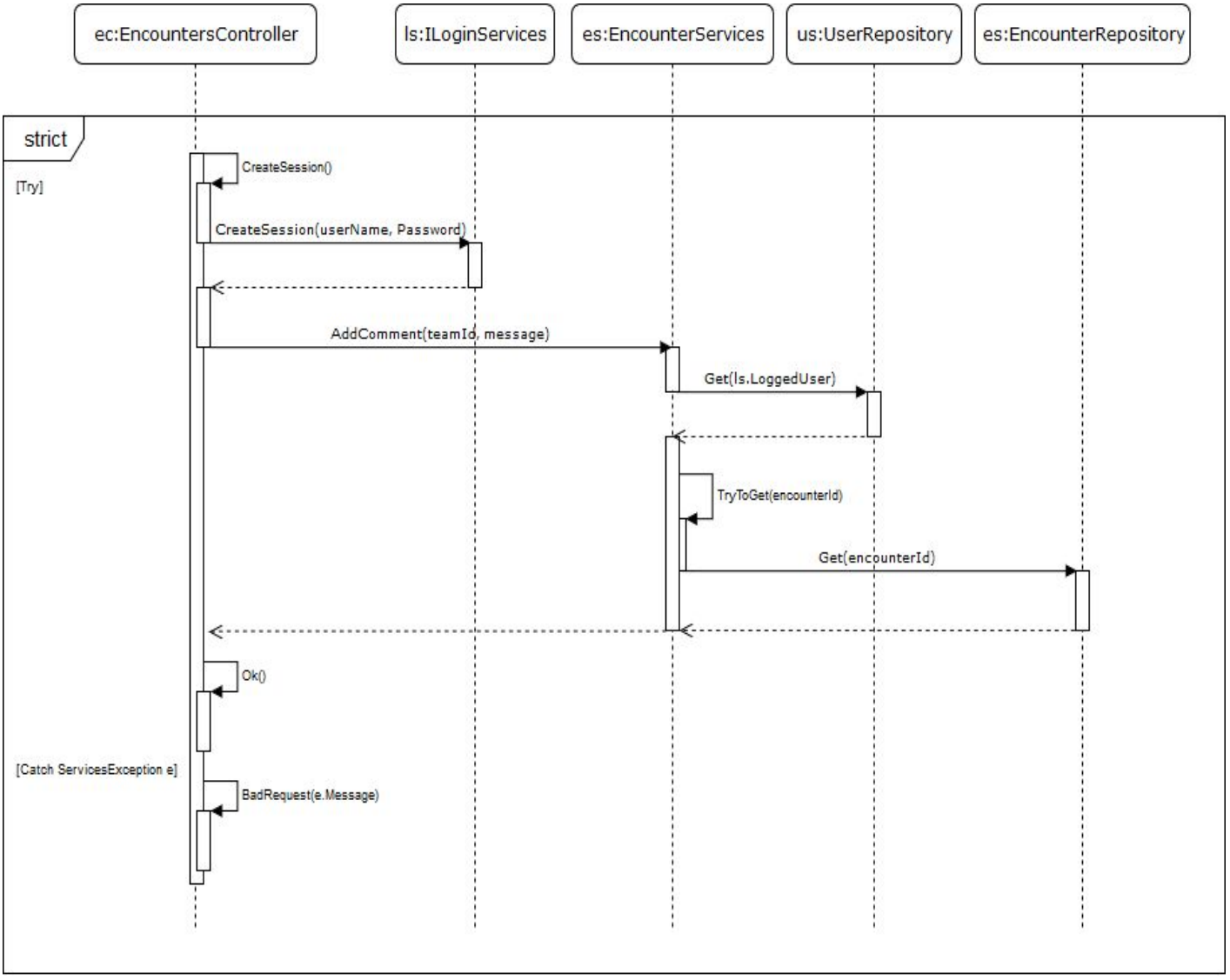
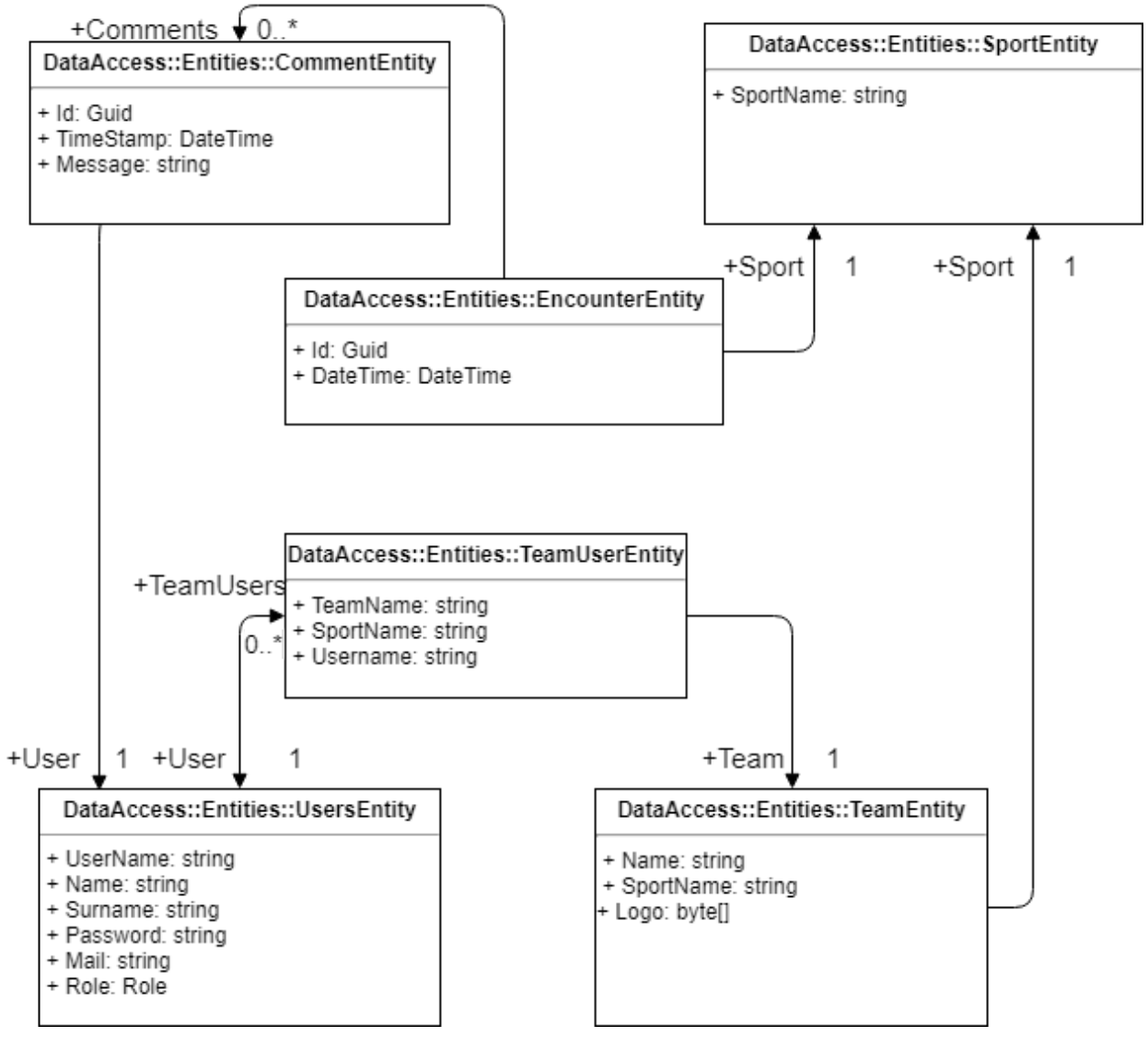


Diagrama de secuencia Agregar Comentario

Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

Diagrama de Entidades de acceso a datos

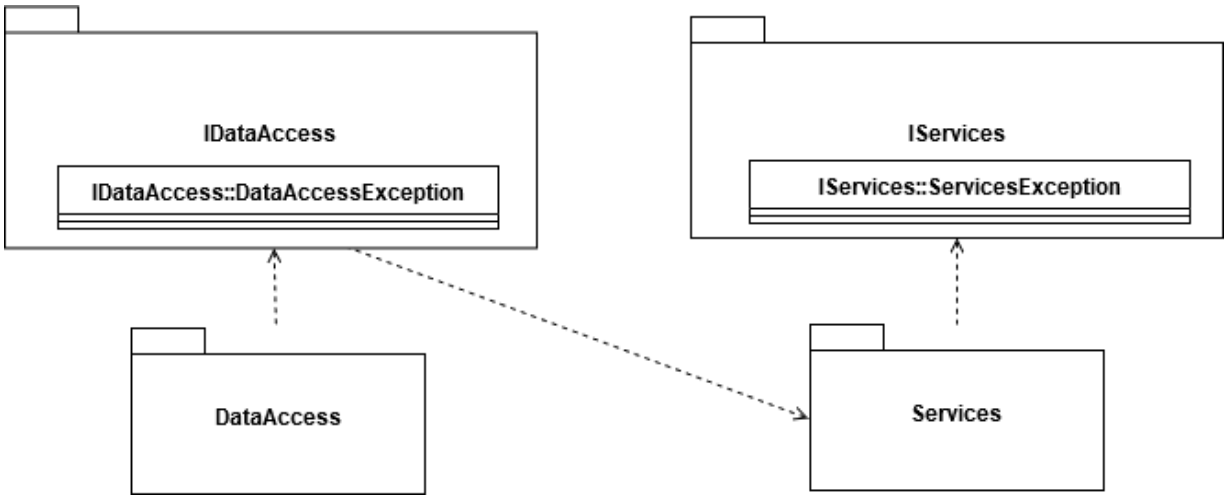


Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

Manejo de excepciones

Para el manejo de excepciones se usaron la menor cantidad posible. Para las excepciones de acceso a datos, se lanza una excepción dentro del paquete IDataAccess que proporciona un mensaje de cuál fue el problema que dió lugar a ésta.

Cada método de los servicios que coordina eventos y accede a datos captura esta excepción y lanza una nueva excepción localizada en el paquete IServices, esta excepción contiene un mensaje que informa desde donde fue lanzada y la excepción interna que fue arrojada desde el acceso a datos.



Excepciones principales en el sistema.

Cobertura de las Pruebas Unitarias

Para el desarrollo del obligatorio se utilizó la metodología Test Driven Development “Chicago”, primero se desarrollaron las pruebas para las capas más bajas (DataAccess) y luego de ser implementada se implementaron las capas más superiores que dependen de

éstas. De esta forma se logró reducir la cantidad de Mocks necesarios para el desarrollo, siendo estos necesarios solamente en los test de los controllers.

De un análisis de cobertura para las 189 pruebas mstest desarrolladas se desprenden los siguientes resultados:

Paquete	Bloques cubiertos	% de bloques cubiertos
IDataAccess	9	67%
DataAccess	394	87%
IServices	88	73%
Services	502	72%
Domain	480	84%
GenericEntityRepository	321	85%
WebApi	680	75%
Total	**2474**	**87%**

Adicionalmente a las pruebas unitarias se realizaron sesenta pruebas automatizadas en Postman con el fin de comprobar que la aplicación funciona correctamente, y puedan probarse cosas que no son posibles desde los tests, ya que son tecnología que ejecuta el framework, como el token de autenticación y los campos requeridos en el cuerpo de las requests.

Manual de Instalación

Para instalar el deploy de la aplicación en entornos de trabajo Windows es necesario tener el siguiente software instalado:

- Internet Information Server
- SQL Server Management Studio 2014

- Dotnet Core Runtime

Se deben seguir los siguientes pasos:

1. En la raíz directorio en la rama master del repositorio se encuentra una carpeta que contiene dos archivos .bak uno con una base de datos vacía y otro con datos de prueba. Adicionalmente y por seguridad también existen dos archivos .sql como alternativa a los .bak. Estas tablas deben ser importadas a mediante SQL Server Management Studio 2014.

2. Dentro de la carpeta del deploy localizar el archivo appsetings.json y cambiar el connection string por el del servidor SQL que tiene el backup de la base de datos importado.

3. Localizar y ejecutar el IIS y verificar que exista el módulo AspNetCoreModule haciendo clic en módulos.

4. Una vez hecho esto, se debe copiar la carpeta del deploy en la siguiente ubicación C:\inetpub\wwwroot

5. Dentro de IIS agregar un nuevo sitio web.

6. Quitar el CLR por defecto.

7. Volver SQL server, dirigirse a la carpeta Security y crear un nuevo inicio de sesion y colocar en el login name IIS APPPOOL\[nombre elegido para la aplicacion].

*Este procedimiento puede variar de acuerdo a las configuraciones del entorno en donde se quieran ejecutar.