Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2
Obligatorio 2

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 2

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

Índice

Diseño de Aplicaciones 2
Obligatorio 2

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

Declaraciones de Autoría

Nosotros, Eusebio Durán, Nicolás Eirin, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizabamos el segundo obligatorio de la materia 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.

Diseño de Aplicaciones 2
Obligatorio 2

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

Descripción General

Para esta segunda instancia de obligatorio, se implementaron a la primer solución propuesta, los nuevos requerimientos planteados en la letra del segundo obligatorio, y lo discutido por el foro de aulas abierto para este fin. Además ahora, el sistema cuenta con una aplicación de Front-end Single Page Application para que el usuario pueda interactuar con el mismo. Dicha aplicación es un proyecto de Angular completamente independiente al Back-end, que se comunica con éste cuando necesita datos a través de los Endpoints definidos en la primer entrega.

Análisis de la Solución a nivel de Componentes

En esta sección se analizan principios a nivel de los componentes que contiene la solución presentada, para entender por qué esta fue organizada de esta manera.

Cohesión

- **Equivalencia liberación/reuso:** Las clases de un componente tiene que tener un motivo por el cual estar juntas. Todos los componentes de la solución tienen motivos por los cuales sus clases están juntas. Generalmente las clases de los componentes tienden a tener mayor acoplamiento entre sí. Un componente donde existe poco acoplamiento es la WebApi, las clases de este componente tienen muy poco acoplamiento entre sí, existen muy pocas dependencias entre ellas. Lo cual es lógico, es esperable que un controller no dependa de otro. El motivo por el cual estas clases están juntas, es porque tienen las mismas responsabilidades a nivel de funcionalidades, la responsabilidad de los controllers es atender las peticiones de entrada y salida de información y llamar a los servicios correspondientes.

- **Clausura común:** Las clases que cambian por el mismo motivo y al mismo tiempo se encuentran en un mismo paquete. Un componente que claramente cumple con este principio es el DataAccess, si se modifica o implementa un nuevo sistema de almacenamiento de datos, probablemente tenga impacto sobre la mayoría o todas las clases de este componente.

- **Reuso común:** La mayoría de los componentes también pueden ajustarse a este principio, por ejemplo en el caso del dominio, no tendría sentido rehusar solo algunas clases de éste ya que tienen un alto acoplamiento entre sí.

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

Acoplamiento

- **Dependencias acíclicas:** Para que el impacto de cambio sea el mínimo posible, es decir para tener que recompilar la mínima cantidad de componentes al realizar un cambio, la solución se diseñó de tal manera que no existan ciclos en el grafo de dependencias.

- **Dependencias estables:** El sentido de las dependencias se hace de acuerdo a la estabilidad de los paquetes, es por esto que los paquetes más complicados de cambiar, como el dominio de la aplicación, que también es el que cambia con menos frecuencia, no tengan dependencias hacia otros paquetes. Como contraposición el paquete Dominio, es muy estable, pero presenta pocas abstracciones, por lo que es difícil de extender y se encuentra en la zona de dolor (se profundizará en esto más adelante).

- **Abstracciones estables:** Para la primera instancia del sistema, se diseñaron los componenetes de tal manera que se cumpliera el Principio de Inversión de Dependencias, este diseño, también permite la comunicación entre componentes por medio de abstracciones estables. Si se observa el diagrama de componentes las abstracciones del componente Services se encuentran en otro componente IServices para que quede claro que el impacto de modificar Services no produzca que se recompilen los componentes que requieran IServices. De igual manera ocurre con el paquete DataAccess que provee la interfaz IDataAccess y con las excepciones que estos lanzan. De esta forma, las abstracciones a parte de permitir la extensibilidad de los componentes también los protege del impacto de tener que recompilar toda la solución frente a un cambio, teniendo que recompilar menor cantidad de componentes según el caso.

Diseño de Aplicaciones 2
Obligatorio 2

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

Métricas a Nivel de Componentes

Las métricas de código a nivel de assemblies fueron automatizadas completamente con el Plugin para el IDE Visual Studio NDepend. Los resultados provistos por el mismo son los siguientes:

Assemblies	# lines of code	# IL instruction	# Types	# Abstract Types	# lines of comment	% Comment	% Coverage	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
EirinDuran.WebApi v1.0.0.0	888	10710	47	0	6	0.67	-	0	192	0.87	1	0	0
EirinDuran.Services v1.0.0.0	351	2375	17	1	0	0	-	1	43	1.88	0.98	0.06	0.03
EirinDuran.Logger v1.0.0.0	4	19	1	0	0	0	-	1	3	1	0.75	0	0.18
EirinDuran.IServices v1.0.0.0	75	364	23	11	0	0	-	38	0	0.65	0	0.48	0.37
EirinDuran.IDataAccess v1.0.0.0	3	18	3	2	0	0	-	22	1	0.67	0.04	0.67	0.2
EirinDuran.GenericEntityRepository v1.0.0.0	175	1195	8	1	5	2.78	-	11	31	1.62	0.74	0.12	0.1
EirinDuran.FixtureGenerators.RoundRobin v1.0.0.0	54	372	1	0	0	0	-	0	8	1	1	0	0
EirinDuran.FixtureGenerators.AllOnce v1.0.0.0	27	169	1	0	0	0	-	0	9	1	1	0	0
EirinDuran.Domain v1.0.0.0	191	1182	19	2	0	0	-	37	4	1.63	0.1	0.11	0.56
EirinDuran.DataAccess v1.0.0.0	350	3048	27	2	1	0.28	-	6	41	2.41	0.87	0.07	0.04
EirinDuran.AssemblyLoader v1.0.0.0	29	161	1	0	0	0	-	1	6	1	0.86	0	0.1
DefaultPositionTableGenerator v1.0.0.0	38	274	1	0	0	0	-	0	8	1	1	0	0

*Quedaron excluidos de los análisis los assemblies de Test que consideramos, no pertenecen a la solución presentada.

Cohesión Relacional (H)

La cohesión relacional deseable por componente es $1.5 \leq H \leq 4.0$. Si se observa la tabla anterior ningún componente supera la cohesión máxima recomendada. Los componentes que exponen $H < 1.5$ pueden justificarse:

- IServices e IDataAccess brindan interfaces para la comunicación con otros componentes, por lo que es lógico que sus clases tengan muy poca relación entre sí. Además de tener excepciones y objetos de transferencia entre capas (DTOs).

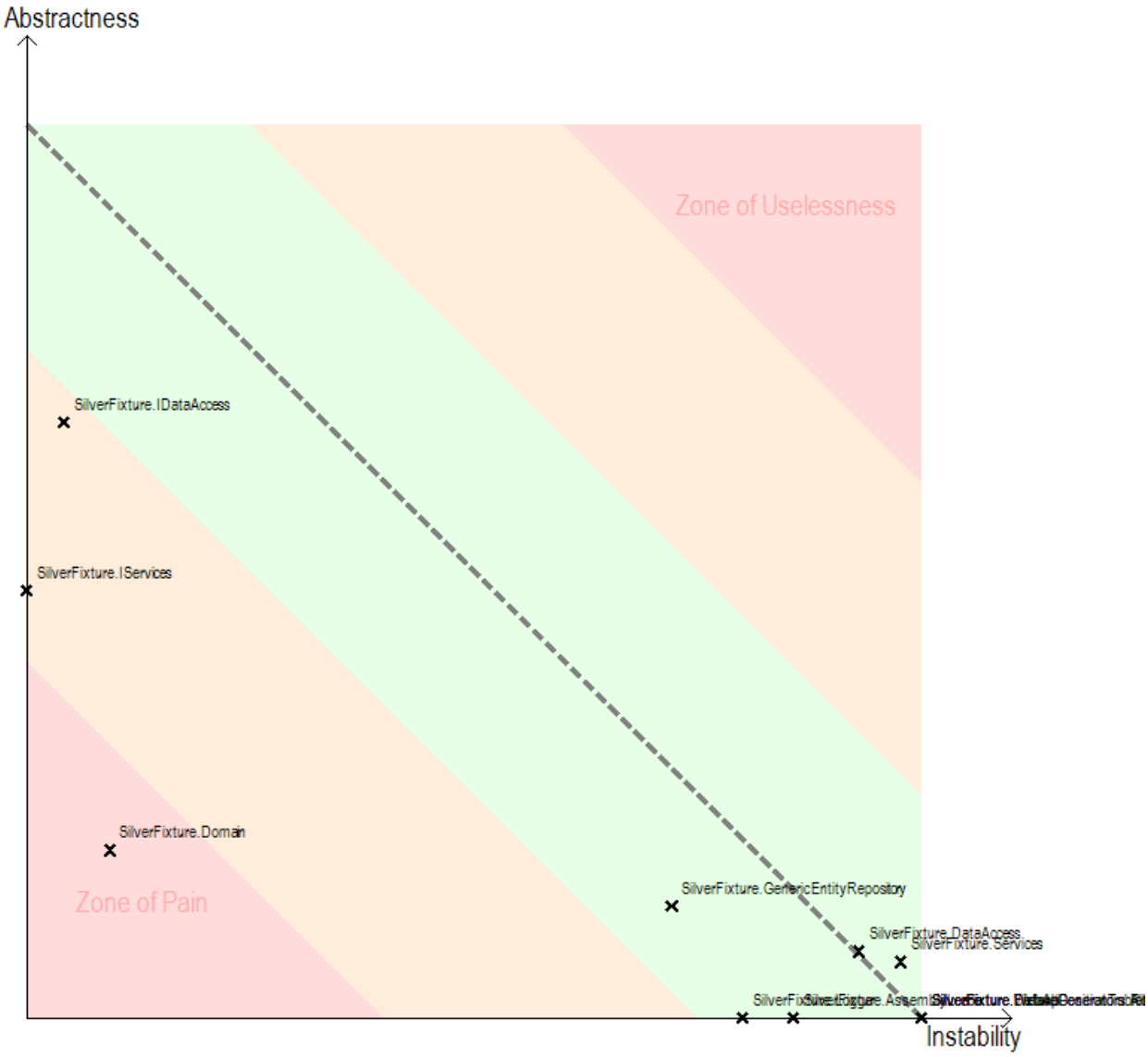
- FixtureGenerators.RoundRobin y FixtureGenerators.AllOnce al igual que AssemblyLoader son mecanismos para proporcionar algoritmos sin nececesidad de recompilar la solución, si

este sistema no hubiese sido estos componentes no existirían y los algoritmos pasarían a ser parte del dominio, aumentando la cohesión relacional.

- WebApi como fue mencionado en la primer parte tiene clases que no dependen entre sí, pero tienen un motivo por el cual estar juntas. Además tiene un acoplamiento eferente muy grande debido que la clase Startup de este componente define configuraciones de instanciación propias del framework de ASP.NET entre otras, por lo que se ve obligado a conocer más tipos de los que necesita.

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

Gráfico Inestabilidad vs Abstracciones



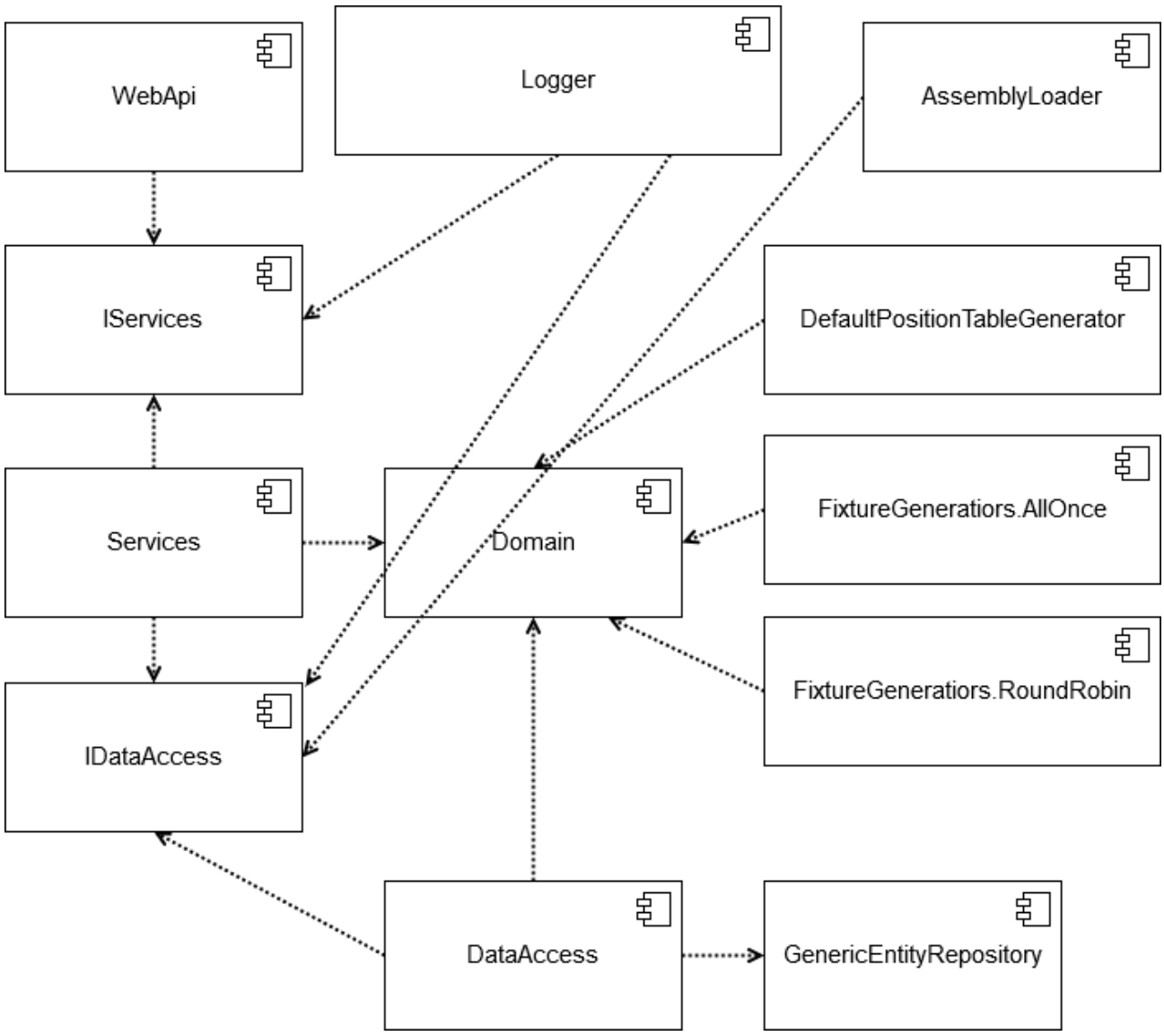
Los resultados obtenidos en el gráfico de arriba no necesariamente reflejan en realidad del nivel de abstracción, los componentes Services y DataAccess podrían considerarse más abstractos en la práctica. El problema surge porque las abstracciones por las que los demás componentes se comunican con éstos se hacen a través de otros componentes separados

IServices e IDataAccess que proveen interfaces para la comunicación. Si las abstracciones estuvieran dentro de los componentes Services y DataAccess, se obtendrían muchos mejores valores en el gráfico. Pero como estas interfaces fueron pensadas para cumplir con el principio de Inversión de Dependencias, esto no tendría sentido. También podrían haber sido colocadas en los componentes que las usan, pero se tomó la decisión de dejarlas separadas.

Por otro lado el dominio es quien menos interfaces provee, por lo que su nivel de abstracción es muy bajo, solucionar este problema requiere pensar el diseño del Dominio de tal manera que se provea mayor cantidad de interfaces para permitir la extensibilidad. Debido a restricciones de tiempo, que solo existe un componente en la zona de dolor, y que la probabilidad que este componente cambie debido a los requerimientos es muy baja, se tomó la decisión de dejarlo en dicha zona. Análogamente el dominio presenta mayor estabilidad respecto a los demás componentes.

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

Dependencias entre Componentes



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

Calidad del Código

Basándonos en el libro *"Código Limpio, Manual de estilo para el desarrollo ágil de software" (Robert C. Martin)* se aplicaron las siguientes técnicas para asegurar la construcción de un mejor código:

- Se utilizaron consistentemente **nombres nemotécnicos** en la declaración de variables, en la declaración métodos y nombres de clases.

- Se **evitó el uso de encodings** a excepción de las interfaces que por estándar de c# comienzan con “I” por ejemplo: IMyInterface, de las clases que solo tienen por responsabilidad transportar datos que terminan en “DTO” y de las excepciones propias que terminan en “Exception” y algunos casos puntuales donde vale la pena diferenciar.

- Se utilizaron **funciones pequeñas** que solo tienen una única responsabilidad. Si se necesitan funciones auxiliares, estas están declaradas más abajo del método que se auxilia de ellas y son privadas a la clase.

- Se manejaron **excepciones propias** pensadas de manera genérica y a éstas se les agregó una descripción para que el programador tenga más información y pueda saber por qué se produjo la excepción, por ejemplo ServicesException es la excepción que lanzan los servicios, junto con un mensaje que describe lo que ocurrió.

- Se intentó **evitar los comentarios**, solo se utilizaron cuando fue absolutamente necesario, y se trató que sea el código el que refleje lo que hace.

- El **sangrado del código** se hizo respetando las configuraciones que trae por defecto del formateador de código para C# y Typescript.

- Las **clases mantienen una estructura lógica** que va desde lo más general hasta lo más profundo, comienza por declaración de variables, luego los métodos, etcétera. De la misma forma existe una separación vertical y un agrupamiento lógico. Las líneas de código tienen un ancho menor a doscientos caracteres.

- En muy pocos casos no se respetó la **Ley de Demeter**, sobretodo en las pruebas unitarias.

Métricas de Calidad de Código

NDepend estimó para el proyecto (Back-End) una deuda técnica de 6.72%.

Diseño del Front-End

El Front-End que permite interactuar con la WebApi es un proyecto independiente realizado en Angular versión 7, para el diseño de los componentes y elementos con los que el usuario interactúa se utilizó la biblioteca gráfica Material Design, y los íconos oficiales provistos por ésta. Adicionalmente se usaron hojas de estilo de bootstrap.

El código del proyecto de angular sigue las buenas prácticas recomendadas en la página oficial de angular.

Internamente el proyecto se divide en servicios, componentes, clases (entidades), y directivas. Los servicios manejan los requests que van hacia el backend y los errores que el servidor potencialmente pueda devolver. Además existen servicios para la protección de las rutas de acuerdo a los permisos del usuario logueado en el sistema.

El token correspondiente al mecanismo de autenticación se guarda el en “local storage” y se declara un nuevo header con este token en cada método del servicio. Esto genera repetición de código que podría haberse evitado interceptando las requests y añadiendo el token. Por falta de conocimiento, se optó por dejar el código repetido, siendo este un aspecto a mejorar.

Los objetos JSON que el servidor envía se mapean a clases definidas en el proyecto.

Respecto a los componentes están separados en tres archivos, uno con todo el HTML, otro con la hoja de estilos CSS propia de dicho componente (los estilos comunes a más de un componente se encuentran en una hoja global de estilos) y otro con las funciones Typescript de dicho componente.

La navegabilidad de la página ocurre a través de rutas o a través de ventanas modales. Cada vez que se modifica información en el backend de la aplicación recarga solo la información que es nueva, pero no toda la página, este comportamiento es el esperado en una Single Page Application.

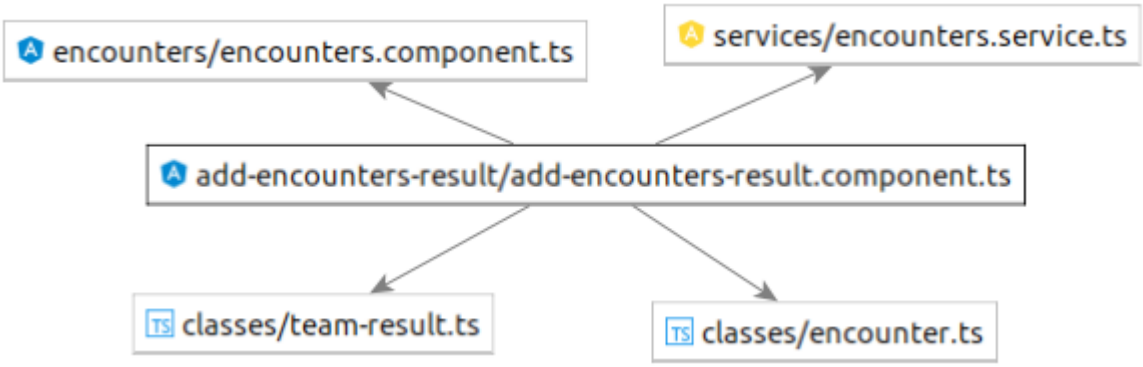
Para evitar request no válidos y descongestionar la red, los datos en los ingresos de formularios que no necesitan acceder a datos son validados desde el lado del Front-End.

Por último, la forma de cambiar la ruta en la que se encuentra la aplicación de backend es modificando la variable WEB_API_URL.

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

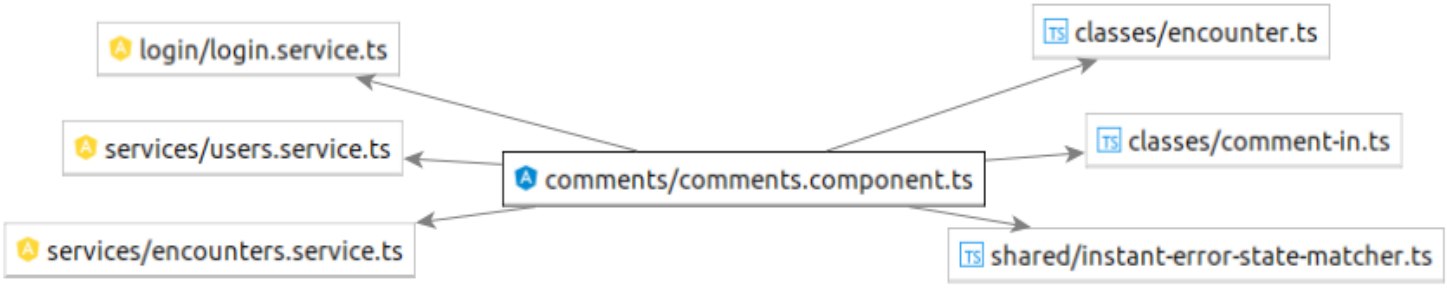
Diagramas de Dependencias de los Componentes más Importantes

Agregar un Resultado



El diagrama anterior muestra las dependencias componente add-encounter-result con los demás clases y servicios del sistema. Utiliza las clases team-result y encounter para mapear el resultado desde/hacia el backend/los servicios a clases propias, es modal del componente encounter y utiliza los servicios que provee encounter.services.

Agregar un Comentario a un Encuentro



El mecanismo de realización de un comentario al igual que el de agregar un resultado de un encuentro, utiliza las clases encounter y comment-in para mapear los datos, a partir de del servicio users.service obtiene los comentarios para los encuentros que el usuario sigue. Desde el servicio encounters.service se puede agregar un comentario a un encuentro puntual mapeado a la clase comment-in.

Para evitar que el usuario pueda enviar un comentario con un formato inválido (vacío) la directiva instant-error-state-matcher muestra un error en pantalla y desactiva el botón de envío del comentario. Este mecanismo es usado en la mayoría de los formularios que componen a la solución de Front-End con el fin de que no se envíen datos incorrectos que pueden ser validados desde el Front-End.

Puntos de acoplamiento al Dominio

Como el impacto de cambiar el dominio del sistema es muy grande, creamos una serie de puntos de acoplamiento en la areas del código que van a cambiar frecuentemente que permiten cambiar características del dominio sin tener que recompilarlo. Esto se realiza a través de dos mecanismos, la inyección de dependencias y de él cargado de DLLs dinámicamente.

Manejo de Generadores de Fixtures

Para poder agregar, eliminar o cambiar generadores de fixtures sin afectar el resto del sistema y de manera de poder hacerlo en tiempo de ejecución movimos los generadores de fixtures a componentes separados y utilizamos reflection para cargarlos en tiempo de ejecución. Para esto el dominio tiene definida la interfaz de IFixtureGenerator que todos los generadores de fixtures deben implementar.

Se podría haber definido la interfaz en un componente separado de manera de que los generadores no dependan del dominio pero optamos por no hacerlo ya que agregan complejidad al sistema al tener un componente más y tener que abstraer los objetos de dominios usados en un fixture y además no aporta mucho ya que como nadie depende de los generadores si estos se ven afectados por un cambio en el dominio, el impacto de actualizar el generador es mínimo.

Cambiar un generador es tan simple como agregar, sacar o reemplazar los DLLs en la carpeta FixtureGenerators que está localizada junto al resto de los componentes. Esta carpeta es creada automáticamente y cargada con los generadores actuales luego de cada build mediante un script de MSBuild. Cuando se está creando un publish es necesario copiar la carpeta desde el build al publish.

Si se quiere tercerizar la creación de generadores de fixtures solo hace falta darles una copia del DLL (No el código) del dominio para que puedan acceder a la interfaz IFixtureGenerator. Luego solamente se debe colocar el DLL del fixture generator en la carpeta Fixtures Generator.

Manejo de Generador de Posiciones

Otro punto del dominio que es propenso al cambio es el generador de posiciones ya que el dia de mañana se podría cambiar el sistema de puntajes. Para proteger al dominio del cambio se utilizó la inyección de dependencias. Como en el caso de los generadores de fixtures, el dominio tiene una interfaz IPositionTableGenerator que los generadores de posiciones, que se implementan en componentes diferentes, implementan. En caso de cambiar el generador de posiciones el cambio se vería limitado a el propio componente del generador y el componente del WebApi solamente por el startup.

Decidimos separar todo el sistema de las posiciones de los encuentros en sí mismo ya que los encuentros son algo estable que no tiende a cambiar mientras que los generadores de posiciones son más inestables. La desventaja de de tener estas dos cosas separadas es que no se puede aprovechar de el poliformismo para diferenciar si un encuentros es de a solo dos equipos o de a varios. Pero para haber implementar eso de esa manera se hubiesen tenido que utilizar Factories que hubiesen complejizado el sistema al tener que inyectar la factory en todas las áreas en donde es necesario crear encuentros, que es una gran parte del sistema.

Infraestructura

La infraestructura consiste en clases para acceder a recursos externos como el acceso a datos, manejo de archivos o un sistema de logs. Estas clases están basadas en interfaces definidas en la capa de aplicación (en IServices) de manera de utilizar inyección de dependencias. El Core de nuestro sistema (la capa de aplicación y de dominio) no debería depender de factores externos por esto se utiliza la inyección de dependencias para aislar el cambio de la infraestructura.

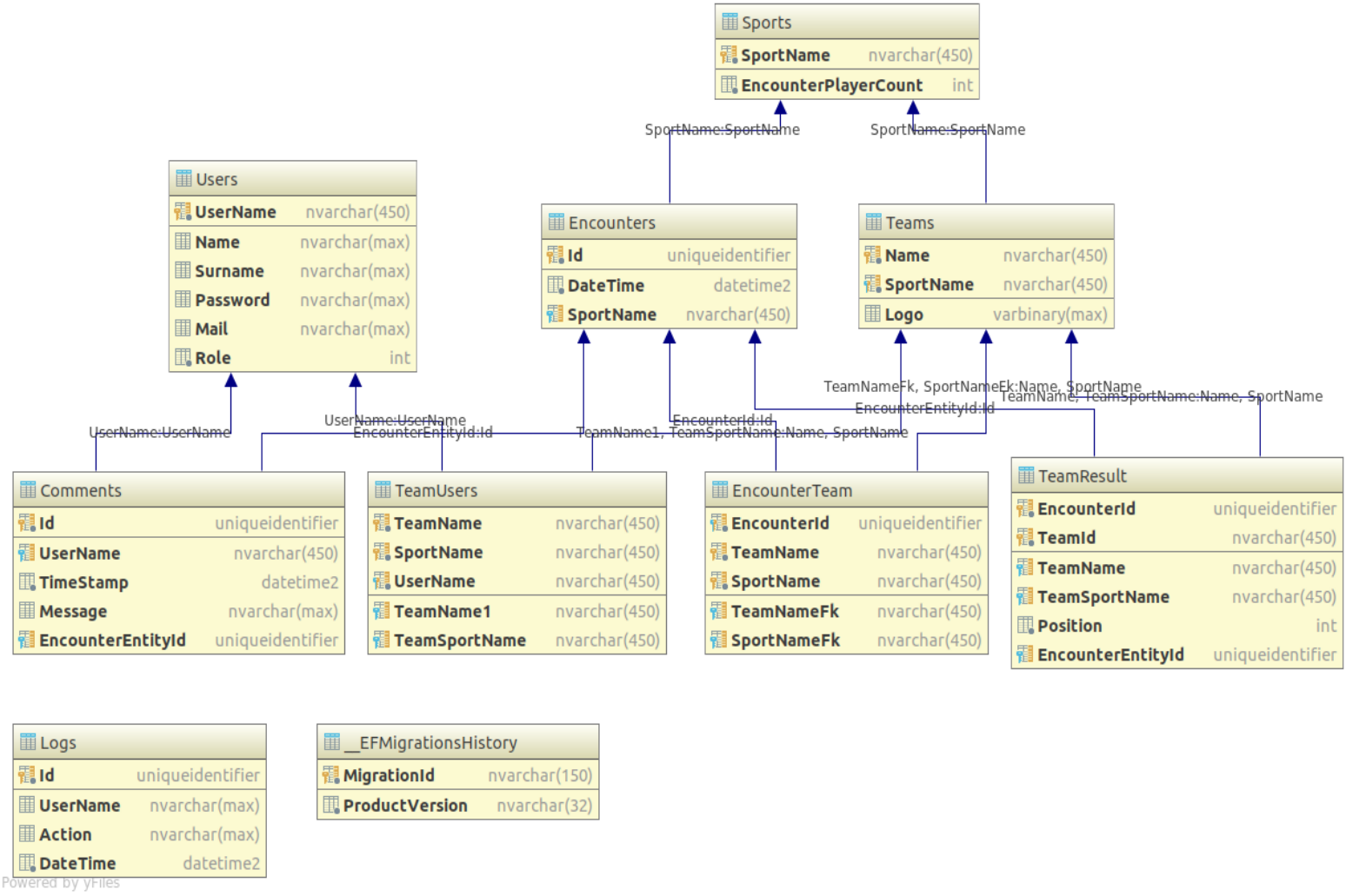
Logs

El sistema de auditoría fue implementado utilizando el patrón de interceptor, que es similar al proxy, agregando un filtro de ASP.NET que se ejecuta antes de cada llamada a los controles y utilizando un ILogger registra la acción realizada y el autor. Se utilizó este patrón para no tener que agregarle la responsabilidad de logear a los controller y para que el cambio de el logeo no afecte a los controllers ya que los controller ni siquiera saben que existe este middleware.

Nosotros interpretamos que el sistema de logeo tenía que ser independiente del sistema a nivel de código y a nivel de que se utilice otra forma de guardar los logs. Por esto igual que al resto de las clases de infraestructura se utilizó dependency injection. Si se quisiese por ejemplo guardar los logs en un archivo de texto o otra base de datos lo único que se debería hacer es implementar otro ILogger y inyectar al startup.

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

Modelo de Base de Datos



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. Utiliza puntos de acoplamiento para facilitar el cambio.

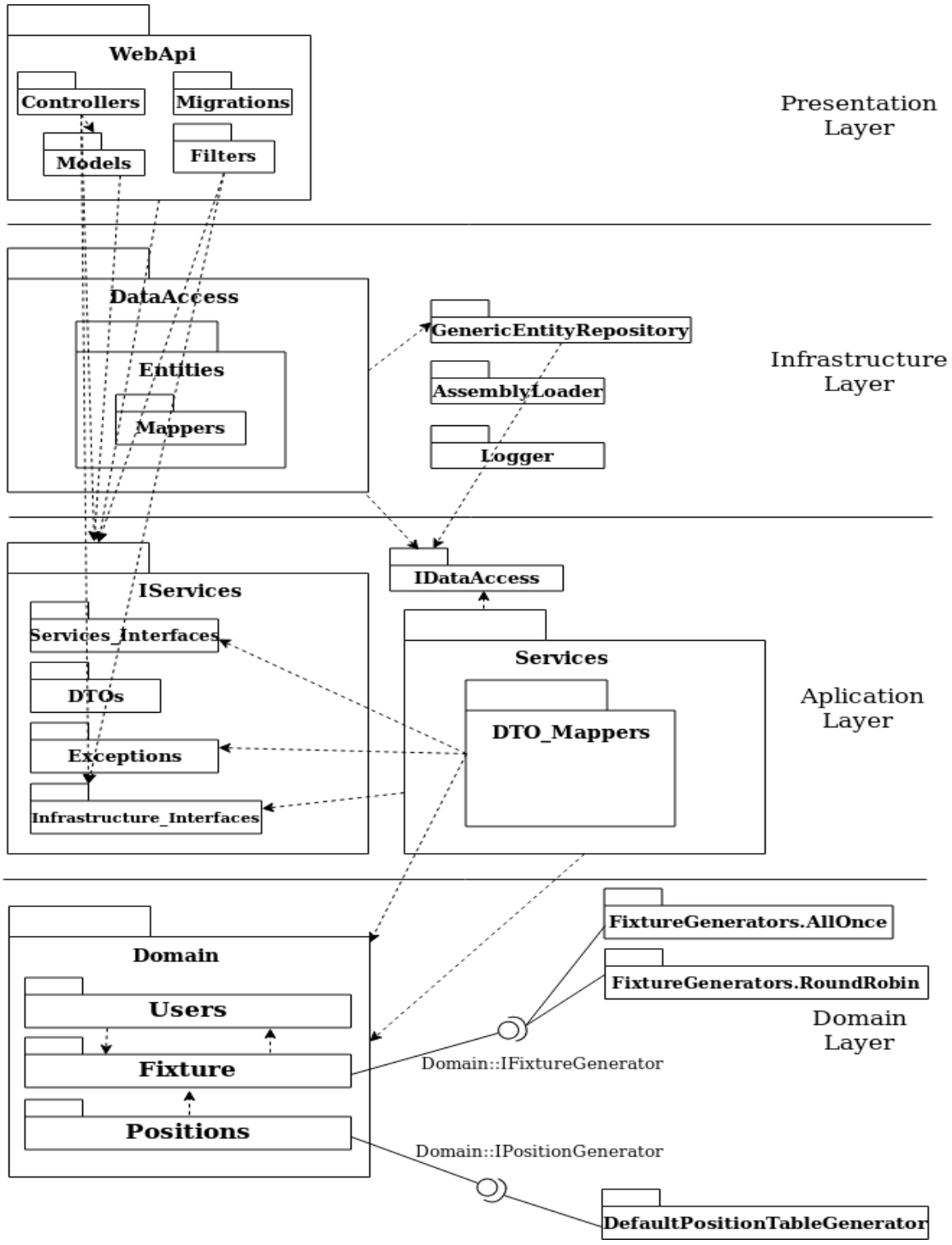
- **Application Layer:** Expone una serie de funciones que puede realizar el sistema y la realiza coordinando entre el dominio, el acceso a datos y posiblemente otros elementos externos.

- **Infrastructure Layer:** Esta capa se encarga de proveer acceso a recursos externos como el acceso a datos o sistema de auditoría al Core del sistema (capas de aplicación y dominio) mediante interfaces definidas en IServices.

- **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)

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:

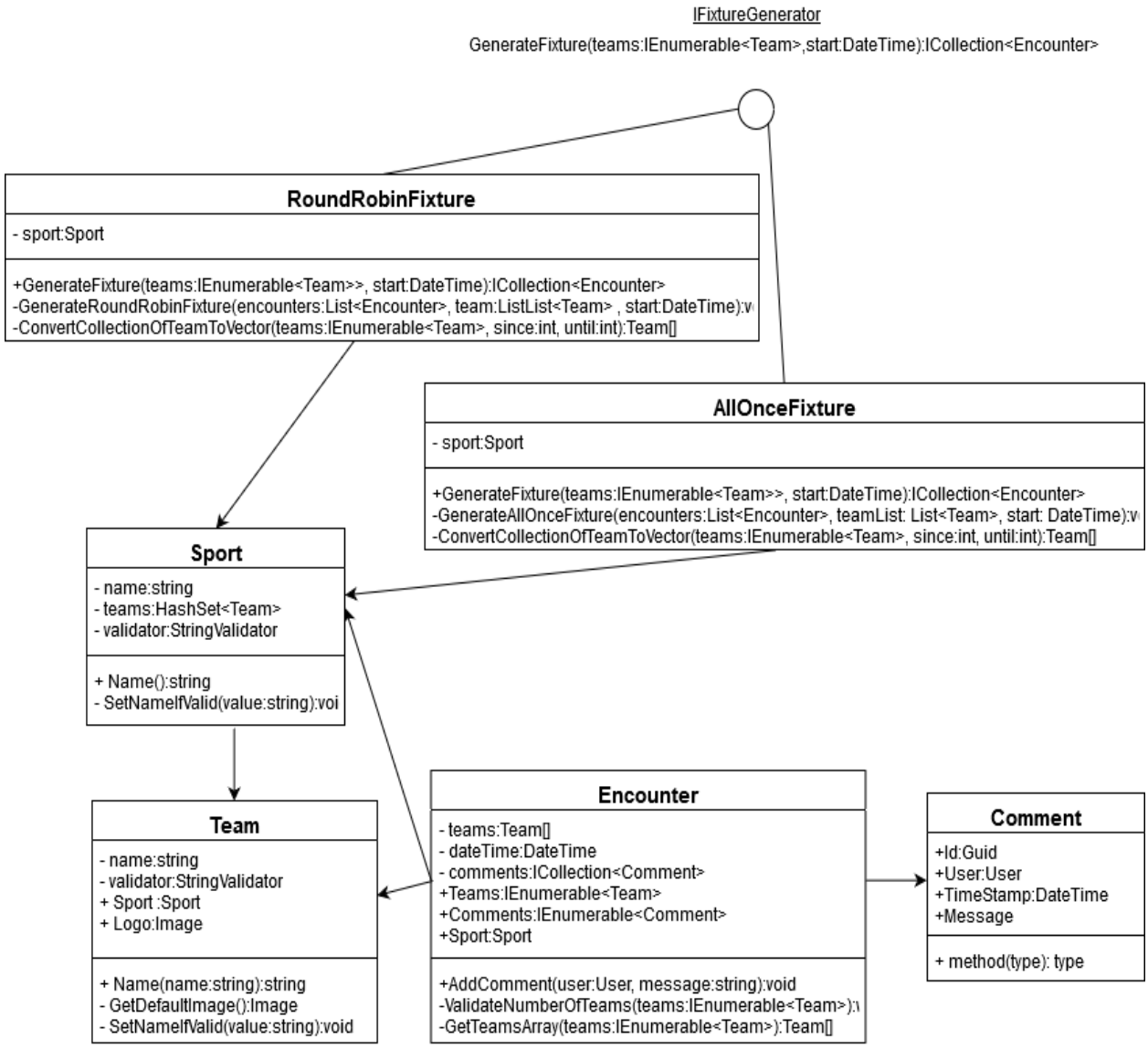


Diagrama de clases del subpaquete de dominio Fixture

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:

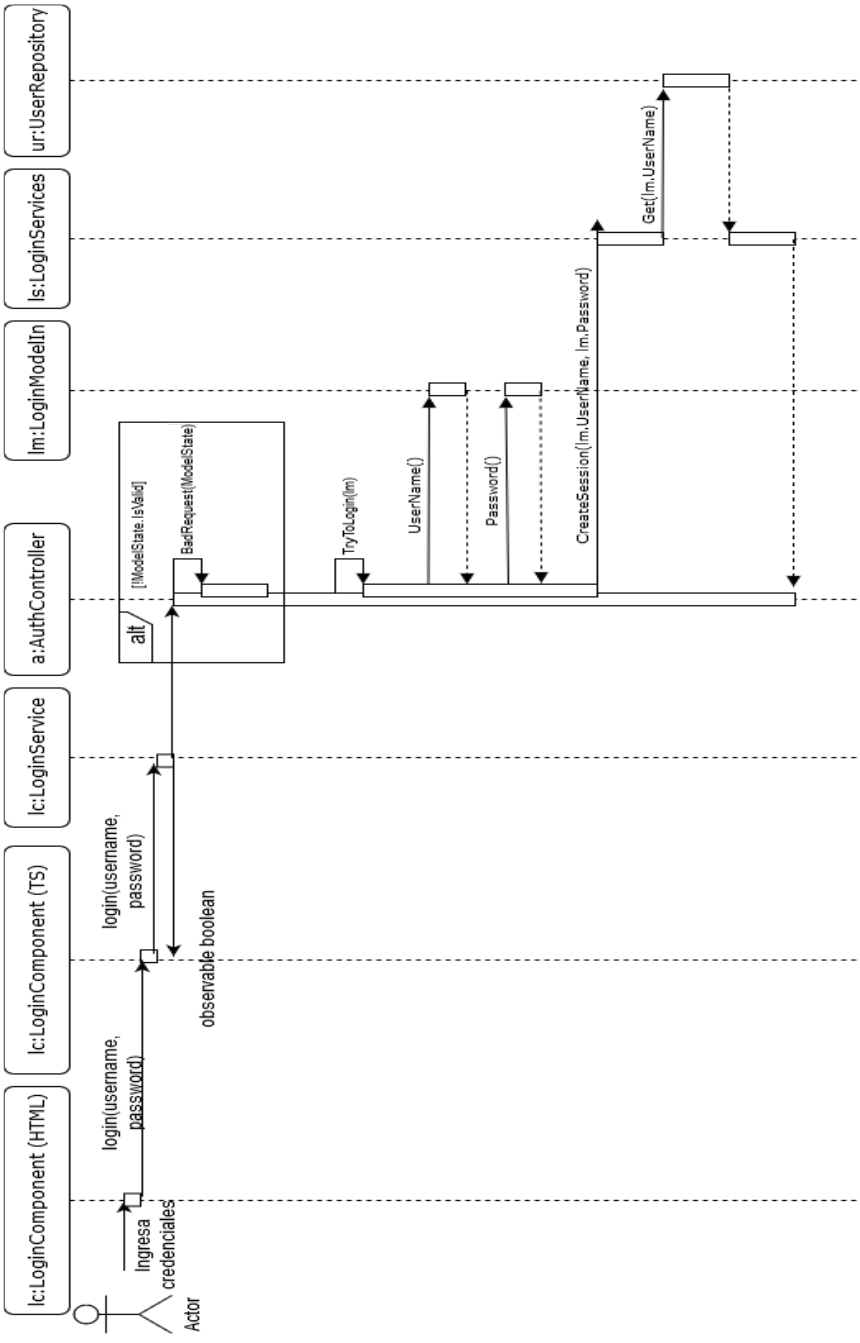


Diagrama de secuencia del proceso de autenticación.

El login service (Front-End) guarda el token y lo mantiene por la fecha de validez fijada en el mismo. Recordando la justificación de más arriba donde los servicios del backend también validan contra la base de datos que el usuario siga existiendo en el sistema. Supongamos que el usuario se dé de baja el mismo y aun mantenga el token, si no estuviese validado desde los servicios, un usuario que no existe podría seguir operando en la aplicación, ya que posee un token válido. Lo mismo ocurriría si un administrador de la aplicación borra al usuario, y dado que este tiene aún un token válido, si no se valida contra la base de datos, puede seguir operando aunque no exista.

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.

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

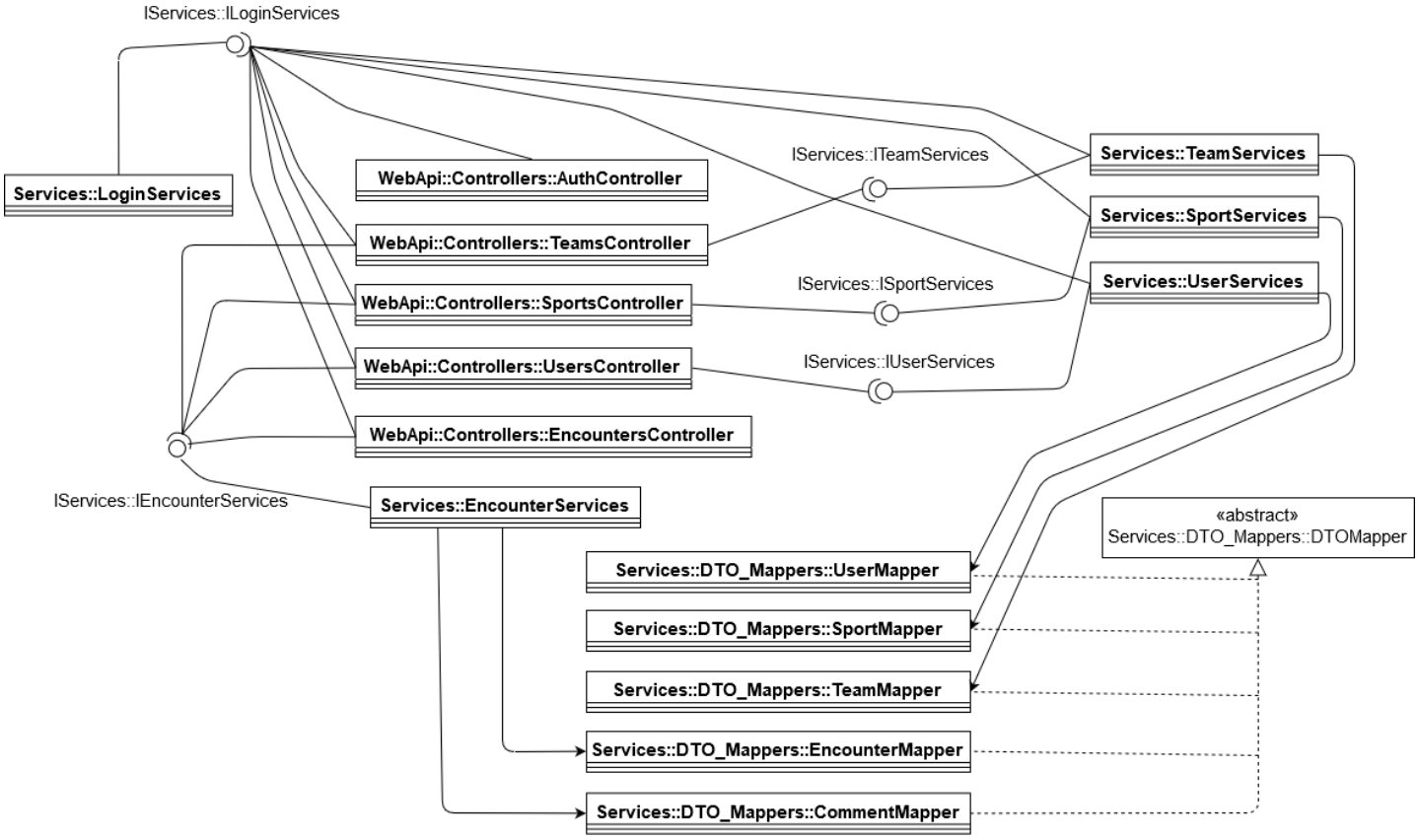


Diagrama UML que muestra las relaciones entre los paquetes WebApi, IServices y Services

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.

Cambios respecto a la primer entrega

Se separó el encounter services en tres clases.

- **EncounterSimpleServices**: Tiene la responsabilidad de realizar las operaciones CRUD sobre los encuentros.
- **EncounterExtendedServices:** Tiene la responsabilidad de realizar queries específicas para obtener encuentros.
- **Fixture Services:** Tiene la responsabilidad de crear fixtures.

También se agregó el PositionServices que tiene la responsabilidad de obtener las posiciones de los diferentes equipos.

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.

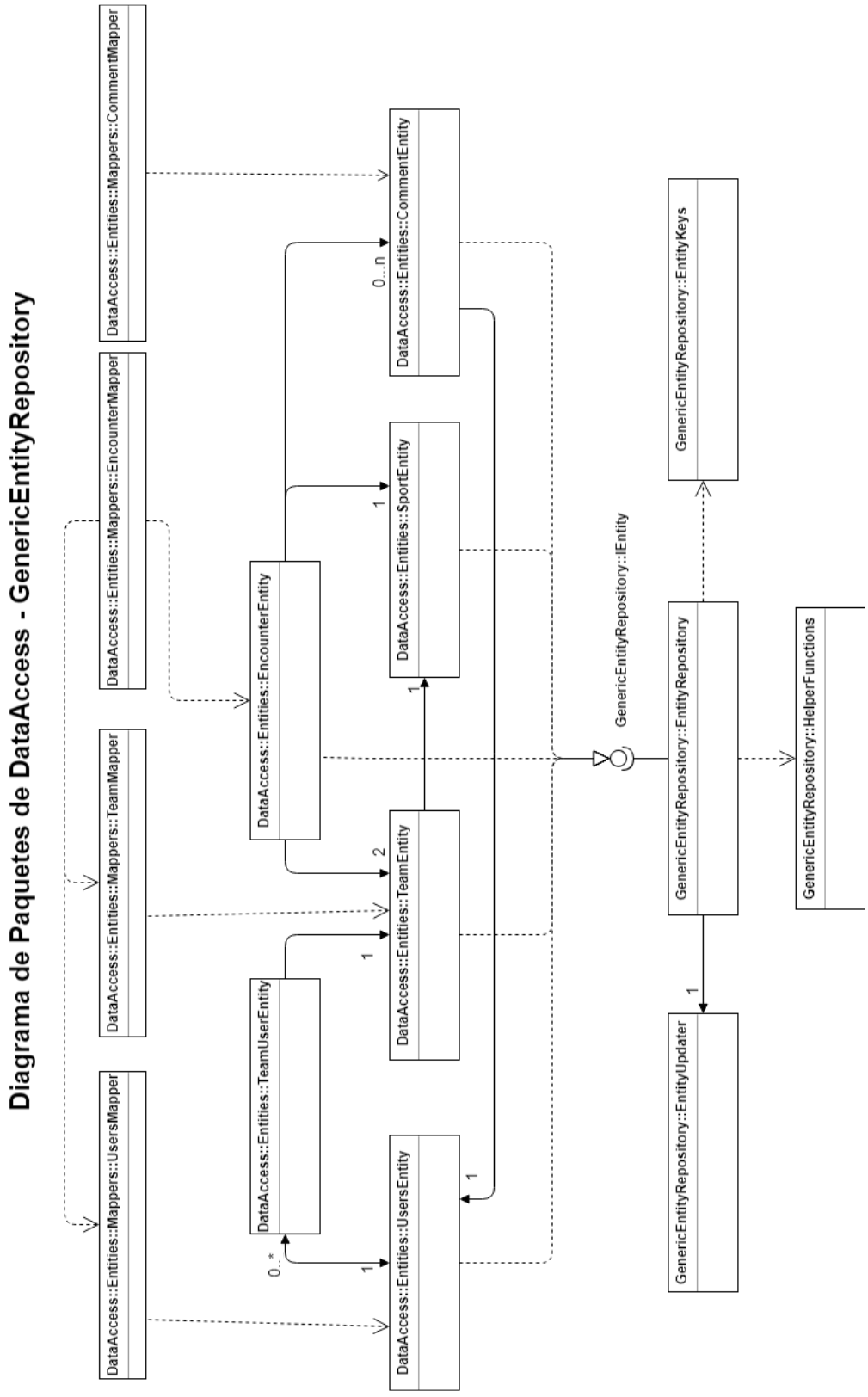


Diagrama de Paquetes de DataAccess - GenericEntityRepository
DataAccess::Entities::Mappers::CommentMapper
DataAccess::Entities::Mappers::EncounterMapper
DataAccess::Entities::Mappers::TeamMapper
DataAccess::Entities::Mappers::UsersMapper
DataAccess::Entities::CommentEntity
DataAccess::Entities::EncounterEntity
DataAccess::Entities::SportEntity
DataAccess::Entities::TeamEntity
DataAccess::Entities::TeamUserEntity
DataAccess::Entities::UsersEntity
0..n
1
2
0..*
GenericEntityRepository::IEntity
GenericEntityRepository::EntityKeys
GenericEntityRepository::EntityRepository
GenericEntityRepository::HelperFunctions
GenericEntityRepository::EntityUpdater

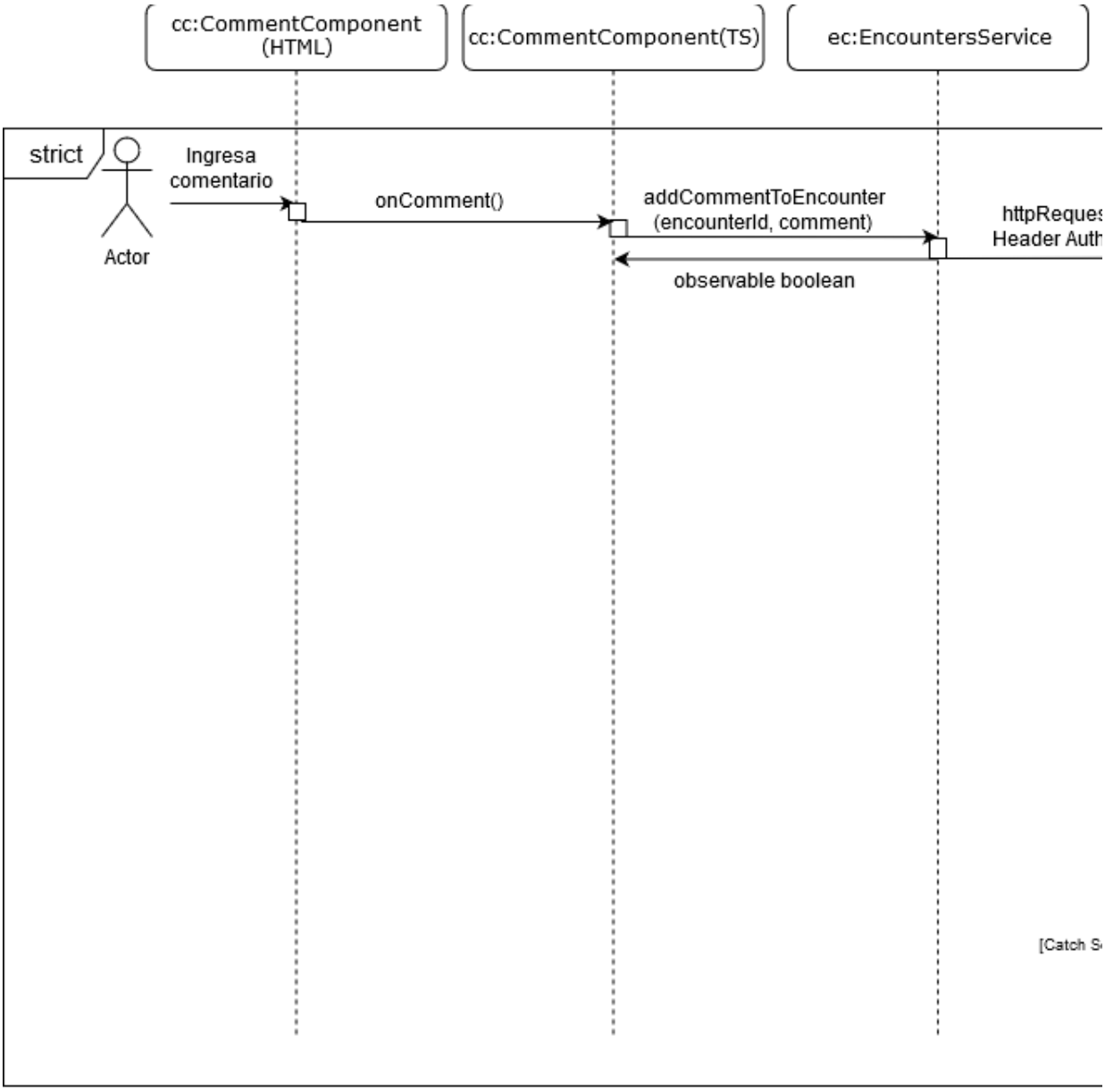


Diagrama de secuencia Agregar Comentario (continúa)

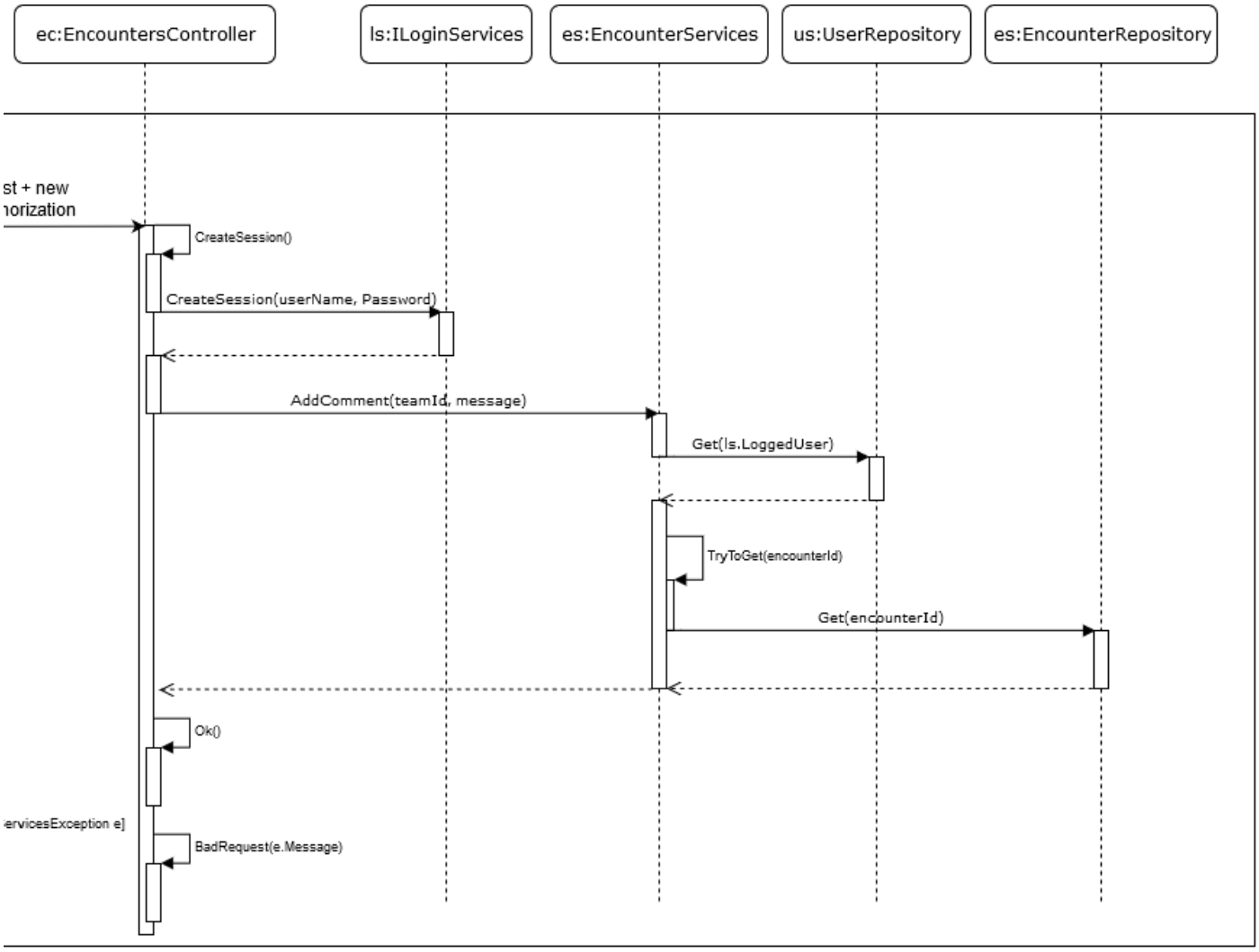
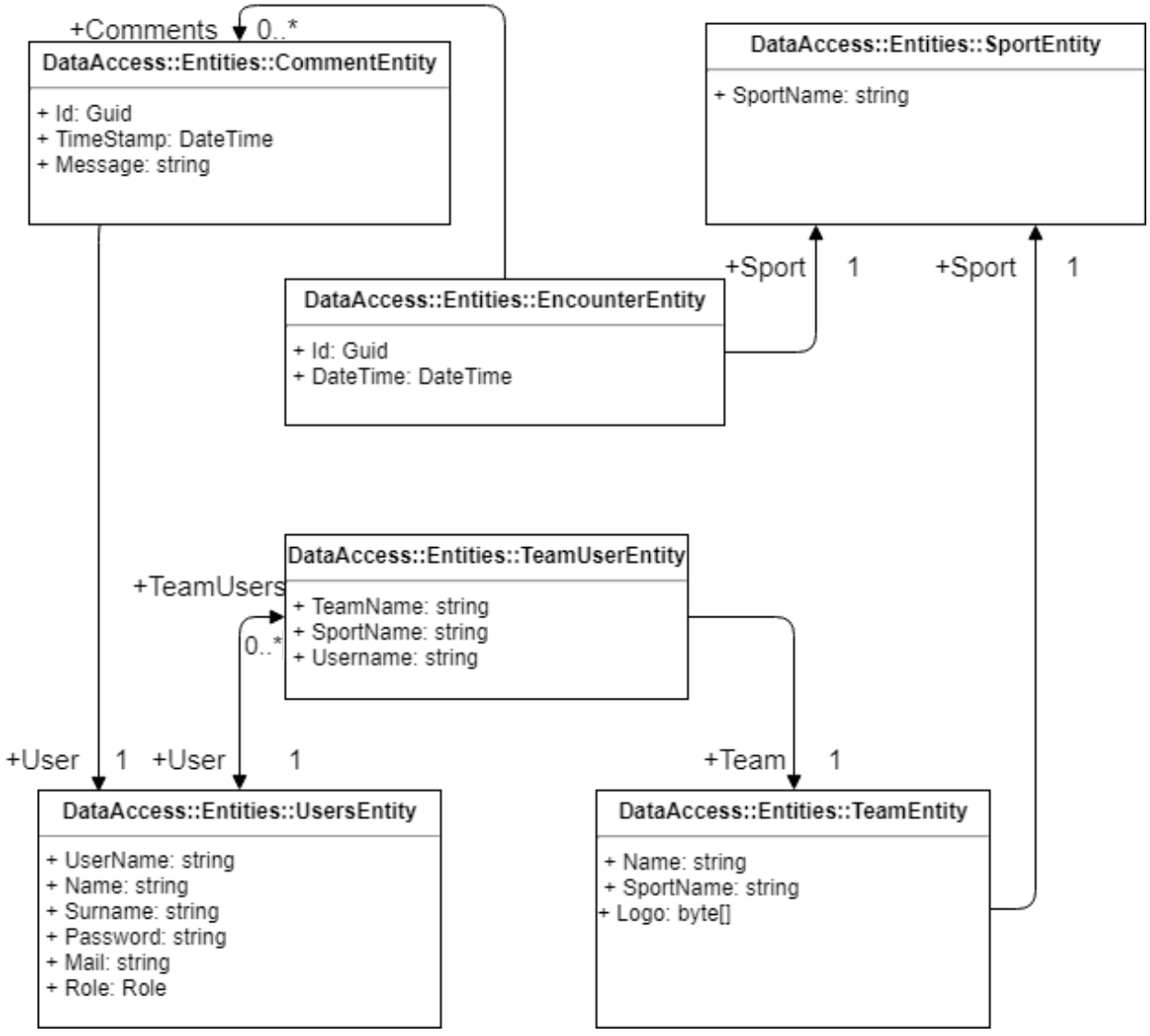


Diagrama de secuencia Agregar Comentario

Para agregar un comentario, se completa el formulario disponible para esto en la interfaz web de la aplicación de Front-End, luego se hace el request con el token authorization en el header la información del comentario es recibida por el EncountersController quién la transforma y la envía al EncounterServices, quién solicita al DataAccess el encuentro con ese Id, le agrega el comentario y guarda los cambios en la base. Si todo el proceso

anterior ocurre sin errores, el Front-End solicita todos los comentarios nuevamente, y recarga la lista.

Diagrama de Entidades de acceso a datos

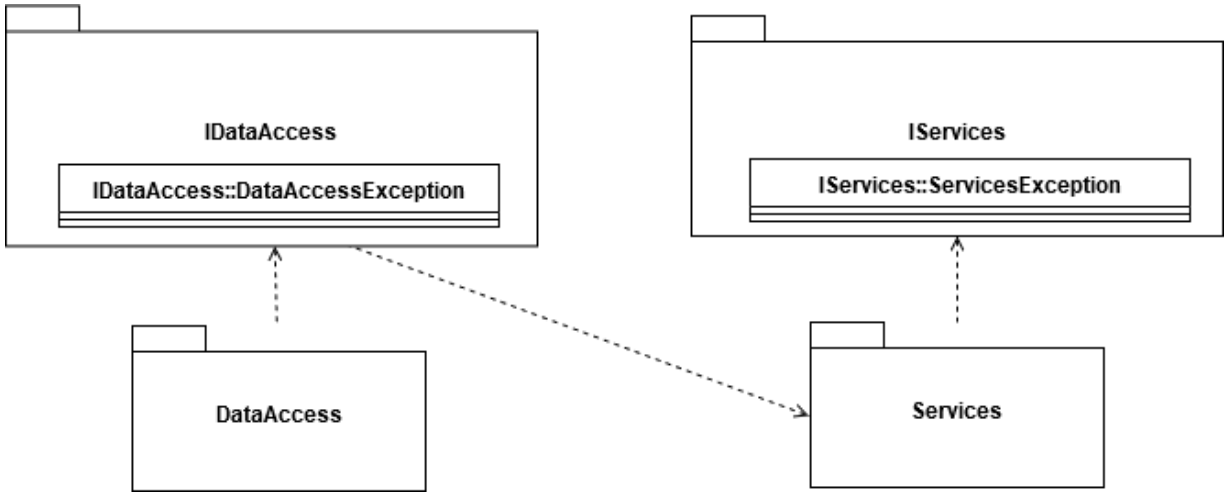


Universidad ORT Uruguay – Facultad de Ingeniería
Eusebio Durán (202741) - Nicolás Eirin (200111)

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.

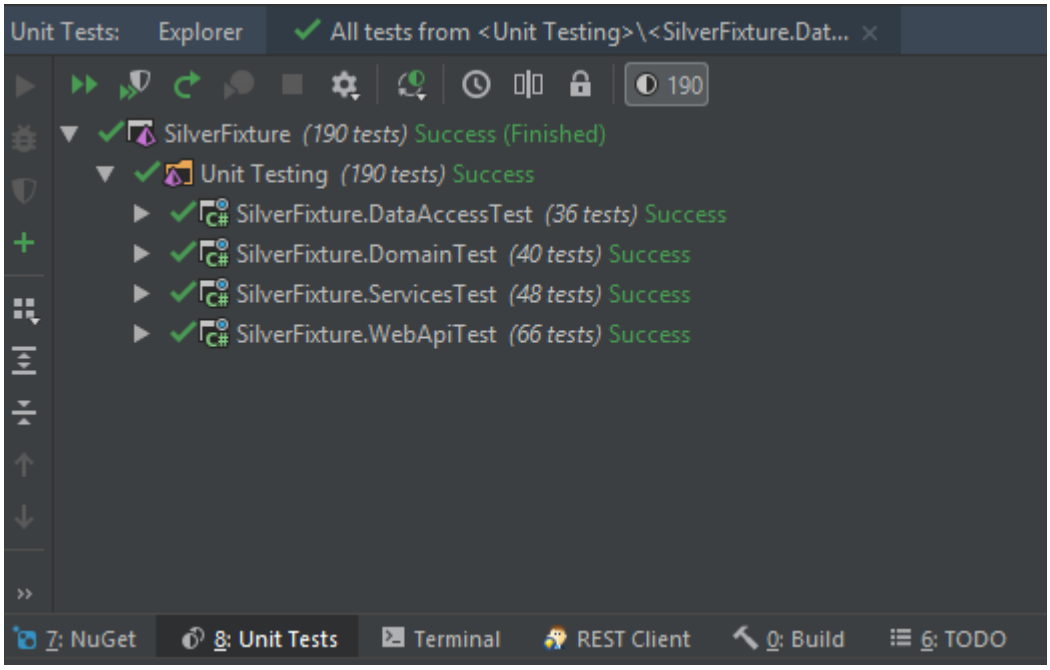


Excepciones principales en el sistema.

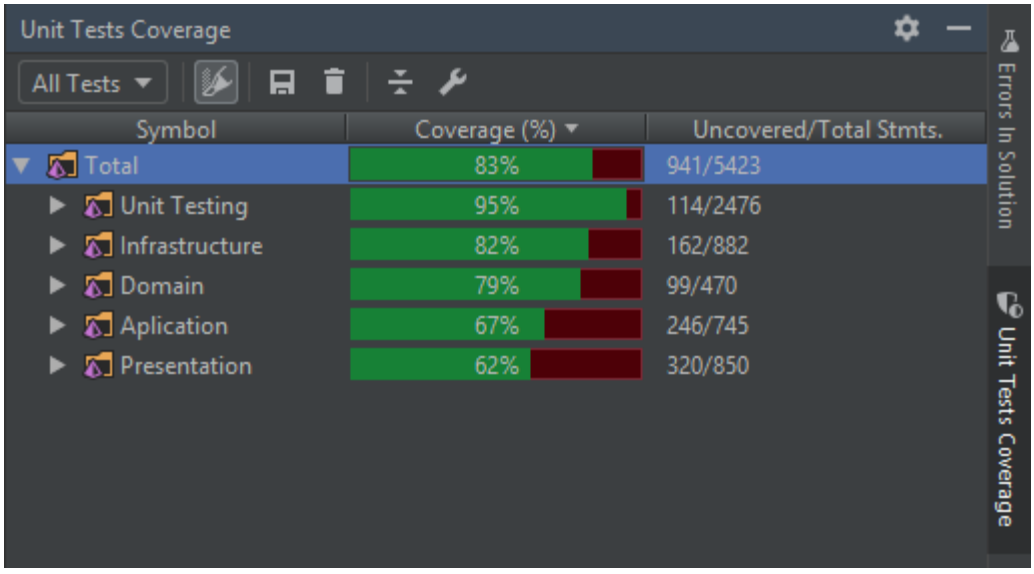
Cobertura de las Pruebas Unitarias

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.

Se realizaron 190 pruebas unitarias y también una colección de pruebas de postman para comprobar el correcto funcionamiento de los controllers.



Captura de pantalla de los tests implementados



Captura de pantalla de la cobertura de código