Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

FANTASYLEAGUE: APLICACIÓN WEB PARA CONVERTIRSE EN MÁNAGER ON-LINE DE FÚTBOL Y COMPETIR CON AMIGOS

Alumno: José Carpio Blanca

Tutor: Prof. D. José Ramón Balsas Almagro
Dpto: Informática

Junio, 2023

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don José Ramón Balsas Almagro , tutor del Trabajo Fin de Grado titulado:*"FantasyLeague: aplicación web para convertirse en mánager on-line de fútbol y competir con amigos"*, que presenta José Carpio Blanca, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Junio de 2023

El alumno: Los tutores:

José Carpio Blanca José Ramón Balsas Almagro

Índice de contenidos

Índice de ilustraciones

Índice de tablas

1. Introducción

El fútbol es el deporte que más seguidores tiene en el mundo[1]. Bajo esta premisa, no es ninguna novedad la existencia de gran cantidad de aplicaciones o juegos de este deporte ya sea con la intención de sacar beneficio de ello o simplemente por el amor a este deporte.

Realizar una aplicación web sobre este deporte posibilita una gran oportunidad de mercado debido a su demanda, a la vez que, al ser un mercado tan explotado, la aplicación debe de ser lo suficientemente única y diferente para no aburrir al cliente con aplicaciones redundantes centradas en el mismo tema.

El contexto al que va dirigido este trabajo es la realización de una aplicación web de carácter lúdico sobre fútbol donde se le dé la oportunidad al cliente de crear ligas con sus amigos, donde cada cliente pueda manejar un equipo conformado por los jugadores de una misma liga real, como veremos en los siguientes apartados.

Por poner de ejemplo otras aplicaciones parecidas que existen en el mercado, podemos citar a Laliga Fantasy Marca, gestionado por el diario Marca, o el Biwenger, gestionado también por otro diario deportivo como el AS. Este tipo de aplicaciones son parecidas al contexto del trabajo, aunque se centran solo en los jugadores de la liga española de fútbol.

1.1. Motivación o propósito

La idea general del trabajo, como se ha mencionado antes, es crear una aplicación web que permita a los usuarios crear ligas de fútbol para jugar con amigos o con otros usuarios de la aplicación.

El usuario podrá elegir 11 jugadores reales entre todos los clubes de la liga elegida para formar su plantilla. Una vez elegida la plantilla, tras cada jornada obtendrá una puntuación dependiendo del rendimiento real de los jugadores, siendo el objetivo del jugador maximizar esta puntuación.

Como cada semana se juega una jornada de cada liga (salvo casos especiales, como un parón de selecciones), tras acabar dicha jornada se revisarán las estadísticas de cada partido y se repartirán los puntos en función del rendimiento de cada jugador.

Por ejemplo, si en una jornada se juega un Barcelona - Real Madrid con resultado 0-1 con gol de Benzema, todos los jugadores del Barcelona recibirán una penalización de puntuación por perder, todos los jugadores del Real Madrid recibirán más puntos por ganar el partido, y en especial Benzema recibirá una puntuación extra por marcar el gol.

De esta forma se asigna una puntuación a cada jugador de cada club que va variando según su rendimiento real en cada jornada de liga, permitiendo a los usuarios de la aplicación poder cambiar su plantilla entre jornadas.

1.2. Objetivos

- Realizar un estudio de necesidades y soluciones para el contexto seleccionado.
- Diseñar un sistema informático especialmente orientado a la utilización de arquitecturas, principios de diseño y metodologías de desarrollo web.
- Seleccionar un entorno de desarrollo basado en tecnologías web e implementar un prototipo de aplicación web que satisfaga las necesidades que se hayan determinado.

1.3. Metodología

Las fases del diseño software de una aplicación son el análisis preliminar, donde se especifica el problema, el diseño del software, donde se especifica la solución, la implementación, las pruebas y el mantenimiento.

Existen diferentes metodologías que afrontan estas etapas del diseño de diferentes maneras, como lo son el desarrollo o ciclo en cascada, el desarrollo iterativo o el desarrollo ágil.

El ciclo de vida en cascada es el método más clásico para afrontar un problema informático, en el cual el proyecto progresa a través de una secuencia ordenada de pasos partiendo desde el análisis hasta las pruebas del sistema.

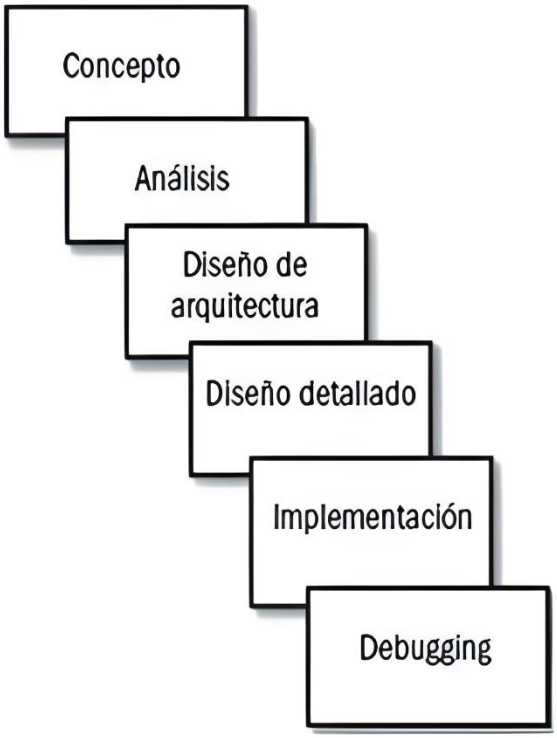

Ilustración 1: ciclo de vida en cascada [2]

Este tipo de metodologías no aceptan bien los cambios y los errores que se puedan producir en el proyecto y dependiendo de la rapidez en la que se detecte el error puede suponer un alto coste de tiempo en el desarrollo del proyecto. Además, hoy en día no suelen utilizarse en el desarrollo de proyectos, por lo que queda descartada.

La metodología escogida para el desarrollo del trabajo es la metodología basada en el ciclo de vida iterativo[3], que consiste en la iteración de varios ciclos de vida en cascada, que se repiten hasta obtener un producto que satisfaga al cliente.

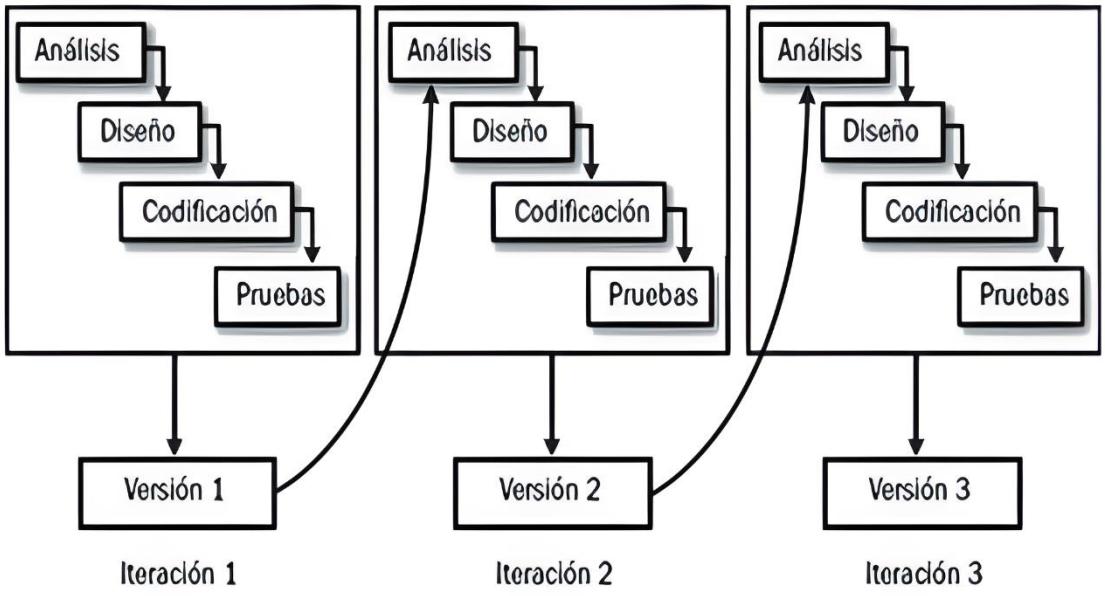

Ilustración 2: ciclo de vida en iterativo [2]

La elección de este tipo de metodología, respecto a por ejemplo una ágil, es que las metodologías ágiles son especialmente buenas cuando no están muy claros los requisitos del sistema y necesita una constante revisión por parte del cliente en el producto que se está desarrollando.

En este trabajo sucede una casuística inusual, que es la de que la empresa y el cliente son la misma entidad (en este caso la persona que realiza el trabajo de fin de grado), por lo cual no es necesario seguir una metodología ágil ya que, aunque puede haber cambios a lo largo del desarrollo del proyecto, no serán tan abultados al ser la misma persona la que analiza e implementa el sistema.

Por último, mencionar también la metodología elegida para el plan de pruebas, que será la metodología TDD, sobre la que más adelante en el documento se detallará con más profundidad.

1.4. Estructura del documento

Este documento se ha organizado de la siguiente manera: en el apartado 2 realizaremos toda la fase del análisis preliminar, así como un estudio de aplicaciones existentes en el contexto propuesto y de tecnologías candidatas para el desarrollo del proyecto.

También se hablará sobre la propuesta final de solución, detallando los requisitos que contendrá nuestro sistema, así como una planificación inicial del proyecto junto a un pequeño de estudio viabilidad y un modelo de dominio inicial.

En el apartado 3 se plantea todo el diseño del sistema, incluyendo diagramas entidad-relación, de clases o de secuencia entre otros, para finalizar mostrando el diseño de la interfaz propuesta y el plan de pruebas a desarrollar.

El apartado 4 tratará sobre todo el proceso de implementación del sistema informático, empezando por un diagrama de arquitectura que engloba todo el sistema para determinar su alcance, y terminando por un análisis exhaustivo que contenga los detalles más relevantes sobre la implementación del proyecto.

En el apartado 5 trataremos los resultados obtenidos por el plan de pruebas desarrollado en el apartado 3, así como las conclusiones en el apartado 6 en las que se analizarán entre otros aspectos el nivel de cumplimiento de los objetivos planteados y una algunas reflexiones sobre el proyecto abordado.

Para terminar, en el apartado 7 podemos ver todas las citas bibliográficas de referencia que permitan ayudar o ampliar toda la información o conocimiento que se trata durante el proyecto, junto a un par de apéndices que permitan instalar y guiar al usuario a lo largo del proyecto o sistema desarrollado, con el objetivo de apoyar y favorecer el entendimiento del mismo.

2. Análisis

Antes de empezar a analizar tecnologías, vamos analizar algunas aplicaciones existentes, como las ya nombradas en la introducción, donde podamos conocer ideas generales o inconvenientes que nos puedan ayudar a entender mejor el contexto y desarrollo de la aplicación.

2.1. Análisis preliminar de aplicaciones existentes

En primer lugar, me gustaría resaltar algunas ideas generales de LaLiga Fantasy de Marca[1], que al ser la única aplicación oficial de este estilo patrocinada por LaLiga, es de donde más ideas podemos sacar sobre un funcionamiento general.

En esta aplicación, cada jugador recibe un equipo aleatorio al iniciar una liga, y el mercado de fichajes se está actualizando constantemente cada día con 12 jugadores nuevos[4].

Durante este tiempo, los jugadores de una liga pueden pujar por los jugadores si están interesados en su compra. Quien consiga hacer la oferta más alta, obtiene el jugador. Si no se han realizado ofertas por el jugador, ya no se puede transferir hasta el próximo período de fichajes.

Los usuarios también pueden vender sus jugadores, donde el resto de jugadores podrán pujar por ellos, permaneciendo en el mercado como máximo tres días.

En caso de que los jugadores que quieres vender no sean pujados, el sistema realiza una oferta automática con una cantidad aleatoria entre un 10% de su valor por encima o por debajo.

Respecto a la alineación para cada jornada, ésta puede ser elegida un día antes de cada jornada y se puede editar a partir de la finalización de la misma.

En esta aplicación, los jugadores son puntuados en función de su desempeño real. Cada jugador, al acabar su partido, debe ser puntuado por un administrador del sistema.

[1] https://www.laligafantasymarca.com/

En LaLiga Fantasy no solo importa las acciones reales del jugador, si no que también tiene en cuenta las calificaciones dadas por periodistas en relación con el desempeño del jugador.

PUNTOS OFENSIVOS	
1	Partido Jugado
2	60' jugados o más
5	Gol de Portero o Defensa
4	Gol de Centrocampista
3	Gol de Delantero
3	Asistencias
1	Ocasiones Creadas
2	Penalti Provocado

PUNTOS DEFENSIVOS	
4	Portería a cero (POR/DEF)
2	Portería a cero (CEN)
1	Portería a cero (DEL)
5	Penalti Parado
1	Por cada dos paradas

PUNTOS NEGATIVOS	
-2	Penalti Fallado
-2	Gol en propia puerta
-2	Por cada dos goles recibidos POR/DEF
-1	Por cada dos goles recibidos CEN/DEL
-1	Por cada tarjeta amarilla
-3	Por cada tarjeta roja

PUNTOS BONUS	
1	Por cada dos remates a portería
1	Por cada tres regates
2	Por cada dos centros al área
1	Por cada cinco recuperaciones
-1	Por cada cinco pérdidas
1	Por cada cinco despejes

Ilustración 3: Sistema de puntuación de LaLigaFantasy [4]

Al finalizar la jornada, se dan 50 000€ por cada punto obtenido, que se puede utilizar más tarde para realizar ofertas o fichar jugadores.

Un ejemplo de una plantilla una vez terminada una jornada la podemos ver en la ilustración 4. Una vez acaba la temporada, el usuario que más puntos haya obtenido a lo largo de las jornadas es el ganador de la liga.

Ilustración 4: Ejemplo de equipo en LaLiga Fantasy [4]

En general, y aunque en este apartado solo vamos a ver exhaustivamente la anterior aplicación, el funcionamiento es similar para todas las aplicaciones de este tipo.

Podríamos poner de ejemplo una por una otras aplicaciones similares, pero todas se basan en el mismo funcionamiento: formas un equipo, gestionas tu plantilla para la siguiente jornada de liga, una vez acabados los partidos los jugadores son puntuados, el usuario recibe dinero dentro del juego a partir de esos puntos que puede usar para vender jugadores o intentar mejorar su equipo, y vuelta a empezar.

A pesar de que todas las aplicaciones centradas en esta temática funcionan de forma similar, ¿por qué hay tantas empresas interesadas en realizar este tipo de aplicaciones?

Una posible respuesta es la publicidad que te puede proporcionar, pues como ya hemos mencionado en algunos ejemplos en la introducción, muchas de estas aplicaciones son gestionadas por diarios deportivos, y tener una aplicación de este estilo puede atraer a más usuarios que quizás no sean lectores del periódico pero si usuarios de la aplicación, con lo cual consigues más retención de usuarios.

Otro motivo puede ser intentar fortalecer la marca y marcar presencia en el mercado, pues una aplicación de este estilo lo suficientemente única y de calidad puede aumentar la visibilidad de la empresa.

Por último, y aunque no creo que sea un motivo principal, también puede servir este tipo de aplicaciones para recopilar datos sobre las preferencias del usuario que permitan a la empresa personalizar los contenidos de la aplicación, o en el caso de LaLiga Fantasy de Marca, recomendar artículos que contengan los gustos y preferencias del usuario.

Sin embargo, en el mercado actual casi no existen aplicaciones de este tipo que unifiquen todas las ligas o deportes, puesto que la mayoría solo se centran en una liga nacional.

Por lo tanto, uno de los objetivos será unificar todas las ligas en una sola aplicación, de forma que si eres un gran seguidor de otra liga que no sea la de tu nación, puedas alternar entre jugar a tu liga nacional o cualquier otra liga sin cambiar de aplicación, sistema o sitio web.

Como conclusión, algunas ideas o requisitos a destacar de este análisis preliminar puede ser el sistema de gestión de plantillas, donde se seleccionan 11 jugadores y se valida dicho plantel antes de la jornada, así como el sistema de puntaje, que se adaptará a nuestra aplicación.

Es destacable también el sistema de ofertas a jugadores que ya están fichados por otros jugadores, que estará fuertemente inspirado por aplicaciones como ya mencionadas, pues algo que ya funciona de serie no hay por qué cambiarlo.

Respecto a los aspectos a descartar, tanto los menús como los sistemas de compra/venta de jugadores suelen ser complejos al realizarse en tiempo real, por lo que este sistema se realizará de cero.

Otro aspecto descartable también puede ser el uso de la publicidad, que se eliminará de nuestra aplicación.

Más información sobre este tipo de aplicaciones y su historia la podemos encontrar en [5]

2.2. Estudio de tecnologías

Antes de comenzar a analizar las tecnologías o enfoques con las que se pueden desarrollar el proyecto, me gustaría justificar el uso de una solución web frente a soluciones como una aplicación nativa o el uso de un CMS[2] generalista.

El motivo principal de la elección de una aplicación web se basa en la accesibilidad, puesto que una página web es accesible desde cualquier dispositivo con acceso a internet, lo que significa que los usuarios no necesitan descargar ni instalar nada en sus dispositivos. Algo similar pasa en las actualizaciones, pues los usuarios no tienen necesidad de instalar nuevas versiones de la aplicación.

Otro punto fuerte de las soluciones web frente a aplicaciones nativas o CMS es su flexibilidad y escalabilidad, puesto que una aplicación web especialmente adaptada podría soportar un mayor número de usuarios sin requerir una gran cantidad de recursos adicionales.

Estas ventajas junto a otras como el costo o mantenimiento, permite a las aplicaciones web estar un paso por delante frente a otras soluciones mencionadas para este tipo de proyectos, por lo que será la solución elegida para afrontar el problema.

[2] Content Management System (Sistema de gestión de contenido)

Mencionar también que no todos los proyectos tienen por qué tener una única solución, si no que son complementarias. Por ejemplo, se podría utilizar una combinación de soluciones, donde se utiliza una aplicación nativa para ofrecer una experiencia al usuario más rica y una solución web para permitir el acceso desde cualquier dispositivo conectado a internet.

Una vez definidos el problema a afrontar de forma breve, los objetivos y la metodología a utilizar, es hora de analizar qué tipo de arquitectura deberá usar nuestra aplicación. Para ello tenemos dos opciones: una arquitectura SSR[3] o CSR[4].

Una arquitectura SSR implica que el servidor genera la página HTML completa y la envía al cliente. Esto significa que la mayor parte del procesamiento y renderizado se realiza en el servidor antes de que se entregue la página al cliente.

No obstante, tiene varias limitaciones, como que los clientes deben ser implementados con el mismo lenguaje de programación que el usado en el servidor. Además, servidor y cliente deben ejecutarse en el mismo servidor de aplicaciones (como por ejemplo Payara, Tomcat etc).

Esto puede provocar que la capacidad de escalado sea limitada puesto que los clientes y el servidor deben ejecutarse en el mismo hardware y dentro del mismo servidor de aplicaciones. Además, no admite clientes en dispositivos móviles o implementados mediante frameworks 100% Javascript, como lo son ReactJs o Angular.

Una posible solución a estas limitaciones es la de pasar a un esquema distribuido o arquitectura CSR donde el servidor se comunica con cualquier potencial cliente a través de una API. Algunas ventajas de este tipo de arquitecturas son:

- El cliente puede estar en cualquier lenguaje de programación
- Permite usar cualquier framework en el front-end
- Permite usar una aplicación web, móvil o desktop

[3] Server-Side Rendering
[4] Client-Side Rendering

Destacar también que este tipo de enfoque está fuertemente relacionado con el paradigma SOA (Arquitectura Orientada a Servicios) que trata de construir aplicaciones a partir de un conjunto de servicios conectados a una red y que en la actualidad se suele materializar a través de una API basada en REST.

Antes de iniciar a enumerar posibles tecnologías o frameworks, también es importante señalar las ventajas e inconvenientes de la utilización de frameworks de desarrollo de software.

Un framework es una colección de software con un comportamiento genérico que puede ser personalizado para realizar un comportamiento específico. El objetivo no es otro que aunar principios de diseño, patrones y buenas prácticas en la implementación de aplicaciones con características o necesidades similares.

Sin embargo, su a veces excesiva curva de aprendizaje o gran acoplamiento de la aplicación (que supone dificultades para migrar a otros frameworks), lo que plantea ciertas limitaciones en la elección de un framework ideal para nuestro proyecto.

En cambio, un correcto uso del framework junto a un buen diseño de la aplicación puede minimizar los posibles inconvenientes que se nos planteen a lo largo de un proyecto.

2.2.1. VueJS

Empezando por los frameworks de desarrollo en la parte del cliente, VueJS[5] es un framework progresivo de programación en el cliente que utiliza conceptos exitosos de otros frameworks como Angular y React. Creada por Evan You en 2014 y con licencia de uso de tipo MIT, Vue se basa en el uso de HTML, CSS y Javascript.

Vue utiliza un renderizado declarativo basado en componentes para crear interfaces de usuario de forma rápida, agradable y sencilla. VueJS se encuentra en la categoría de aplicaciones de tipo SPA (Single Page Application), que se trata de un enfoque donde el navegador se descarga una versión mínima de .html junto a un .js que se encargará de controlar toda la web.

[5] https://es.vuejs.org

Algunas características generales de VueJS son las siguientes:

- La curva de aprendizaje es la más sencilla entre los frameworks más populares, como pueden ser React o Angular.
- Se trata, como ya he indicado antes, de un framework progresivo, lo que significa que es ideal para migrar y adaptar proyectos existentes hechos en otras tecnologías y pasarlos poco a poco a Vue.
- Vue mantiene un DOM virtual propio, lo que permite a la biblioteca determinar qué partes del DOM han cambiado comparando contenidos entre la versión nueva y la interna, y utilizando el resultado para determinar cómo actualizar eficientemente el DOM del navegador.
- Vue es un framework sin opinión, lo que significa que no te va a guiar por una forma única de hacer las cosas, a diferencia de por ejemplo Angular, que es más estricto en cuanto a las decisiones de arquitectura y te obligará en cierta forma a utilizar determinadas tecnologías o características.

Más información sobre Vue la podemos encontrar en [6,7]

2.2.2. ReactJS

ReactJS[6] es una biblioteca Javascript para crear interfaces de usuario. Desarrollado por Facebook en 2013 y con licencia de tipo MIT, React intenta ayudar a los desarrolladores a construir aplicaciones que usan datos que están cambiando continuamente.

React se compone de componentes reutilizables que conforman partes de la interfaz de usuario. Tener estos componentes facilita el desarrollo porque no tenemos que repetir código reiterativo, si no que bastaría con importar el componente en la parte del código donde se necesite.

Al igual que Vue, React también se encuentra en la categoría de aplicaciones de tipo SPA, lo que conduce a una renderización más rápida sin recargas de la página. Una característica de React es que su enfoque se suele centrar más en programación pura Javascript, utilizando HTML y CSS sólo como complementos.

[6] https://es.react.dev/

Otras características generales de React son:

- React es, posiblemente, el framework Javascript más popular del mundo. Esto implica tener una comunidad muy activa en la que puedes contribuir y obtener ayuda en caso de necesitarlo.
- React permite acelerar el desarrollo y un mayor rendimiento de las aplicaciones gracias a la reutilización de sus componentes.
- Al igual que Vue, React es otro framework sin opinión.
- React utiliza una extensión de la sintaxis de Javascript que amplía las características de ES6 llamada JSX. Esta extensión permite combinar la lógica y el marcado de Javascript en un componente.
- Como Vue, React también mantiene un DOM virtual propio, lo que provoca un mayor rendimiento en la aplicación.

Otras lecturas recomendadas sobre React y su funcionamiento las podemos encontrar en [8,9].

2.2.3. Spring Framework

Ahora vamos a analizar un par de frameworks de desarrollo en el lado del servidor, empezando por Spring[7].

Spring fue lanzado en el año 2002 por Rod Johnson bajo la licencia Apache 2.0, siendo una plataforma Java de código abierto. Spring se encarga de simplificar el desarrollo de aplicaciones, como se puede ver en [10]: *"Spring handles the infrastructure so you can focus on your application"*. Spring se basa en POJOs sin necesidad de introducir dependencias con la arquitectura usada, como Servlets o EJB.

A diferencia de frameworks en el cliente como Vue o React, Spring se considera un framework con opinión en el desarrollo de aplicaciones Java. Está basada en "convención" frente a configuración y utiliza una selección de bibliotecas de terceros y módulos spring para resolver problemas habituales. Spring es considerado el framework más popular para Java a nivel empresarial, y se puede considerar también como el padre de otros frameworks de este lenguaje de programación.

[7] https://spring.io/

Algunas características de Spring son:

- Permite tecnologías como la inyección de dependencias, validación de datos o conversiones de tipos.
- Soporte a DAO, ORM o JDBC.
- Gestión de transacciones.
- Implementación de rutinas transversales (AOP)
- Spring facilita la implementación de la seguridad en la aplicación con librerías específicas como Spring Security.

Más características e información sobre Spring la podemos encontrar en [10,11].

2.2.4. Java EE

Java EE (Java Enterprise Edition) es una plataforma de software para el desarrollo de aplicaciones empresariales en el servidor desarrollado por Sun MicroSystems (posteriormente adquirida por Oracle) en la década de 1990. Java EE es un software de código abierto, con una licencia de uso gratuito y una licencia comercial opcional para aquellos que desean soporte y actualizaciones.

Java EE proporciona un conjunto de especificaciones para el desarrollo de aplicaciones escalables y seguras. Entre sus características incluyen el uso de componentes como servlets, páginas JSP, JSF u EJBs entre otros, y tecnologías para la gestión de transacciones, seguridad o integración con bases de datos.

En 2017, Oracle anunció que iba a transferir la responsabilidad de Java EE a la fundación Eclipse. Desde entonces, la plataforma se ha renombrado como Jakarta EE[8] y su última versión operativa es la 10.0

Más información sobre Jakarta EE la podemos encontrar en [12]

2.2.5. MySQL

En programación es prácticamente inevitable trabajar con algún tipo de sistema de gestión de bases de datos. Cualquier programa que imaginemos tarde o temprano necesitará almacenar datos en algún lugar.

[8] https://jakarta.ee/

MySQL[9] es el sistema de gestión de bases de datos relacional más extendido en la actualidad al estar basada en código abierto. Desarrollada originalmente por MySQL AB, fue adquirida por Sun MicroSystems en 2008 y a su vez comprada por Oracle Corporation en 2010.

MySQL es un sistema que cuenta con una doble licencia. Por una parte es de código abierto, pero por otra, cuenta con una versión comercial gestionada por la compañía Oracle.

Algunas características de MySQL son:

- Puede manejar grandes volúmenes de datos y soportar miles de conexiones simultáneas, lo que la convierte en una opción adecuada para aplicaciones de alta carga por su gran escalabilidad.
- Está diseñado para ser rápido y eficiente gracias a su motor de almacenamiento InnoDB que es capaz de realizar operaciones de lectura y escritura de alta velocidad.
- MySQL admite la replicación y el clustering, lo que permite que los datos se distribuyan entre varios servidores para mejorar la disponibilidad y la tolerancia a fallos.
- Por último, cuenta con una gran comunidad de desarrolladores y usuarios que ofrecen soporte, documentación y recursos para ayudar a otros usuarios a utilizar el sistema de manera más efectiva posible.

Mencionar también que MySQL entra dentro de la categoría de bases de datos relacionales[13], que en resumen son bases estructuradas en tablas, normalizadas y que utilizan SQL como lenguaje de consulta.

2.2.6. NoSQL

Las bases de datos NoSQL surgen como respuesta a las necesidades de muchas aplicaciones web actuales: trabajar con grandes volúmenes de datos, satisfacer muchas transacciones por segundo y proporcionar bajos tiempos de respuesta.

[9] https://www.mysql.com/

Este tipo de bases de datos surgen en la década de los 2000, donde empresas como Google, Amazon o Facebook estaban lidiando con enormes cantidades de datos y necesitaban una forma más eficiente y escalable de almacenar y gestionar los datos.

NoSQL es un término que agrupa un conjunto heterogéneo de bases de datos que tienen en común no implementar el modelo relacional, y en general carecer de un esquema predefinido, como pueden ser MongoDB[10] o CouchDB[11].

Algunas características de este tipo de bases de datos son:

- A diferencia de las bases de datos relacionales que organizan los datos en tablas, NoSQL utiliza diferentes estructuras de datos como documentos, grafos o pares clave-valor.
- Los sistemas de gestión de bases de datos NoSQL están diseñadas para permitir añadir nuevos servidores de una manera sencilla (scale-out)
- Las bases de datos NoSQL están diseñadas para aprovechar múltiples servidores de bajo coste y disponen de estrategias de replicación que hacen que la caída de un servidor no paralice el sistema.
- Por su simplicidad, el rendimiento de las bases de datos NoSQL supera al de las bases de datos relacionales tradicionales.

Estas y otras más características sobre bases de datos NoSQL y sus diferencias con las bases de datos tradicionales las puedes encontrar en [14].

2.3. Propuesta de solución

Tras analizar todas las posibles soluciones y tecnologías con las que abordar este proyecto, es hora de elegir una posible solución, que no tiene por qué ser la perfectamente ideal, para la resolución del problema.

En primer lugar, vamos a empezar a detallar la arquitectura de la aplicación. Como ya se ha mencionado previamente, una arquitectura donde el cliente y el servidor están alojados en el mismo servidor de aplicaciones puede tener sus ventajas, aunque la solución más utilizada en el ámbito real y empresarial es la de un

[10] https://www.mongodb.com/
[11] https://couchdb.apache.org/

esquema distribuido, gracias a su gran flexibilidad donde cualquier cliente puede conectarse a tu servidor si sabe cómo hacerlo.

Esta arquitectura a su vez también tiene dos enfoques: un enfoque monolítico, donde la aplicación se desarrolla como una única unidad cohesiva e indivisible, en la que todas las funciones y componentes están interconectados y empaquetados juntos en una sola aplicación, y otro enfoque basado en microservicios, que se enfoca en descomponer una aplicación en servicios independientes, altamente cohesivos y escalables, que permiten una mayor agilidad y velocidad de desarrollo entre otras características.

El segundo enfoque permite conectar nuestro backend incluso con diferentes bases de datos, donde un microservicio puede estar conectada a una base de datos NoSQL y otro microservicio a una MySQL, lo que permite una mayor flexibilidad en la aplicación a la hora de gestionar los datos.

Aunque hoy en día están de moda los microservicios en el mundo empresarial, nuestra elección es la de seguir inicialmente un enfoque monolítico, pues donde se llega a ver realmente el rendimiento y potencial de enfoques basados en microservicios suelen ser en aplicaciones con grandes volúmenes de datos, donde la aplicación maneja todo tipos de datos e incluso recurren a distintas bases de datos de almacenamiento.

En nuestra aplicación esta casuística no se da, además de que no contiene un excesivo volumen de datos (que además pueden ser actualizados año a año) ni un modelo de dominio lo suficientemente complejo para usar microservicios. Por esto se ha decidido que una arquitectura monolítica es más que suficiente, pues además suelen ser más fáciles de implementar y desarrollar.

Por lo tanto, la arquitectura escogida para afrontar el proyecto será la de un enfoque distribuido monolítico.

Una vez escogida la arquitectura, es hora de elegir los tipos de frameworks o tecnologías que me acompañarán en el desarrollo.

En la parte del cliente, se han analizado los dos frameworks que más o menos están siendo más utilizados a nivel global. Sin embargo, aquí la experiencia del desarrollador juega un papel clave, ya que se ha trabajado previamente con Vue pero no con React. Con el objetivo de no tener una curva de aprendizaje demasiado elevada al principio del proyecto, el desarrollo de software en el cliente será con el framework VueJS.

En la parte del servidor no sucede lo mismo que en la parte del cliente, pues el desarrollador ha tenido la posibilidad de trabajar tanto con Spring como con Jakarta EE.

No obstante, como se ha mencionado en el análisis, Jakarta no es un framework en sí, por lo que puede dar más complicaciones en la configuración, por ejemplo en la capa de persistencia. Por detalles como este y porque la experiencia con el framework Spring al fin y al cabo ha sido más profunda que con Jakarta EE, el framework de desarrollo en el lado del servidor será Spring Framework.

Finalmente, solo queda una cosa por detallar: la base de datos a utilizar. Las bases de datos NoSQL se han ganado una gran popularidad en los últimos años gracias al gran rendimiento que ofrecen.

A pesar de ello, y como sucede con los enfoques basados en microservicios, que algo se vuelva popular no significa que tenga que ser usado a la fuerza, y al considerar que la aplicación a desarrollar es relacional por naturaleza debido a que el uso de claves tanto primarias como foráneas están muy presentes entre las consultas de la aplicación, la base de datos elegida será MySQL.

En resumen, la propuesta de solución elegida es la de realizar una aplicación utilizando un esquema distribuido monolítico, que combina el uso de VueJS como framework en la parte del cliente y Spring como framework en la parte del servidor, y el uso de una base de datos relacional MySQL.

2.4. Requisitos del sistema

Detallada la propuesta de solución, es hora de comenzar a enumerar los requisitos que tendrá nuestro sistema. Como estamos usando una metodología clásica, se ha tomado la decisión de definir el sistema mediante requisitos debido a que esta forma es más detallada y más específica que por ejemplo hacerla mediante historias de usuarios.

Una metodología clásica también se acopla perfectamente a esta especificación mediante requisitos gracias a su trazabilidad, donde los requisitos se documentan de forma estructurada y se le asignan un identificador único.

A continuación, vamos a enumerar tanto los requisitos funcionales, donde se tratan acciones o comportamientos que debe tener un sistema para satisfacer las necesidades o expectativas del cliente (se centran en el “qué”), como los no funcionales, que se centran en el “cómo” el sistema debe hacerlo y otros aspectos como las características o restricciones del sistema.

2.4.1. Requisitos funcionales

RF01 – El usuario debe poder iniciar sesión utilizando su usuario y su clave en la aplicación.

RF02 – El usuario debe poder crear una liga (cualquiera entre la española, inglesa, alemana, francesa o italiana) dentro de la aplicación.

RF03 – El usuario debe poder ver la clasificación de una liga en la que participa en cualquier momento.

RF04 – El usuario debe poder unirse y buscar una liga dentro de la aplicación

RF05 – Los administradores deben ser capaces de actualizar los resultados de los partidos de cada jornada, y de actualizar las estadísticas de los jugadores de dicho partido.

RF06 – Para que una liga funcione correctamente, debe de haber un mínimo de 2 personas en ella y un máximo de 20 personas.

RF07 – Todos los usuarios al empezar una liga contarán con un presupuesto inicial de 300 millones para crear su primera plantilla.

RF08 – Los usuarios deben poder buscar un jugador por su alias (en ocasiones, hay jugadores que no se les conoce por su apellido o nombre. Por ejemplo, el alias de Leo Messi es “Messi”, pero el alias de Pablo Gavilán es “Gavi” o de Pedro González es “Pedri”).

RF09 – Los usuarios deben poder fichar jugadores para su plantilla, de forma que si el jugador está libre su precio será igual a su valor de mercado. En caso de que el jugador ya pertenezca a otro jugador, éste deberá mandar una oferta por él y el otro jugador podrá aceptar o rechazarla. Dicha oferta no podrá superar el 125% del valor del jugador

RF10 – El usuario podrá fichar un jugador que pertenezca a otro usuario activando su cláusula de rescisión, de forma que el usuario paga el precio de la clausura y no necesita negociar con el otro jugador. Esta cláusula nunca podrá superar una cifra superior al 125% del valor del jugador.

RF11 – A la hora de vender jugadores, un usuario podrá vender su jugador a un usuario o al propio mercado. En caso de venderlo al mercado, obtendrá un 80% de ganancias del valor de traspaso del jugador. Si un jugador está suspendido o lesionado, el usuario puede venderlo por el mismo valor de mercado.

RF12 – Los usuarios deben tener su plantilla validada un día antes del inicio de la jornada si quieren puntuar en dicha jornada. Esta validación comprobará que todos los jugadores de la plantilla están disponibles para jugar y de que hay 11 jugadores en el plantel.

RF13 – Los administradores deben ser capaces de finalizar la jornada una vez se ha puntuado el último partido real para que los usuarios puedan ver su puntuación.

RF14 – Al finalizar la jornada, los usuarios que hayan puntuado positivamente, obtendrán un presupuesto de valor ((puntuación / 2) * 10), de forma que si un usuario puntúa con 6 puntos en una jornada, obtendrá 30 millones.

Los usuarios que han puntuado negativamente o hayan obtenido una puntuación de 0 puntos, siempre obtendrán 60 millones para intentar puntuar mejor la próxima jornada.

RF15 – Cada jugador tendrá una puntuación al finalizar una jornada de liga, que se regirá según la siguiente tabla:

Parámetro	Condición	Puntaje
Gol	Cualquier jugador	+2
Asistencia	Cualquier jugador	+1
Victoria	El equipo gana el partido	+1
Derrota	El equipo pierde el partido	-1
Portería imbatida	El portero consigue que no le metan gol	+2
Intercepciones	Cada 3 intercepciones exitosas	+1
Regates	Cada 3 regates exitosos	+1
Tarjeta amarilla	El jugador recibe una tarjeta amarilla	-1
Tarjeta roja	El jugador recibe una tarjeta roja	-2
Goles encajados (portero)	Cada gol encajado	-1
Goles encajados (defensa,centrocampista o delantero)	Cada 2 goles encajados	-1
Penalty	El jugador falla un penalty	-2

Tabla 1: Tabla del requisito RF15 sobre puntuaciones

2.4.2. Requisitos no funcionales

RNF01 – Se utilizará una base de datos relacional para almacenar la información de la aplicación.

RNF02 – La interfaz se desarrollará siguiendo un diseño responsive con el fin de garantizar la correcta visualización en diferentes dispositivos.

RNF03 – La interfaz debe ser sencilla y vistosa, de forma que no se usen más de 3 gamas de colores.

RNF04 – El sistema debe proporcionar mensajes de error que sean informativos y orientativos para la comprensión del usuario.

2.5. Planificación inicial de tareas

Como la metodología elegida es una metodología clásica iterativa, el desarrollo de la aplicación se distribuirá en tres iteraciones lo más homogéneas posibles *(Véase la tabla 2)*.

El trabajo de fin de grado supone 12 créditos, que son aproximadamente 300 horas. La elección de tres iteraciones para el desarrollo está condicionado por este tiempo de trabajo, donde en cada iteración se trabajará aproximadamente 100 horas (aunque este tiempo puede variar dependiendo de las dificultades que se presenten).

En la primera iteración se van a implementar todo lo referente con la creación de ligas, así como su almacenamiento, la lógica de unión de los usuarios a una liga, etc. En resumen, los requisitos que se implementarán son los requisitos 2, 4, 7, 14, 6 y 3

En la segunda iteración trabajará en la interacción de los usuarios con los jugadores dentro de una liga, como sus fichajes, ventas o gestión de plantillas de cada usuario. Estos son los requisitos 9, 11, 8 y 10

En la última iteración se ha implementado todo lo referente a la seguridad de la aplicación, sistema de puntajes o comprobaciones de plantillas, o lo que es lo mismo, se implementarán los requisitos 1, 5, 15, 13 y 12.

En resumen, la planificación en iteraciones ha quedado de la siguiente manera (marcada con una X en la iteración que se ha completado el requisito):

Nº de requisito	Iteración 1	Iteración 2	Iteración 3
RF01			X
RF02	X		
RF03	X		
RF04	X		
RF05			X
RF06	X		
RF07	X		
RF08		X	
RF09		X	
RF10		X	
RF11		X	
RF12			X
RF13			X
RF14	X		
RF15			X

Tabla 2: Planificación del desarrollo en iteraciones

Nótese como los requisitos no funcionales no están indicados en la tabla, ya que son requisitos más generales que se han ido desarrollando en todas las iteraciones.

Por ejemplo, no hay una iteración donde se haya implementado un interfaz responsive y en otra no, si no que se ha implementado una interfaz responsive en todas las iteraciones donde ha hecho falta crear una nueva interfaz dentro de la aplicación.

Con la anterior tabla tenemos la planificación de las iteraciones donde se desarrollarán los requisitos, pero en este proyecto no solo se ha desarrollado una aplicación, si no que, como en todos los proyectos, se pierde un tiempo en el estudio del problema y todo el análisis preliminar que ya he indicado.

Todo el proceso de planificación del proyecto se puede ver en la ilustración 5, donde se puede ver el tiempo que se empleará en el análisis desde que se asignó este proyecto, hasta que se empiece a implementar el desarrollo indicado en la tabla 2.

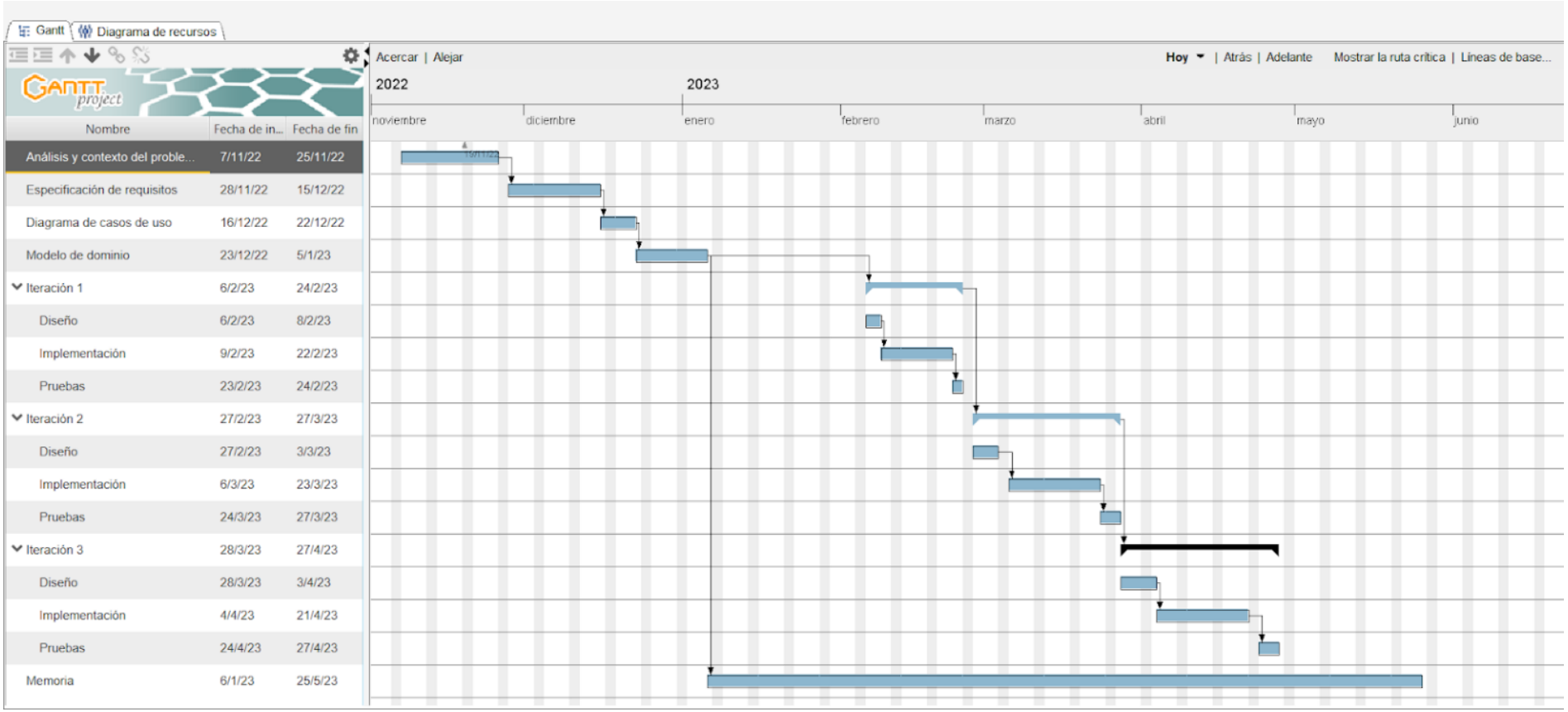

Ilustración 5: Diagrama de Gantt del proyecto

Como se puede apreciar, desde noviembre (mes de comienzo) hasta los siguientes dos meses se realizará todo el análisis previo que incluyen todo el estudio de tecnologías que he indicado antes, así como el análisis del problema que incluyen diagramas que veremos más adelante como puede ser el de dominio o el de clases. Posteriormente, los meses restantes se dedicarán al desarrollo del proyecto.

2.5.1. Evolución de la planificación inicial

Analizando en retrospectiva la evolución del proyecto una vez realizado el mismo, de enero a febrero se realizó un parón, debido a la convocatoria de exámenes ordinaria 1, para empezar desde febrero hasta finales de marzo con el desarrollo de la aplicación. Importante recalcar también que durante esta etapa no hubo ninguna variación significativa en la planificación, pues se implementaron las iteraciones satisfactoriamente tal y como estaba planeado.

Como resultado, el proyecto se realizó según la planificación inicial +7, es decir, con una semana de retraso, por lo que se finalizó en mayo, lo que supone 4 meses completos de desarrollo.

En conclusión, la planificación inicial fue bastante acertada (cosa que no siempre ocurre) y el tiempo y las fases se han ajustado a las expectativas iniciales de estimación.

2.6. Estudio de viabilidad

Para el estudio de la viabilidad del proyecto se ha realizado un análisis previo de costes que incluyen el coste de licencias o suscripciones de aplicaciones software, el salario de los empleados y el costo de infraestructuras y hardware.

Como ya se ha mencionado antes, el proyecto se ha realizado con el framework VueJS en la parte del cliente y el framework Spring en la parte del servidor, además del uso de la base de datos MySQL.

También hay que tener en cuenta el uso de aplicaciones externas que sean usadas durante el desarrollo de todo el proyecto, es decir, desde el análisis hasta la implementación. En este caso, se va a tener en cuenta la aplicación Visual Paradigm[12] que ha sido la que se ha utilizado para desarrollar los diferentes diagramas que van a ser usados durante el desarrollo.

[12] https://www.visual-paradigm.com/

VueJS utiliza el tipo de licencia de tipo MIT. Este tipo de licencia permite a los usuarios utilizar, modificar y distribuir el software bajo ciertas condiciones, pero en general es una licencia de software libre y de código abierto, por lo que no supone ningún costo adicional el uso de este framework para el proyecto.

En el caso de Spring framework, éste utiliza la licencia Apache 2.0, que es una licencia de software libre y de código abierto que permite a los usuarios distribuir el software de Spring de forma gratuita, por lo que tampoco supone un costo adicional.

Para MySQL existen dos tipos de licencia, una licencia GPL que permite a los usuarios distribuir el software de forma libre, y otra comercial para usuarios que deseen utilizar el software MySQL de forma que no cumpla con los términos de la licencia GPL, como puede ser integrar MySQL en una aplicación comercial propietaria sin necesidad de distribuir el código fuente o bajos los propios términos de la empresa.

También existe otra opción similar y gratuita para la producción, mariaDB[13], que se trata de una base de datos relacional de código abierto y una bifurcación de MySQL, que mantiene la compatibilidad con la mayoría de aplicaciones y herramientas desarrolladas para MySQL y puede ser una alternativa confiable y de alto rendimiento.

En nuestro caso, con la licencia GPL de MySQL será suficiente, por lo que tampoco supone ningún gasto extra.

Por último, es hora de analizar el coste de uso de una aplicación que ayude el análisis y desarrollo del software, como puede ser Visual Paradigm. Durante el desarrollo del trabajo, esta herramienta se ha utilizado bajo la licencia académica que proporciona la Universidad de Jaén para su estudiantado, pero en el caso de abarcar este proyecto bajo un ámbito empresarial, lo ideal sería obtener una subscripción de la misma. En este caso, se ha decidido que con la licencia estándar de Visual Paradigm, con un costo de 349€/año, sería suficiente.

[13] https://mariadb.org/

Una vez analizado los costes de licencias y productos software, es hora de decidir el equipo de trabajo. Lo ideal sería contratar a un analista de software que permita analizar y desarrollar una solución al problema lo suficientemente adecuada para su posterior desarrollo. Basándome en el convenio colectivo de Telefónica publicado en BOE[15], el salario de un analista es de 2277€/mes.

Asimismo, es necesario contar con un equipo de desarrollo que permita implementar correctamente la aplicación detallada por los analistas. En este caso, el equipo de trabajo está desarrollado por una sola persona, pero lo normal en un ámbito empresarial es que el equipo de trabajo lo compongan un grupo de personas. El sueldo de un desarrollador, según el convenio anteriormente mencionado, es de 2277€ que será pagado durante 4 meses que ha durado el desarrollo.

Por último, solo quedan especificar las infraestructuras o equipos hardware que serán usados para la correcta implementación del proyecto. Lo adecuado sería contar con un equipo informático lo suficientemente capaz que ayuden al equipo de trabajo a conseguir los objetivos sin muchos imprevistos. Para este proyecto, se ha elegido el portátil MSI Bravo 15[14] por el coste de 991€, aunque esta elección puede ser a gusto del comprador.

Finalmente, vamos a elegir la infraestructura hardware donde se almacenarán los datos de la aplicación. Este servidor puede ser físico, que suelen ser más costosos pero ofrecen un mayor rendimiento, o servidores virtuales o en la nube, que suelen ser opciones más económicas si tu aplicación no contiene un gran volumen de datos.

Por ese motivo, me he decantado por la segunda opción, en específico por el servicio que ofrece DigitalOcean[15] de servidores en la nube que cuestan 5€/mes.

DigitalOcean es un proveedor de servicios de infraestructura en la nube que permite a los desarrolladores implementar y escalar aplicaciones en servidores virtuales en la nube. En general, DigitalOcean destaca por su facilidad de uso, escabilidad, una comunidad activa de desarrolladores y sobre todo por sus precios asequibles, que ha sido el principal motivo de elección del servicio.

[14] amzn.to/3ovyWy9
[15] https://www.digitalocean.com/pricing/app-platform

Este servicio no es necesario en la etapa de desarrollo, pero sería interesante asumir este coste para experimentar con un entorno de pruebas en un proveedor en particular. Lógicamente en producción se debería elegir un proveedor concreto (que no tendría por qué ser este) y que probablemente tenga prestaciones y precios más superiores.

Una vez analizados el coste de cada servicio a utilizar, hay que prorratear el coste a el tiempo de uso que realmente tiene el producto que compramos.

Esto se debe aplicar tanto para las licencias como para el equipo informático que adquiramos. Como la única licencia de uso comercial es la de Visual Paradigm, debemos prorratear su coste, que en resumen sería 350 / 12 = 29’17€/mes * 6 meses = 175€. Respecto al equipo informático, si consideramos una vida útil estimada de 4 años (48 meses) el cálculo sería: 911 / 48 = 18’98€ * 6 = 113’875€.

En resumen, el presupuesto y costes del proyecto quedaría de la siguiente manera:

Herramienta	Número	Coste individual	Coste total
VueJS	1	0€	0€
Spring Framework	1	0€	0€
MySQL	1	0€	0€
Visual Paradigm	1	350€	175€
Analista/Programador	1 * 4 meses	2277€	9108€
Equipo informático	1	991€	113.875€
DigitalOcean	1 * 4 meses	5€	20€
Total			9416.875€

Tabla 3: Tabla de costes del proyecto

Como en este proyecto el analista y programador es la misma persona, se ha contado por igual durante los meses que ha durado el proyecto en su totalidad (análisis y desarrollo).

En total, se quedaría en 9416.875€, en el cual 175€ se destinan a software, 133.875€ se destinan a hardware, y 9108€ se destinan al salario del equipo de trabajo.

2.7. Modelo de dominio

Teniendo en cuenta los requisitos mencionados en el punto 2.3, el modelo de dominio se podría quedar de la siguiente manera:

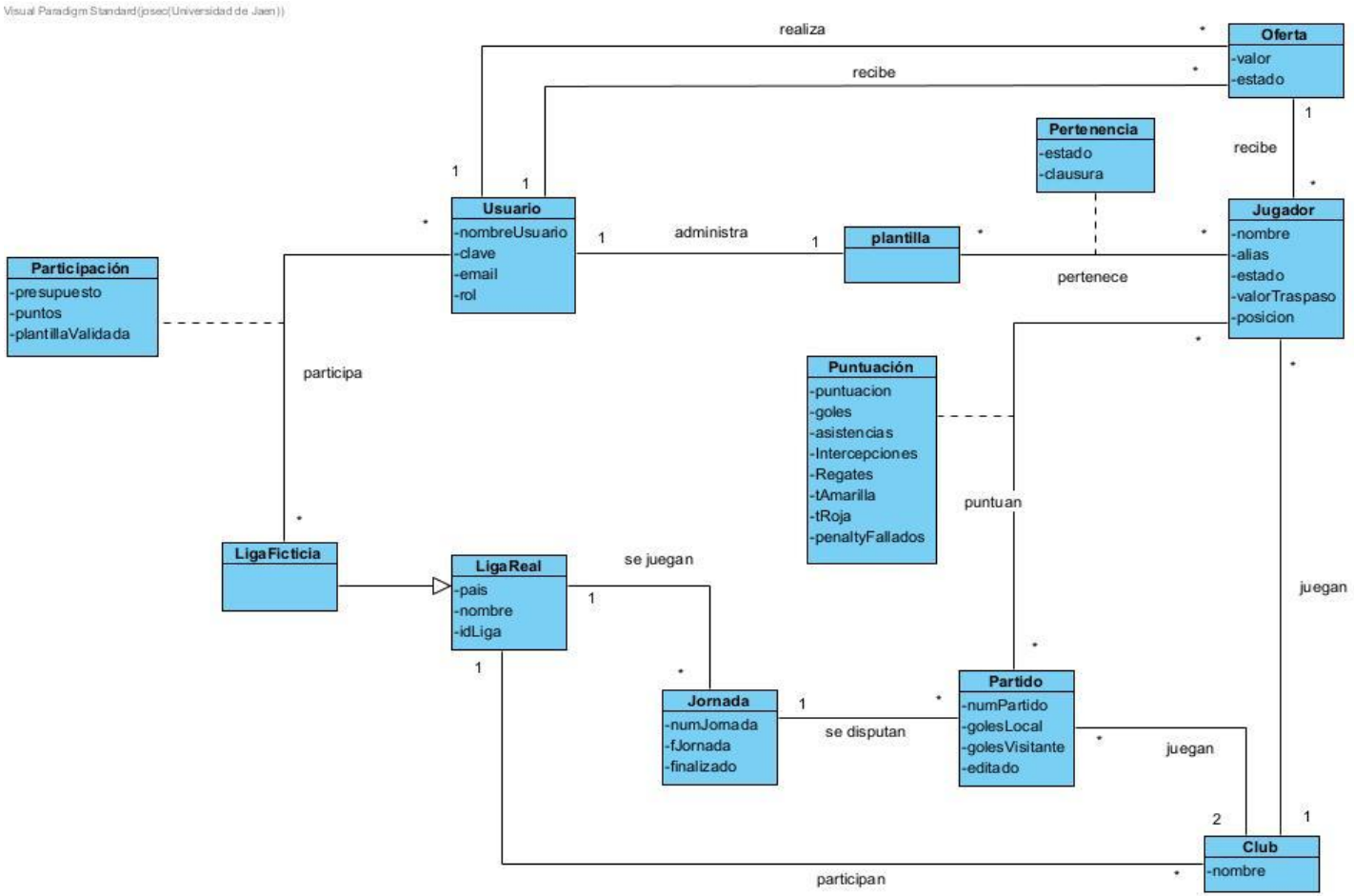

Ilustración 6: Modelo de dominio

Como se puede observar, la abstracción entre las ligas reales y las ligas donde participan los usuarios de la aplicación se ha transformado en una herencia entre LigaReal y LigaFicticia.

Realmente en la aplicación existen 5 ligas reales, de las cuales heredan todas las demás ligas en las que participan los usuarios. Estas 5 ligas son las que realmente tienen jornadas y participan clubes en la vida real, por lo que simplemente conociendo el país de la LigaFicticia de la que hereda, se sabe cuál de las 5 ligas ha creado el usuario.

Esta herencia se hace para que cada vez que se crea una liga no haya que crear otra vez nuevas jornadas, o haya que añadir otra vez los 20 clubes que existen por liga en cada creación.

De esta forma, y de forma externa, solo estarían almacenadas en la aplicación 100 (20 * 5) clubes y 190 (38 * 5) jornadas, con 1900 (38 * 10 * 5) partidos que se juegan por temporada juntando todas las ligas. Estos datos cambian por temporada, por lo que se aprovechará el parón de vacaciones en verano para actualizar los clubes, jornadas y partidos de cada liga.

De forma similar pasaría con la clase jugadores, donde cada club tiene aproximadamente 25 jugadores. Si hacemos cálculos tenemos un total de 2500 (25 * 20 * 5) jugadores por temporada en nuestra aplicación.

Tanto la clase LigaReal, como Jornada, Partido, Club o Jugador, ya estarán almacenadas en la base de datos y son clases que sólo se actualizarán, ya que un usuario nunca podrá crear o borrar estas clases, y solo los administradores podrán modificarlas (en el caso que hiciesen falta actualizarlas, como por ejemplo en la RF13 donde los administradores deben editar los partidos con el resultado real).

La relación entre usuarios y ligas se ha modelado con la clase Participacion, que se trata de una clase de asociación ya que los usuarios pueden participar en muchas ligas y las ligas pueden tener muchos usuarios.

Algo parecido sucede en la relación entre la plantilla de un jugador y los jugadores en sí, que se modela con la clase de asociación de pertenencia, ya que un jugador puede estar en muchas plantillas de ligas diferentes.

Realmente, en los requisitos RF10 y RF11 estás creando y eliminando pertenencias de la base de datos, aunque para el usuario, estás fichando y vendiendo jugadores. Igualmente, esta parte se verá con más detalle más adelante.

2.8. Casos de uso

Partiendo de los requisitos mencionados en el apartado 2.3, los diferentes casos de uso de nuestra aplicación se pueden resumir en el siguiente diagrama:

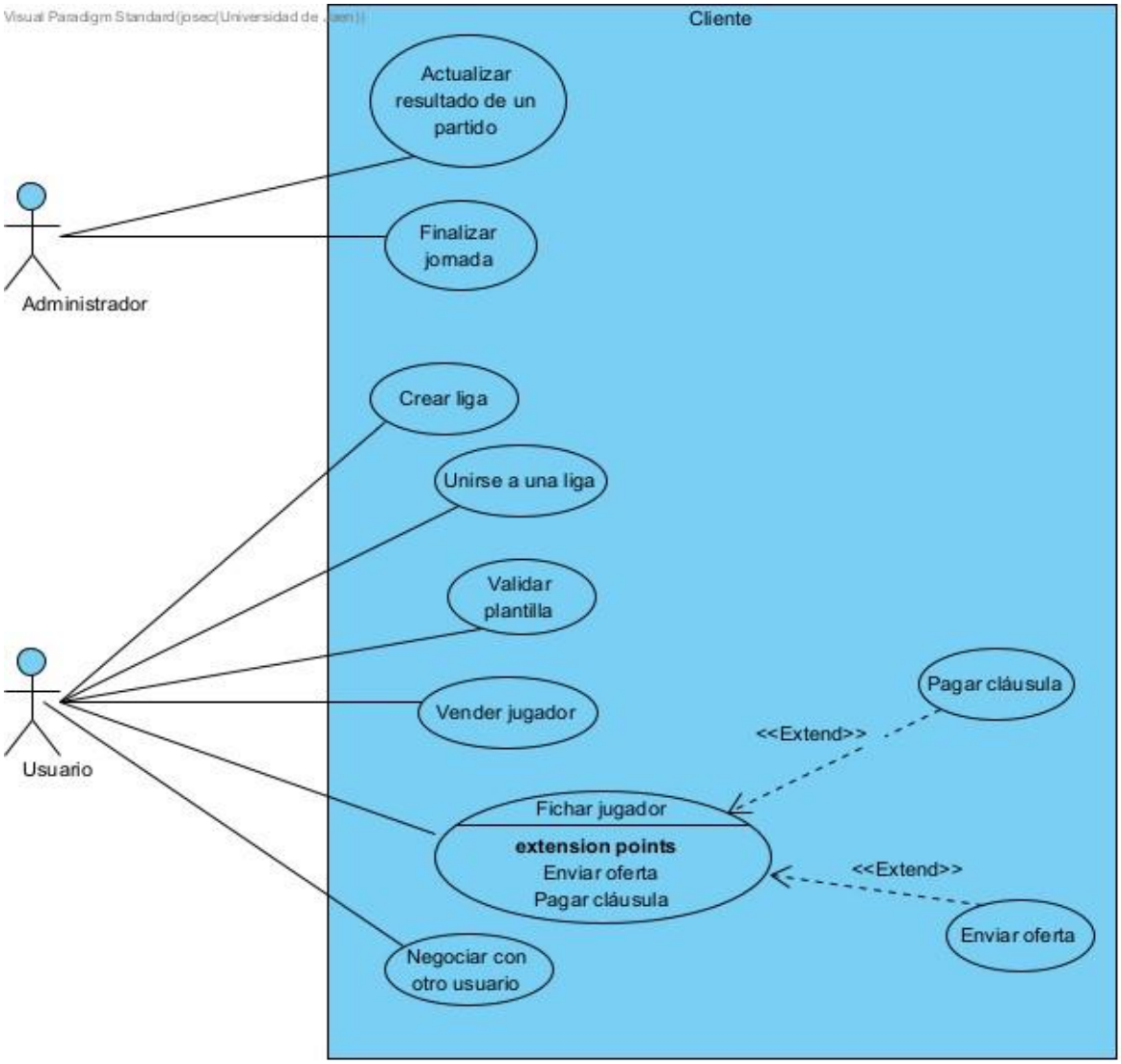

Ilustración 7: Casos de uso

Como se puede apreciar en la ilustración anterior, el usuario interactúa con los casos de uso en la parte del cliente, que se comunicarán mediante el API Rest en la parte del servidor.

Un caso excepcional pasa a la hora de fichar un jugador, donde el actor, al intentar fichar un jugador, éste puede estar libre o comprado por otro usuario, por lo que dependiendo de este condicionante este caso de uso puede incluir enviar una oferta o pagar la cláusula de rescisión directamente.

A continuación, vamos a detallar algunos casos de uso que puedan resultar más complejos en la aplicación (ya que muchos de estos casos de uso son bastante triviales), incluyendo en cada uno una tabla con actores, precondiciones y descripción, y un diagrama de actividad del caso de uso.

Los casos de uso que analizaremos con más detalle son: fichar jugador, negociar con otro usuario, actualizar el resultado de un partido y unirse a una liga.

2.8.1. Caso de uso: fichar un jugador

Empezando por el caso de uso de fichar un jugador, éste se relaciona con otros dos casos de uso, pagar cláusula y enviar oferta. Dichos casos de uso se activarán cuando el jugador ya ha sido previamente fichado por otro usuario, como veremos más tarde en el diagrama de actividad.

Toda la información sobre este caso de uso la podemos ver en la tabla 4 y en la ilustración 8:

Número	Atributo	Contenido
1	Nombre	Fichar jugador
2	Actores	Usuario
3	Descripción	El usuario ficha un nuevo jugador para su equipo en una liga determinada
4	Precondiciones	El jugador debe pertenecer a la liga en la que participa el usuario
5	Postcondiciones	El jugador pertenece al usuario que solicita ficharlo

Tabla 4: Documentación del caso de uso de fichar un jugador

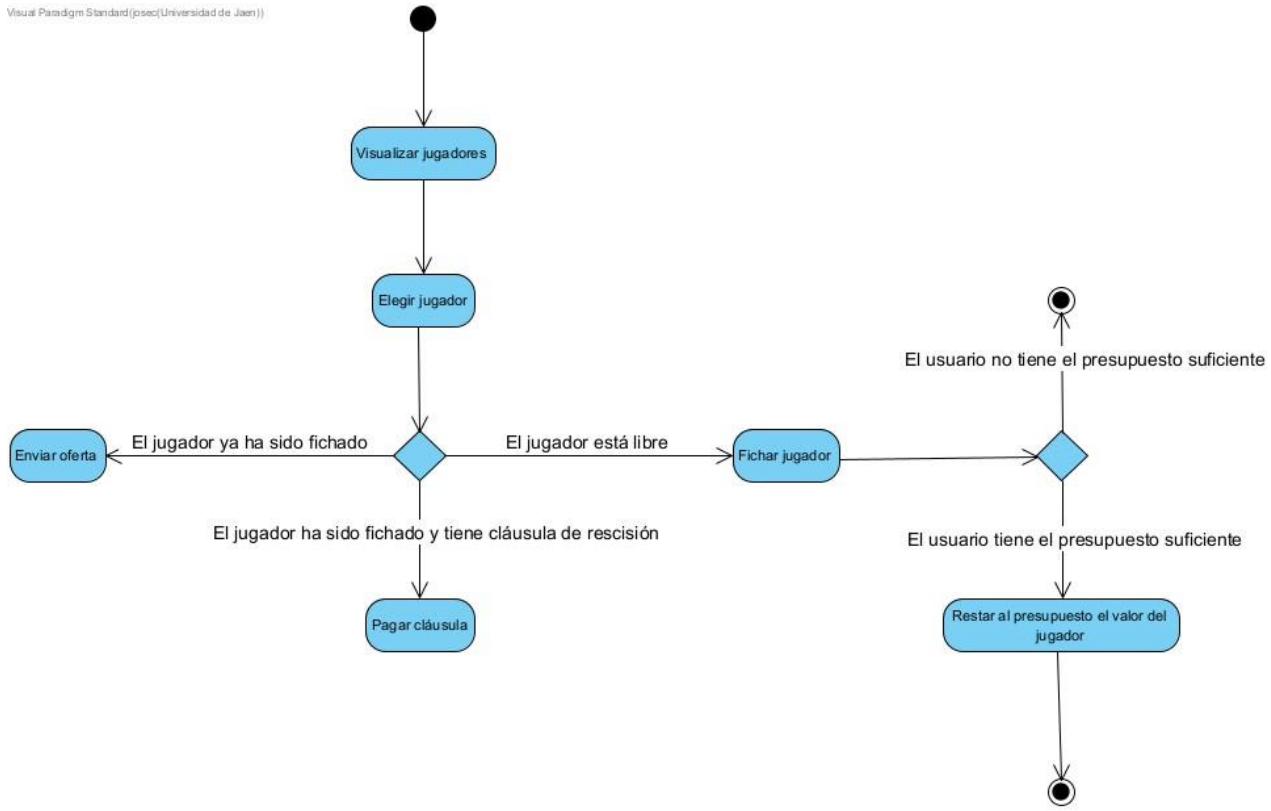

Ilustración 8: Diagrama de actividad de fichar un jugador

Como podemos ver, el usuario elige un jugador para ficharlo, y dependiendo de si está fichado o no se moverá a otro caso de uso o no. El caso de uso finaliza si el usuario no tiene el presupuesto suficiente o si el usuario ha podido fichar el jugador satisfactoriamente.

Otros casos de uso similares son, evidentemente, enviar oferta, pagar cláusula y vender jugador, donde se realizan operaciones similares con los presupuestos de los usuarios.

2.8.2. Caso de uso: negociar con otro usuario

Para el caso de uso de negociar con otro usuario, la salida dependerá de la respuesta del usuario en la negociación, como veremos a continuación.

Número	Atributo	Contenido
1	Nombre	Negociar con otro usuario
2	Actores	Usuario
3	Descripción	El usuario negocia una oferta por un jugador realizada por otro usuario.
4	Precondiciones	Debe existir al menos una oferta por algún jugador que pertenece al usuario que recibe la oferta
5	Postcondiciones	El jugador sigue pertenenciendo al usuario si se rechaza la oferta, o pasa a pertenecer al usuario que realiza la oferta si se acepta

Tabla 5: Documentación del caso de uso de negociar con otro usuario

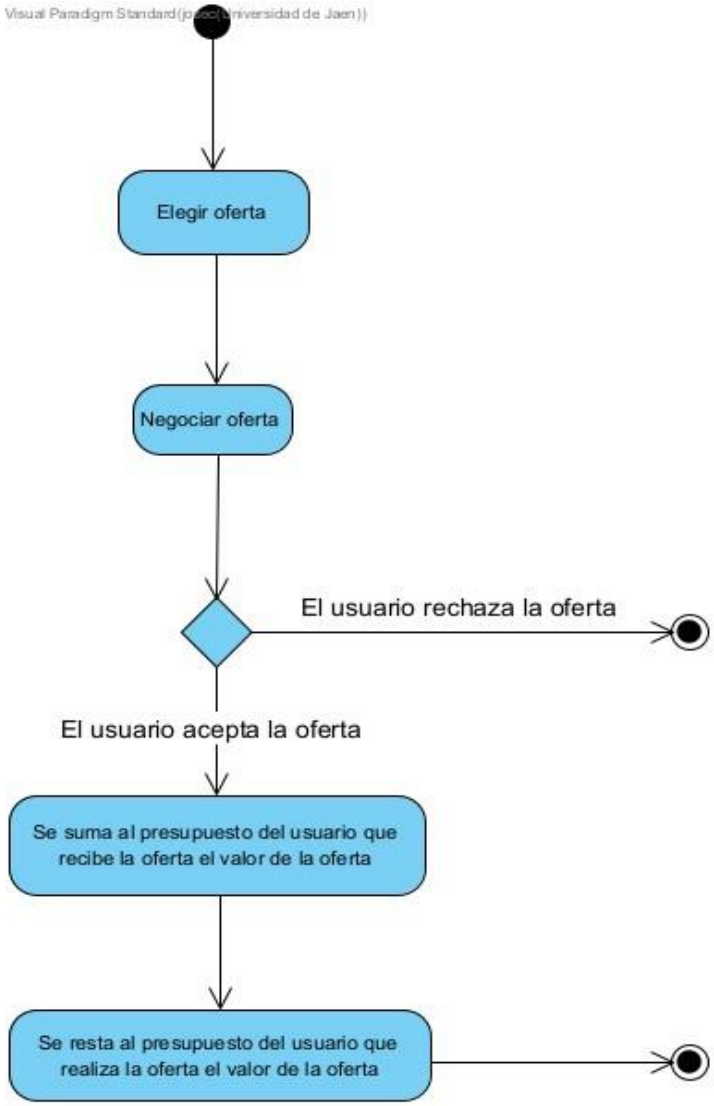

Ilustración 9: Diagrama de actividad de negociar con otro usuario

Como se puede apreciar, el usuario elige la oferta que ha recibido y la acepta o rechaza según sus intereses.

2.8.3. Caso de uso: actualizar el resultado de un partido

Este caso de uso es quizás más simple que los otros dos, pero es importante recalcar el orden en el que se realizan las operaciones, pues este caso incluye puntuar a los jugadores que han participado en un partido, y esto sólo se puede hacer una vez se ha guardado el resultado del partido.

Número	Atributo	Contenido
1	Nombre	Actualizar el resultado de un partido
2	Actores	Administrador
3	Descripción	El administrador actualiza el resultado de un partido y puntúa a los jugadores que han participado en él
4	Precondiciones	El partido ha acabado en la vida real, y se tienen los resultados y las estadísticas de los jugadores
5	Postcondiciones	El resultado del partido se ha actualizado y se han puntuado a los jugadores acorde a sus estadísticas del partido

Tabla 6: Documentación del caso de uso de actualizar el resultado de un partido

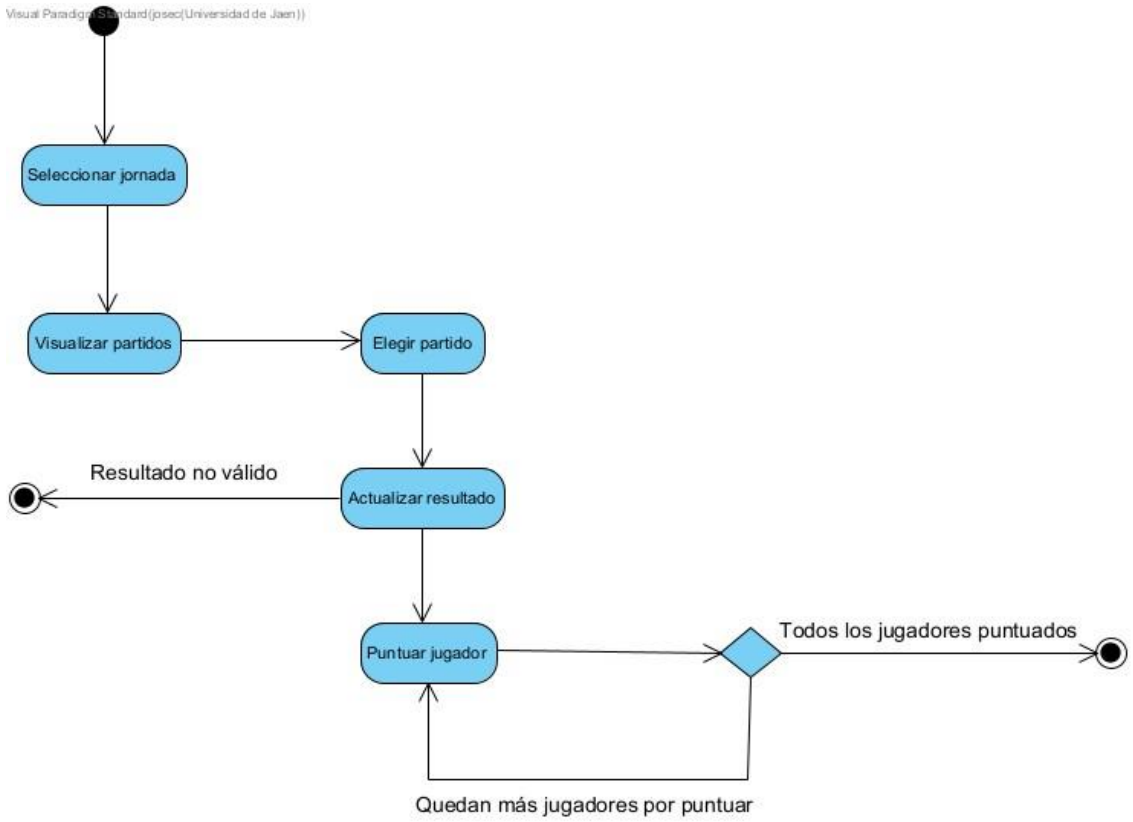

Ilustración 10: Diagrama de actividad de actualizar resultado de un partido

Como ya hemos mencionado, este caso de uso es más simple de comprender, donde sólo saldríamos en el caso de que intentemos actualizar el partido con un resultado no válido (por ejemplo -1 – -2) o una vez se han puntuado todos los jugadores del partido.

2.8.4. Caso de uso: Unirse a una liga

Por último, vamos a detallar el caso de uso de unirse a una liga. Como vamos a ver a continuación, este caso de uso, junto a los que quedan, son ya bastante triviales, por lo que será el último que analizaremos con detalle.

Número	Atributo	Contenido
1	Nombre	Unirse a una liga
2	Actores	Usuario
3	Descripción	El usuario se une a liga de la aplicación
4	Precondiciones	El usuario está registrado en la aplicación y la liga ha sido creada previamente
5	Postcondiciones	Se suma un nuevo participante a la liga seleccionada.

Tabla 7: Documentación del caso de uso de unirse a una liga

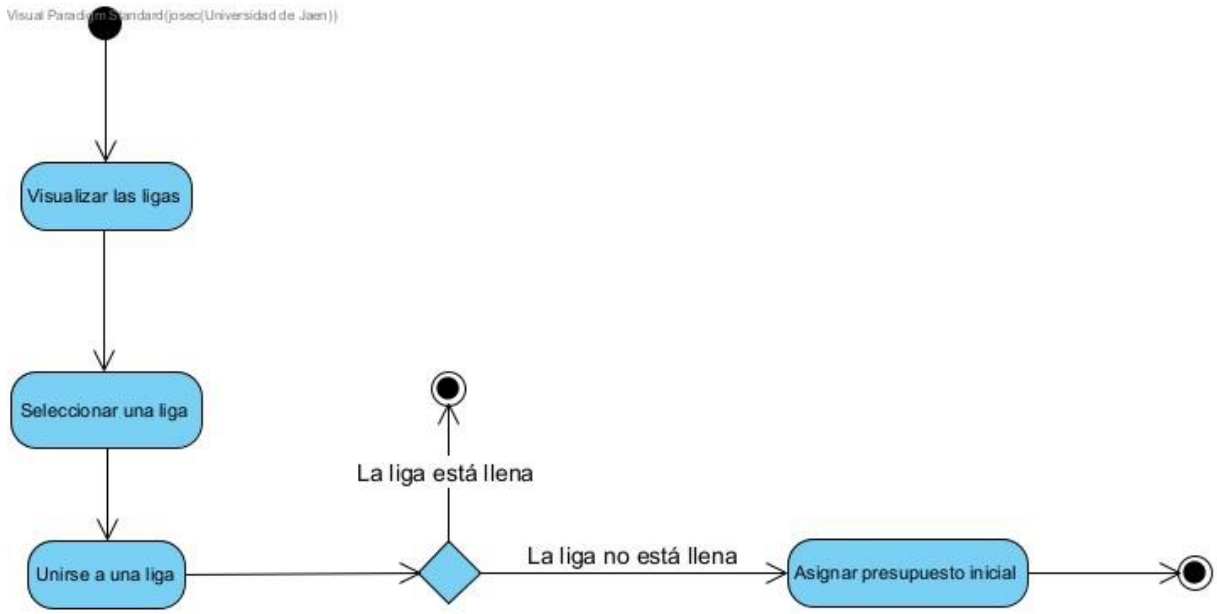

Ilustración 11: Diagrama de actividad de unirse a una liga

El usuario solicita unirse a una liga seleccionada previamente, y si la liga no excede el número máximo de participantes se une a la liga y se le asigna el presupuesto inicial.

Otro caso de uso similar al anterior es el de crear una liga, donde el escenario de excepción sería que otra liga en la aplicación contenga el mismo nombre que le queramos asignar a la nueva liga.

Los demás casos de uso son muy similares al de la ilustración 11: se siguen unos pasos previos hasta que se llega a una condición, donde si se cumple satisfactoriamente según los requisitos mostrados en el apartado 2.3 (como por ejemplo, la validación de una plantilla) se completa el caso de uso, y si no, se completan escenarios alternativos.

3. Diseño

El diseño de la aplicación a nivel de servidor se realizará siguiendo una arquitectura de 3 capas:

- Una primera capa, de dominio, donde se realiza toda la lógica de negocio y donde se implementa las reglas reales del negocio: cómo la información es creada, procesada y almacenada.
- La segunda capa, llamada capa de persistencia, simplifica el acceso a la base de datos eliminando las características particulares de cada sistema de gestión de bases de datos.
- Por último, se ha implementado una capa de servicios, que hace accesible la funcionalidad de la lógica de negocio a una aplicación externa mediante protocolos estandarizados y políticas de seguridad.

Estas tres capas se irán implementando en cada iteración del proyecto, cumpliendo con los requisitos indicados en cada iteración. A la anterior enumeración le falta una capa, la capa de presentación o aplicación, donde se realiza toda la interacción persona-ordenador y que se implementa en la parte del cliente.

Es importante mencionar estas capas a la hora de realizar el diseño de la aplicación ya que, como veremos más adelante, podremos separar los diferentes diagramas de clases de diseño en sus respectivas capas, permitiéndonos dividir la aplicación en paquetes de alta cohesión.

Un ejemplo de todas las capas (incluyendo la base de datos) y cómo se relacionan entre ellas la podemos ver en la ilustración 12:

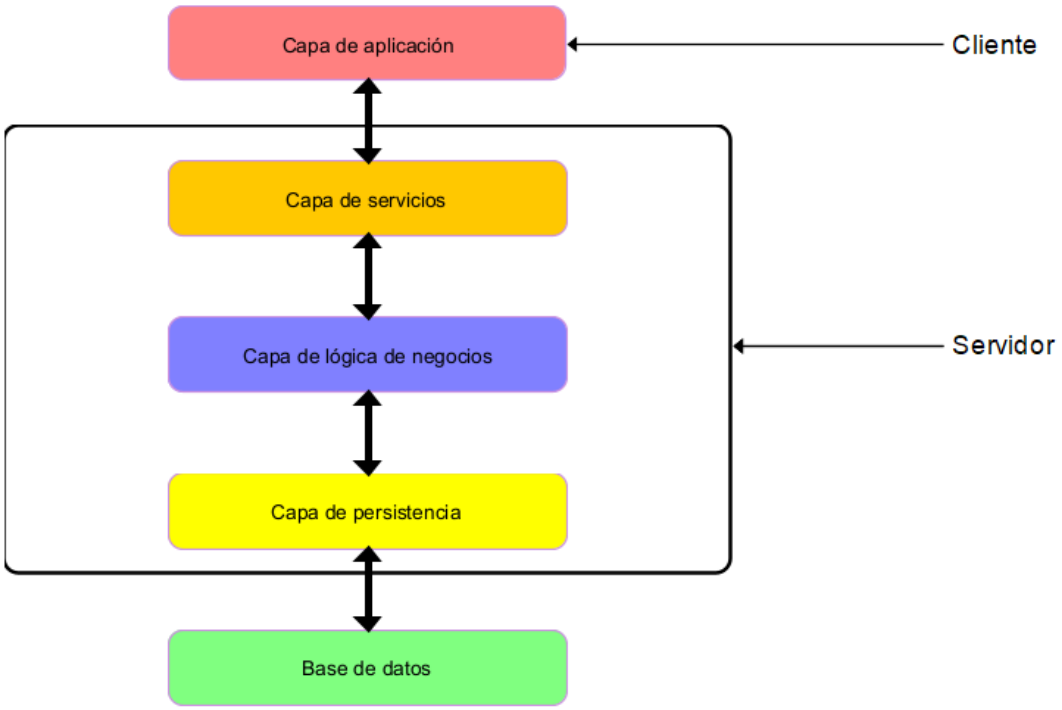

Ilustración 12: Diagrama de capas del diseño

3.1 Evolución del diseño

La evolución del diseño la podemos ver en la evolución del diagrama de clases, que ha ido cambiando conforme avanzan las iteraciones del proyecto.

Por ello, creemos que antes de comentar el diagrama entidad-relación definitivo, es importante mostrar dicha evolución, con el objetivo de presentar los cambios más relevantes debido a decisiones importantes de diseño posibles gracias al planteamiento iterativo empleado, ya que el diagrama de clases final contendrá unas relaciones bastante diferentes respecto al modelo de dominio inicial planteado, y mostrar dicho diagrama final puede resultar sorpresivo sin comentar antes todos los cambios.

Empezando por la primera iteración, este diagrama no ha sufrido muchos cambios respecto al modelo de dominio debido a que en esta iteración el objetivo era centrarnos en lo referente a la lógica de las ligas, así como su creación, que contengan las jornadas adecuadas junto a sus partidos etc.

Antes de empezar a mostrar dicha evolución, vamos a utilizar un patrón para facilitar la visualización de los diagramas: usaremos clases en verde para las nuevas clases que se hayan implementado en una iteración, amarillas para las modificadas respecto a la modificación anterior y azul si la clase ni ninguna relación ha sido modificada.

El diagrama con el que se empezó a trabajar es el siguiente:

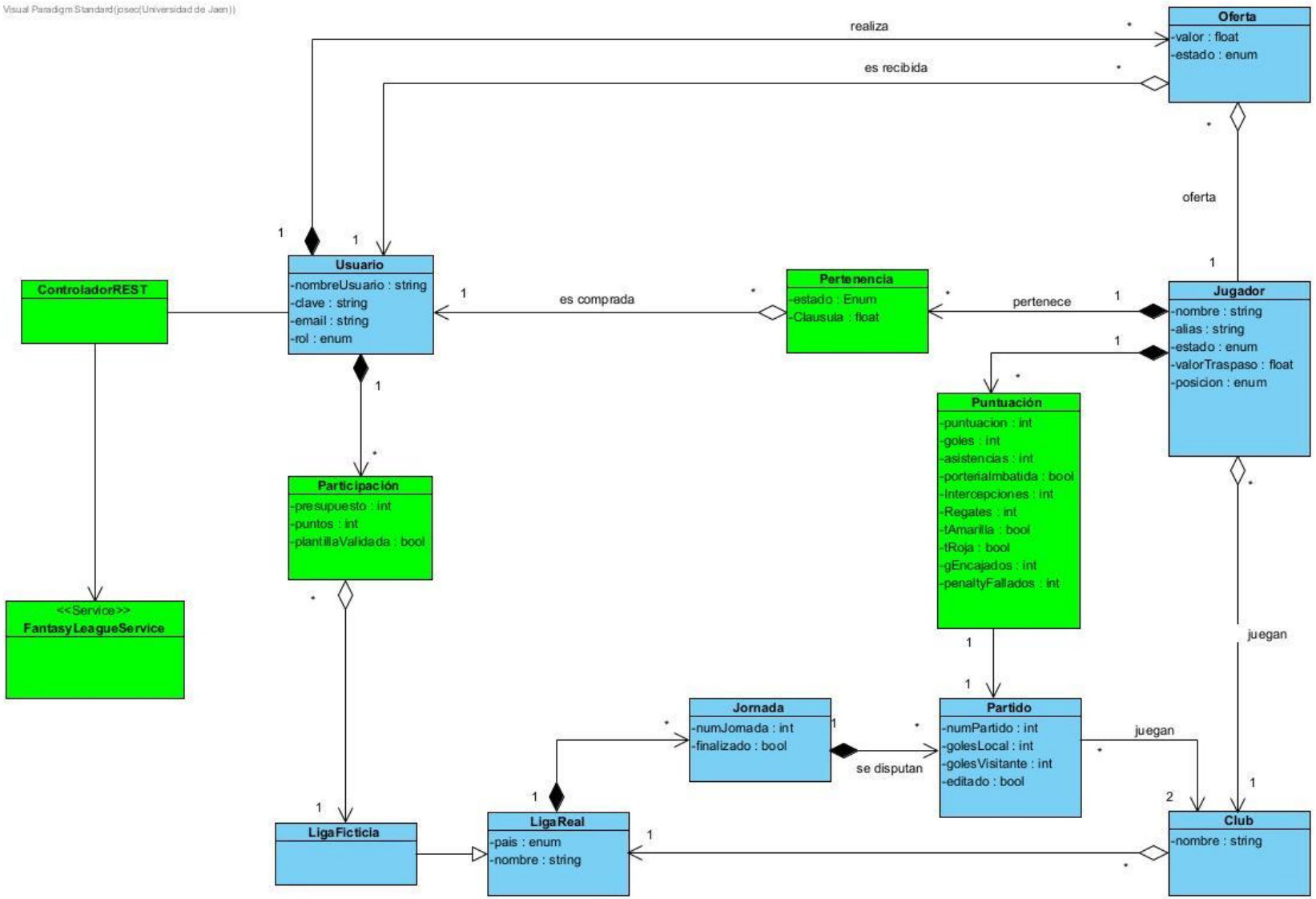

Ilustración 13: Diagrama de clases en la primera iteración

Como se puede apreciar, se ha eliminado la clase “Plantilla” que actuaba como intermediario entre los usuarios y los jugadores. Esta eliminación sólo supone más responsabilidad a la clase Pertenencia, puesto que no supone un gran cambio en el resto de clases de la aplicación.

También se han transformado las clases de asociación, tales como Participacion o Pertenencia, ya que los lenguajes de programación orientados a objetos no soportan este tipo de artefactos de análisis.

En su lugar, se han convertido en una clase normal y se ha utilizado una combinación de composición y agregación para capturar la semántica, aunque evidentemente se pierde significado realizando este cambio.

Por último, se ha detallado los tipos de relaciones y su navegabilidad, intentando hacer de ésta última la relación entre clases lo más cómodo posible.

Para la segunda iteración, se produce lo que es posiblemente el mayor cambio al diagrama de clases, ya que se modeló toda la interacción de los usuarios con los jugadores, y el diagrama anterior tenía un grave problema en esta parte.

El diagrama quedó de la siguiente forma:

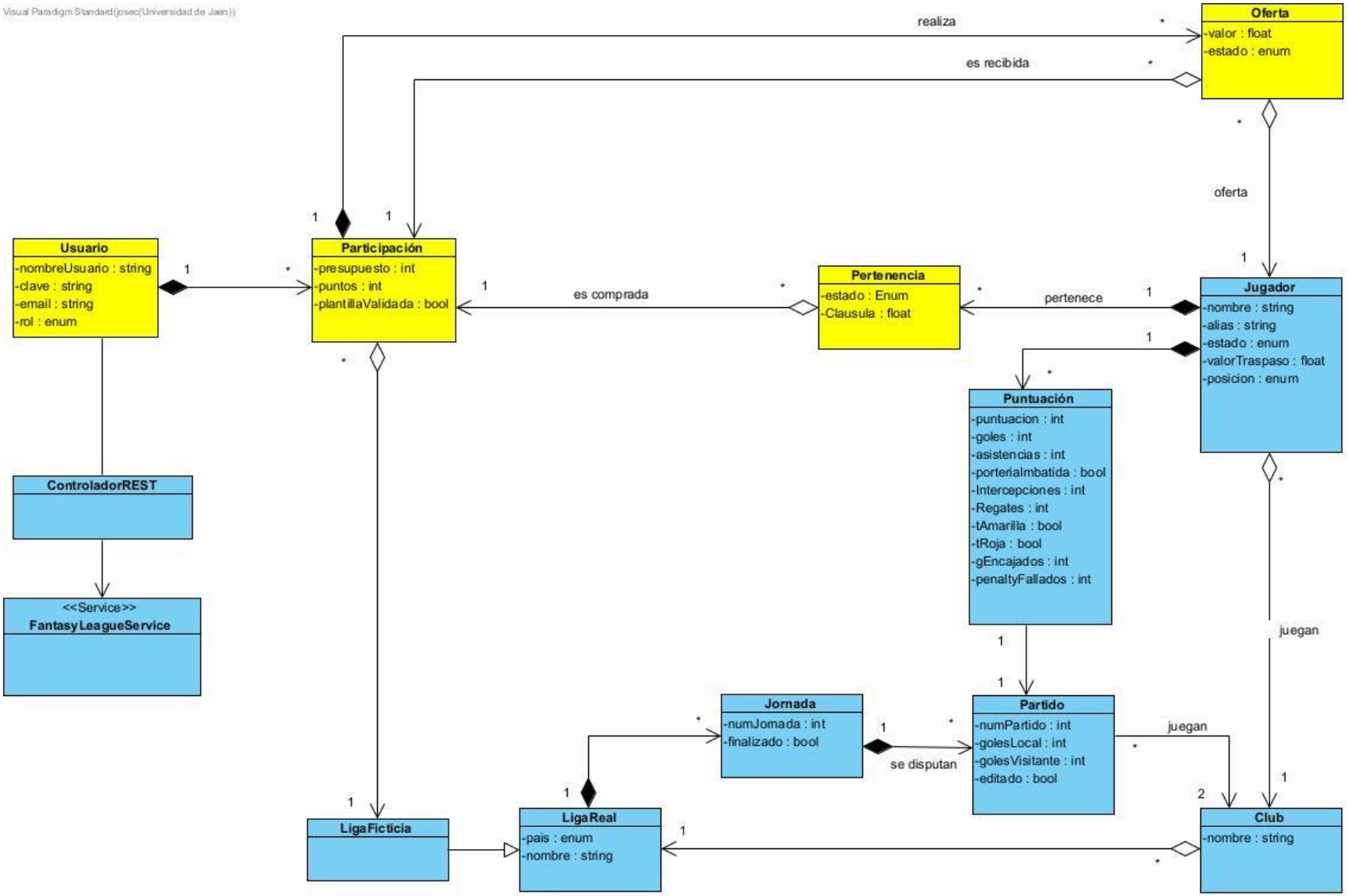

Ilustración 14: Diagrama de clases en la segunda iteración

Tal y como podemos observar en la ilustración 14, se ha producido un gran cambio respecto a las relaciones entre los usuarios y las pertenencias y las ofertas.

Este cambio se debe a que, como estaba antes, un usuario administraba sus pertenencias (que a fin de cuentas son los jugadores que tiene un usuario) pero nunca se hacía referencia a la liga donde se encuentran esas pertenencias, lo que provocaba que si un usuario por ejemplo estaba en 10 ligas diferentes pero del mismo país, podría incluso tener jugadores repetidos pero no se sabe a qué ligas pertenecen cada uno.

Este problema se soluciona delegando esa responsabilidad a la clase Participacion, que se convierte en el experto de información y en el gestor en todo lo referente a las ligas.

En la última iteración no cambió el diagrama de forma significativa, si no que se añadieron las nuevas clases que actuarían como servicios (como por ejemplo de seguridad). Sin embargo, sí que sufrió un único cambio en la relación entre ofertas y jugadores, como se puede observar a continuación:

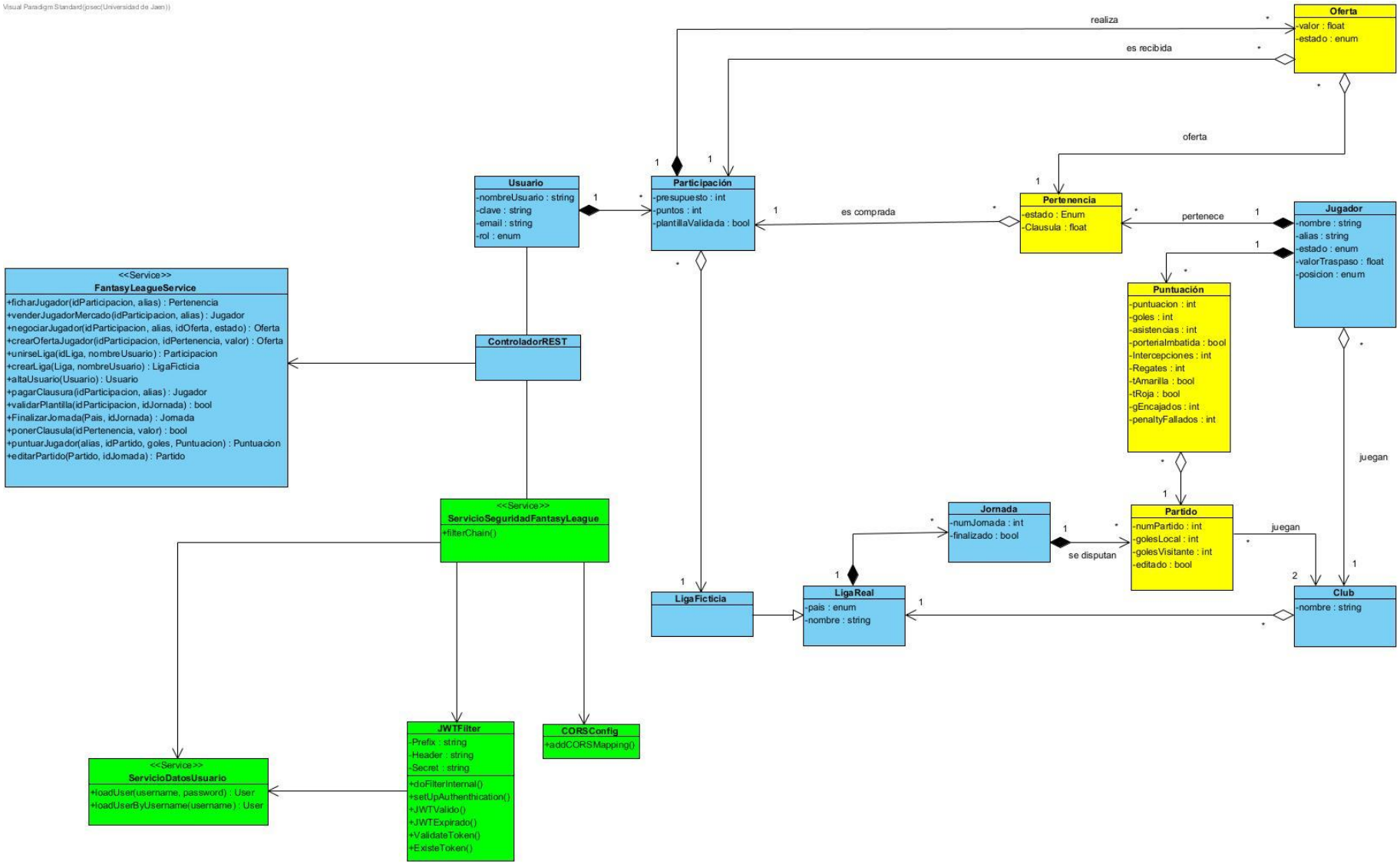

Ilustración 15: Diagrama de clases en la tercera iteración

Este cambio viene motivado un poco por la explicación de los cambios de la segunda iteración, y aunque la aplicación podría funcionar perfectamente con el diagrama de clases de la segunda iteración, realizar este cambio es lo idóneo ya que a pesar de que a vistas del cliente éste realiza una oferta a un jugador perteneciente a un usuario, realmente internamente se está realizando una oferta a una pertenencia de una participación.

También se ha cambiado el tipo de relación entre puntuación y partido, ya que esta relación era de 1 a 1 pero realmente un partido puede tener muchas puntuaciones (tantas como jugadores hayan participado en el partido).

Este diagrama completaría la capa de dominio y de servicios, sin embargo, nos queda la capa de persistencia, que veremos a continuación.

3.2 Diagrama Entidad-Relación

Para el diseño de la capa de persistencia, se va a utilizar un ORM[16], en concreto JPA[17], que facilite el manejo de la información almacenada en la base de datos.

JPA proporciona una forma de mapear objetos a tablas de una base de datos y facilita la interacción con la base de datos utilizando una interfaz orientada a objetos.

Aunque esta parte se describe con más profundidad en la etapa de implementación, es importante mencionarlo aquí para aclarar cómo se generan las tablas finales a partir de las entidades/relaciones del modelo.

Antes de mostrar el diagrama entidad-relación de la aplicación, vamos a mostrar el diagrama ORM desde donde se generará el diagrama entidad-relación, que no es más que el diagrama de la ilustración 16 pero sólo con las clases o entidades que conforman la capa de dominio:

[16] Object-Relational Mapping (Mapeo Objeto-Relacional)
[17] Java Persistence API

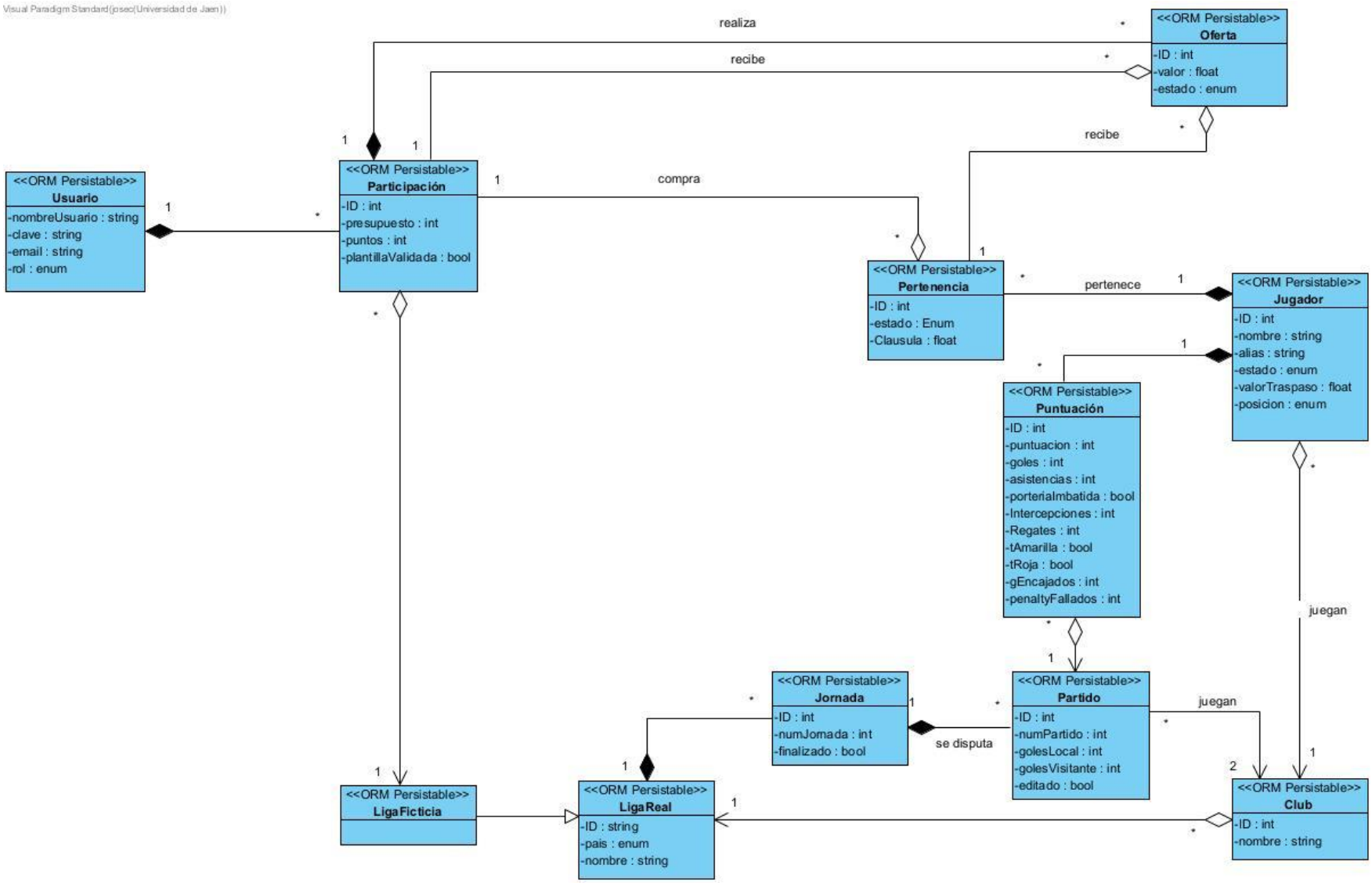

Ilustración 16: Diagrama ORM

Como se puede observar, se han indicado todas las clases como entidades persistentes y se ha creado un nuevo atributo llamado “ID” en todas las clases que no tenían un identificador único bien definido, esto es, en todas las clases salvo Usuario, donde su clave primaria será el nombre de usuario, y LigaReal, donde a pesar de que se ha creado un atributo ID, éste realmente será un atributo de tipo string con el nombre convertido a minúscula y sin espacios.

A partir del diagrama ORM podemos obtener de forma “automática” el diagrama entidad-relación de nuestra aplicación (esto lo podemos hacer fácilmente con herramientas como Visual Paradigm), como podemos ver en la ilustración 17:

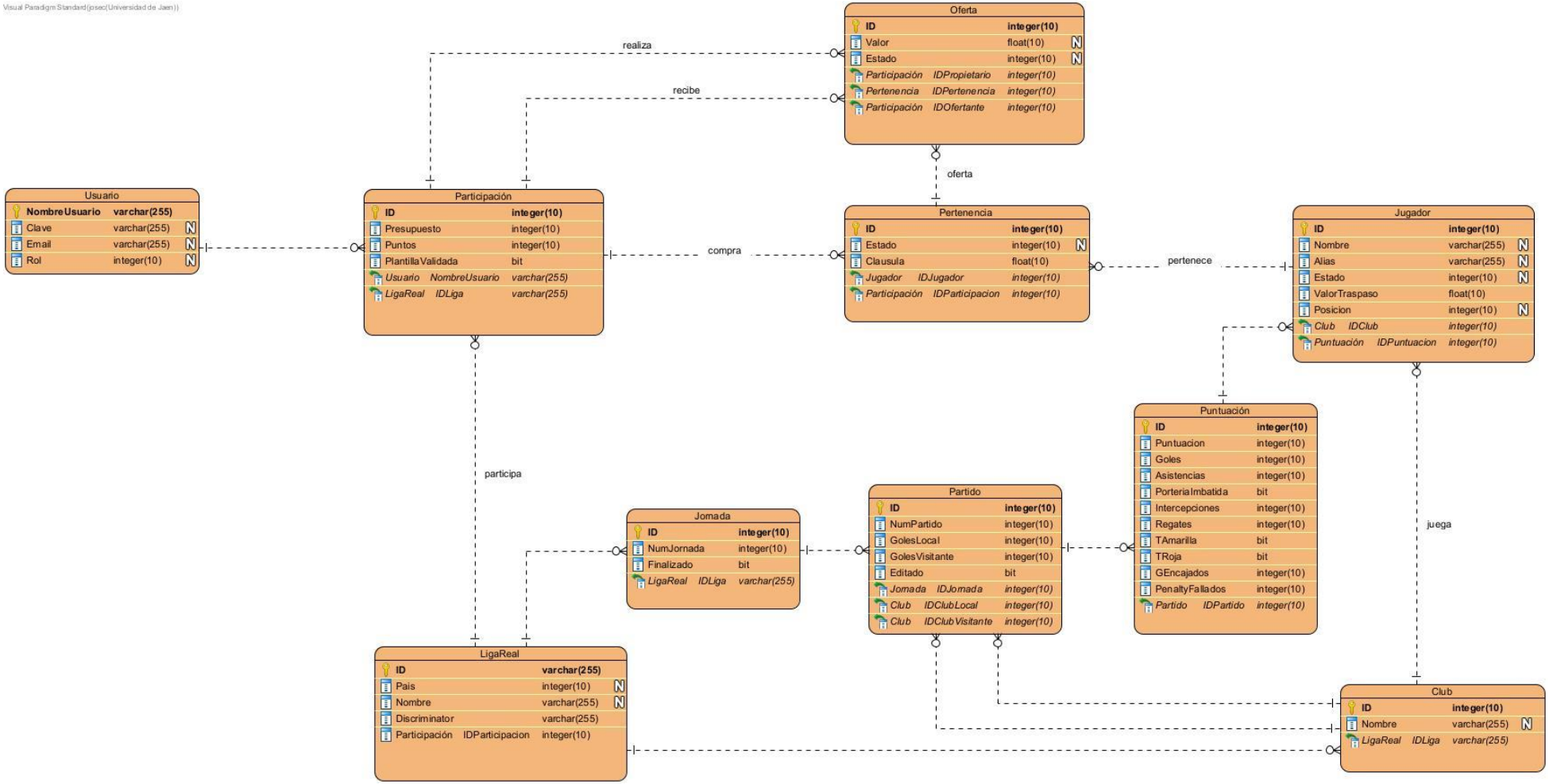

Ilustración 17: Diagrama Entidad-Relación

Al tener el diagrama ORM, este diagrama se puede realizar casi de forma trivial, aunque tiene un par de singularidades que me gustaría comentar:

- La entidad Oferta contiene la clave primaria de dos participaciones, así como lo indican sus relaciones: un id que será del que realiza la oferta (IDOfertante) y otro que será del que la recibe (IDPropietario)
- La entidad Partido, como se puede ver en el diagrama de clases, tiene una relación de muchas a 2 respecto a la entidad Club.

  Esta relación se podría modelar como muchas a una, sin embargo, como en un partido sólo juegan 2 clubes, se ha modelado como una doble relación de muchos a uno para que la entidad Partido contenga la clave primaria de los dos clubes que participan en él.

- La clase LigaFicticia tiene una relación de tipo herencia con LigaReal. Esto se puede modelar de varias maneras, como por ejemplo creando una tabla para cada clase o almacenando ambas en una sola tabla.

  Como realmente sólo existen 5 entidades de tipo LigaReal, que son las que gestionan los administradores, y las demás entidades siempre serán de tipo LigaFicticia, ya que son las entidades que crean los usuarios, se ha elegido almacenar ambas clases en una sola tabla cuyos 5 primeros registros serán de tipo LigaReal y las demás de tipo LigaFicticia, ya que realmente para los usuarios solo existe la clase LigaFicticia.

  Para ello se crea un atributo llamado dtype o Discriminator que indica el nombre de la clase a la que pertenece esa tupla.

Como ya hemos mencionado anteriormente, la conversión entre diagrama ORM y el diagrama Entidad-Relación se hace de forma “automática” si utilizamos herramientas como Visual Paradigm.

Sin embargo, puede resultar interesante demostrar que efectivamente estas tablas están normalizadas. Aunque existen 5 formas normales, se suele realizar la normalización hasta la tercera forma normal, como describiremos a continuación:

- Primera Forma Normal: Establece que cada columna en una tabla debe contener valores atómicos e indivisibles. Además, cada columna debe tener un nombre único y estar identificada por una clave primaria.
- Segunda Forma Normal: Establece que una tabla en 1NF debe eliminar las dependencias parciales. Esto significa que cada columna no clave en la tabla debe depender completamente de la clave primaria.
- Tercera Forma Normal: Establece que una tabla en 2NF debe eliminar las dependencias transitivas. Esto significa que no debe haber dependencias entre las columnas no clave. Si una columna no clave depende de otra columna no clave, se debe separar en una nueva tabla.

En nuestro caso, y para no hacer este análisis demasiado largo, vamos a comprobar si está normalizada la tabla Participacion, ya que es la tabla que más relaciones tiene.

En primer lugar, podemos ver que está en primera forma normal ya que la tabla contiene una clave primaria (ID) que es única.

También está en segunda forma normal ya que los atributos ”Puntos” y “Presupuesto” dependen únicamente de la clave primaria y no tienen dependencias parciales unos de otros.

Finalmente, la tabla también está en tercera forma normal ya que los atributos como “nombreUsuario” o “IDLiga” hacen referencia a otras tablas, por lo que se han eliminado los atributos transitivos de la tabla Participacion para crear las nuevas tablas (Usuario y LigaReal respectivamente).

Una vez tenemos las tablas podemos hablar ahora sí sobre una estimación aproximada de los registros que contendrán nuestra aplicación. En su momento ya hicimos un cálculo sobre las entidades principales de la aplicación (como jugadores o partidos) en el apartado 2.6, pero ya podemos hacerla sobre otras entidades como las puntuaciones u otros registros de otras tablas que pueden ser fácilmente calculables.

Un resumen de la estimación de registros para algunas tablas la podemos ver en la tabla 8:

Tabla	Cálculo	Número de registros
LigaReal	5 ligas reales	>=5
Jornada	38 jornadas x 5 ligas	190
Partido	38 x 5 x 10 partidos	1900
Club	20 clubes x 5 ligas	100
Jugador	25 jugadores x 20 x 5	2500
Puntuacion	2500 jugadores x 1900 partidos	<=4 750 000

Tabla 8: Estimación de registros por tabla

Respecto a las demás tablas, el número de registros pueden ir escalando dependiendo de la popularidad del producto, pero más o menos los registros de las tablas indicados en la tabla 8 serán estáticos, siendo los registros de LigaReal siempre mayores o iguales que 5 y los registros de la tabla Puntuacion menores o iguales a 4 750 000.

3.3 Diagramas de clases

Una vez analizado la evolución del diagrama de clases y obtenido el diagrama entidad-relación, ahora sí podemos mostrar el diagrama de clases de forma completa de nuestra aplicación. Hay que recalcar que, al ser una aplicación cliente-servidor, debemos dividir el diagrama de clases en dos.

3.3.1. Diagrama de clases del servidor

En la parte del servidor, se utilizará el patrón DAO[18] para implementar la capa de servicios. Dicho patrón suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, por lo que se puede considerar como un adaptador específico para relacionar la lógica de negocio y la capa de persistencia. Algunas características de este patrón de diseño son las siguientes:

- El patrón DAO abstrae y encapsula todos los accesos a la fuente de datos.
- Oculta los detalles de implementación de la fuente de datos a sus clientes
- La interfaz expuesta no cambia cuando cambia el esquema de almacenamiento de la fuente de datos.

[18] Data Access Object

- Por cada tabla de una base de datos relacional existirá un DAO. Los métodos de esta clase dependen de la aplicación, pero suelen implementarse en 4 funciones básicas: Create, Read, Update, Delete (conocida como CRUD por sus iniciales)

La traducción de estos objetos se realiza a través de una clase intermedia, denominada DTO, que calca la estructura de una tabla de la fuente de datos y almacena temporalmente los datos.

Así, el DAO utilizará el DTO para transportar los datos desde la base de datos hasta la capa de lógica de negocio o viceversa. Más información sobre este patrón de diseño y su aplicación la podemos encontrar en [16].

Por lo tanto, lo que se envía desde el servidor hasta el cliente no son más que DTOs, que son clases cuyos atributos vienen dados por los atributos guardados en la entidad persistente correspondiente.

Como el diagrama de clases de la aplicación es bastante grande, se le ha asignado un color a cada capa y cada clase tendrá un color dependiendo de la capa donde pertenezca. La asignación de colores es la siguiente: el color azul para las clases que pertenezcan a la capa de dominio, amarillo para las clases que pertenezcan a la capa de persistencia y verde para las clases que pertenezcan a la capa de servicios:

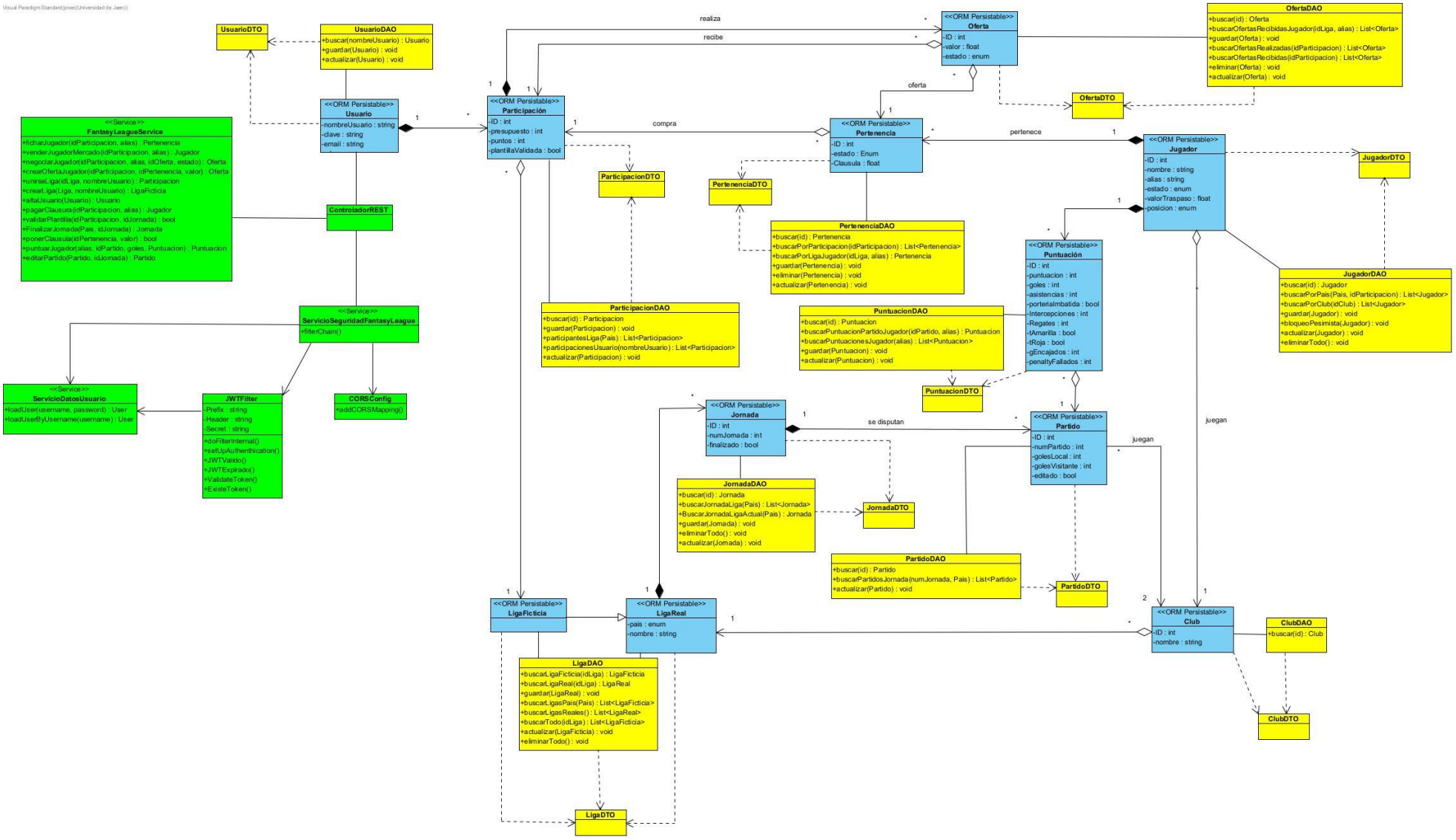

Ilustración 18: Diagrama de clases del servidor

Para no hacer el diagrama excesivamente grande, se han obviado los atributos de las clases DTO, que básicamente tienen los mismos atributos que se muestran en el diagrama entidad-relación (ilustración 17).

3.3.2. Diagrama de clases del cliente

Respecto al diagrama de clases en la parte del cliente, podemos representar cada componente Vue como una clase en el diagrama, identificando los atributos y los métodos de cada componente, además de incluir otras clases o módulos JavaScript en el diagrama.

Antes de explicar los detalles del diagrama de clases, hay que comentar que se ha utilizado la arquitectura MVC en la capa de presentación. El objetivo de aplicar este tipo de arquitectura es conseguir un bajo acoplamiento entre la capa de presentación y la del modelo del sistema, basándose en el principio de separación modelo-vista, que indica que los objetos del modelo no deberían conocer los modelos de la vista.

Esta interacción se puede realizar de forma clásica, donde el usuario interactúa con el controlador y éste lo hace con la vista y el modelo, o una más moderna, que es la que usaremos nosotros, donde el usuario interactúa con los elementos de la vista y ésta interactúa con el controlador y modelo.

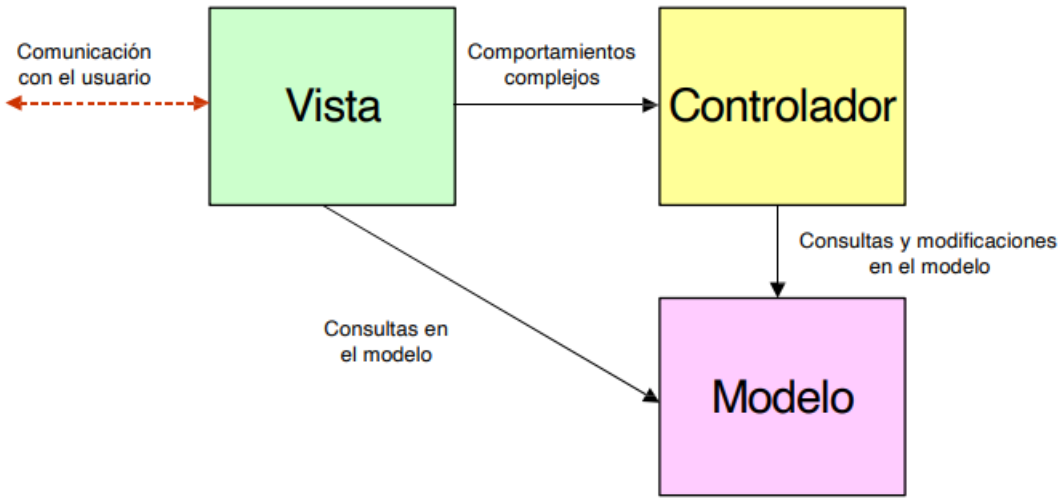

Ilustración 19: Arquitectura MVC [17]

Para evitar hacer este diagrama excesivamente grande, hemos obviado algunos métodos/atributos genéricos como pueden ser métodos de paginación y ordenación, para centrarnos en atributos/métodos más específicos de cada componente de forma que con sólo verlos se pueda saber cuál es la función de ese componente en la aplicación.

Por último, vamos a utilizar una gama de colores para diferenciar clases con diferente funcionamiento como hemos hecho en el diagrama de clases en la parte del servidor, con el objetivo de facilitar su comprensión.

En este caso vamos a diferenciar las vistas con el color verde, el control en amarillo y el modelo en rosa (al igual que en la ilustración 19).

Sin entrar en más detalles, el diagrama de clases en la parte del cliente es el siguiente:

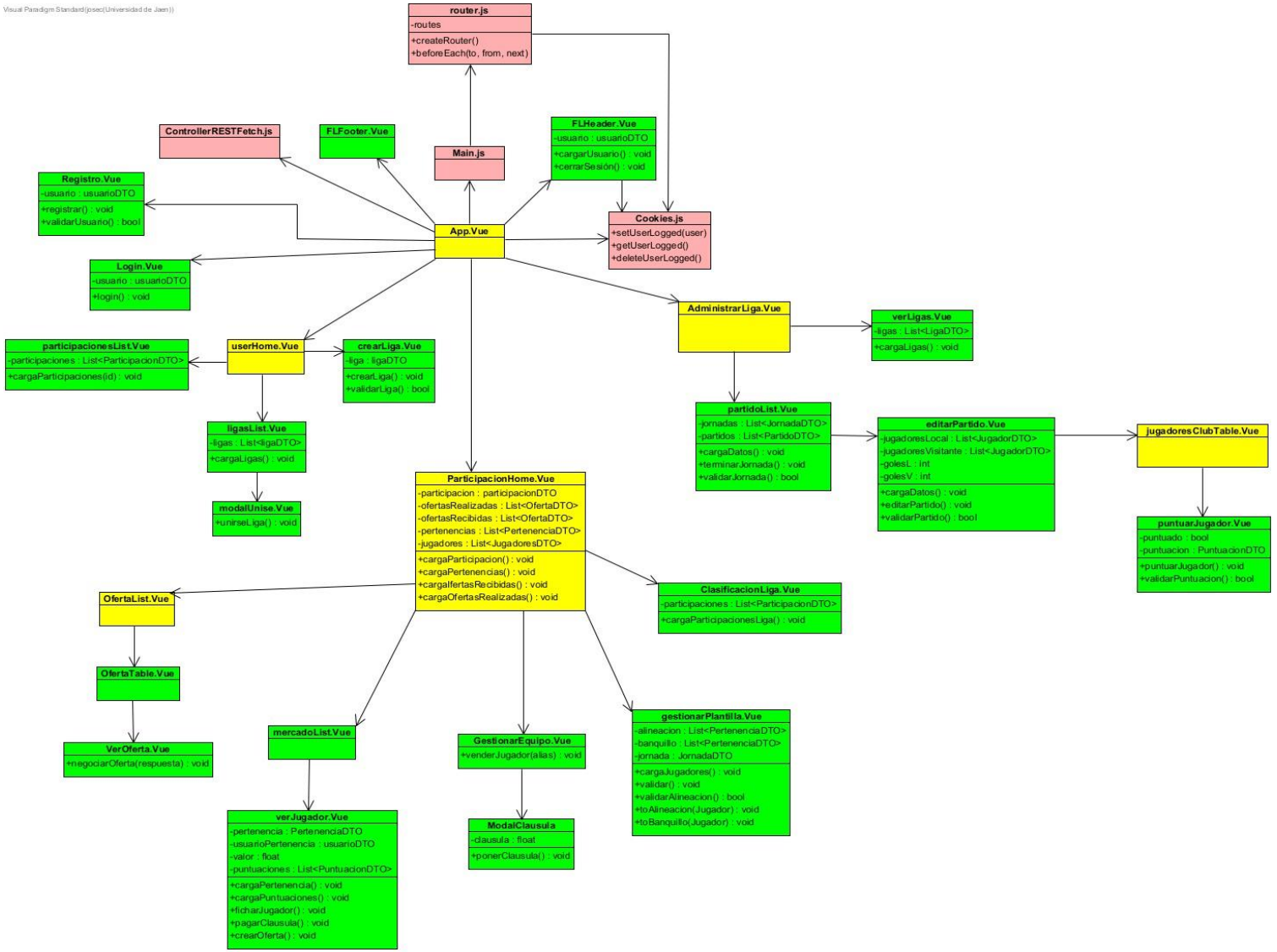

Ilustración 20: Diagrama de clases del cliente

Como ya explicaremos en el apartado de implementación, Vue tiene una singularidad, y es que es capaz de propagar atributos de unos componentes a otros.

Esto se puede ver por ejemplo entre el componente ParticipacionHome.Vue y OfertaList.Vue, donde en ParticipacionHome.Vue se cargan las ofertas, y OfertaList.Vue solo tiene que inyectar dichos atributos previamente cargados, por lo que no hace falta cargarlos de nuevo en ese componente.

Algo similar podemos ver también entre la relación de App.Vue y ControllerRESTFetch.js, donde App carga el fichero y lo propaga a los demás componentes.

En nuestro diagrama de clases existe un componente raíz, App.Vue que se encarga de renderizar todas las demás vistas (indicadas previamente por router.js). Este componente también puede cargar otro componente de control, como es el caso de userHome.Vue, que se encarga de renderizar una vista diferente (en este caso las vistas con las que se relaciona) según el input que indique el usuario.

Respecto al diagrama no hay mucho más que comentar, pues observando los atributos y métodos más o menos describen qué podemos esperar de cada componente, aunque igualmente podremos analizar en más profundidad la relación entre clases y componentes cuando veamos la capa de presentación.

3.4 Diagramas de secuencia

En este apartado vamos a analizar los aspectos más relevantes del proyecto, con el objetivo de explicar el funcionamiento interno de nuestra aplicación. Recalcar que no vamos a explicar todas las funcionalidades, solo las más complejas, pues las demás son muy similares en cuanto a funcionamiento.

En resumen, vamos a mostrar los diagramas de secuencias de las funcionalidades Fichar Jugador, Negociar Oferta y Finalizar Jornada.

Como en este diagrama intervienen tanto el diagrama de clases por parte del cliente como del servidor, vamos a indicar en amarillo aquellas clases que pertenecen al diagrama de clases del cliente y en azul los que pertenecen al servidor.

Es importante recordar el uso de la arquitectura MVC que ya hemos explicado anteriormente, pues podemos ver cómo en los diagramas de secuencia las clases interactúan siguiendo este esquema.

3.4.1. Diagrama de secuencia: fichaje de jugador

Vamos a empezar por el diagrama de secuencia de Fichar Jugador, que la podemos ver en la ilustración 21:

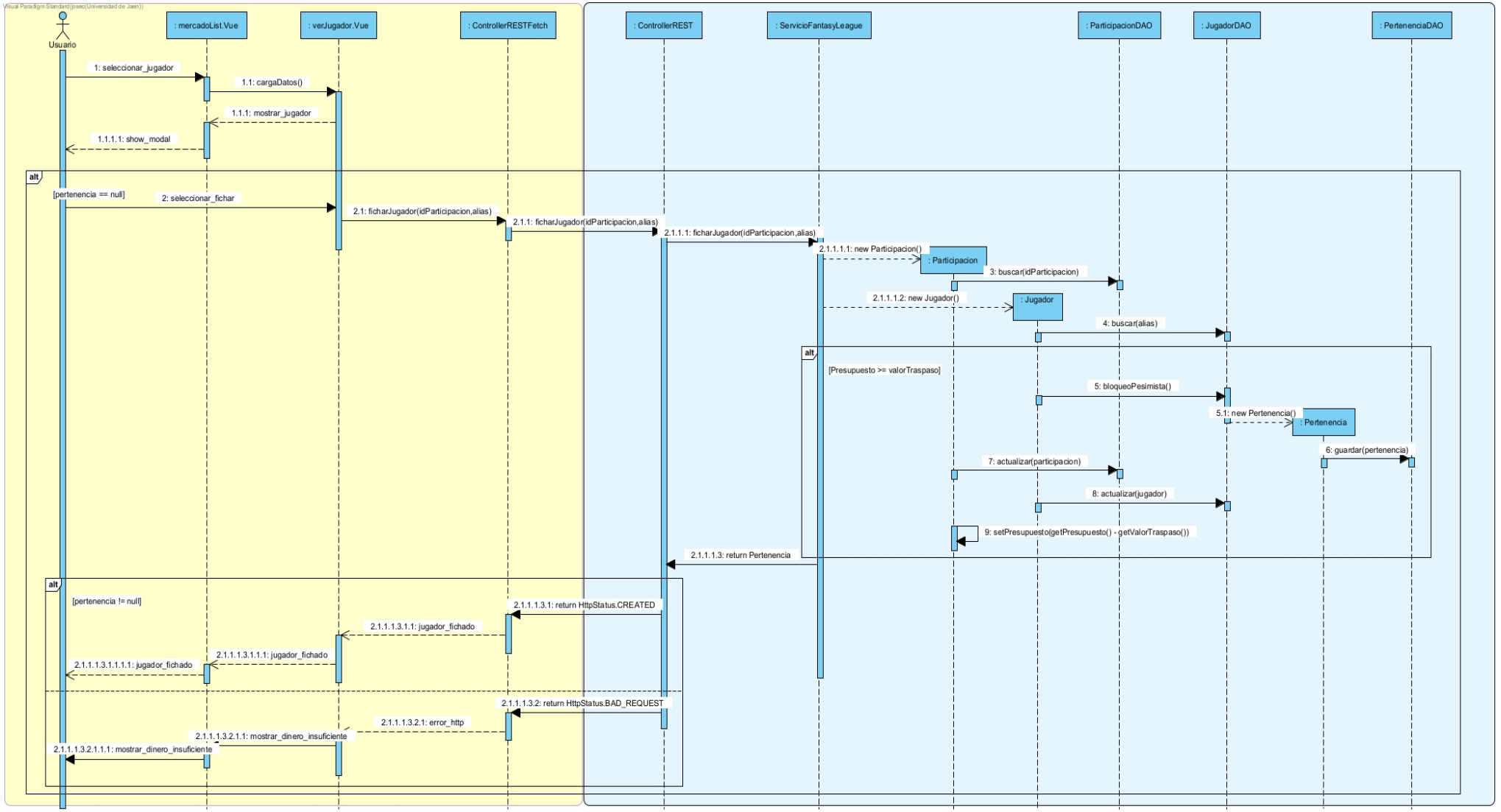

Ilustración 21: Diagrama de secuencia de fichar jugador

Fichar un jugador depende de si el jugador ya ha sido fichado previamente por otro jugador. En ese caso, se entraría en el caso de uso de crear una oferta, o de pagar una cláusula si el jugador tiene alguna (es decir, si la cláusula es mayor que 0).

Para poder visualizar mejor el diagrama de secuencia, hemos eliminado los otros casos de uso a los que se hace referencia y nos hemos centrado únicamente en el caso de que el jugador no ha sido comprado y el usuario desea comprarlo.

Primero, el usuario selecciona al jugador que quiere fichar en el componente MercadoList.Vue, que contiene un listado con los jugadores disponibles. Éste activa el componente verJugador.Vue que muestra todos los datos del jugador.

Si el jugador no pertenece a otro jugador (es decir, no ha sido comprado previamente), el usuario podrá pulsar un botón que permita fichar el jugador. Dicho botón dentro del componente verJugador.Vue llamará a la función correspondiente de la clase ControladorRESTFetch.js, que se encarga de hacer las llamadas al servidor.

En la parte del servidor, el controlador llama a la función ficharJugador() del servicio que se encarga de hacer lo necesario para que se cumpla la transacción.

Después, se busca el jugador y la participación para comprobar que existen, y se comprueba que el presupuesto de la participación es mayor que el valor de traspaso del jugador. Si es el caso, se bloquea el jugador para que no lo pueda comprar nadie más concurrentemente, y se crea la pertenencia correspondiente. Por último, se actualizan y guardan las entidades en sus DAOs correspondientes.

Una vez terminada la función del servicio, el controlador del servidor devolverá un código de respuesta dependiendo de si se ha podido crear la pertenencia o no. Si se ha podido crear, se enviará un código de respuesta 201 (CREATED) indicando que la transacción se ha completado exitosamente y se mostrará dicha retroalimentación al usuario.

En el caso de que la pertenencia no se haya podido crear porque la participación no contiene el presupuesto necesario, se enviará un código de respuesta 400 (BAD_REQUEST) debido a que el usuario no contiene el presupuesto suficiente para permitirse fichar al jugador,

3.4.2. Diagrama de secuencia: negociar una oferta

Después de analizar el caso de uso de Fichar Jugador, podemos ver el diagrama de secuencia de Negociar Oferta en la siguiente ilustración:

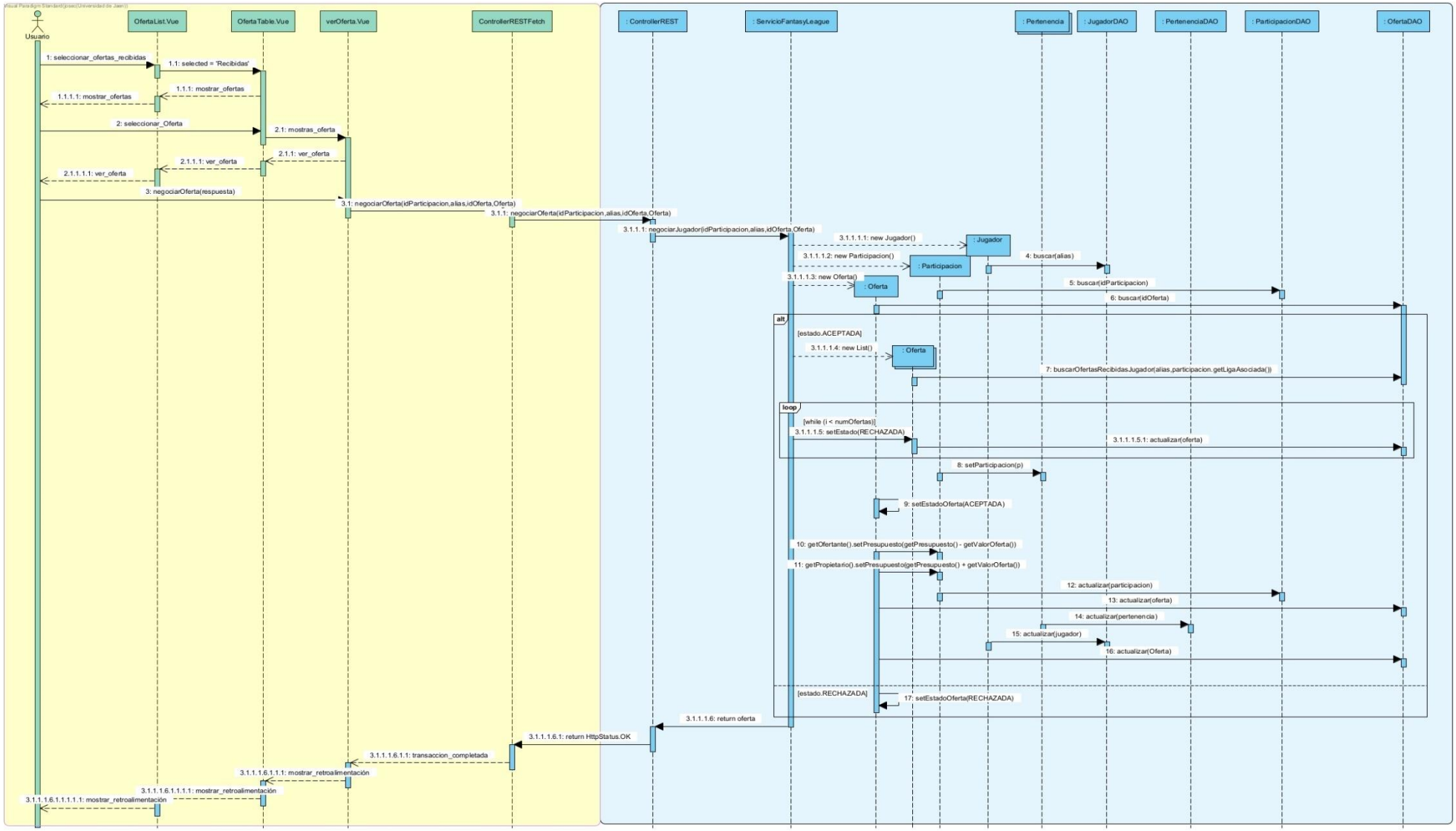

Ilustración 22: Diagrama de secuencia de negociar jugador

Como podemos observar, el usuario realmente interactúa con tres componentes, ya que OfertaList.Vue puede mostrar las ofertas realizadas o recibidas según el input que seleccione el usuario. Dependiendo de la respuesta que ha dado el usuario, el cliente se conecta con el servidor.

Si la respuesta ha sido que acepta la oferta, debemos rechazar todas las posibles ofertas que el usuario contenga además de la oferta en la que estamos trabajando. Además, actualizamos la pertenencia y el presupuesto de ambas participaciones.

En caso de que el usuario rechace la oferta, simplemente cambiamos el estado a RECHAZADA. En ambos casos, se devuelve un código de respuesta 200, indicando siempre que la oferta ha sido modificada correctamente y mostrándoselo al usuario.

3.4.3. Diagrama de secuencia: finalizar una jornada

Por último, vamos a mostrar el diagrama de secuencia de Finalizar Jornada, de la que se encarga el administrador:

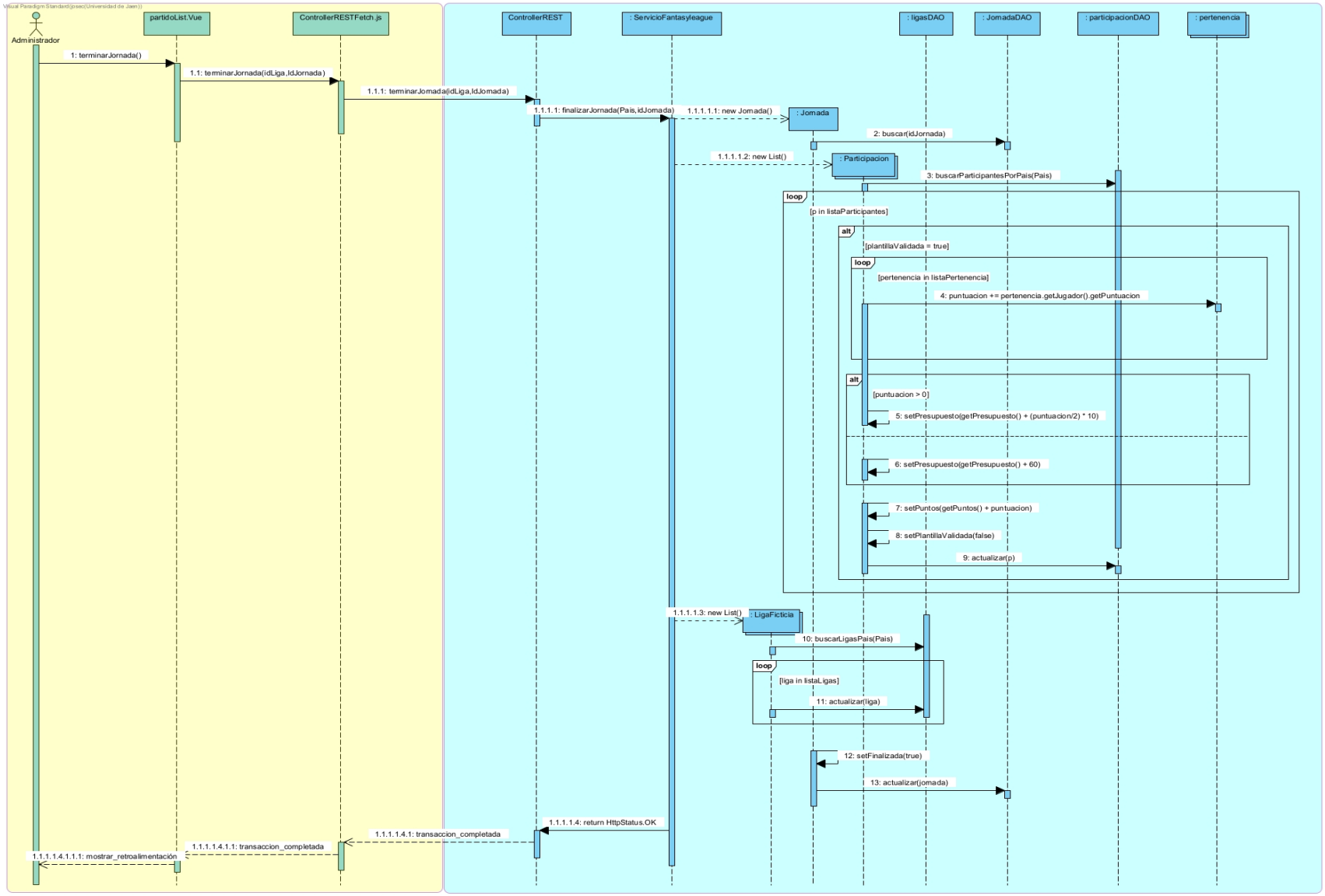

Ilustración 23: Diagrama de secuencia de finalizar jornada

El anterior diagrama es algo más simple de entender que el resto, aunque finalizar una jornada no es una operación tan trivial como se puede pensar, pues hay que buscar todos los participantes que tengan una participación en una liga del mismo país que la jornada a finalizar, y comprobar si tienen su plantilla validada. Si es el caso, se recoge toda la puntuación que han ido acumulando sus jugadores y se le asigna la puntuación total como puntos de la participación.

También se le asigna el presupuesto según el RF14. Por último, no nos debemos olvidar de actualizar todas las ligas cuyos participantes se han modificado y asignar como finalizada la jornada.

3.5 Diseño de la interfaz

La interfaz de la aplicación está formada por aquellos elementos con los que interaccionan agentes externos a la aplicación. En nuestro caso, al tener dos aplicaciones diferentes, para conectarlas es necesario un API REST que permita a la aplicación cliente interaccionar con la aplicación servidor.

En este apartado no vamos a ver solo el diseño de esta API REST, si no que también mostraremos el diseño de la interfaz gráfica de nuestra aplicación a través de wireframes con el objetivo de comunicar de manera clara y efectiva la funcionalidad y las opciones disponibles en la aplicación.

Este proceso de diseño también agiliza la productividad y aumenta la eficiencia, pues de esta forma no tenemos que crear toda esta etapa en la implementación desde cero, permitiendo mantener la consistencia y coherencia de la aplicación, y facilitando la comprensión para usuarios externos.

3.5.1 Diseño del API REST

La interfaz de la aplicación está formada por aquellos elementos con los que interaccionan agentes externos a la aplicación. En nuestro caso, al tener dos aplicaciones diferentes, para conectarlas es necesario un API REST que permita a la aplicación cliente interaccionar con la aplicación servidor.

Un servicio web REST es un patrón de arquitectura que no define un protocolo ni formato de mensaje concreto integrado en el protocolo HTTP. Está mejor adaptado a clientes con capacidades limitadas como móviles o accesos desde una página web mediante JavaScript y está orientado al acceso a recursos.

Este tipo de servicios codifican nuestro servicio como una serie de recursos accesibles mediante URLs. Dichas URLs nunca deben incluir referencias a operaciones a realizar sobre el recurso.

Las operaciones se realizan mediante los métodos estándar HTTP (GET, PUT, DELETE, POST) y utiliza los códigos de respuesta estándar para indicar el resultado de la operación (200 OK, 201 CREATED, etc).

Todo el proceso de documentación del API REST se ha realizado de forma dinámica a partir del controlador que se ha implementado, y siguiendo la guía mostrada en [18].

Como las URLs implementadas en el controlador son demasiadas como para dar una visión de conjunto clara, vamos a resumir en una tabla los recursos principales junto a una breve explicación de los que se encarga cada uno, como podemos ver en la tabla 9:

Recurso	Métodos HTTP	Descripción
/login	POST	Este es una caso especial, pues, a contrario de los demás recursos, se ha especificado un recurso único y diferenciado del resto para realizar el inicio de sesión en la aplicación
/usuario	POST, GET	Los recursos que empiecen por /usuario comprenderán todas las acciones que puede hacer el usuario (que no la participación de un usuario), como puede ser dar de alta un nuevo usuario, crear una liga o ver las participaciones asociadas a ese usuario
/participacion	ALL	Este recurso comprende la mayoría de operaciones que se pueden hacer en la aplicación, desde fichar un jugador, pasando por crear una oferta o vender un jugador, pues realmente la clase Participacion es el punto central donde se cometen la mayoría de transacciones. También utiliza todos los métodos HTTP disponibles
/liga	GET, PUT	Aquí se realizan las operaciones asociadas a una liga y no una participación, como puede ser obtener las jornadas de una liga, ver las ofertas para un jugador dentro de una liga o ver sus participantes.
/jugador	GET	Este recurso permite obtener las puntuaciones asociadas a un jugador o mostrar un jugador en concreto
/pertenencia	PUT	Este recurso solo comprende un URL, que permite poner cláusula a un jugador. Realmente se podría meter dentro de /participacion, puesto que una pertenencia depende de su participación, pero como detallaremos más adelante, se han acortado algunos recursos con el objetivo de no hacer URLs demasiado extensas
/admin	GET, POST, PUT	Este recurso comprenden todas las operaciones que puede realizar un usuario con permisos de administrador, como puede ser puntuar un jugador o finalizar una jornada.

Tabla 9: Recursos principales del controlador

Algunas consideraciones a tener en cuenta respecto al diseño del API REST:

- El controlador contiene 20 operaciones GET, 7 operaciones POST, 8 operaciones PUT y 1 operación DELETE
- Se ha evitado el uso de verbos o indicaciones de la operación a realizar en la propia URL

- Se ha evitado el uso de parámetros en la URL para indicar valores de atributos durante una creación o actualización
- Una URL puede tener asociados diferentes métodos HTTP que permita diferentes acciones u operaciones
- Para evitar consultas excesivamente largas, se ha realizado un filtrado evitando indicar en las URLs aquellos recursos que realmente no son necesarios para llevar a cabo una operación.

  Por ejemplo, para mostrar todas las ofertas que tiene una participación, la URL ideal sería "/usuario/{nombreUsuario}/participacion/{id}/ofertas", pero como para saber las ofertas asociadas a una participación basta con saber el id de la participación, se ha acortado a "/participacion/{id}/ofertas".

Como ya veremos más adelante, esto también se ha realizado a nivel de seguridad de la aplicación, pero para facilitar el acceso a los recursos por roles, se ha añadido un /admin al principio de cada URL cuyos recursos sólo pueden acceder los usuarios con rol de administrador.

A continuación, vamos a mostrar las URLS implementadas junto al método que utilizan además de una breve descripción, aunque toda la información sobre las URLS (códigos de respuesta, parámetros etc) utilizadas en el controlador la podemos encontrar en el documento compartido (véase Apendice III) o iniciando la aplicación y accediendo a http://localhost:8080/swagger-ui/index.html#/.

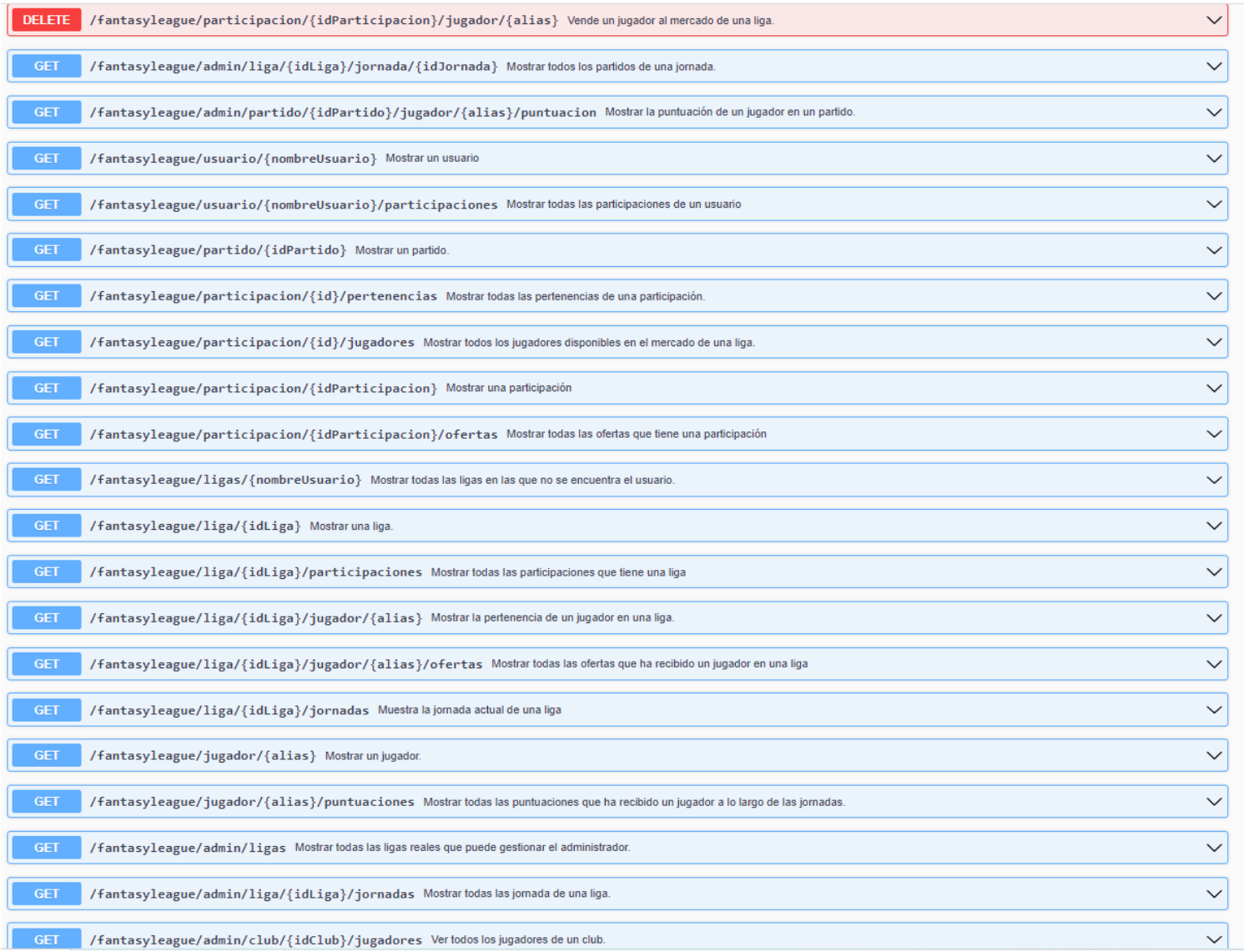

Ilustración 24: Operaciones GET y DELETE del controlador

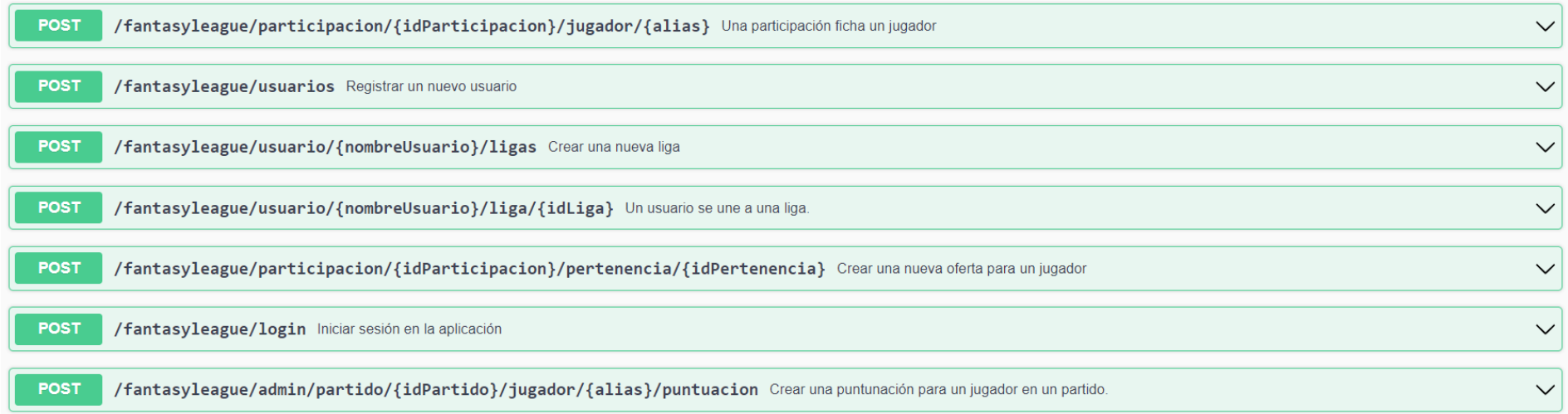

Ilustración 25: Operaciones POST del controlador

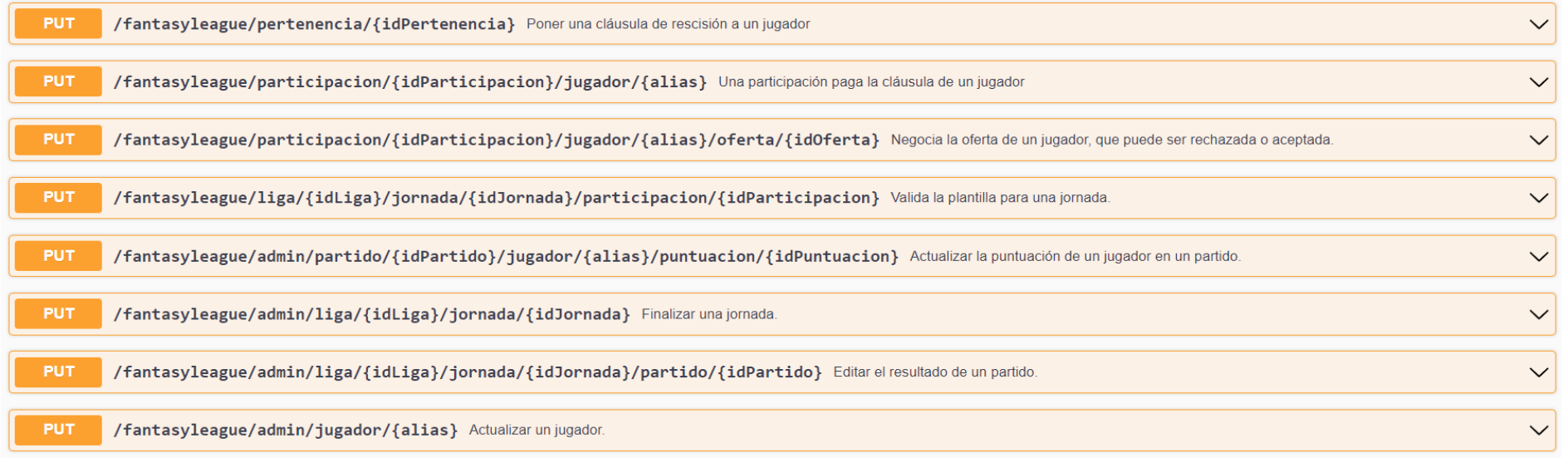

Ilustración 26: Operaciones PUT del controlador

Ilustración 27: Objetos con los que interactúa el API REST

Como son muchos los métodos implementados, vamos a mostrar cómo se vería uno de ellos completamente, para dar una visión de cómo se verían los demás.

En este caso vamos a mostrar la operación de vender un jugador, que es el único método DELETE en nuestra aplicación:

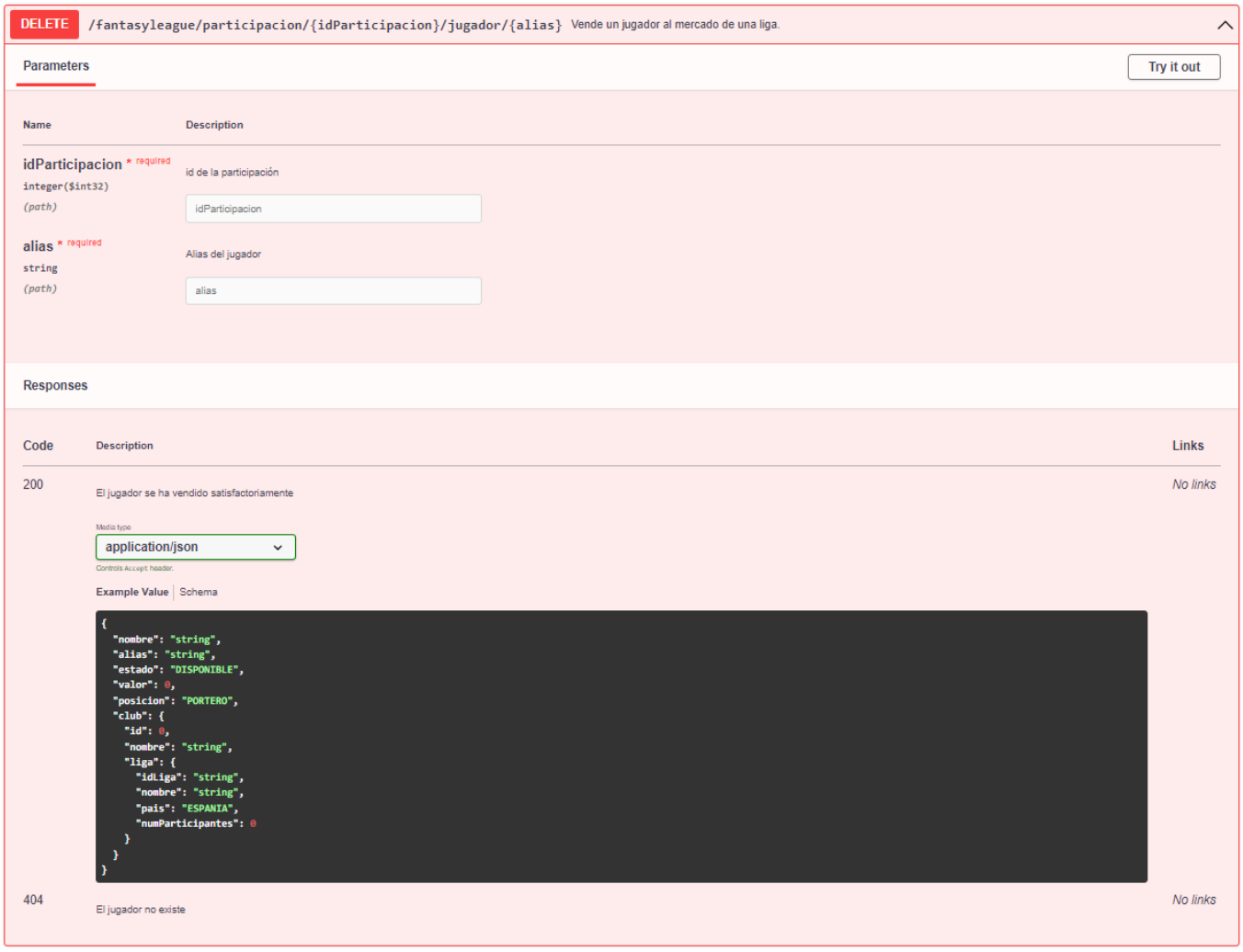

Ilustración 28: Operación completa de vender un jugador al mercado

La anterior ilustración permite ver la descripción de los atributos que utiliza, así como la estructura del objeto en formato JSON con el que interactúa y los códigos de error que puede enviar.

3.5.2 Diseño de la interfaz gráfica

Para el diseño de la interfaz gráfica de nuestra aplicación web se ha intentado hacer de la forma más intuitiva y simple posible, sin sobrecargar de colores la pantalla siguiendo el requisito RNF03.

Aquí vamos a mostrar algunos wireframes de los casos de uso más representativos de nuestra aplicación, ya que muchas pantallas que no mostraremos son realmente listas para seleccionar jugadores o ligas o formularios para registrarse o crear nuevas ligas.

Empezando por el principio, el wireframe para iniciar sesión en la aplicación lo podemos ver en la ilustración 29:

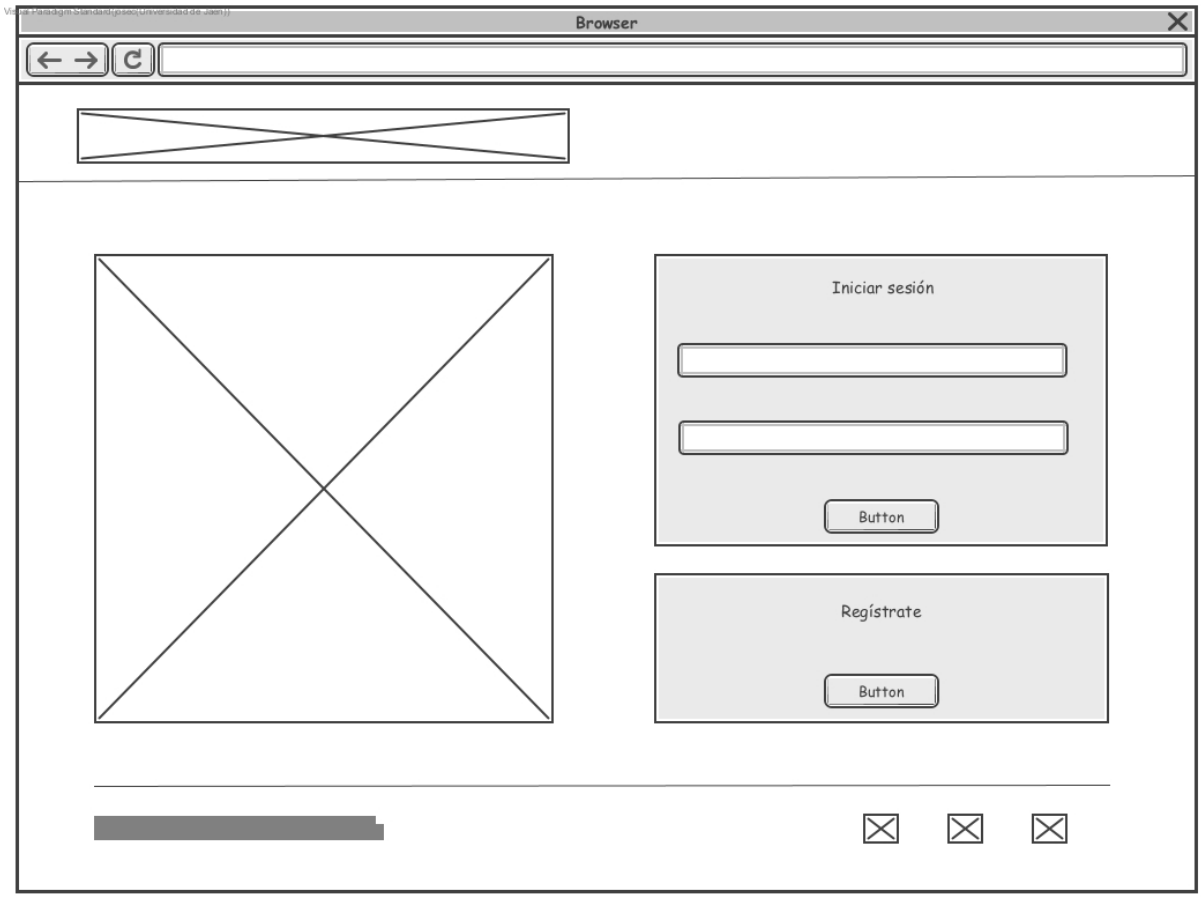

Ilustración 29: Wireframe de inicio de sesión

Más tarde analizaremos en un diagrama jerárquico toda la lógica de redireccionamiento de la aplicación, que indique como llegar a todas las pantallas desde la página de inicio, aunque en la ilustración anterior se puede intuir. Esta página nos permitirá redirigirnos a una página de registro o a diferentes páginas dependiendo de si nuestro rol al registrarnos es el de un usuario o un administrador.

Una vez hemos iniciado sesión como usuarios, la siguiente página que es mostrada al usuario es la siguiente:

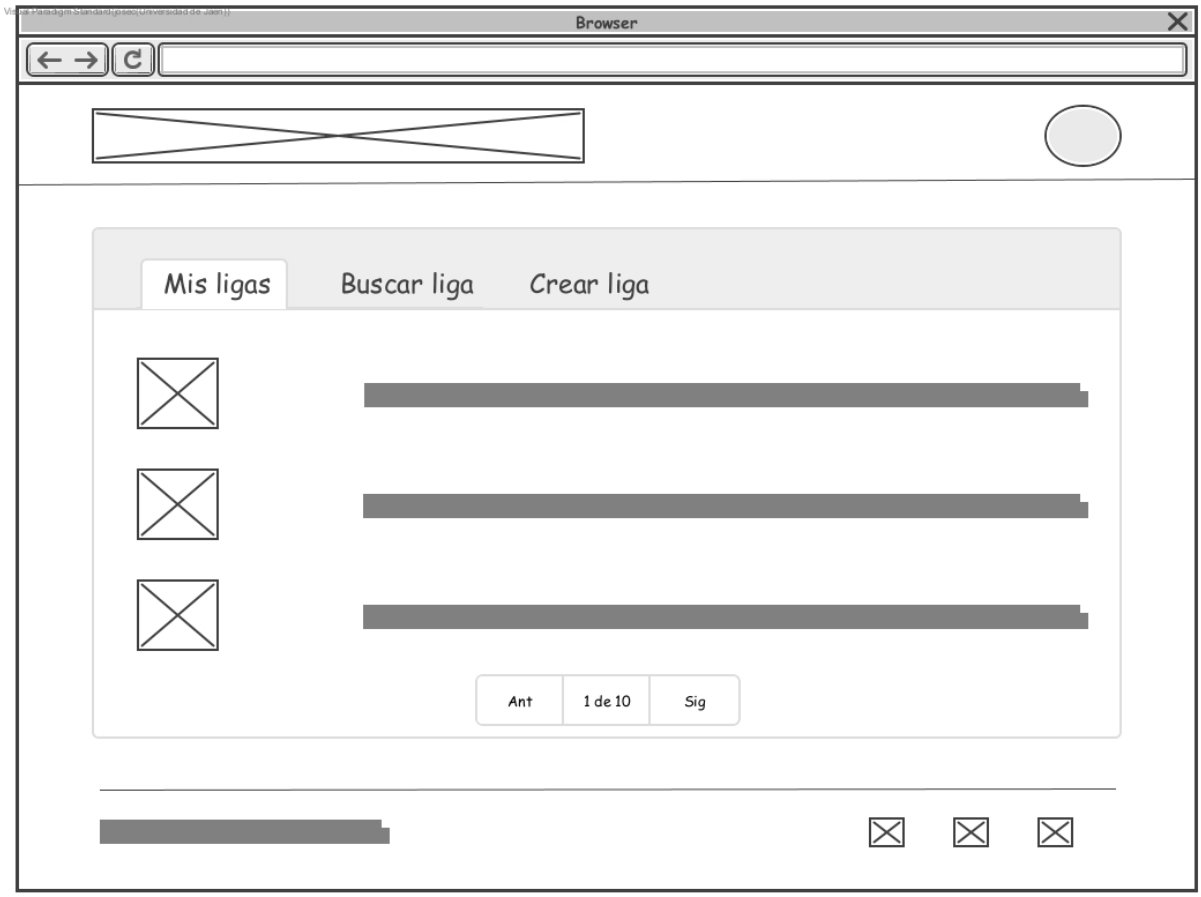

Ilustración 30: Wireframe de la pantalla "mis ligas"

Como se puede apreciar, esta pantalla permite al usuario seleccionar la liga que desea administrar. Destacar también que en cualquier tabla se produce una paginación del mismo estilo que en la anterior ilustración (si fuese necesario) y que el diseño de muchas pantallas, como veremos adelante, está inspirado en el uso de tabs, de esta forma el usuario navega por nuestra forma de forma más intuitiva y no teniendo que pulsar enlaces que redireccionan a las pantallas que deseen.

En la pestaña de “buscar liga” se mostrará una tabla con todas las ligas y en la pestaña de “crear liga” se mostrará un formulario que permita crear una nueva liga

El siguiente wireframe muestra la pantalla en la que interactúa el usuario una vez seleccionada la liga en la que participa:

Ilustración 31: Wireframe de la pestaña "gestionar plantilla"

En la anterior ilustración podemos ver que al acceder a una participación en específico, el número de tabs se amplía, ya que ahora se muestran las funcionalidades principales de la aplicación.

Aquí solo vamos a mostrar dos funcionalidades más (fichar un jugador y gestionar equipo, visualizables en las ilustraciones 32, 33 y 34), aunque las demás funcionalidades son bastantes intuitivas: en la pestaña clasificación mostrará un listado con cada usuario que participe en la misma liga ordenados por los puntos obtenidos y en la pestaña ofertas se mostrarán las ofertas recibidas y realizadas.

Además, en esta pestaña se ha añadido un círculo junto al número de ofertas nuevas que ha recibido el usuario por sus jugadores. De esta forma el usuario no necesita estar constantemente revisando esta pestaña revisando se ha recibido ofertas por sus jugadores.

Respecto a la ilustración 31, esta pantalla sirve para validar una plantilla para una jornada. En la ilustración se muestra un ejemplo, pero ambas tablas de banquillo o alineación pueden ser tan grandes como jugadores tenga el usuario en su equipo. Además, en las tablas se debe mostrar una foto del jugador junto a su alias para que el usuario pueda identificar fácilmente qué jugador está alineando.

Para fichar un jugador, el usuario selecciona la pestaña “mercado”. En dicha pestaña, se muestra una tabla con todos lo jugadores de la liga, en las que el usuario los puede buscar por nombre u ordenarlos por su valor de traspaso entre otras acciones:

Ilustración 32: Wireframe de la pestaña "Mercado"

Cuando el usuario pulsa esta pestaña, se muestra una tabla con todos los jugadores de la liga junto con sus datos y una foto de perfil. Sin embargo, lo interesante se muestra al pulsar el botón “fichar”, como podemos ver en la ilustración 33:

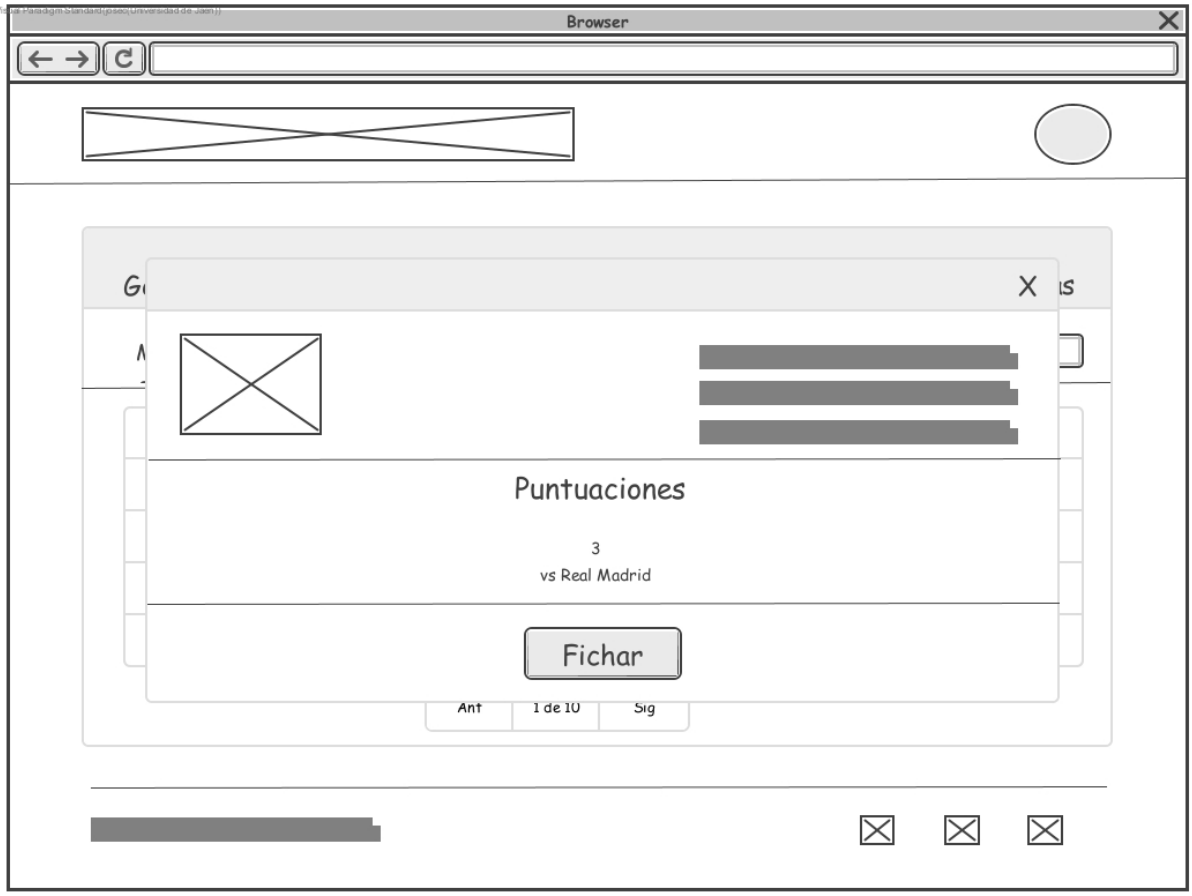

Ilustración 33: Wireframe del Modal para fichar un jugador

Al pulsar el botón “fichar”, se muestra una ventana con toda la información relevante del jugador. Esta ventana puede cambiar si el jugador ya pertenece a otro jugador, donde se indicará a qué usuario pertenece y cambiaría el botón “fichar” por dos botones llamados “pagar cláusula” (si tuviese cláusula) y “enviar oferta”.

Para finalizar, vamos a mostrar la pestaña “gestionar equipo”, que aunque solo contiene un listado con los jugadores fichados que tiene el jugador, también incluye otras funcionalidades como vender el jugador o ponerle una cláusula:

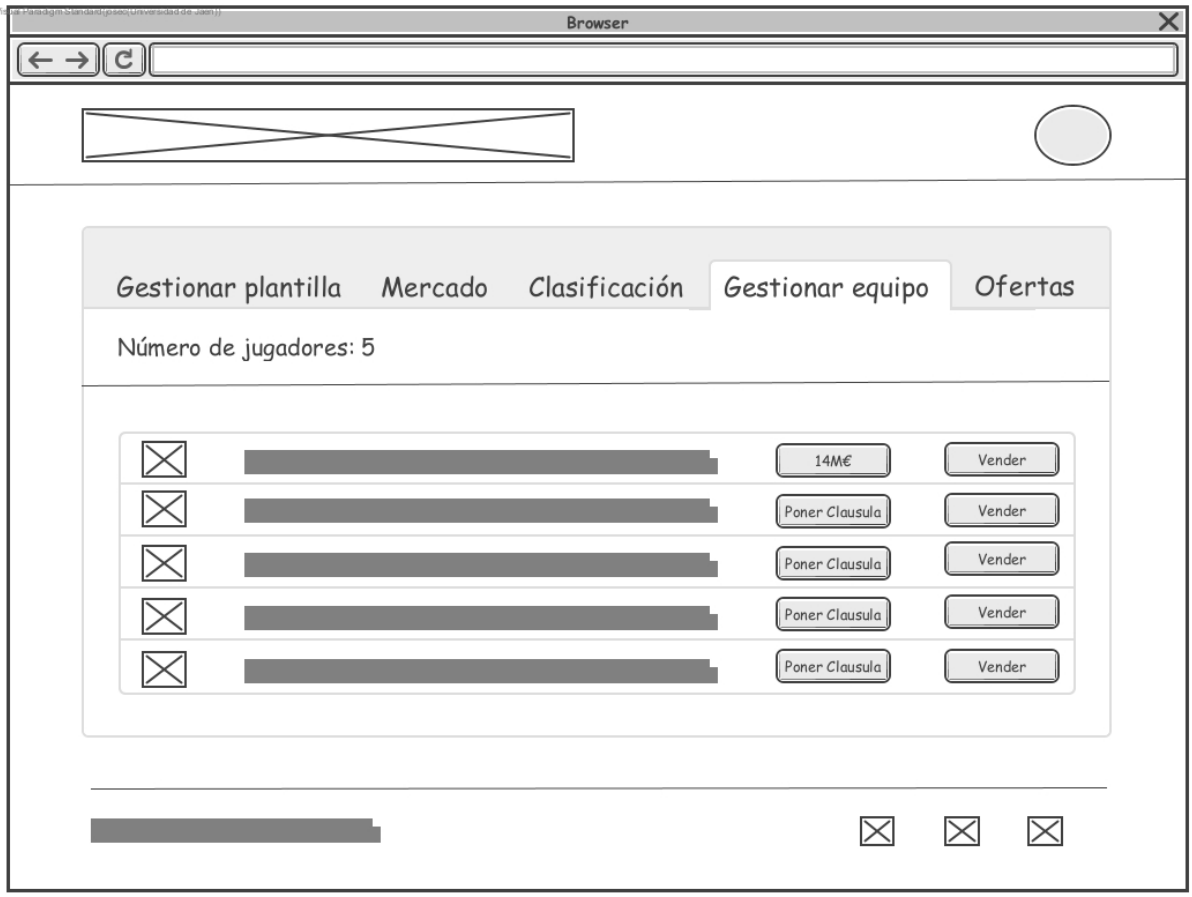

Ilustración 34: Wireframe de la pestaña "Gestionar equipo"

Esta pantalla muestra todos los datos del jugador junto a su foto más la opción de añadir/modificar su cláusula de rescisión.

Una vez mostrados los wireframes principales para el usuario, vamos a mostrar un par de pantallas para mostrar la interfaz que contiene las funciones principales para un administrador. Vamos a empezar con un wireframe que permite al administrador seleccionar un partido:

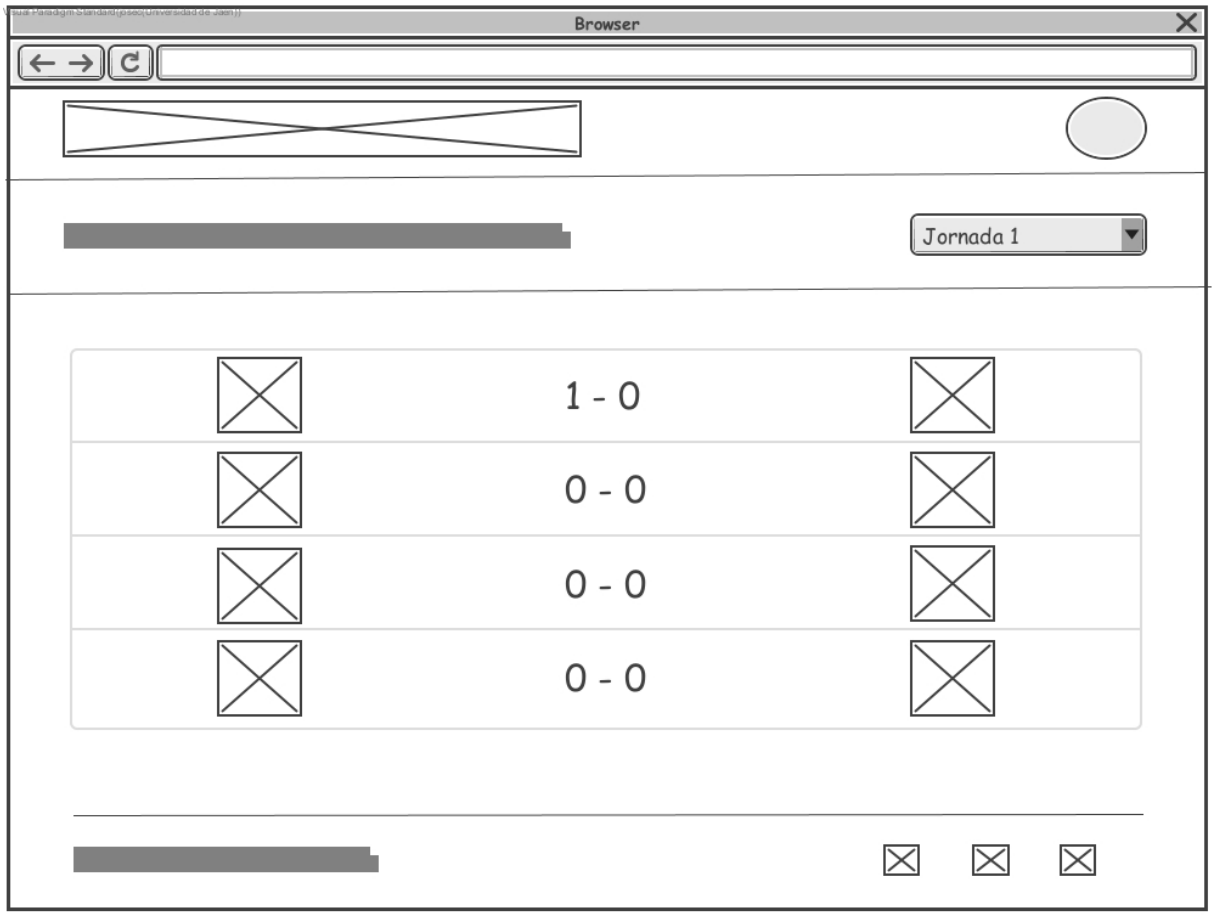

Ilustración 35: Wireframe de la pantalla de seleccionar un partido (administrador)

En la anterior ilustración podemos ver cómo, una vez el administrador ha seleccionado la liga que desea gestionar, se le muestran todos los partidos eligiendo una jornada en concreto. De forma similar al funcionamiento de fichar jugador, el administrador selecciona el partido que desea editar y se abre una ventana que permite su edición, como podemos ver en la ilustración 36:

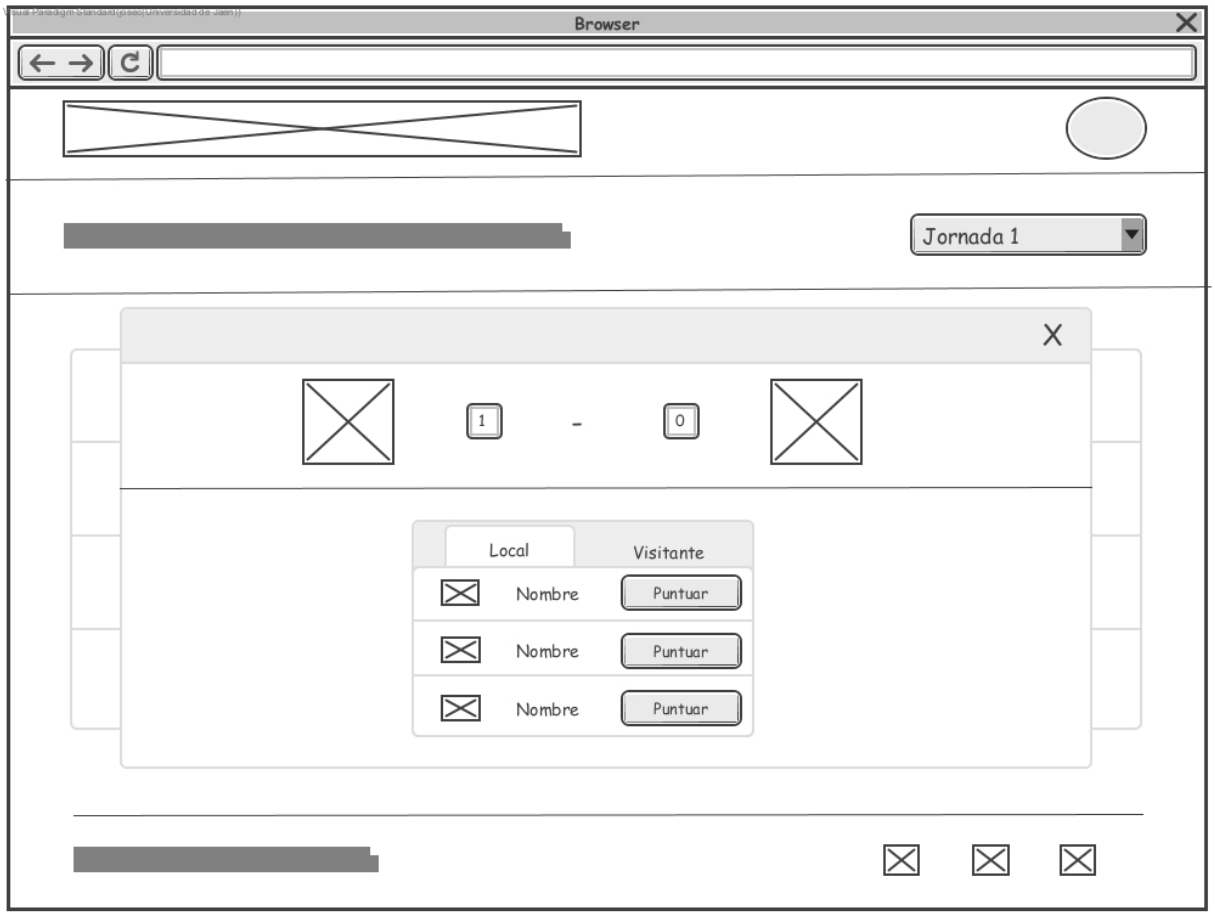

Ilustración 36: Wireframe de la pantalla de editar un partido (administrador)

Al seleccionar un partido se muestra una ventana que permite editar un resultado y seleccionar un jugador para puntuarlo. Al pulsar el botón “puntuar” de un jugador, se mostrará otra ventana con el formulario necesario para puntuar un jugador en específico en un partido.

Cabe recalcar que estos diseños no son totalmente fieles al diseño final, ya que puede cambiar algún botón o algún elemento según la tecnología con la que hagamos la interfaz, pero sí que comparten la misma distribución de los elementos y el resultado final se asemejará en un 90%.

Para finalizar este apartado, vamos a mostrar un diagrama jerárquico que nos permita analizar las conexiones entre las pantallas y sus componentes asociados. Este diagrama nos permitirá saber cómo acceder a cualquier página desde la pantalla inicial de nuestra aplicación:

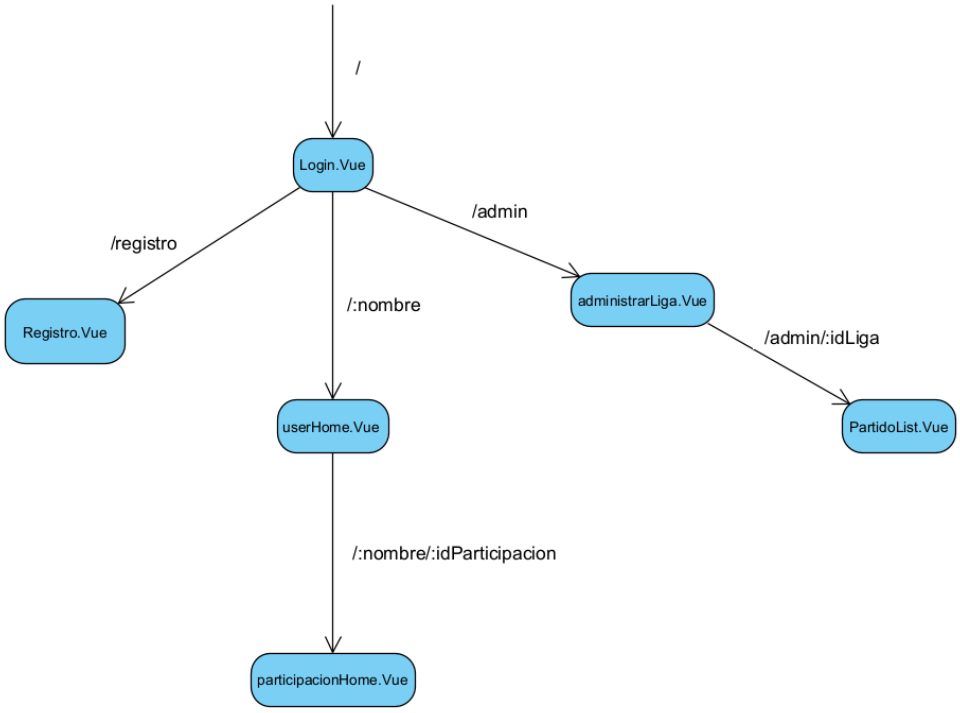

Ilustración 37: Diagrama jerárquico de las conexiones entre páginas

Como se puede observar, este diagrama es bastante simple, ya que realmente lo interesante está dentro del componente, pues aunque aquí solo se indique un componente, éste a su vez puede asociarse con otros 5 dentro de él. En ese caso es interesante comparar el diagrama anterior con el diagrama de clases en la parte del cliente (ilustración 20), donde se pueden ver todas las relaciones entre los componentes de la aplicación.

Destacar también que aquellas rutas que contienen el carácter ":" antes de la ruta (por ejemplo /:nombre) significa que son rutas dinámicas, es decir, que dependiendo del nombre de usuario, esta ruta se puede llamar /jcb00034 o cualquier otro nombre (o id en otro caso).

Por último, pero no menos importante, también existe un componente, llamado forbidden.Vue, en la que se accede con la url /error. Este componente realmente se relaciona con todos los demás vistos en la ilustración 37, y se llevará hasta él si se produce algún intento de vulnerar la seguridad de la aplicación.

Por ejemplo, si yo me registro en la aplicación como un usuario, no podré acceder a los componentes referenciados con “/admin”, o si me registro como “jcb00034”, no podré acceder al componente userHome.Vue del usuario “joseCarpio” ni a participaciones de ligas donde no esté unido.

Este componente es importante resaltarlo a parte, pues aunque no lo hemos indicado en el diagrama jerárquico, existe en la aplicación y un error puede llevarnos hasta él.

3.6 Plan de pruebas

El plan de pruebas diseñado para la aplicación se basa en la utilización de Test Driven Development (TDD) para la parte del servidor y pruebas de aceptación por parte del usuario en el cliente. Esto se debe a que realmente las funcionalidades importantes están implementadas en el backend (como hemos ido mostrando en los diagramas de secuencia), mientras que en el frontend la mayoría de funciones van a ser asíncronas y pueden provocar ciertos problemas de sincronización a la hora de probarlas.

La metodología TDD consiste en escribir sólo el código necesario para arreglar un test que está fallando. Esta metodología se basa en un ciclo, llamado ciclo TDD, donde primero se escribe la prueba (rojo), después se escribe el código que responde a la prueba (verde) y por último simplificar el código (azul).

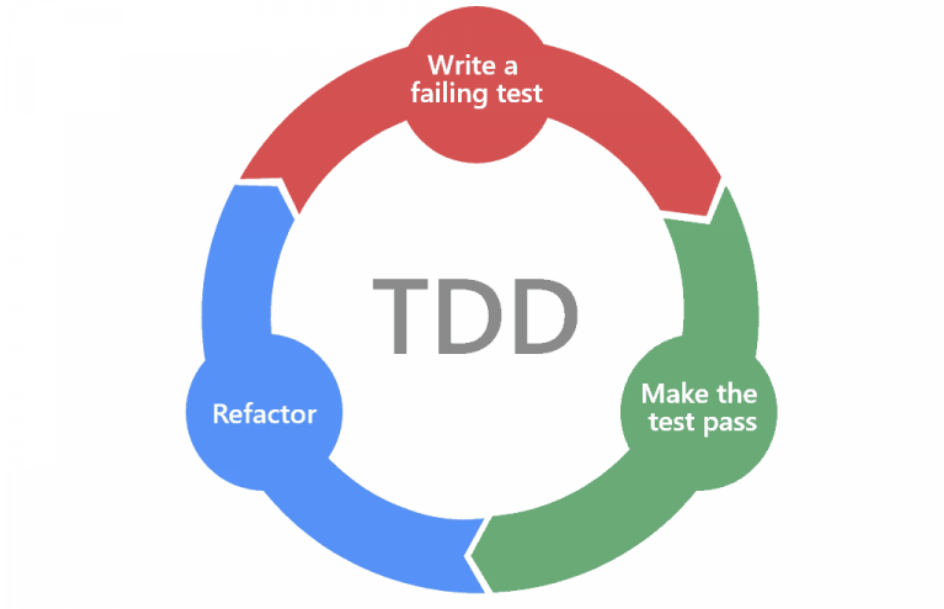

Ilustración 38: Ciclo TDD [19]

La elección de esta metodología se debe a los beneficios que puede proporcionar una buena práctica de esta metodología, como por ejemplo que proporciona evidencias concretas de que el software funciona, favorece la eliminación de código repetitivo gracias a la refactorización y proporciona un código más limpio y simple al solo codificar lo que se necesita.

Para la correcta implementación de este enfoque, se va utilizar el módulo jUnit en la parte del servidor, que es un framework de pruebas para Java que proporciona un conjunto de bibliotecas y anotaciones que permiten a los desarrolladores escribir y ejecutar pruebas automatizadas.

Respecto a la planificación de pruebas, se van a realizar pruebas de unidad e integración en la parte del servidor, y pruebas de aceptación en la parte del cliente.

En las pruebas de unidad se realiza un testing independiente de cada clase de la aplicación mostradas en la ilustración 18. Debido a que dichas clases no contendrán muchas funciones, simplemente se comprobará que su código hace lo que se debe.

El enfoque de las pruebas de unidad es el de las técnicas de caja blanca, consistente en el análisis minucioso del código y que permite comprobar el paso de parámetros en cada función (esto nos será útil para comprobar el correcto funcionamiento del Bean Validation, que detallaremos más adelante en la implementación). En este tipo de pruebas no se puede aplicar TDD al no favorecer la factorización.

En las pruebas de integración, se comprobará que las distintas unidades se relacionan correctamente entre sí. Este tipo de enfoque es habitual en las pruebas de caja negra, consistente en el análisis de los requerimientos funcionales y que son independientes de la implementación: se enfocan en el resultado que se espera, sin importar cómo está programado. En este tipo de pruebas sí se aplicará la metodología TDD.

Por último, se van a plantear pruebas para la verificación automática de los métodos del API REST diseñado. Estas pruebas se realizarán con jUnit y en específico con la biblioteca TestRestTemplate, que permite realizar peticiones HTTP, y LocalServerPort, que permite enviar las peticiones desde un puerto aleatorio al puerto donde está escuchando el API REST.

Hay que recalcar que estas pruebas se centrarán en que los códigos de respuesta sean correctos y no en que se han modificado los datos de forma correcta, ya que ese tipo de comprobaciones se harán en las pruebas de integración.

En las pruebas de aceptación, se comprobará que el frontend está correctamente implementado desde el punto de vista del cliente antes de poner en marcha el sistema. Estas pruebas se centrarán en probar el correcto funcionamiento de la aplicación, validar el cumplimiento de los requisitos y aspectos de usabilidad.

Es importante indicar que el plan de pruebas se centrará en comprobar todas las funcionalidades detalladas en la especificación de los casos de uso mediante las pruebas en el servidor, considerando que no hay problemas tanto en las acciones principales como en los itinerarios alternativos de cada caso y en la validación y comprobación de que se han cubierto todos los requisitos, tanto funcionales como no funcionales, mediante las pruebas en la parte del cliente.

También se diseñarán una serie de pruebas, tanto en la parte del cliente como en el servidor, que comprueben que cada perfil de usuario realiza sus acciones específicas en la aplicación, y que, por ejemplo, un usuario no puede realizar las acciones de un administrador.

4. Implementación

4.1. Arquitectura

Recordando un poco la propuesta de solución explicada en el apartado 2.2, la arquitectura elegida es una aplicación monolítica, utilizando el framework Vue.js en la parte del cliente y el framework Spring en la parte del servidor. También se utilizará una base de datos MySQL que permita almacenar los datos de la aplicación.

Aunque estas tecnologías son las principales, no son las únicas que se utilizan en todo el proyecto, como detallaremos a continuación:

Para la integración de Vue.js en la parte del cliente, se utilizará el framework Vite. Vite es un framework de desarrollo web creado para mejorar la velocidad y la productividad en el desarrollo de aplicaciones web front-end.

Más información sobre Vite la podemos encontrar en [20], pero resumiendo, Vite permite un desarrollo en tiempo real eficiente, reduce el tiempo de inicio de la aplicación y es compatible con varios lenguajes y frameworks populares, aunque es especialmente conocido por su integración con Vue.js

Hay que destacar también el uso de Bootstrap en la parte del cliente, que es un framework CSS de código abierto ampliamente utilizado para el desarrollo de sitios web y aplicaciones responsive. Bootstrap proporciona una colección de estilos CSS predefinidos, que junto al uso de Vue.JS, proporciona componentes reutilizables y listos para usar según las directrices que establece Vue.

Para la capa de persistencia en la parte del servidor, se va a utilizar un ORM, como ya se mencionó anteriormente. Concretando un poco más esta parte, lo que realmente se utilizará será Hibernate, que es un framework de ORM para el lenguaje de programación Java, y que actuará como proveedor de persistencia subyacente.

Resumiendo un poco, el objetivo principal de Hibernate es simplificar el desarrollo de aplicaciones que utilizan bases de datos relacionales eliminando gran parte del código de bajo nivel y las tareas repetitivas asociadas al acceso de datos.

Un resumen de todos los elementos que constituyen nuestra arquitectura la podemos ver en la ilustración 39:

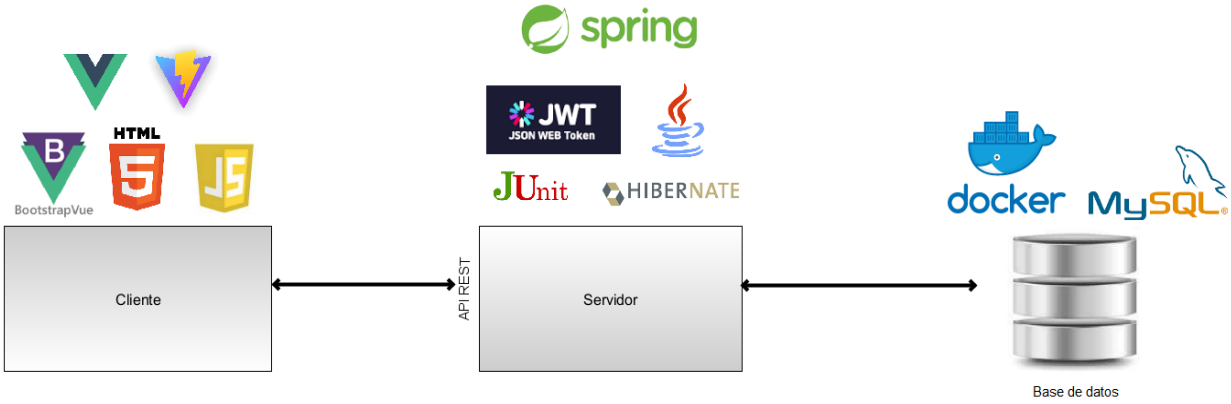

Ilustración 39: Diagrama arquitectónico del sistema

La ilustración anterior proporciona un resumen de las tecnologías usadas en la arquitectura del sistema.

Cabe recalcar que hay elementos, como Spring Security o Router-Vue, que no se indican en el diagrama al ser bibliotecas de los frameworks Spring y Vue respectivamente. Todos estos recursos externos están documentados en el apartado 4.2 con más detalle.

4.2. Detalles sobre implementación

En este apartado vamos a comentar los aspectos técnicos sobre la implementación, así como el uso de bibliotecas, frameworks o recursos empleados,

Al tener dos aplicaciones, vamos a dividir en dos subapartados indicando los aspectos más relevantes de cada aplicación. Igualmente, antes de entrar en temas más específicos de cada aplicación, resaltar que se ha utilizado la versión 3.3.1 para Vue,JS y la versión 2.5.5 para Spring framework, integrada en la versión 17 de Java para Maven.

4.2.1. Estructura de los contenidos de las aplicaciones

Un vistazo rápido a la distribución de componentes de la aplicación cliente lo podemos encontrar en la ilustración 54:

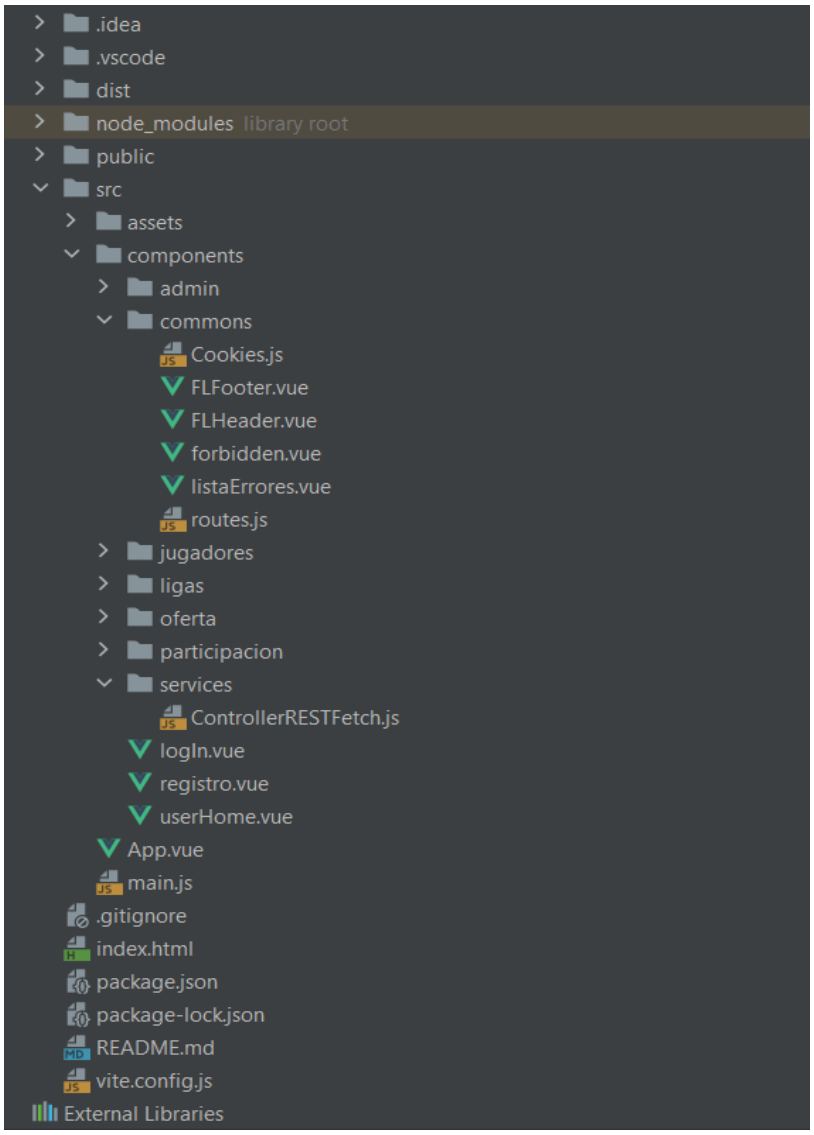

Ilustración 40: Esquema de contenidos de la aplicación cliente

Esta aplicación contiene las imágenes utilizadas en el proyecto en la carpeta “assets”, en la carpeta “components” se encuentran todos los componentes utilizados.

Estos componentes están distribuidos en carpetas según la entidad en la que trabajan, aunque también se suministra dos carpetas adicionales, “commons” para componentes o clases que utilizan los demás componentes, y la carpeta “services” donde se encuentra la clase que realiza peticiones al API REST del servidor.

Respecto a la aplicación servidor, sus clases están distribuidas en paquetes según a la clase a la que pertenezcan, como podemos ver en la ilustración 55:

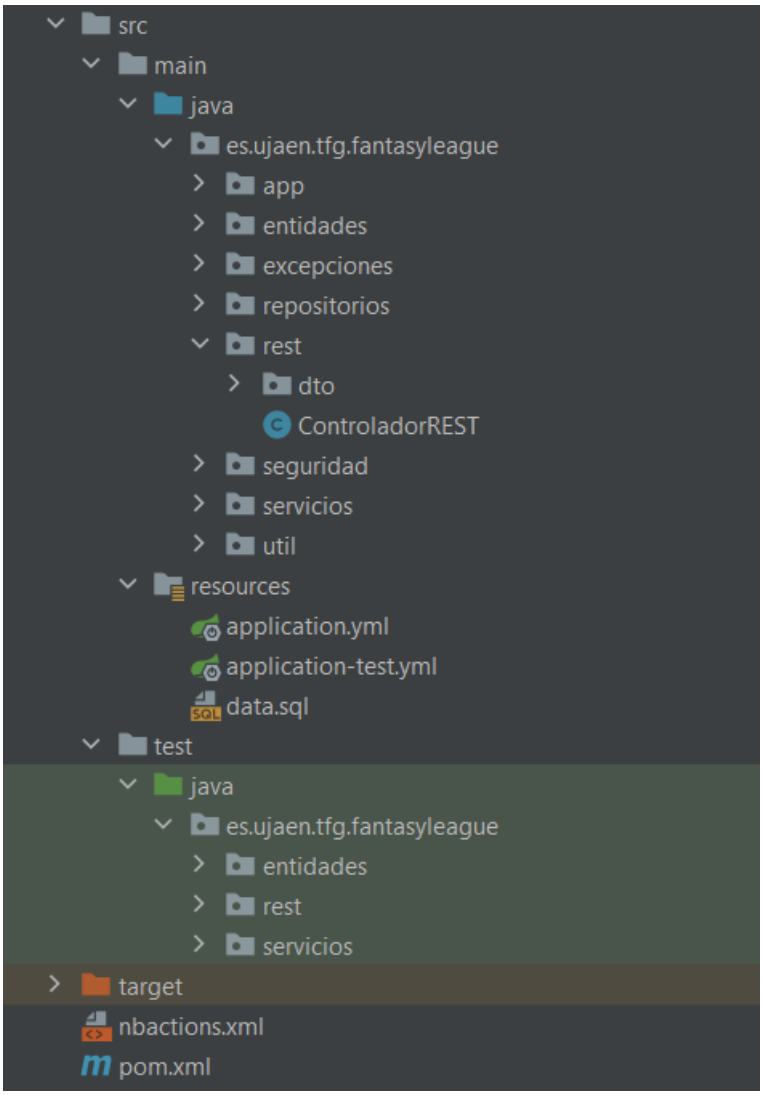

Ilustración 41: Esquema de contenidos de la aplicación servidor

Esta aplicación está distribuida en tres carpetas: una principal donde se encuentran todas las clases de la aplicación, llamada “main”; otra donde podemos encontrar los ficheros de configuración de la aplicación, que se encuentra dentro de la otra aplicación, llamada “resources”, y por último la carpeta de testing.

4.2.2. Implementación de la capa de dominio

Vamos a empezar por el servidor, ya que realmente es la aplicación más compleja en cuanto a funcionalidades y emplea más recursos y bibliotecas externas.

Aunque al estudiar las tecnologías propuestas para afrontar el proyecto ya mencionamos algunas citas bibliográficas importantes, toda la información sobre los recursos utilizados en este apartado y su implementación la podemos encontrar en [21].

Para facilitar la configuración de la aplicación Spring, se utilizará Spring Boot que simplifica y acelera este proceso de configuración inicial. Spring Boot ofrece varias ventajas, pero una de las más importantes es la configuración automática que ofrece, al detectar y configurar automáticamente las dependencias y los componentes necesarios para la aplicación, además utilizar Apache Maven o Gradle como herramientas de gestión de dependencias, que facilita la incorporación y la gestión de dependencias en el proyecto.

Al igual que en el diseño, vamos a comentar los aspectos más relevantes de cada capa implementada, siendo éstas la capa de dominio, persistencia y servicios.

En la capa de dominio, lo más resaltable es el uso de Bean Validation en las clases del sistema. El Bean Validation o Validación de beans permite evitar mucho código de comprobación reiterativo y de escaso interés.

Este proceso se centra sobre todo en garantizar que los argumentos de operaciones o los atributos de las entidades tengan valores correctos, evitando por tanto inconsistencias.

Realizar esta comprobación de manera programática implica incluir comprobaciones explícitas reiterativas que “ensucian” el código del dominio realmente relevante.

Con la utilización de Bean Validation, todo este código repetitivo nos lo podemos ahorrar mediante un conjunto de anotaciones, además de permitirnos validar los parámetros de cualquier operación de un Bean.

Un ejemplo de uso lo podemos encontrar en la ilustración 40, más concretamente en la clase usuario:

```
@Entity
public class Usuario implements Serializable {

    @Id
    @NotBlank
    String nombre;

    @NotBlank
    String clave;

    @Email
    String email;

    @NotNull
    Rol rol;
```

Ilustración 42: Ejemplo de uso de Bean Validation en la clase Usuario

Como se puede apreciar en la ilustración anterior, se hace uso de las anotaciones @NotBlank para indicar que un atributo de tipo String no puede ser una cadena vacía, @NotNull para indicar que ese atributo no puede ser igual a Null, y la anotación @Email que indica que ese atributo debe contener una dirección de email válida.

4.2.3. Implementación de la capa de persistencia

En la capa de persistencia, se ha utilizado la especificación JPA haciendo uso del framework Hibernate para ayudar con el mapeado de objetos a la base de datos. JPA usa anotaciones en el código Java para indicar la información de mapeado, ya que este mapeado de una clase a una tabla relacional no es 100% automático.

Aunque aquí vamos a detallar y resumir como se ha realizado este proceso, es quizás el proceso más complejo de la aplicación, por lo que una profundización mayor de todos los conceptos que vamos a tratar la podemos ver en [22].

Un pequeño adelanto del uso de algunas anotaciones la podemos ver en la ilustración 40, aunque vamos a explicar detalladamente como se ha realizado este mapeado a continuación:

- Toda entidad de la aplicación debe contener la anotación @Entity y debe poseer un constructor sin argumentos.
- Hay que marcar el atributo o propiedad que hace de clave primaria de la tabla con la anotación @Id. También podemos pedir que JPA autogenere el valor de las claves primarias (un ejemplo lo veremos en la ilustración 41).
- El mapeado de relaciones es la parte más delicada del mapeado de una clase. Aunque existen 4 tipos de relaciones, hemos podido analizar a lo largo del diseño de la aplicación que en nuestro diagrama de clases sólo contiene relaciones del tipo uno a muchos y muchos a uno, lo cual simplifica bastante este proceso.

  En estos casos, se debe poner @OneToMany en la clase que contenga la colección de referencias, y @ManyToOne en la clase opuesta. Además, en el caso de que esta relación fuese bidireccional, hay que agregar el elemento mappedBy para ligarla con la relación principal en la otra clase.

- Todas las relaciones @OneToMany utilizan una carga perezosa de los datos, mientras que las relaciones @ManyToOne utiliza la carga en cascada. Sin embargo, hay excepciones en el caso de las relaciones @OneToMany, donde algunas relaciones, como por ejemplo entre las clases Usuario y Participacion, donde Usuario sólo se relaciona con esa clase y lo más normal es que siempre que la clase Usuario realice una operación incluya dentro de ella su colección de Participacion, utilicen una carga en cascada al existir una gran probabilidad de acceder a los objetos relacionados.

  Otro ejemplo lo podemos encontrar entre Participacion y Pertenencia, pues aunque Participacion es la clase que más relaciones tiene, lo más normal es que las operaciones que más tiempo esté realizando sea con la clase Pertenencia para gestionar su plantilla, vender jugadores etc.

- En nuestra aplicación, se produce una herencia entre LigaReal y LigaFicticia. En su momento ya explicamos cómo se iba a resolver este mapeado (apartado 3.2), por lo que para realizar este mapeado de herencia es necesario añadir la anotación @Inheritance con la estrategia SINGLE_TABLE.

Un ejemplo de todo este proceso lo podemos ver en la clase Oferta, mostrada en la ilustración 41:

```
@Entity
public class Oferta implements Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    int id;

    @Positive
    float valor;

    @NotNull
    EstadoOferta estado;

    @Valid @ManyToOne
    Participacion ofertaLiga;

    @Valid @ManyToOne
    Participacion pPropietario;

    @Valid @ManyToOne
    Pertenencia ofertaJugador;
```

Ilustración 43: Ejemplo de mapeado en la clase Oferta

En el ejemplo anterior se asigna como clave primaria el atributo id, con la estrategia IDENTITY, que indica el uso de columnas autoincrementadas. También se puede apreciar varias anotaciones de Bean Validation y las relaciones @ManyToOne que contiene esta clase.

JPA utiliza un lenguaje de consulta específico, llamado JPQL[19], que se utiliza para realizar consultas orientadas a objetos y está diseñado para ser independiente de la base de datos subyacente.

[19] Java Persistence Query Language

Este lenguaje proporciona una sintaxis similar a SQL pero se centra en las entidades y sus atributos en lugar de las tablas y sus columnas. Este será el lenguaje de consulta en caso de que necesitemos realizar algún tipo de consulta a la base de datos.

Otro punto importante es la gestión de transacciones en Spring, dónde Spring Boot ya proporciona un gestor de transacciones por nosotros. Normalmente una operación se realiza en el contexto de una transacción, aunque se pueden definir una serie de atributos que definen la política de transacción que debe ser aplicada.

Estos atributos pueden ser varios, pero nosotros solo nos vamos a centrar en los atributos *Propagation* y *Read-Only*, y los demás atributos, como el nivel de aislamiento, los vamos a dejar por defecto.

Para el atributo *Propagation*, existen bastantes políticas, aunque nos hemos centrado en usar sólo las más interesantes: *Required*, que indica que la operación se debe ejecutar en una transacción existente, y en el caso de que no existe, se crea una nueva, y *Supports*, que indica que la operación no requiere de una transacción, pero se puede ejecutar en una existente.

En resumen, se utilizará un método de propagación *Supports* para consultas o búsqueda de entidades en la base de datos, y *Required* para las demás operaciones, como guardar o actualizar en la base de datos. Además, si una operación no realiza modificaciones en la base de datos, la transacción será declarada con el atributo *Read-Only*.

Un ejemplo de consulta JPQL y de todo el proceso de gestión de transacciones la podemos ver en la ilustración 42 con la clase RepositorioPartido (o partidoDAO):

```java
@Repository
@Transactional(propagation = Propagation.REQUIRED)
public class RepositorioPartido {

    @PersistenceContext
    EntityManager em;

    @Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
    public Partido buscar(int id) {
        return em.find(Partido.class, id);
    }

    @Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
    public List<Partido> buscarPartidosJornada(int numJornada, Enums.Pais pais) {
        try {
            return em.createQuery("select p from Partido p where p.jornada.numJornada = :numJornada and p.jornada.liga.pais = :pais",
                    .setParameter("numJornada", numJornada)
                    .setParameter("pais", pais)
                    .getResultList();
        } catch (Exception e) {
            return null;
        }
    }

    public void actualizar(Partido p) {
        em.merge(p);
    }
}
```

Ilustración 44: Ejemplo de repositorio en la clase RepositorioPartido

En la ilustración anterior podemos ver como utilizamos la anotación @Repository para indicar a Spring que la siguiente clase es un repositorio, y utilizamos el atributo *Propagation_Required* en toda la clase.

De forma específica, para consultas y búsqueda de entidades, se indica *Propagation_Supports* y el atributo *Read-Only* activado como hemos explicado antes. En este ejemplo también podemos ver un ejemplo de uso de JPQL, donde es importante incluir la consulta con try-catch ya que si la consulta no encuentra ninguna entidad lanzará una excepción. Capturamos la excepción y devolvemos null en ese caso, para no interrumpir toda la operación que se estaba realizando.

También se usará la anotación @Transactional en aquellas operaciones en el servicio donde se necesite interactuar con los objetos relacionados de algunas clases (por ejemplo, para finalizar una jornada (ilustración 23) se necesita en la operación las pertenencias asociadas a una participación).

Antes de explicar cómo se ha realizado la conexión con la base de datos, es importante explicar cómo se han implementado los DTOs, pues al utilizar Java 17.0, esta implementación es relativamente sencilla si aplicamos el uso de records. Los records son una nueva característica introducida en Java 14 como una forma concisa de definir clases inmutables.

Un ejemplo de record lo podemos ver en la clase DTOPertenencia:

```
public record DTOPertenencia(
        int id,
        DTOJugador jugador,
        DTOParticipacion participacion,
        float clausura) {

    public DTOPertenencia(Pertenencia p){
        this(p.getId(),new DTOJugador(p.getJugador()),new DTOParticipacion(p.getParticipacion()),p.getClausura());
    }

    public Pertenencia aPertenencia(){
        return new Pertenencia(id, jugador.aJugador(), participacion.aParticipacion(), clausura);
    }

}
```

Ilustración 45: Clase DTOPertenencia implementada con record

Estas clases sólo se utilizarán para enviar datos al frontend y para transformar en clases los datos enviados desde el frontend al backend, por lo que son realmente útiles y es uno de los motivos por el cual se ha utilizado una versión de java más actualizada.

4.2.4. Conexión con la base de datos

Para finalizar la implementación en la capa de persistencia, solo nos queda realizar la conexión con la base de datos. Nosotros hemos utilizado Docker para implementar la base de datos.

Docker es una plataforma de contenedores de software que facilita la creación, implementación y ejecución de aplicaciones en entornos aislados llamados contenedores. Estos contenedores incluyen todo lo necesario para ejecutar una aplicación, incluyendo el código, las bibliotecas, dependencias y configuraciones.

El contenedor de MySQL creado con Docker se ejecutará de forma aislada en su propio entorno, con su propia instancia de MySQL y las configuraciones que se hayan proporcionado. Toda la documentación necesaria sobre el uso de Docker y opciones de configuración específicas para la imagen de MySQL la podemos encontrar en [23].

Resumiendo un poco este apartado, lo único que tenemos que ejecutar una vez descargado Docker es ejecutar el siguiente comando para instalar la imagen:

```
docker run -d -p 33060:3306 --name mysql-db -e MYSQL_ROOT_PASSWORD=secret mysql
```

Ilustración 46: Comando para instalar la imagen MySQL en Docker

El anterior comando descarga e instala la imagen oficial de MySQL, define secret como clave de root y lo asocia al puerto 33060. Después, solo debemos crear la nueva base de datos (llamada fantasyleague) junto a un nuevo usuario otorgándole los permisos necesarios para trabajar con la base de datos. Más información sobre la configuración correcta de la base de datos la podemos ver en el Apéndice I.

Aunque ya profundizaremos más en el apartado 5, de forma similar se ha creado otra base de datos para el testing, llamada fantasyleague_test.

En este caso, se configurará una base de datos H2 en memoria para ser utilizada en el entorno de testing de la aplicación. Esto nos permitirá realizar consultas a la base de datos de forma más rápida que si lo hiciésemos de forma normal, agilizando todo el proceso de testing.

Para configurar la conexión a la base de datos, debemos configurar el fichero application.yml (o application.properties) de la siguiente forma:

```
spring.sql.init.mode: always

spring.datasource.url: jdbc:mysql://localhost:33060/fantasyleague
spring.datasource.username: fantasyleague
spring.datasource.password: secret
spring.datasource.data: classpath:data.sql

spring.jpa.properties.javax.persistence.schema-generation.database.action: none

spring.data.jpa.repositories.bootstrap-mode: default

spring.jpa.defer-datasource-initialization: true
```

Ilustración 47: Fichero de configuración application.yml

De forma similar, debemos configurar dicha conexión a la base de datos de testing en el fichero application_test.yml:

```
spring.datasource.url: jdbc:h2:mem:fantasyleague_test;MODE=MYSQL;DATABASE_TO_LOWER=TRUE;DB_CLOSE_DELAY=-1

spring.jpa.properties.javax.persistence.schema-generation.database.action: drop-and-create
```

Ilustración 48: Fichero de configuración application_test.yml

Finalmente, es importante destacar también el uso de un fichero, llamado data.sql, donde se encuentran todos los datos que se insertarán automáticamente en la base de datos al iniciar la aplicación o reiniciar el entorno de ejecución, de forma que no haya que volver a insertarlos manualmente.

4.2.5. Implementación de la capa de servicios

La capa de servicios incluye cualquier servicio detallado en el diagrama de clases, como por ejemplo el API-REST o la seguridad de la aplicación. Como todo el API-REST ya se ha detallado en profundidad en el apartado 3.5.1, nos vamos a centrar sobre todo en la seguridad de la aplicación.

Toda la seguridad se ha implementado mediante Spring-Security, que es un framework de seguridad que proporciona funcionalidades para autenticación, autorización y protección contra ataques en aplicaciones web y servicios RESTful.

Nuestra aplicación web está implementada con JavaScript y se ejecuta en el navegador, lo que supone un problema ya que los navegadores impiden que este tipo de aplicaciones accedan a un servidor distinto del que se descargó la página web, para evitar ataques y problemas de seguridad (same-origin policy).

La política Cross-Origin Resource Sharing (CORS) relaja la anterior política de seguridad, permitiendo que una aplicación pura JavaScript acceda a un servicio RESTful de confianza.

En nuestro caso, vamos a activar CORS utilizando la anotación @CrossOrigin en nuestro controlador, y creando una nueva clase que active esta política de seguridad, como podemos ver en la ilustración 47:

```
@Configuration
@EnableWebMvc
public class CORSConfig implements WebMvcConfigurer{

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/fantasyleague/**").allowedOrigins("*").allowedMethods("*").allowedHeaders("*");
    }
}
```

Ilustración 49: Clase de configuración para CORS

Para aumentar la seguridad de la aplicación en el almacenamiento de los usuarios que se registran en ella, se ha utilizado un codificador (en específico, el BCryptPasswordEncoder) que permite encriptar la clave de todos los usuarios de la aplicación.

Esta clave se almacenará codificada en la base de datos pero se mostrará sin codificar al usuario, de forma que nunca se pueda visualizar la contraseña encriptada.

Para que un usuario pueda iniciar sesión en la aplicación, primero se debe comprobar que el usuario existe, y después que la contraseña que se envía desde el frontend es igual a la contraseña codificada almacenada en la base de datos (evidentemente se debe codificar la contraseña enviada desde el frontend también para hacer la comprobación).

La autorización de los usuarios se ha realizado mediante el uso de tokens JWT[20], que es un estándar para la definición de tokens flexibles y muy seguros.

Estos tokens se pasan desde el campo “Authorization” de la cabecera HTTP, indicando el tipo “bearer”. El esquema de comunicación entre el usuario y la aplicación la podemos ver en la siguiente imagen:

[20] Java Web Tokens

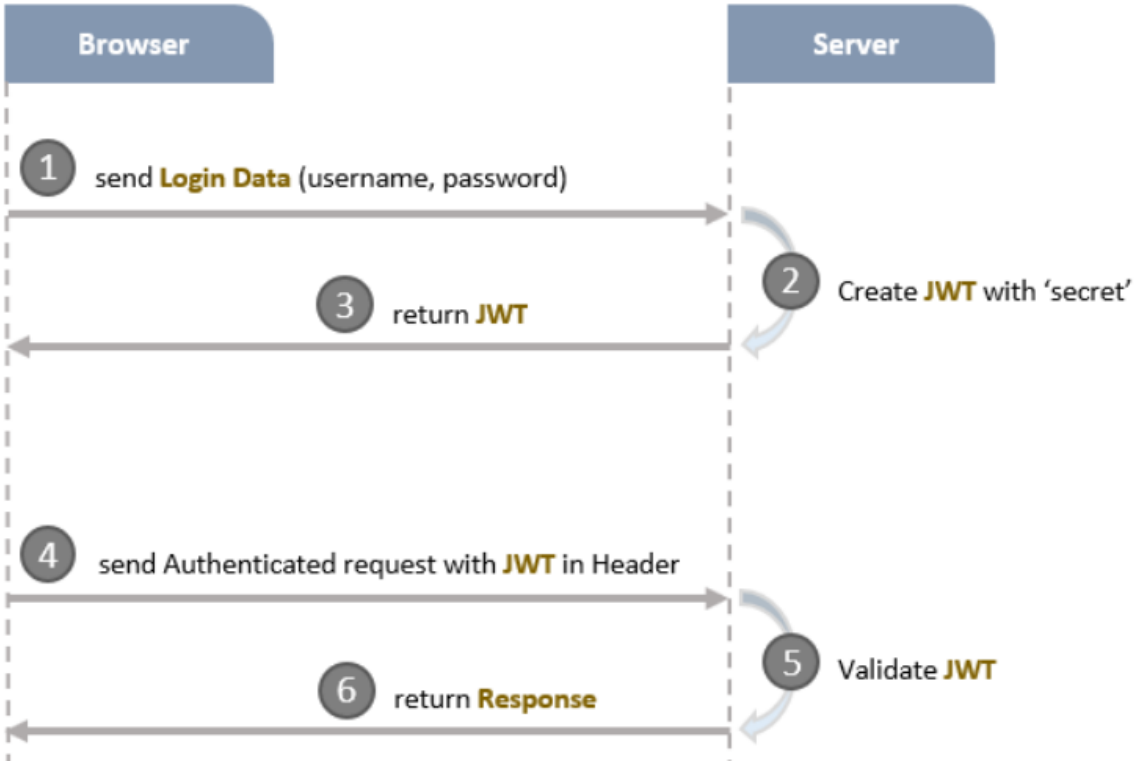

Ilustración 50: Esquema de comunicación entre fontend y backend utilizando tokens JWT [24]

Una vez creado el token, éste se envía al frontend que se encarga de almacenar el token mientras el usuario mantiene la sesión iniciada en la aplicación.

Cada vez que el usuario manda una petición HTTP al API REST del backend, se aplicará un filtro JWT que comprobará que efectivamente la petición contiene el campo "Authorization" con el token existente, que el token enviado pertenece al usuario que realiza la petición, y que el token no ha expirado.

Por último, solo nos queda configurar la seguridad dentro de Spring. Esto lo podemos hacer definiendo un filtro de seguridad dentro de la clase ServicioSeguridadFantasyLeague de la siguiente forma:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    http.csrf().disable().cors();

    http.authorizeRequests().antMatchers(HttpMethod.POST, "/fantasyleague/login").permitAll();
    http.authorizeRequests().antMatchers(HttpMethod.POST, "/fantasyleague/usuarios").permitAll();
    http.authorizeRequests().antMatchers("/fantasyleague/admin/**").hasRole("ADMIN");
    http.authorizeRequests().antMatchers("/fantasyleague/**").hasRole("USUARIO");

    http.addFilterBefore(new JWTFilter(userDetailsService), UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

Ilustración 51: Filtro de seguridad del backend

Como se puede ver, se permiten las solicitudes POST tanto al login como al registro de usuarios sin realizar autenticación.

Para todas aquellas URLs que comiencen con /admin, requiere que las solicitudes estén autenticadas con el rol ADMIN, y las demás URLs requiere que estén autenticadas con el rol USUARIO.

Debido a que muchas URLs han sido acortadas por su excesiva longitud, no vamos a configurar la autorización de acceso a una URL específica en el backend, (por ejemplo, que en un acceso a una URL de tipo “/usuario/{nombreUsuario}”, el valor de {nombreUsuario} sea igual a la del nombre de usuario del usuario autentificado) si no que esta comprobación se realizará a nivel de frontend, como veremos en el apartado 4.2.5

4.2.6. Implementación de la capa de presentación

Como ya hemos explicado en el apartado 4.1, se ha utilizado la herramienta Vite para el desarrollo de la aplicación Vue. El uso de esta herramienta frente a un uso tradicional de Vue se debe a la mejora en la velocidad de desarrollo que ésta ofrece, además de una recarga rápida de módulos y una configuración simplificada.

Una comparativa de velocidad entre el uso de Vue CLI y Vite la podemos ver en la ilustración 50:

	vue-cli	vite	🏆
Scaffolding	~28s	**~5.95s**	⚡
Dev Server	~1.64s	**~0.37s**	⚡
Update	~0.349s	**<0.3s**	⚡
Build	~11.27s	**~2.88s**	⚡

Ilustración 52: Comparativa de velocidades entre Vite y Vue CLI (Fuente: https://bit.ly/3oItCHL)

Este ha sido el principal motivo del uso de Vite, con el objetivo de agilizar todo el proceso de desarrollo del frontend.

Como Vue es una aplicación de tipo SPA, nuestra aplicación contendrá un solo fichero html junto a los ficheros JavaScript que se encargarán de controlar toda la web.

Por lo tanto, en nuestra aplicación solo tendremos un fichero js (llamado main.js) que se encargue de configurar la aplicación Vue y de descargar todas las dependencias necesarias, que estará enlazado al fichero html, y un componente principal, llamado App.Vue, en el que se cargarán todos los demás componentes.

Un ejemplo del componente App.Vue lo podemos encontrar en la ilustración 51:

```
<script setup>
import FLFooter from "./components/commons/FLFooter.vue";
</script>

<template>
  <router-view></router-view>
  <FLFooter/>
</template>
```

Ilustración 53: Componente App.Vue

Como se puede ver, este componente es quizás el más simple de todos, pues simplemente carga un footer estático y utiliza el componente router-view de la biblioteca router-vue para cargar toda la navegación establecida en la aplicación.

También se podría añadir antes del componente router-view un header, pero como ya explicaremos más adelante, este header no es estático como el footer, por lo que es necesario que esté implementado en todas las páginas que necesitan el header al ser diferente dependiendo del componente en el que se encuentre cargado.

Toda la interfaz web, como ya se ha mencionado anteriormente, se ha implementado con Boostrap-Vue-3[21]. Aunque más adelante, en el Apéndice II ofrezcamos un manual de usuario, es recalcable mencionar el uso de Modals en el diseño de la interfaz.

Un componente Modal son mensajes de diálogos simplificados que admite una serie de casos de uso, que puede comprender desde notificaciones hasta contenido personalizado.

Un ejemplo del uso de este componente lo podemos ver en el wireframe de la ilustración 33 o 36. Con el uso de este componente ofrecemos al usuario una experiencia cómoda y simple mediante ventanas de mensajes con las funcionalidades necesarias sin tener que implementar nuevos componentes.

[21] https://bootstrap-vue.org/

Para la navegación entre páginas se ha utilizado el router de Vue, también conocido como router-vue[22].

```
const routes = [
    { path: '/', component: login },
    { path: '/registro', component: registro },
    { path: '/admin',name:'admin', component: administrarLiga},
    { path: '/admin/:idLiga',name:'partidos', component: partidoList},
    { path: '/error',name:'forbidden', component: forbidden},
    { path: '/:nombre',name:'home', component: userHome},
    { path: '/:nombre/:idParticipacion',name:'participacion', component: participacionHome}
]

3 usages
export const router = createRouter( options: {
    history: createWebHistory(),
    routes
})
```

Ilustración 54: Implementación de router-vue en la aplicación

Este router también se encarga de comprobar que un usuario con rol USUARIO no acceda a componentes que necesitan el rol ADMIN, o que usuarios cuya sesión está activa puedan acceder al “home” de otros usuarios mediante la función BeforeEach.

También debe de encargarse que un usuario sólo pueda acceder a las participaciones que se relacionan con ese usuario, y no a la de las demás. Si alguna política de seguridad se incumple, este router redirigirá a una página llamada forbidden.Vue que indica al usuario que está realizando una acción prohibida.

Para realizar estas comprobaciones de seguridad, router-vue se comunica con el fichero ControllerRESTFetch.js, que se encarga de realizar las peticiones HTTP al API REST de la parte del servidor, y de la clase Cookies.js, que se encarga de almacenar la sesión activa una vez un usuario realiza el login de nuestra aplicación.

[22] https://router.vuejs.org/

```
async login() {
  try {
    let user = {
      nombre: '',
      rol: '',
      token: ''
    }
    await ctrllREST.login(this.usuario)
        .then(data => {
          user.nombre = data.nombre
          user.rol = data.rol
          user.token = data.token
        });

    await Cookies.setUserLogged(user);
    if (user.rol === "USUARIO") {
      this.$router.push({name: 'home', params: {nombre: user.nombre}});
    }else{
      this.$router.push({path:'/admin'});
    }
  } catch (err) {
    this.errorMsgs.validacion = "Usuario o contraseña erróneos"
  }
}
```

Ilustración 55: Función login que almacena el usuario en una Cookie

Dependiendo de si la clase Cookie.js contiene un usuario almacenado o no (o lo que es lo mismo, si un usuario ha iniciado sesión o no en nuestra aplicación), el header permitirá al usuario cerrar sesión en la aplicación. Por este motivo no se utiliza un header estático, sino que genera esta opción de forma dinámica dependiendo de si un usuario inicia sesión o se registra en la aplicación.

También es importante almacenar el token enviado desde el backend, ya que será necesario para las demás peticiones que vayamos a realizar al servidor.

5. Pruebas

Como el plan de pruebas diseñado para la aplicación está basado en pruebas de usuario en el cliente y en el uso de TDD para la parte del servidor, nos vamos a centrar en comentar los resultados en ambas aplicaciones.

Comenzando por la parte del cliente, se ha utilizado un enfoque de caja negra partiendo de los casos de uso detallados.

Durante el periodo de desarrollo, se ha comprobado que el comportamiento de los casos de uso era el esperado tanto para las situaciones correctas (la operación se realiza como se espera), como para las acciones alternativas (se detecta algún error en la operación).

El número de pruebas realizadas puede variar dependiendo del caso de uso, sabiendo que mínimo habrá el doble de pruebas que casos de uso detallados (prueba correcta + incorrecta), aunque realmente sabemos que son muchas más, pues las pruebas incorrectas se prueban por cada error introducido.

Con todo esto, podemos hacer una estimación, pues por ejemplo hay casos de uso como el de finalizar una jornada, donde solo contiene la situación correcta y una situación alternativa de error, y casos de uso como el de validar una plantilla donde se tienen 5 situaciones alternativas al comprobarse el número de jugadores, sus posiciones etc.

En total, hemos considerado que cada caso de uso contiene una media de 3 pruebas, que serían un total de 30 pruebas realizadas (10 casos de uso * 3) en la parte del cliente.

En la parte del servidor, estas pruebas se han realizado mediante el framework jUnit, como ya explicamos en el apartado 3.6, más concretamente jUnit 5[25].

Para integrar jUnit a nuestro proyecto solo debemos añadir el siguiente código al fichero pom.xml:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>
```

Ilustración 56: Integración de jUnit al proyecto

jUnit proporciona una serie de anotaciones y bibliotecas que simplifican todo el proceso de testing, como veremos en los ejemplos posteriores.

El número de pruebas desarrolladas totales ha sido de 31, de las cuales 14 han sido pruebas de unidad y 17 pruebas de integración tanto del controlador REST como de la capa de servicios en general mediante TDD.

Para las pruebas de unidad, debido a que las clases no contienen demasiadas funciones independientes se ha probado que todo el Bean Validation funciona correctamente en una entidad, además de probar cada función que contenga la clase en el caso de que tuviese alguna. Un ejemplo de las pruebas de unidad para la clase Participacion la podemos encontrar en la ilustración 57:

```
public class ParticipacionTest {

    public ParticipacionTest() {
    }

    @Test
    void validacionParticipacionTest(){
        LigaFicticia ligaF = new LigaFicticia(Enums.Pais.FRANCIA, "PruebaLiga");
        Usuario u = new Usuario("Jose","jcb00034","jcb00034@red.ujaen.es",Enums.Rol.USUARIO);
        Participacion p = new Participacion(u, ligaF);

        Validator validator = Validation.buildDefaultValidatorFactory().getValidator();
        Set<ConstraintViolation<Participacion>> violations = validator.validate(p);

        Assertions.assertThat(violations).isEmpty();
    }
}
```

Ilustración 57: Ejemplo de prueba de unidad para la clase Participacion

Para las pruebas de integración, se ha realizado todo el proceso TDD descrito en el apartado 3.6. Estas pruebas se han dividido en dos paquetes: un paquete que

contiene todas las pruebas para los servicios, y otro que contiene todas las pruebas para el controlador REST.

Aunque lo ideal es que cada función o método contenga una prueba independiente, en muchos casos para agilizar el proceso se han realizado pruebas para diferentes funciones en una sola unidad de test. Esto se debe a que muchas operaciones del servicio están relacionadas entre sí.

Por ejemplo, para vender un jugador, primero lo tienes que fichar, por lo que para evitar código repetitivo este tipo de operaciones se han realizado de forma conjunta, en el que primeramente se prueba todo lo relacionado con la función FicharJugador() y después se realiza el mismo proceso para VenderJugador(), pero todo en la misma unidad de test.

Para las pruebas del API REST, como ya se mencionó en el diseño de las pruebas se ha utilizado adicionalmente la biblioteca TestRestTemplate y LocalServerPort. Un ejemplo de configuración de todas estas bibliotecas la podemos encontrar en la ilustración 58:

```
@LocalServerPort
int localPort;

@Autowired
MappingJackson2HttpMessageConverter springBootJacksonConverter;

TestRestTemplate restTemplate;

/**
 * Crear un TestRestTemplate para las pruebas
 */
@PostConstruct
void crearRestTemplateBuilder() {
    RestTemplateBuilder restTemplateBuilder = new RestTemplateBuilder()
            .rootUri("http://localhost:" + localPort + "/fantasyleague")
            .additionalMessageConverters(List.of(springBootJacksonConverter));

    restTemplate = new TestRestTemplate(restTemplateBuilder);
}
```

Ilustración 58: Ejemplo de configuración para las pruebas del API REST

Esta biblioteca nos proporciona diferentes formas de realizar el testing mediante peticiones HTTP, sin embargo, como en la mayoría de nuestras peticiones nos hace falta el campo “Authorization” para proporcionar el token JWT, realizaremos las peticiones de la siguiente forma:

```
DTOUsuario u = new DTOUsuario("admin", "secreto", "admin@admin.com", Enums.Rol.ADMINISTRADOR, "");

ResponseEntity<DTOUsuario> response = restTemplate
        .postForEntity(
                "/login",
                u,
                DTOUsuario.class
        );
u = response.getBody();
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);
headers.set("Authorization", u.token());
HttpEntity<?> request = new HttpEntity<>(null, headers);

ResponseEntity<DTOPartido> r1 = restTemplate
        .exchange("/partido/1",
                HttpMethod.GET,
                request,
                DTOPartido.class);
```

Ilustración 59: Ejemplo de testing en el API REST

El ejemplo mostrado en la ilustración 59 pertenece al test realizado para editar un partido, pero todos contienen la misma estructura inicial: será necesario iniciar sesión con un usuario de la aplicación que nos permita obtener un token JWT e insertar este token en el campo “Authorization” dentro del header. El campo request contendrá un objeto, en el caso de que la petición lo requiera, y el header de la petición.

Los resultados de las pruebas han sido satisfactorias, y aunque no ha habido demasiados errores gracias al uso de TDD, si que ha permitido cambiar la forma de pensamiento inicial sobre una operación respecto a cómo se ha implementado finalmente.

Un ejemplo de esto lo podemos ver en la operación VenderJugadorMercado(), donde inicialmente, o tal y como expresa el requisito RF11, si un usuario vende un jugador al mercado este obtendrá el 80% de las ganancias.

Sin embargo, este requisito no comprende lo que pasa con los demás usuarios, ¿y si algún usuario ha realizado ya una oferta por él? Podría haber alguna laguna

donde el usuario venda el jugador al mercado y además acepte la oferta que ha realizado otro usuario por él, obteniendo el doble de ganancias.

La forma de solucionarlo es similar al caso de uso de negociar oferta (que podemos ver en el diagrama de secuencia de la ilustración 22), donde se debe buscar todas las ofertas para ese jugador y rechazarlas manualmente. Algo similar sucede al pagar una cláusula, donde también se deben rechazar todas las ofertas, pero en el caso de vender un jugador al mercado esta operación no era tan trivial como las otras dos.

6. Conclusiones

Como conclusiones del proyecto, analizaremos en primer lugar si se han cumplido los objetivos inicialmente planteados (apartado 1.2).

El primer objetivo trata sobre realizar un estudio de necesidades y soluciones para el contexto seleccionado. Este objetivo se ha cumplido debido a todo el análisis preliminar realizado (apartado 2) no solo de tecnologías que finalmente se han utilizado en el proyecto, si no también de aplicaciones existentes similares al contexto del proyecto junto a algunas tecnologías, como lo son las bases de datos NoSQL o frameworks como React, donde el sistema se podría haber implementado igualmente y donde tratamos sus ventajas e inconvenientes para la solución final.

El segundo objetivo trata de diseñar un sistema informático orientado al desarrollo web. Este objetivo también se ha cumplido, pues se ha escogido una metodología web a utilizar (apartado 1.3) y se ha realizado todo el proceso de diseño (apartado 3), con la ayuda de diagramas tanto de casos de uso como de clases, de secuencia o incluso de arquitectura del sistema (apartado 4.1) para ayudar a explicar el funcionamiento de todo el sistema en su conjunto.

Es importante mencionar también que el sistema informático propuesto está compuesto por dos aplicaciones independientes, que interactúan mediante un API REST.

Por último, el tercer objetivo trata sobre seleccionar un entorno de desarrollo que permita implementar un prototipo de aplicación web que satisfaga las necesidades propuestas. Este objetivo se ha completado, como hemos podido ver durante todo el apartado 4 y 5, con el uso de frameworks como Vue o Spring, y que podemos ver en el código del proyecto proporcionado junto a esta memoria.

Rememorando un poco el enfoque utilizado en el proyecto, creo que el uso de dos aplicaciones independientes permite una mayor flexibilidad que si se hubiese implementado todo en un mismo servidor.

Esto puede suponer algunas ventajas en el futuro, como que por ejemplo si se cambia el framework en el cliente, no hace falta tocar el servidor y viceversa, o que en el futuro se podrían añadir nuevos clientes, como por ejemplo aplicaciones nativas o incluso plantearse la posibilidad de monetizar los datos de la aplicación dando acceso a terceros desarrolladores que permitan que sus aplicaciones interactúen con el sistema.

Personalmente, los aspectos que más he disfrutado del proyecto ha sido la implementación en el backend, pues creo que es la parte donde más se puede observar la evolución y mejora en términos de programación, ya que aunque a lo largo del grado se den asignaturas para implementar interfaces, desde el principio se tratan asignaturas que enseñen a programar y diseñar algoritmos, que es lo que principalmente se hace en el backend.

También ha sido muy interesante implementar la lógica de negocio de una aplicación completa, no solo en el backend, si no también en el frontend gestionando las respuestas del backend y realizando acciones diferentes según las respuestas del API REST.

Como la parte del frontend es la parte donde menos experiencia acumulaba, ha resultado interesante utilizar en profundidad un framework del estilo de Vue, pues a lo largo del grado no se ve un framework frontend con tanta profundidad como se ha tratado en este proyecto, lo cual ha resultado útil de cara al futuro profesional.

Algunas ideas de mejoras de futuro que podrían ser interesantes, pero que no se han podido abarcar debido al tiempo de desarrollo y el tiempo que se ha empleado en los demás requisitos, es el de crear un buzón de notificaciones, sobre todo para notificar al usuario cuando una oferta se ha rechazado o aceptado.

Esta idea se ha implementado parcialmente mostrando al usuario un cuadrado junto al número de ofertas que ha recibido por sus jugadores, pero se podría ampliar con notificaciones que avisen al jugador cada vez que cambia el estado de una oferta, o cada vez que ha finalizado la jornada, por ejemplo.

También se podría mejorar el proceso de fichar jugadores para formar tu equipo, ya que tal y como está especificado, no puedes validar una plantilla hasta que no haya al menos 2 jugadores en la liga, pero esto no te impide fichar los jugadores que quieras, por lo que podrías crear una liga, fichar a 11 jugadores que te apetezcan y después invitar a tus amigos, lo que sería injusto.

Ambas ideas son relativamente fáciles de implementar con la solución planteada, pues se podría impedir a los jugadores fichar jugadores hasta que no haya 2 o más jugadores en la liga (similar al proceso de validación), y el buzón de notificaciones se podría hacer implementando una clase que se relacione con la clase Usuario, de forma que cada vez que se crea una oferta o se finaliza una jornada, se manda un mensaje al buzón del usuario asociado.

Como conclusión final, creo que la realización de un proyecto de este tamaño ayuda al alumnado a ganar confianza de cara a su futuro profesional, pues mucha gente, a pesar de terminar el grado, tienen lo que se llama el “síndrome del impostor” y se creen que luego a la hora de la verdad no son capaces de hacer funcionar un proyecto completo.

La mejor forma de combatir esto es realizando un proyecto completo por uno mismo, como ha sido este caso, y así comprobar todo lo que se ha aprendido en el grado, que puesto en perspectiva no es precisamente poco.

7. Bibliografía

[1] *World Football Report 2022.* Nielsen. [Online] Disponible en: https://bit.ly/3oTrvRz Acceso: 13/06/2023

[2] A. Canepa Sáenz & C. Erika García. *Comparativas de los modelos de ciclo de vida.* [Online] Disponible en: http://www.repositorio.unacar.mx/jspui/bitstream/1030620191/199/1/acalan%2074-2.pdf Acceso: 25/03/2023

[3] Pressman, Roger S. *Ingeniería del software: un enfoque práctico.* Madrid: McGraw-Hill/Interamericana de España, 1998. [Online] Disponible en Recursos Electrónicos de la Universidad de Jaén

[4] Marca (2019). *Funcionamiento de LaLiga Fantasy.* LaLiga Fantasy. [Online] Disponible en: https://laligafantasy.zendesk.com/hc/en-us/sections/360001615873-Market Acceso: 19/05/2023

[5] *Fútbol Fantasy.*Wikipedia [Online] Disponible en: https://es.wikipedia.org/wiki/F%C3%BAtbol_fantasy Acceso: 19/05/2023

[6] *Guía de introducción a Vue.js.* VueJS. [Online] Disponible en: https://es.vuejs.org/v2/guide/ Acceso: 12/05/2023

[7] P. Pšenák & M. Tibensky. *The usage of Vue JS framework for web application creation.* [Online] Disponible en: https://bit.ly/3qjMRYo Acceso: 12/05/2023

[8] *React*. Wikipedia [Online] Disponible en: https://es.wikipedia.org/wiki/React Acceso: 12/05/2023

[9] A. Banks & E. Porcello. *Learning React: Functional web Development with React and Redux.* Sebastopol, California: O'Reilly, 2017. [Online] Disponible en Recursos Electrónicos de la Universidad de Jaén

[10] *Spring Framework Reference Documentation.* Spring Framework. [Online] Disponible en: Spring Framework Acceso: 12/05/2023

[11] W. Wheeler y J. White. *Spring in Practice.* Shelter Island, Manning, 2013. [Online] Disponible en Recursos Electrónicos de la Universidad de Jaén.

[12] *The Complete Guide to Testing on the Jakarta EE Platform.* Payara. [Online] Disponible en: https://bit.ly/3oMy87U Acceso: 13/05/2023

[13] J. Letkowski. *Doing database design with MySQL.* [Online] Disponible en: https://www.researchgate.net/publication/271910489_Doing_database_design_with_MySQL Acceso: 13/05/2023

[14] H. Phiri y D. Kunda. *Comparative Study of NoSQL and Relational Database.* [Online] Disponible en: https://bit.ly/3CeEiRt Acceso: 13/05/2023

[15] *Convenio colectivo de Telefónica publicado en BOE (12 de mayo de 2023).* Indeed. [Online] Disponible en: https://es.indeed.com/career/analista-programador/salaries Acceso: 18/05/2023

[16] S. Bennett, S. McRobb y R. Farmer. *Análisis y diseño orientado a objetos de sistemas: usando UML.* Madrid: McGraw-Hill, 2007, [Online] Disponible en Recursos Electrónicos de la Universidad de Jaén

[17] S. Ben. *Designing the user interface: strategies for effective human-computer interaction.* Boston: Pearson, 2018, [Online] Disponible en Recursos Electrónicos de la Universidad de Jaén

[18] *Documenting a Spring REST API Using OpenAPI 3.0.* Baeldung. [Online] Disponible en: https://www.baeldung.com/spring-rest-openapi-documentation Acceso: 03/06/2023

[19] K. Bent. *Test-Driven Development by Example.* Boston: Addison-Wesley, 2003, [Online] Disponible en Recursos Electrónicos de la Universidad de Jaén

[20] *Documentación oficial de Vite.* ViteJS. [Online] Disponible en: https://vitejs.dev/guide/ Acceso: 04/06/2023

[21] C. Walls. *Spring in Action, 6th Edition.* Shelter Island, New York: Manning Publications, 2022. [Online] Disponible en Recursos Electrónicos de la Universidad de Jaén

[22] M. Keith, J. Keith y M. Schincariol. *Pro JPA 2: Mastering the Java Persistence API.* Berkeley, CA: Appress, 2010 [Online] Disponible en Recursos Electrónicos de la Universidad de Jaén.

[23] *MySQL Docker Official Image.* Docker. [Online] Disponible en: https://hub.docker.com/_/mysql Acceso: 05/06/2023

[24] *Spring Boot + Vue.JS: Authentication with JWT & Spring Security.* Bezkoder. [Online] Disponible en: https://bit.ly/3CaxOTI Acceso: 06/06/2023

[25] *jUnit User Guide.* junit.org. [Online] Disponible en: https://junit.org/junit5/docs/current/user-guide/ Acceso: 21/06/2023

8. Apéndice I. Manual de instalación

8.1. Entorno de desarrollo

Tanto el frontend como el backend se han desarrollados en el IDE intelliJ instalado en la versión 2022.3.2, por lo que este manual proporcionará toda la información para la instalación en este IDE, aunque no se diferencia en mucho con la instalación en otros IDE como NetBeans.

Empezando por la aplicación cliente, lo primero que debemos tener instalado es Node.js[23]. También tenemos que tener instalados los plugins de *Vite Integrated* y *JavaScript and TypeScript* en intelliJ antes de empezar con la creación del proyecto. Después, creamos un nuevo proyecto con el asistente de intelliJ y seleccionamos la opción Vite para Vue, como podemos ver en la ilustración 60:

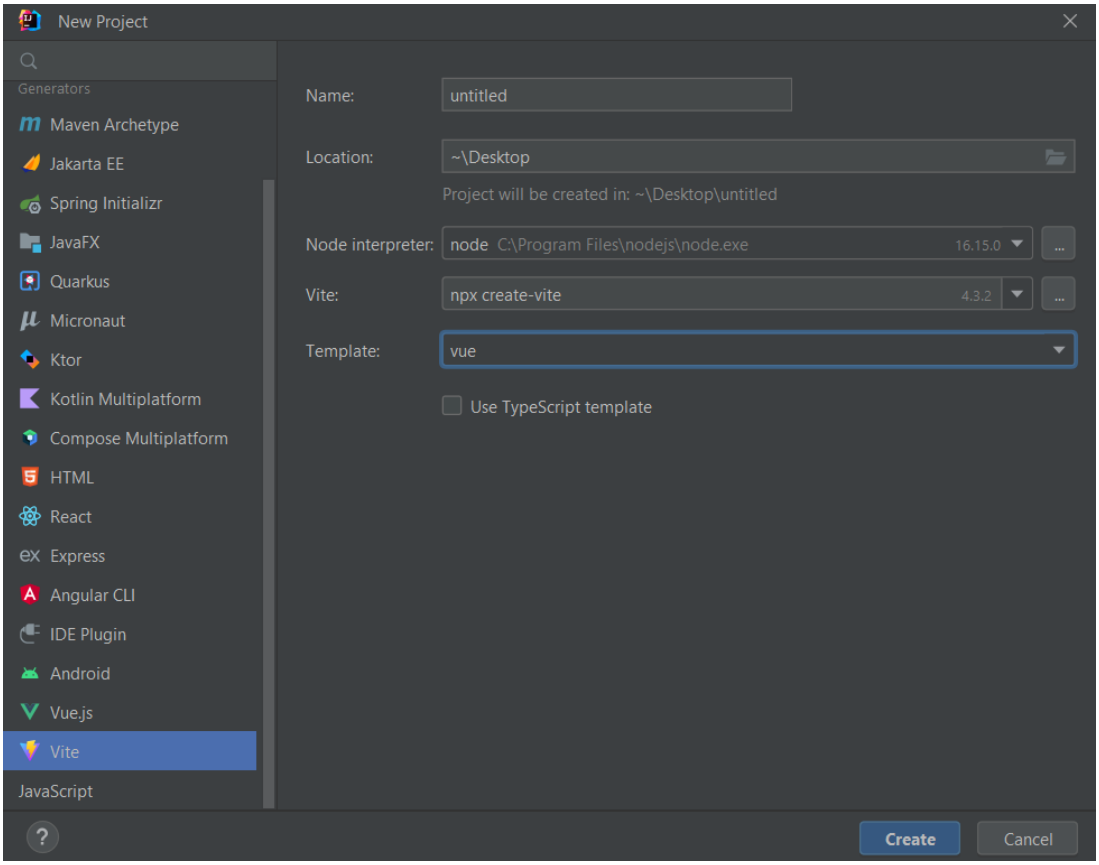

Ilustración 60: Crear proyecto con Vite y Vue

[23] https://nodejs.org/en#download

Para finalizar, solo nos queda ejecutar npm install en el proyecto. Esto lo podemos hacer dirigiéndonos a la carpeta del proyecto desde el terminal o seleccionando “npm install” desde intelliJ. Para finalizar esta instalación, debemos instalar también las dependencias del proyecto. Estas dependencias incluyen Bootstrap-vue-3, vue-router y js-cookie. Para instalarlas, nos vamos al directorio del proyecto desde el terminar y ejecutamos “npm install Bootstrap-vue-3”, “npm install vue-router” y “npm install js-cookie”.

Para la instalación en la parte del servidor, debemos crear un nuevo proyecto desde intelliJ seleccionando Spring Initilizr. En la ventana de configuración, seleccionamos Java con Maven y el JDK 17, como se puede ver en la ilustración 61:

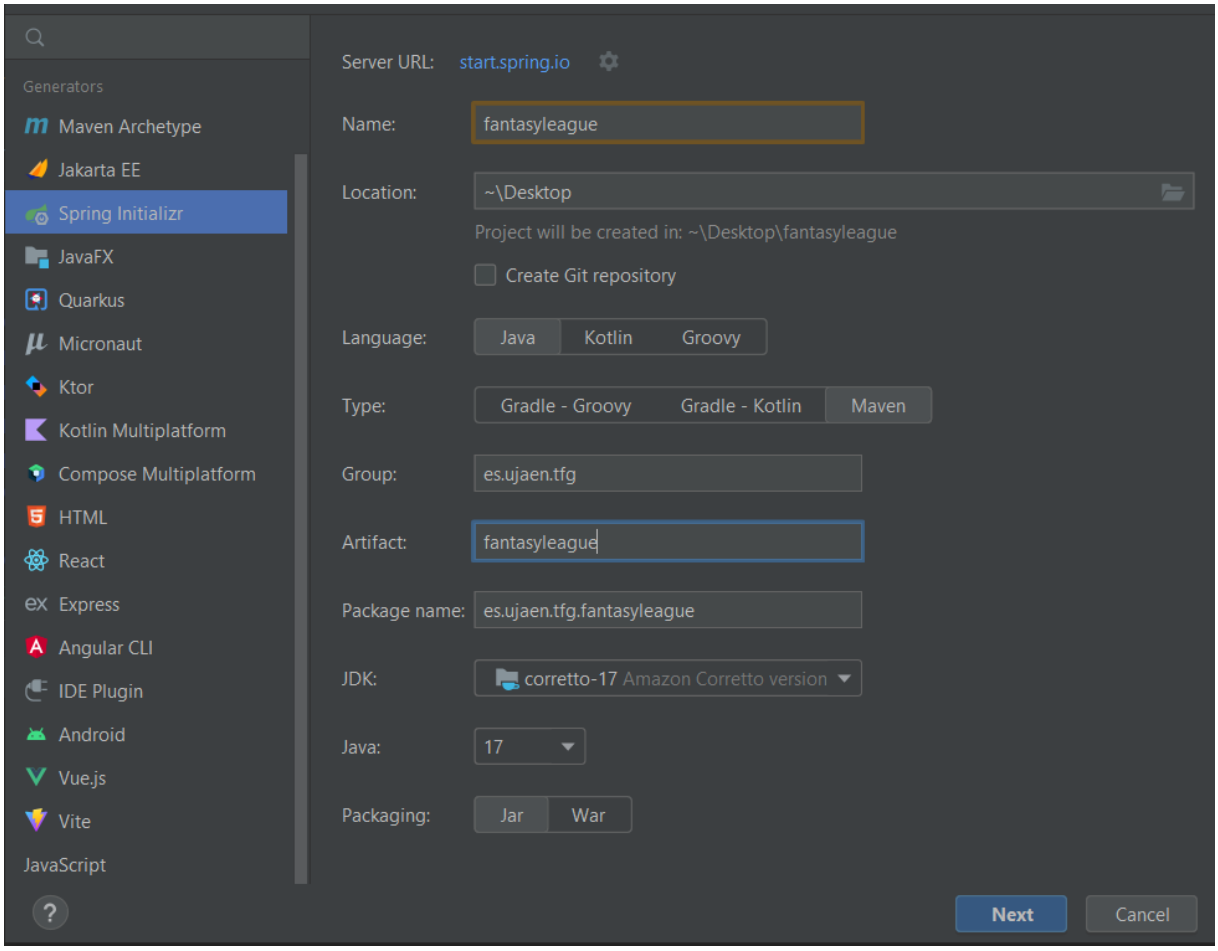

Ilustración 61: Crear proyecto con Spring

Por último, solo nos queda configurar la base de datos. Como ya explicamos en el apartado 4.2.3, ésta se realizó con un contenedor de Docker, por lo que lo primero que tenemos que hacer es crear la imagen correspondiente (ilustración 44). Para crear

la base de datos que almacene nuestras tablas, en el terminal del contenedor debemos ejecutar el siguiente comando:

docker exec mysql-db mysql -psecret -e "create database fantasyleague; use fantasyleague; create user 'fantasyleague' identified by 'secret'; grant all privileges on fantasyleague. to 'fantasyleague'@'%'"*

El anterior comando crea una base de datos junto a un usuario, y asigna los permisos necesarios a ese usuario para poder gestionar la base de datos a su antojo. Finalmente, solo nos queda configurar la base de datos para el testing, que se crea con un comando muy parecido al anterior:

docker exec mysql-db mysql -psecret -e "create database fantasyleague_test; use fantasyleague_test; grant all privileges on fantasyleague_test. to 'fantasyleague'@'%'"*

8.2. Entorno de producción

Para lanzar la aplicación a servidores dedicados, solo debemos irnos al directorio raíz de la aplicación servidor (aquella carpeta que contiene el fichero pom.xml) y poner *mvn package* en la línea de comandos. En el caso de que estemos realizando el proyecto en un IDE, como por ejemplo NetBeans, basta con realizar un *clean and build* del proyecto y se nos cargará el fichero .jar en la carpeta target.

Como para que la consola de comandos te reconozca la orden *mvn* debes tener configurada la carpeta de Maven como una variable global dentro del sistema operativo, vamos a optar por la segunda opción. La salida debería de descargar las dependencias necesarias, compilar el proyecto y crear un nuevo archivo .jar:

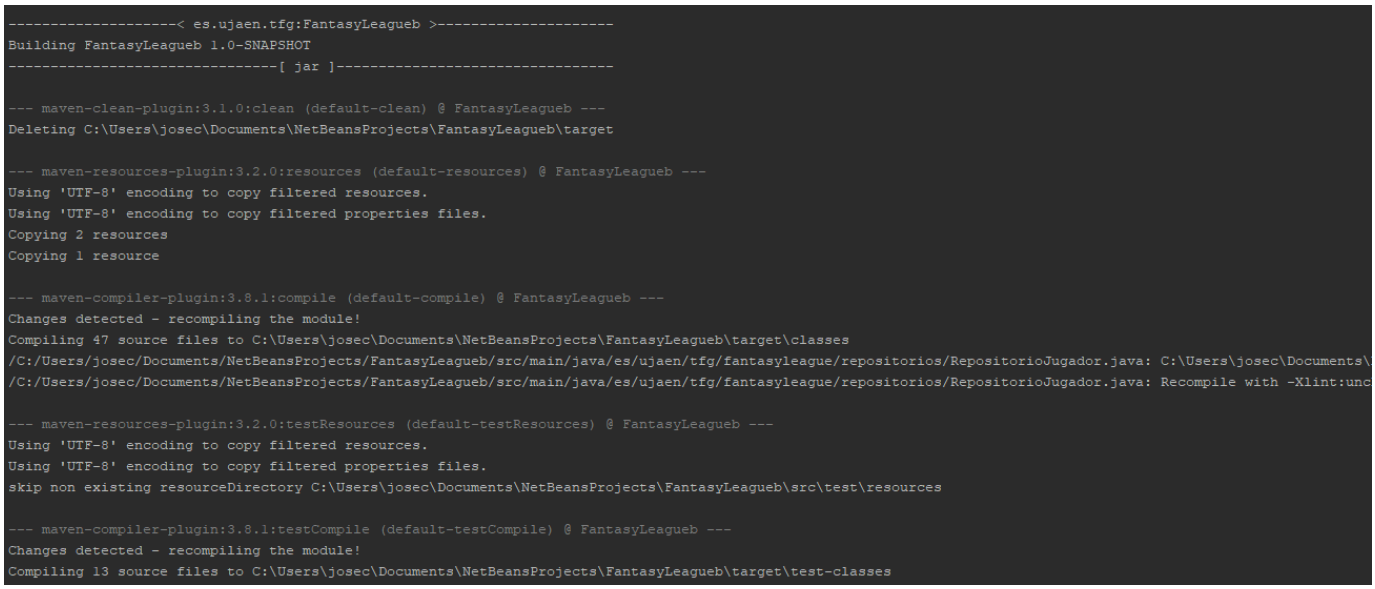

Ilustración 62: Crear fichero .jar

Es importante que antes de realizar esto tengamos configurado el acceso a la base de datos correctamente en nuestro fichero application.yml o application.properties tal y como hicimos en la ilustración 45, utilizando las credenciales necesarias e indicando que no se genere una nueva base de datos cada vez que lancemos la aplicación.

Una vez tenemos el archivo, solo nos queda lanzar la aplicación. Esto lo podemos hacer ejecutando el comando *java -jar fichero.jar* desde la carpeta target:

Ilustración 63: Lanzar la aplicación servidor

Debemos tener cuidado de que nuestra versión de java sea compatible con la versión de java del proyecto (en este caso 17.0), o de otra forma nos lanzará una excepción por incompatibilidad.

En la parte del cliente, lo ideal sería configurar un servidor web que aloje nuestra aplicación. Este servidor puede ser Apache, Nginx o el propio servidor incorporado de Node.js.

Sin embargo, y aunque mencionaremos también cómo se podría lanzar Vue a un servidor dedicado, nosotros hemos utilizado la opción *preview* que nos permite realizar una prueba local con el objetivo de verificar si la compilación de producción se ve correctamente en el entorno local.

Para ello solo debemos ejecutar dos comandos: *npm run build*, que prepara los ficheros necesarios de nuestra aplicación en la carpeta 'dist':

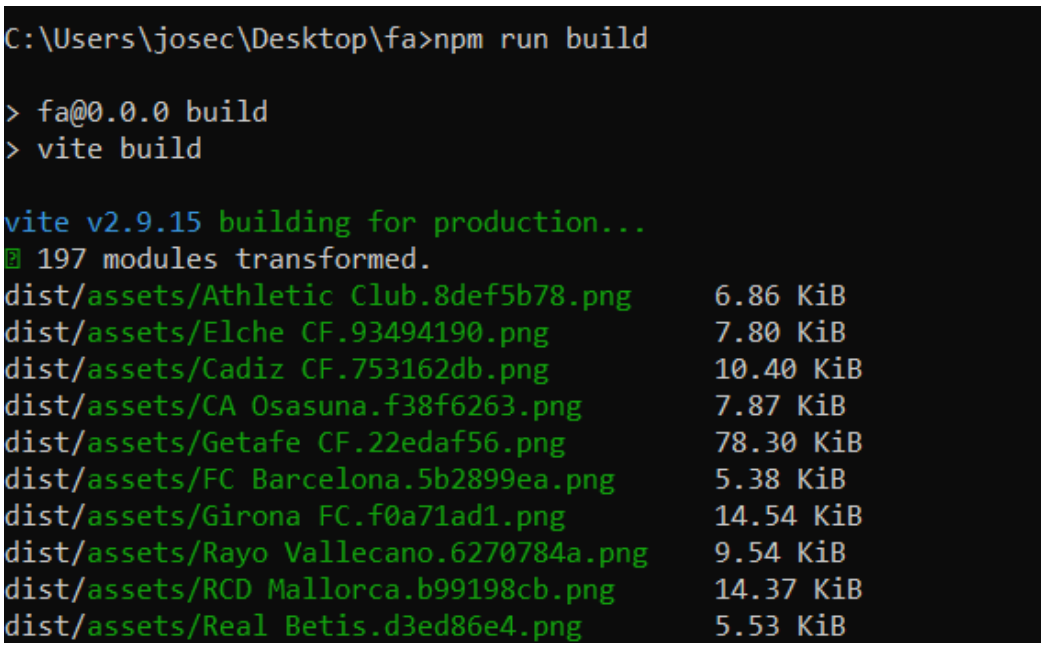

Ilustración 64: Ejecución del comando npm run build

Una vez termina, ejecutaremos *npm run preview*, que arranca el servidor web que sirve los archivos desde dist a *http://localhost* en el puerto que hayamos elegido:

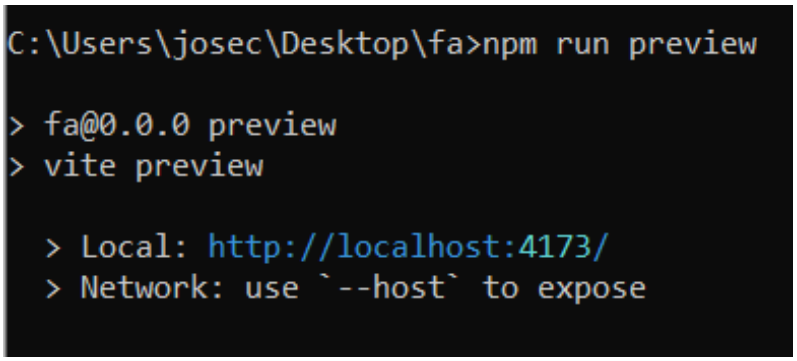

Ilustración 65: Ejecución del comando npm run preview

En el caso de que usemos un servidor dedicado, como los que hemos mencionado, solo debemos mover los ficheros estáticos de la carpeta ‘dist’ a la carpeta raíz del servidor para que pueda cargar nuestra aplicación.

9. Apéndice II. Manual de Usuario

En este apartado vamos a describir algunas de las funcionalidades principales implementadas en el proyecto final de manera sencilla. Un punto interesante sería comparar la interfaz del diseño final de la aplicación con el diseño de la interfaz propuesta en el apartado 3.5.2.

Empezando por el inicio de sesión, vamos a intentar mostrar las interfaces con todos los elementos visuales visibles que la componen, es decir, si por ejemplo un usuario intenta iniciar sesión en la aplicación pero se equivoca al poner la contraseña, vamos a mostrar esa página con el error mostrándose, evitando así tener que poner capturas de pantallas repetitivas con diferentes detalles de funcionamiento.

Cabe recalcar que muchos mensajes en pantallas que vamos a ver, como por ejemplo que al registrarse aparezca que hace falta un email válido, no están cuando se entra por primera vez a esa página, si no que se originan si no ponemos un email válido.

Cuando inicias el proyecto, la primera pantalla que aparece es la siguiente:

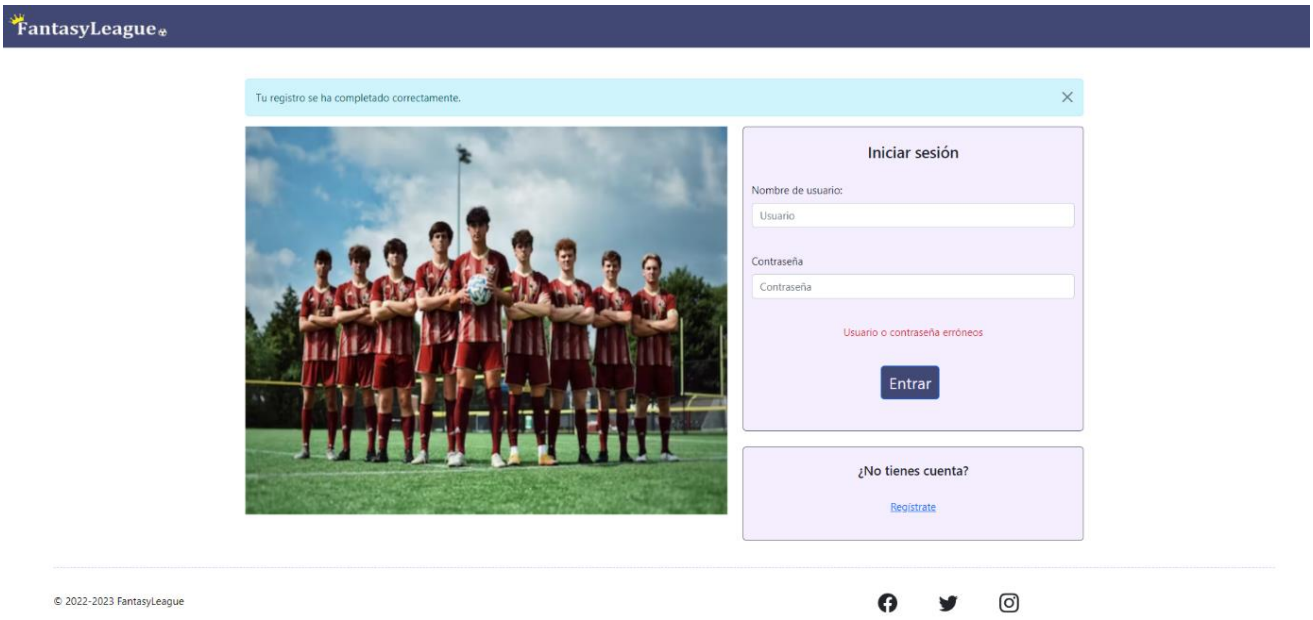

Ilustración 66: Pantalla de inicio de sesión

En la ilustración 66 se puede ver la página de inicio de sesión, muy similar al wireframe propuesto en el diseño de la interfaz, donde se muestra un mensaje de error al usuario si el usuario o la contraseña no son correctos, y en caso de que venga de un redireccionamiento desde la página de registro (ilustración 67) se muestra un mensaje indicando que el registro se ha completado satisfactoriamente. También hay que destacar el uso de migajas de pan, que se repetirá durante algunas pantallas a lo largo de la aplicación, que facilita la navegación entre páginas.

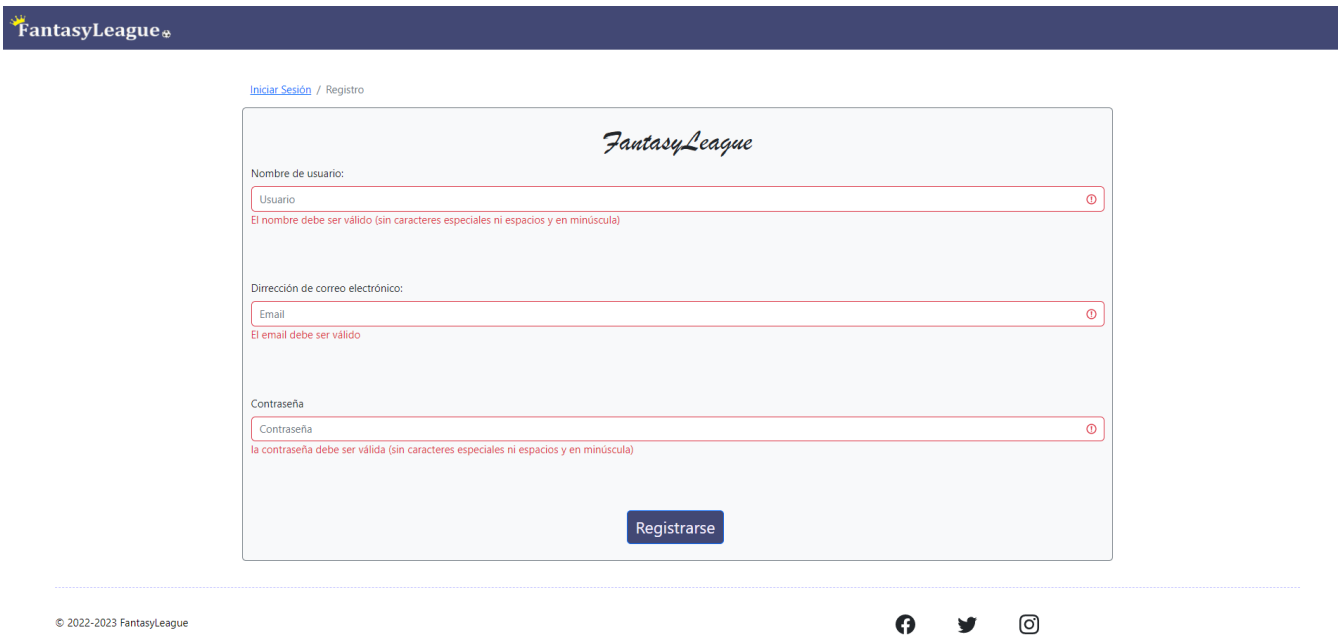

Ilustración 67: Pantalla de registro de usuarios

Vamos a empezar mostrando las funcionalidades principales para un usuario, empezando por la de buscar una liga:

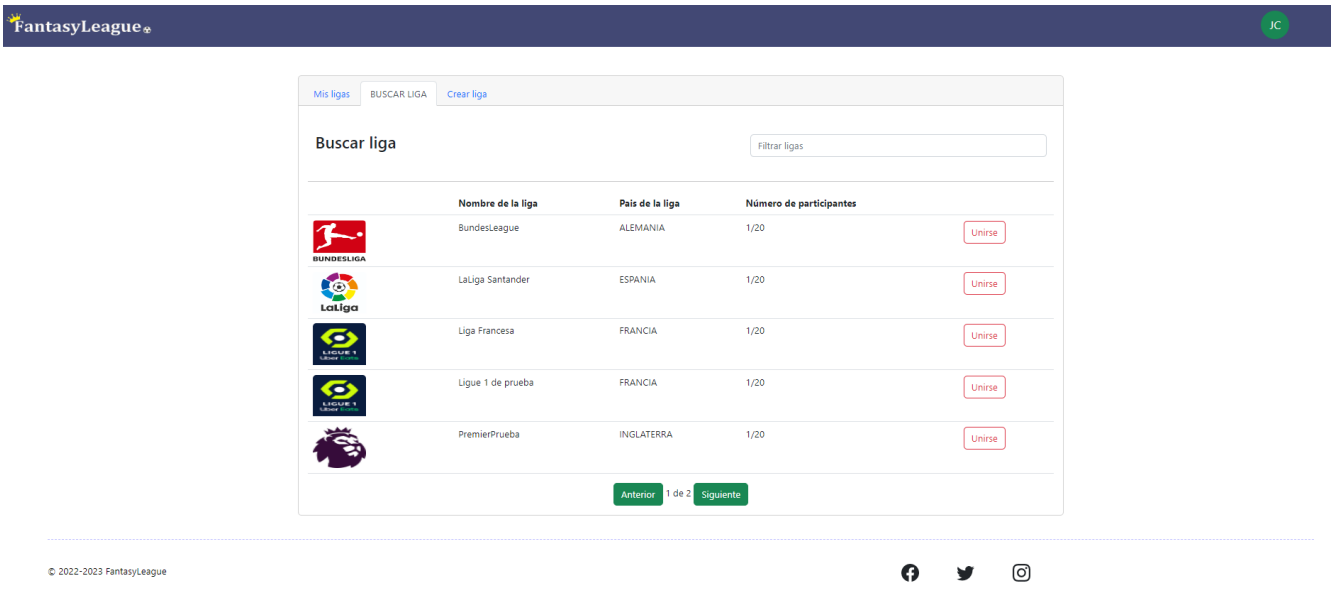

Ilustración 68: Pantalla de buscar una liga

Como se puede apreciar en la ilustración 68, para buscar una liga se nos muestra una tabla con paginación, con la opción de poder buscar una liga por su nombre. Para unirse a una liga, no basta con pulsar el botón “Unirse”, ya que al pulsar este botón saldrá otra ventana indicando si estás seguro de que te quieres unir a esa liga. Esto se hace ya que el proceso de unión de un usuario a una liga es una operación con mucha responsabilidad dentro del sistema, pues en los requisitos aún no está especificado el requisito para abandonarla, y así evitamos posibles equivocaciones.

Como la pestaña de “Mis ligas” solo contiene una tabla con todas las participaciones que contiene el usuario muy similar a la pantalla de buscar una liga, y la pestaña “crear una liga” solo contiene un formulario con los campos necesarios para crear una liga muy similar al formulario de registro. Vamos a omitir ambas pantallas y dirigirnos directamente a las pantallas de gestionar tu plantilla una vez te has unido a una liga.

Otro punto importante que no hemos comentado es que al iniciar sesión se muestra tu avatar en el header arriba a la derecha. Este avatar incluye las dos primeras letras de tu nombre de usuario y al pulsarlo te muestra un pequeño menú que permite cerrar sesión y puede incluir más funcionalidades en el futuro, como vemos en la ilustración 69:

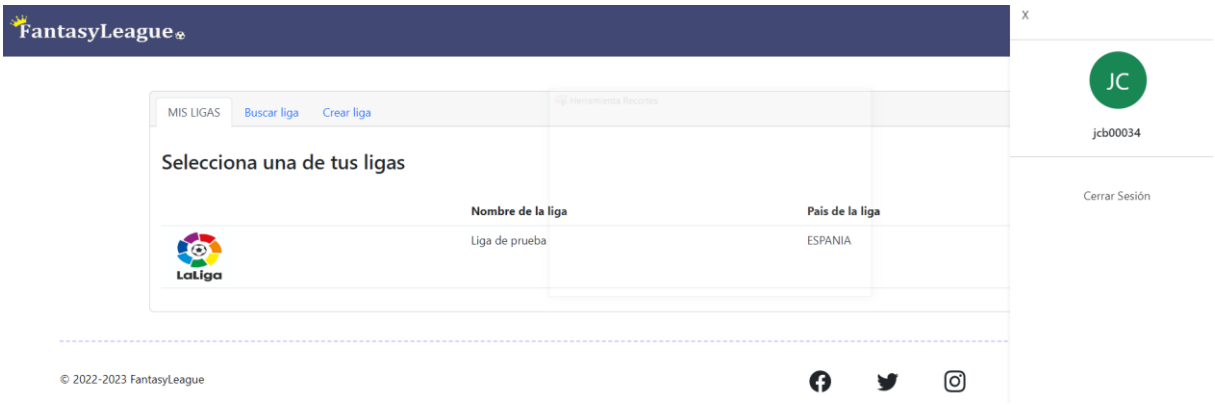

Ilustración 69: Menú de cierre de sesión

Una vez seleccionamos una liga para gestionar tu equipo, la primera pantalla que ve el usuario es la de seleccionar una alineación para la siguiente jornada:

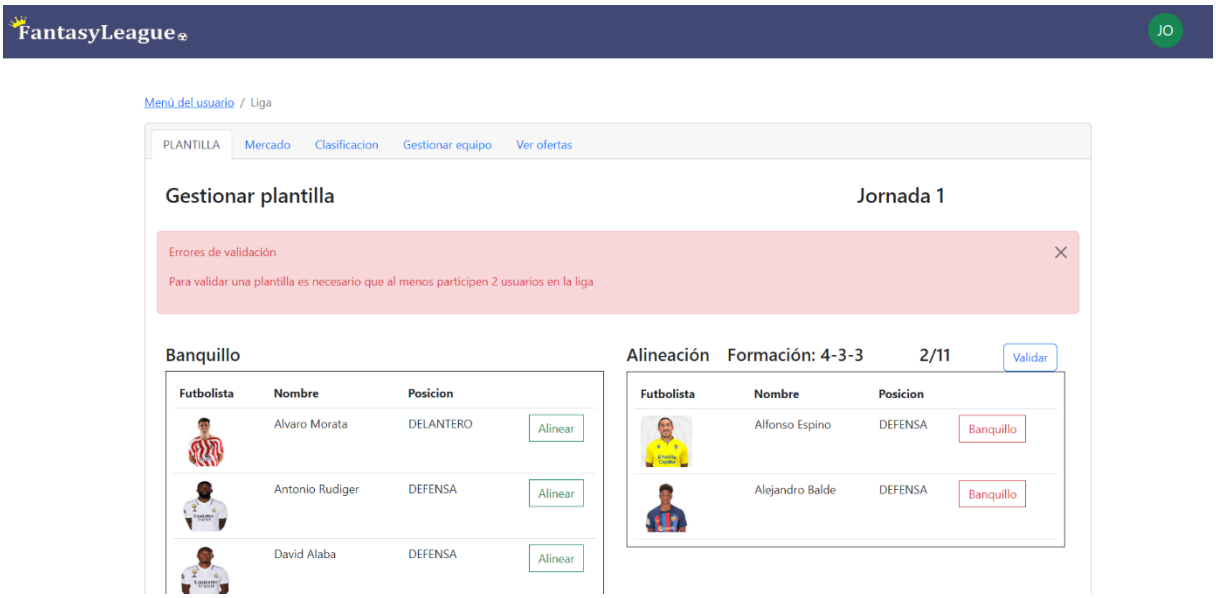

Ilustración 70: Pantalla para gestionar y validar una alineación para la siguiente jornada

Como se puede apreciar, puedes alinear desde el banquillo a la alineación los jugadores que deseas que se encuentren en tu plantilla, y validarlos mediante el botón “validar”. En caso de que la plantilla no cumpla los requisitos, como que por ejemplo no haya 2 personas en la liga para que se pueda jugar en esta liga, no haya 11 jugadores alineados o no se cumplan los requisitos de la formación, saldrá un mensaje de error que indique al usuario lo que está haciendo mal. Esta pantalla también aporta un poco más de información adicional, como por ejemplo la jornada actual en la que se encuentra la liga, la formación de la plantilla y el número de jugadores alineados.

La siguiente pantalla que vamos a ver es la de Mercado, que realmente es un listado de jugadores que contiene todos los jugadores de la liga que no han sido fichados por el usuario. Esta pantalla permite ordenar los jugadores por su precio y buscar jugadores por su alias. Sin embargo, como esta pantalla es realmente simple y no se diferencia mucho con la de la ilustración 68, vamos a centrarnos en la ventana que aparece una vez pulsamos el botón de fichar un jugador, que puede ser diferente dependiendo de si el jugador ya pertenece a un jugador o no, como vemos en las ilustraciones 71 y 72.

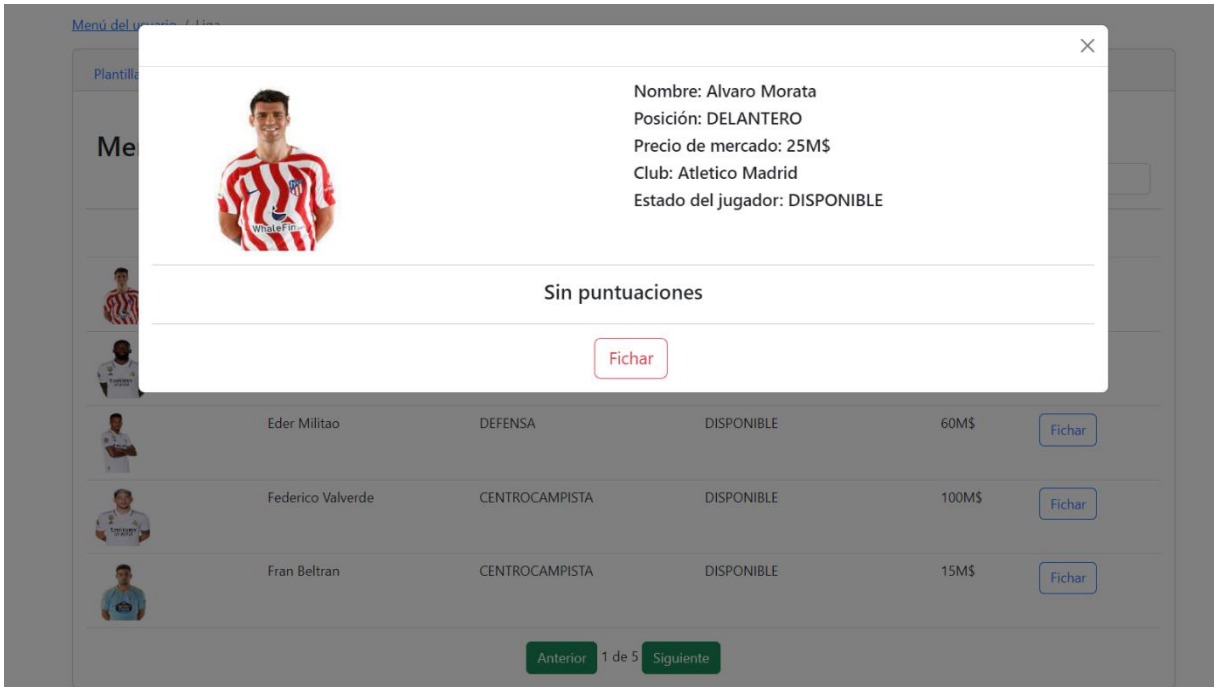

Ilustración 71: Ventana para fichar un jugador libre

En esta ventana se mostrarán todos los datos del jugador y las puntuaciones que ha tenido a lo largo de las jornadas. En este caso, como acaba de empezar la liga esta parte está vacía. Si se quiere fichar un jugador, solo hace pulsar el botón “Fichar”

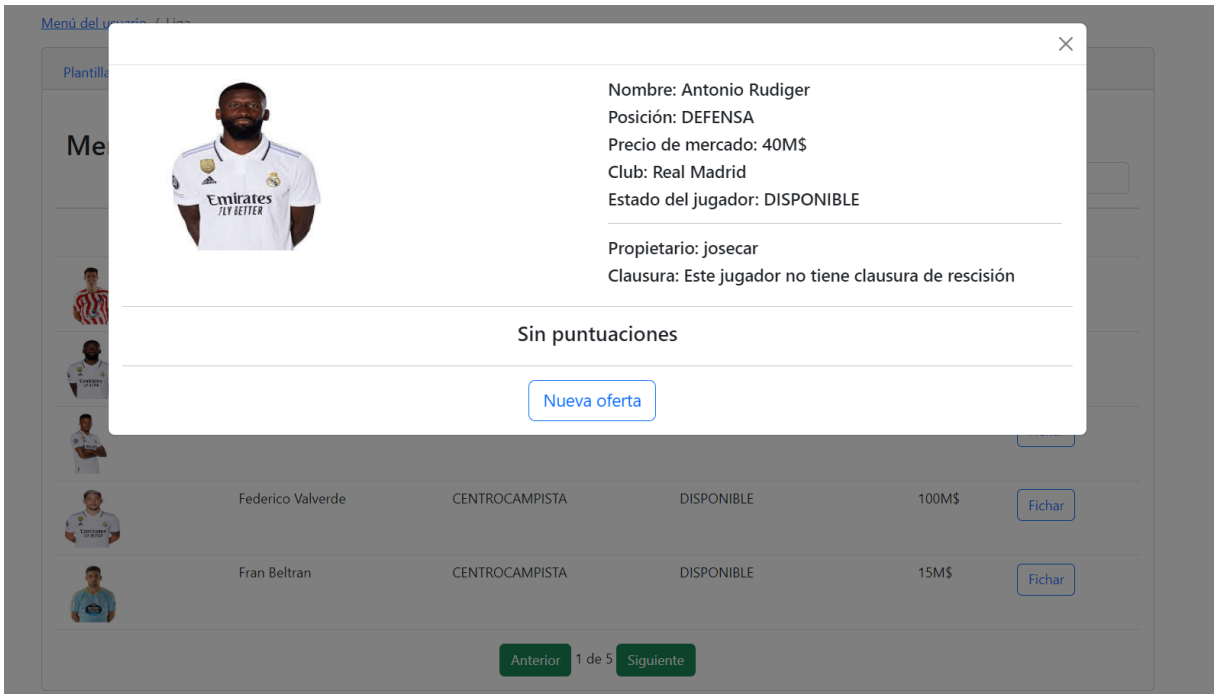

Ilustración 72: Ventana para fichar un jugador que ya ha sido comprado

En este caso, el jugador ya pertenece a otro jugador de la liga, por lo que aparece información adicional sobre el usuario que lo ha fichado y si tiene cláusula de rescisión. Si la tuviera, aparecería otra opción que permita pagar su cláusula. En el caso que pulsemos en “Nueva oferta”, aparecerá otra ventana que permita al usuario poner la cantidad que quiere pagar el usuario por el jugador.

La pantalla de “Gestionar un equipo” se vería de la siguiente forma:

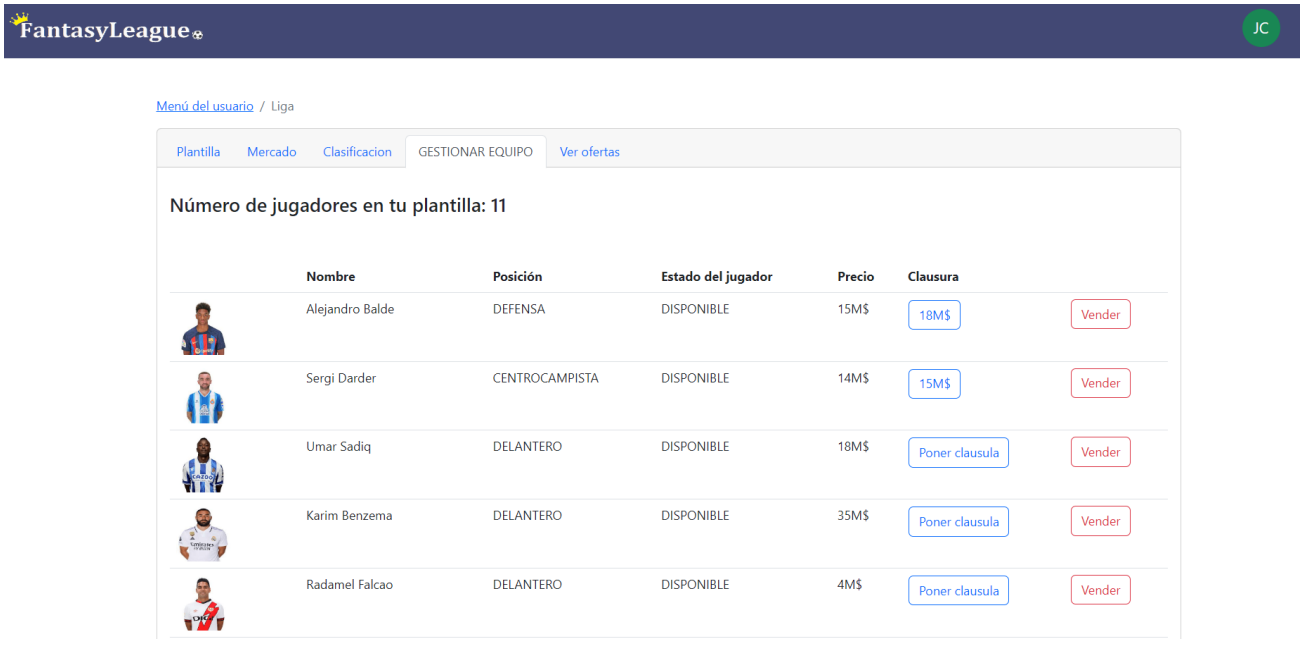

Ilustración 73: Pantalla para gestionar tu equipo

En esta pantalla se pueden vender los jugadores y poner una cláusula a un jugador. Esta opción se ve diferente indicando cuánto vale la cláusula en caso de que el jugador ya tenga una, o indicando “Poner clausula” si no contiene ninguna cláusula.

Por último, vamos a mostrar la pantalla de “Ver ofertas”. Para que esta pantalla se pueda ver con más profundidad, vamos a enviar varias ofertas al jugador “Rudiger” que ya está fichado por otro jugador y vamos a iniciar sesión con el usuario que contiene dicho jugador.

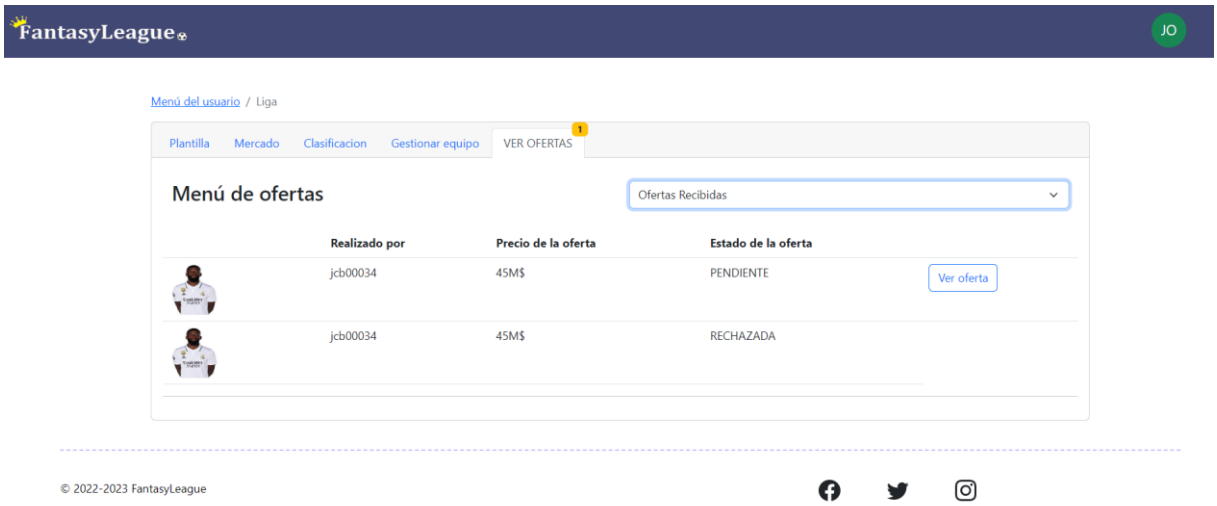

Ilustración 74: Pantalla "ver ofertas"

En esta pantalla se muestran las ofertas recibidas por tus jugadores, pudiendo aceptar o rechazar la oferta e indicando quién realiza la oferta. En caso de que haya nuevas ofertas recibidas, se mostrará el número de ofertas al lado del botón “Ver ofertas” notificando al jugador de las ofertas. También se permite cambiar entre las ofertas recibidas y las realizadas.

Para finalizar este apéndice, vamos a mostrar las funcionalidades más importantes para los administradores. Una vez inicias sesión como administrador, te aparece una vista con una lista que contiene las 5 ligas que puedes administrar.

Como esta pantalla es bastante parecida a la de la ilustración 69 (sin la parte del cierre de sesión), vamos a omitirla para ver cómo se vería la pantalla de editar un partido:

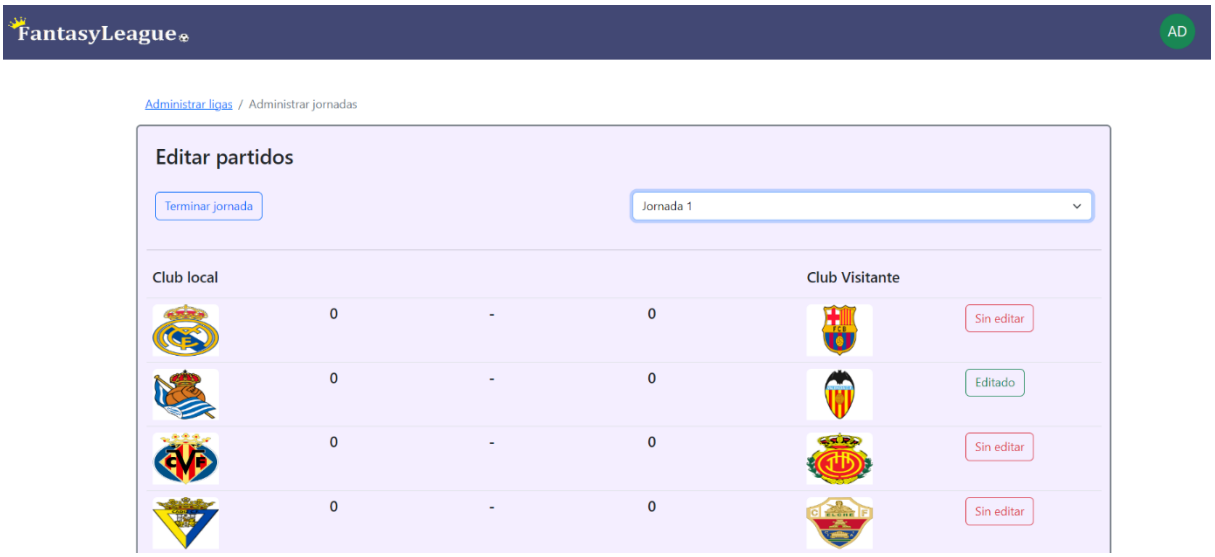

Ilustración 75: Pantalla de "visualizar jornadas"

Con el objetivo de hacer esta pantalla lo más intuitiva posible, seleccionas un partido mediante un botón que indica si el partido ya ha sido editado anteriormente o no.

También permite cambiar entre jornadas y finalizar la jornada. En la siguiente ilustración vemos cómo el administrador puede editar el resultado de un partido y seleccionar a los jugadores que desea puntuar:

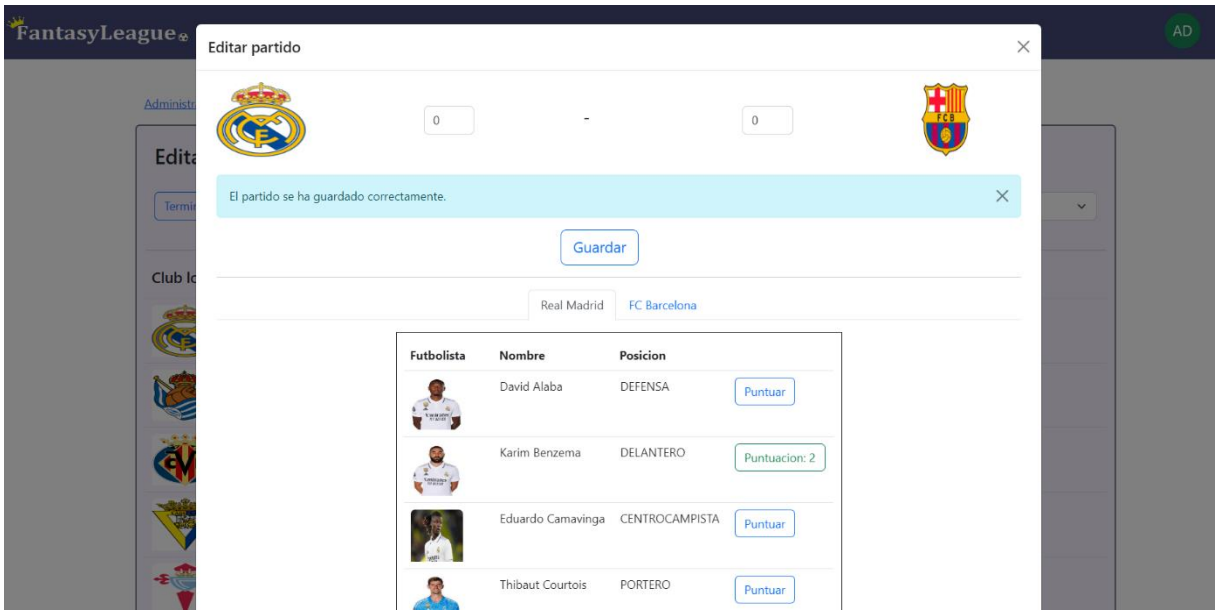

Ilustración 76: Pantalla de "editar partido"

La tabla con los jugadores de los equipos solo aparecerán una vez se ha guardado el partido, como se puede apreciar en la ilustración 76.

La forma de puntuar un jugador es parecida a la de editar un partido, donde se indica si el jugador ya ha sido puntuado, y la puntuación que ha obtenido en su caso. Si pulsamos el botón puntuar en, por ejemplo, el jugador David Alaba, se nos abrirá otra ventana que muestra lo siguiente:

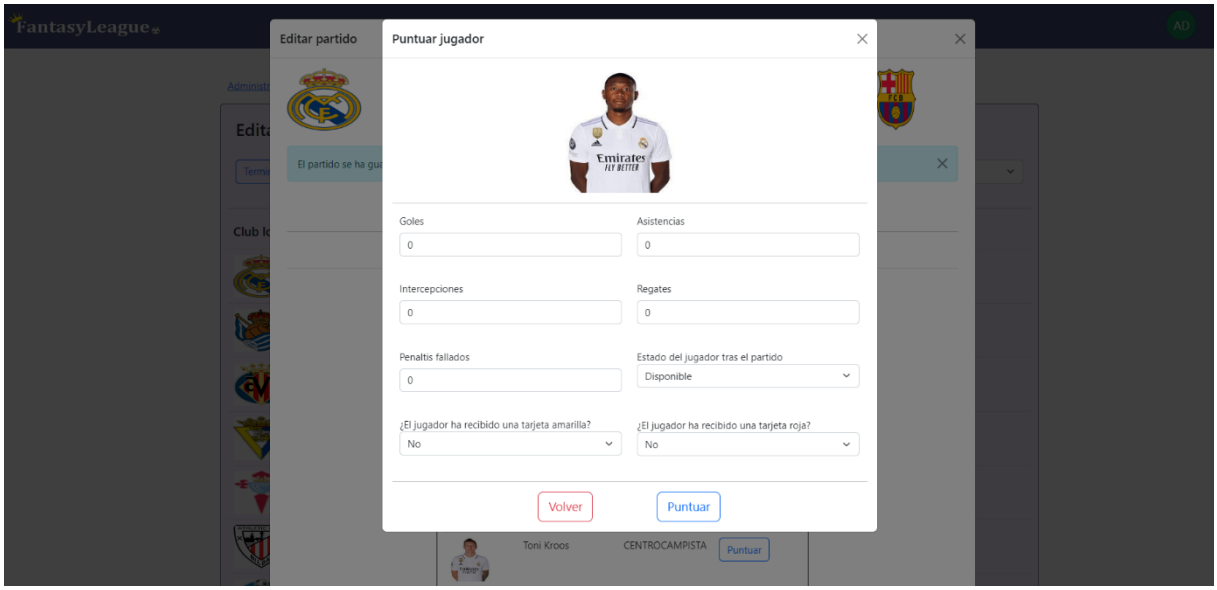

Ilustración 77: Pantalla "puntuar jugador"

En esta ventana se nos permite puntuar al jugador según las estadísticas que ha conseguido en el partido.

El hecho de que tanto la edición como la puntuación de jugadores se hagan mediante ventanas o “Modals” favorece la simplicidad de la aplicación permitiendo al usuario realizar varias operaciones de diferente naturaleza en una misma pantalla, como ya mencionamos en la etapa de diseño, y como se ha visto reflejado en el producto final.

10. Apéndice III. Descripción de contenidos suministrados

Además de la memoria, también se ha suministrado material adicional que permita evaluar todo el trabajo realizado a lo largo del proyecto.

Dicho material se encuentra en un fichero comprimido llamado “materialAdicional.zip” y está organizado a su vez en las siguientes carpetas según la ilustración 78:

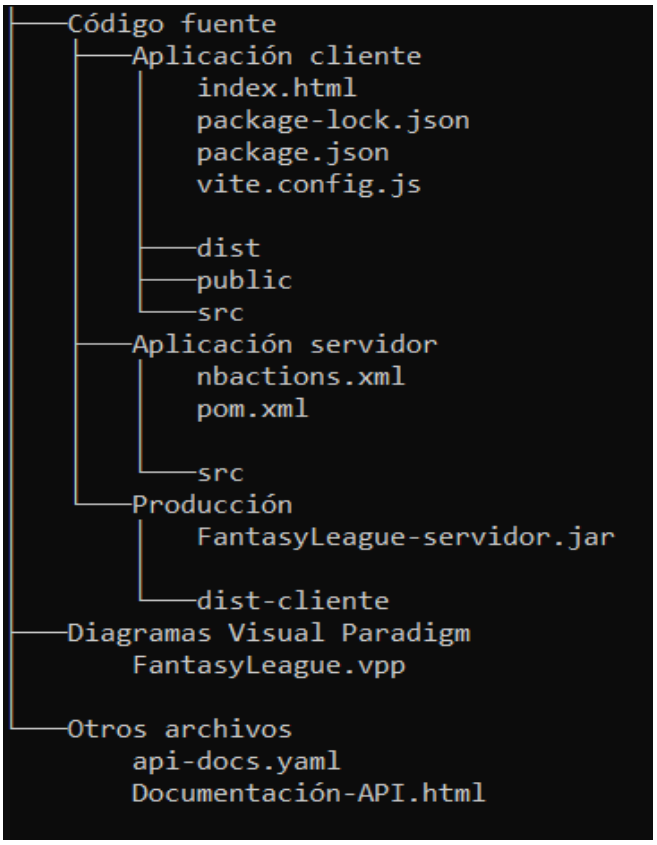

Ilustración 78: Distribución de las carpetas del fichero comprimido

En la primera carpeta nos encontramos con todo lo relacionado con el código fuente de la aplicación.

Esta carpeta, a su vez, está dividida en tres: una carpeta con el código fuente de la aplicación cliente, otra con la aplicación servidor y una carpeta llamada “producción” que contiene el archivo .jar del servidor y la carpeta con la versión front-end de producción.

La segunda carpeta suministrada llamada “Diagramas Visual Paradigm” contiene un proyecto realizado en Visual Paradigm con todos los diagramas mostrados a lo largo de la memoria, como pueden ser los de clases, casos de uso, secuencia etc.

Por último, en “Otros archivos” podemos encontrar ficheros con toda la documentación del API REST del servidor, ya que, aunque ya hablamos de su diseño en el apartado 3.5.1, no pudimos verlo con suficiente profundidad, pues tiene demasiadas clases y no queríamos hacer dicho apartado demasiado largo.

Esta carpeta consta del archivo “documentación-API”, que se trata de un archivo html donde se encuentran todos los métodos documentados (similar a la ilustración 28) de nuestra API, junto a las clases y atributos que utilizan, además de los códigos HTTP que devuelve cada uno.

También se ha suministrado el fichero .yaml que genera la documentación API, para facilitar su visualización mediante otra herramienta si fuese necesario.