Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN

Nombre del Centro

Trabajo Fin de Grado

TeamUp: Prototipo de aplicación web para conocer compañeros en videojuegos en-línea

Alumno: Alberto Longo Gómez

Tutor: Prof. D. José Ramón Balsas Almagro

Dpto: Informática

Junio, 2022

Tabla de contenidos

Tabla de contenidos 1

Tabla de Ilustraciones 4

Relación de Tablas 7

1. Introducción 8

1.1. Motivación 8

1.2. Objetivos 9

1.3. Metodología 10

1.4. Estructura del documento 10

2. Análisis 12

2.1. Análisis preliminar 12

2.1.1. Soluciones existentes 13

2.1.2. Selección de tecnologías 15

2.1.2.1. Tecnologías para back-end 16

2.1.2.2. Tecnologías para front-end 17

2.1.2.2.1. Alcance del framework 17

2.1.2.2.2. Curva de aprendizaje 17

2.1.2.2.3. Comunidad 18

2.1.2.2.4. Lenguaje de programación 18

2.1.2.3. Conclusión 18

2.2. Propuesta de solución 18

2.3. Historias de usuario 19

2.4. Planificación inicial de tareas 32

2.4.1. Propuesta inicial de iteraciones 33

2.4.1.1. Iteración 1 33

2.4.1.2. Iteración 2 33

2.4.1.3. Iteración 3 34

2.4.1.4. Iteración 4 35

2.4.1.5. Iteración 5 36

2.4.2. Evolución de la planificación 37

inicial 2.5. Estudio de viabilidad 38

2.5.1. Estimación de costes software 39

2.5.2. Estimación de costes hardware 40

2.5.3. Estimación de costes humanos 41

2.5.4. Estimación de coste total 42

2.6. Modelo de dominio 42

3. Diseño 46

3.1. Diagrama entidad-relación 46
3.2. Diagrama de clases 50
3.2.1. Servidor 51
3.2.2. Cliente 55
3.3. Diagramas de secuencia 60
3.3.1. Login 60
3.3.2. Modificar perfil de jugador 62
3.3.3. Dar like a otro jugador o equipo 64
3.3.4. Enviar un mensaje a un match 66
3.4. Diseño de la interfaz 68
3.4.1. Estética 68
3.4.2. Storyboard / Diagrama de navegación 70
3.4.2.1. Evolución del diseño de la interfaz 76
3.4.3. API Rest 79
3.4.3.1. Documentación de la API Rest 79
3.5. Plan de pruebas 89
3.5.1. Plan de pruebas en la aplicación servidor 89
3.5.2. Plan de pruebas en la aplicación cliente 91

4. Implementación 92
4.1. Arquitectura 92
4.2. Detalles sobre implementación 94
4.3. Detalles de implementación en el servidor 94
4.3.1. Diagrama de paquetes 94
4.3.2. Especificación del API Rest 97
4.3.3. Generación del servidor a partir de la especificación 101
4.3.4. Conexión del servidor a la base de datos 105
4.3.5. Generación de los end-points 107
4.3.5.1. Definición de los servicios 107
4.3.5.2. Definición de los controladores REST 108
4.3.5.2.1. Gestión de excepciones 109
4.3.6. Sistema de mensajería en el servidor 109
4.4. Detalles de implementación en el cliente 111
4.4.1. Diagrama de paquetes 111
4.4.2. Estructura de carpetas 113
4.4.3. Estructura de un componente 115
4.4.3.1. Estructura de la vista 116
4.4.3.2. Estructura de los estilos 119
4.4.3.3. Estructura de la lógica 120
4.4.4. Validación en el cliente 121
4.4.5. Interacción componente - servicio 125
4.4.6. Autenticación 126

5. Pruebas **131**
5.1. Pruebas en la aplicación servidor 131
5.2. Pruebas en la aplicación cliente 138

6. Conclusiones **144**
6.1. Mejoras y trabajos futuros 145

7. Bibliografía **148**

Apéndice 1. Descripción de contenidos suministrados. **151**

Apéndice 2. Manual de Instalación del sistema **154**
Instalación del Servidor 154
Instalación del Cliente 155

Apéndice 3. Manual de Usuario **157**

Tabla de Ilustraciones

- *Figura 1. Portada de League of Legends. (Riot Games)* 8
- *Figura 2. Portada de Valorant. (Riot Games)* 8
- *Figura 3. Portada de Rainbow Six. (Ubisoft)* 12
- *Figura 4. Sistema de selección de compañeros en E-Pal.* 13
- *Figura 5. Ejemplo de uso de la aplicación Tinder. (TechTudo, 2015)* 14
- *Figura 6. Gráfica de comparativa entre planificación ideal y real.* 37
- *Figura 7. Modelo de dominio del sistema.* 45
- *Figura 8. Gráfico de mapeo objeto-relacional.* 47
- *Figura 9. Diagrama de entidad-relación.* 49
- *Figura 10. Ejemplo gráfico de arquitectura cliente servidor.* 50
- *Figura 11. Arquitectura REST.* 52
- *Figura 12. Representación del patrón DAO.* 52
- *Figura 13. Diagrama de clases de la aplicación servidor.* 54
- *Figura 14. Arquitectura MVC.* 56
- *Figura 15. Arquitectura interna de Angular.* 57
- *Figura 16. Arquitectura MVVM.* 57
- *Figura 17. Diagrama de clases de la aplicación cliente.* 58
- *Figura 18. Diagrama de secuencia del caso de uso: iniciar sesión.* 61
- *Figura 19. Diagrama de secuencia del caso de uso: modificar perfil.* 63
- *Figura 20. Diagrama de secuencia del caso de uso: dar like.* 65
- *Figura 21. Diagrama de secuencia del caso de uso: enviar mensaje a otro jugador.* 67
- *Figura 22. Mockup de la pantalla de inicio.* 69
- *Figura 23. Mockup de la pantalla de registro.* 69
- *Figura 24. Diagrama de navegación del sistema o Storyboard.* 71
- *Figura 25. Mockup de pantalla de inicio de sesión.* 72
- *Figura 26. Mockup de pantalla de registro 2.* 72
- *Figura 27. Mockup de pantalla de registro 3.* 72
- *Figura 28. Mockup de la pantalla principal.* 73
- *Figura 29. Mockup de la pantalla de perfil.* 73
- *Figura 30. Mockup de la pantalla de torneos.* 74
- *Figura 31. Mockup de la pantalla de un torneo en específico.* 74
- *Figura 32. Mockup de la pantalla de contactos.* 75
- *Figura 33. Mockup de la pantalla de un chat.* 75
- *Figura 34. Pantalla de inicio.* 77
- *Figura 35. Pantalla de perfil.* 77
- *Figura 36. Pantalla principal.* 78
- *Figura 37. Endpoints de la entidad jugador.* 80
- *Figura 37.a. Información específica de envío al endpoint POST de la entidad jugador.* 81
- *Figura 37.b. Información específica de respuesta del endpoint POST de la entidad jugador.* 81
- *Figura 37.c. Endpoint de la entidad jugador que obtiene una lista de jugadores recomendados para un equipo.* 83
- *Figura 37.d. Información específica de respuesta del endpoint de la Figura 37.c* 83
- *Figura 37.e. Endpoint de la entidad jugador que elimina un jugador a partir de su identificador.*
- *Figura 38. Endpoints de la entidad equipo.* 84
- *Figura 38.a. Información asociada al endpoint POST de la entidad*

equipo para su registro. 85
- Figura 38.b. Información asociada a la respuesta del endpoint para obtener las preferencias del equipo. 85
- Figura 39. Endpoints asociados a la entidad usuario. 86
- Figura 39.a. Información asociada a la respuesta de un endpoint para recuperar información sobre un usuario. 87
- Figura 40. Endpoints asociados a la entidad juego. 87
- Figura 40.a. Información asociada a la respuesta de un endpoint para obtener información sobre un juego. 88
- Figura 41. Representación gráfica de una prueba de caja negra. 89
- Figura 42. Ejemplo de funcionamiento de la aplicación Postman. 90
- Figura 43. Ejemplo de funcionamiento de la interfaz del editor Swagger para el testeo de endpoints. 91
- Figura 44. Diagrama arquitectónico del sistema. 92
- Figura 45. Diagrama de paquetes del servidor. 95
- Figura 46. Comparativa entre OpenAPI 2.0 y OpenAPI 3.0. 97
- Figura 47. Especificación inicial de la documentación del API. 98
- Figura 48. Definición de un endpoint POST para la entidad juego. 98
- Figura 49. Componente reutilizable “GameDTO” que guarda la información sobre un juego. 99
- Figura 50. Interfaz que genera la especificación creada con OpenAPI 3.0 100
- Figura 51. Endpoint GET /game de la especificación generada. 101
- Figura 52. Plugin Maven para generar el código del servidor asociado a la especificación. 102
- Figura 53. Dependencias de Swagger y OpenAPI. 103
- Figura 54. Opciones de ejecución de la aplicación en Maven. 104
- Figura 55. Archivos generados por la especificación OpenAPI utilizando el plugin generador. 104
- Figura 56. Dependencias de JPA y de la base de datos H2. 105
- Figura 57. Archivo de configuración “application.properties” de Spring. 105
- Figura 58. Ejemplo de repositorio utilizando la interfaz CrudRepository de JPA. 106
- Figura 59. Ejemplo de una anotación de una clase servicio en Spring. 107
- Figura 60. Ejemplo de método de un servicio implementando la lógica de negocio. 107
- Figura 61. Ejemplo de anotación de un controlador REST en Spring. 108
- Figura 62. Ejemplos de métodos de un controlador REST, implementando la interfaz creada por el generador OpenAPI. 108
- Figura 63. Ejemplo de captura de excepciones y mapeado a códigos de respuesta dentro del controlador REST. 109
- Figura 64. Controlador REST relacionado con el sistema de mensajería instantánea. 110
- Figura 65. Método dentro del controlador REST de mensajería encargado de recibir los mensajes y redireccionarlos. 110
- Figura 66. Diagrama de paquetes de la aplicación cliente. 112
- Figura 67. Directorio de carpetas raíz del proyecto de la aplicación cliente. 114
- Figura 68. Directorio de carpetas en /app de la aplicación cliente. 114
- Figura 69. Encabezado del fichero HTML de la vista perfil. 116
- Figura 70. Etiqueta form junto a su contenido dentro de la vista perfil. 117
- Figura 71. Botonera con ejemplos de enlazado de datos en Angular. 118
- Figura 72. Archivo de estilos CSS utilizados para la vista perfil. 119
- Figura 73. Decorador de la clase que actúa como controlador de la vista

perfil. 120
- Figura 74. Método de la clase profile que se encarga de la lógica de borrar un jugador. 121
- Figura 75. Método de inicialización que contiene la creación de un formulario y sus validaciones. 122
- Figura 76. Método de inicialización de un formulario que contiene validaciones asíncronas. 123
- Figura 77. Métodos encargados de gestionar las validaciones asíncronas. 124
- Figura 78. Método encargado de gestionar una validación personalizada. 125
- Figura 79. Decorador de una clase servicio dentro de la aplicación cliente. 125
- Figura 80. Inicialización de un componente, junto a la inyección de servicios y otros componentes. 126
- Figura 81. Servicio encargado de la autenticación dentro de la aplicación cliente. 126
- Figura 82. Métodos encargados de almacenar el token dentro del almacenamiento local del navegador. 127
- Figura 83. Implementación del interceptor para comprobar la seguridad mediante token. 128
- Figura 84. Implementación de una salvaguarda para comprobar previamente los roles antes de redirigir la vista. 129
- Figura 85. Implementación del salvaguarda utilizado cuando el usuario está registrado. 129
- Figura 86. Fichero routing.module, que almacena las rutas de la aplicación junto a sus atributos. 130
- Figura 87. Ejemplo de prueba satisfactoria realizada sobre el endpoint POST /player 132
- Figura 88. Ejemplo de prueba fallida sobre el endpoint POST /player por email repetido. 133
- Figura 89. Ejemplo de prueba fallida sobre el endpoint POST /player por nickname repetido. 134
- Figura 90. Ejemplo de prueba fallida sobre el endpoint POST /player por juego no encontrado. 135
- Figura 91. Ejemplo de prueba fallida realizada sobre el endpoint GET /player/id por estar no autenticado. 136
- Figura 92. Ejemplo de prueba satisfactoria sobre el endpoint POST /auth/login 137
- Figura 93. Ejemplo de prueba satisfactoria sobre el endpoint GET /player/id 138
- Figura 94. Ejemplo de prueba realizada sobre la pantalla de registro, parte 1. 139
- Figura 95. Ejemplo de prueba realizada sobre la pantalla de registro, parte 2. 140
- Figura 96. Ejemplo de prueba realizada sobre la pantalla de registro, parte 3. 141
- Figura 97. Ejemplo de prueba realizada sobre la pantalla de registro, parte 4. 142
- Figura 98. Contenido de la carpeta src. 151
- Figura 99. Contenido de la carpeta TeamUp-Diagrams. 151
- Figura 100. Contenido de la carpeta TeamUp-Front. 152
- Figura 101. Contenido de la carpeta TeamUp-Server. 152
- Figura 102. Contenido de la carpeta build. 153
- Figura 103. Contenido de la carpeta team-up-front. 153
- Figura 104. Lanzamiento del servidor en local a partir del archivo .jar 154

- *Figura 105. Lanzamiento de la aplicación cliente en local.* 155
- *Figura 106. Pantalla de inicio de sesión.* 157
- *Figura 107. Pantalla de registro.* 158
- *Figura 108. Menú desplegable con opciones de navegación.* 159
- *Figura 109. Pantalla de perfil, parte 1* 160
- *Figura 110. Pantalla de perfil, parte 2* 160
- *Figura 111. Pantalla principal, buscando jugadores.* 161
- *Figura 112. Pantalla principal, buscando equipos.* 161
- *Figura 113. Pantalla de chat con usuario2* 162
- *Figura 114. Pantalla de selección de contacto* 162

Relación de Tablas

- *Tabla 1. Multiplicadores de los puntos de usuario dependiendo de la prioridad.* 32
- *Tabla 2. Tabla de costes software.* 39
- *Tabla 3. Tabla de costes hardware.* 40
- *Tabla 4. Tabla de costes humanos.* 41
- *Tabla 5. Tabla de coste total del proyecto.* 42

1. Introducción

Comenzaremos abordando de forma general el contexto al cual va dirigido el proyecto, proponiendo y justificando la motivación que conlleva el trabajo. También estableceremos los objetivos junto a la metodología a seguir, indicando de manera amplia la secuencia de actividades que conlleva el desarrollo del proyecto.

En este proyecto se plantea abordar el estudio y desarrollo de una aplicación basada en tecnologías web con el objetivo de crear una plataforma donde los usuarios puedan conocer y establecer contacto con otros usuarios con la finalidad de realizar proyectos comunes de manera rápida e informal.

1.1. Motivación

La plataforma que pretendemos crear se especializa en el mundo de los videojuegos, sirviendo de plataforma para encontrar compañeros de trabajo con los que fomentar el compañerismo, además de proporcionar una buena experiencia en compañía y conseguir amistades nuevas.

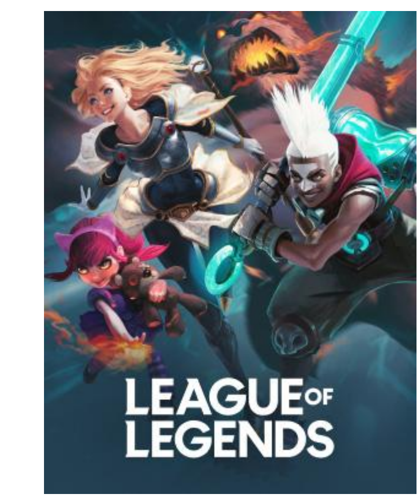	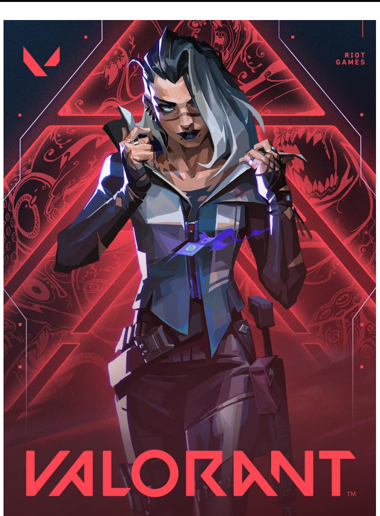
Figura 1. Portada de League of Legends. (Riot Games)	*Figura 2. Portada de Valorant. (Riot Games)*

Los videojuegos, y su faceta más competitiva, los *eSports*, están tomando cada vez más terreno en nuestra sociedad, demostrando que no son solo un *hobby*. Tras la aparición reciente de nuevos juegos competitivos como *Valorant (Figura 2)*, perfilándose como uno de los juegos más exitosos en el apartado competitivo, traído de la mano de Riot Games, los creadores del éxito mundial, *League of Legends (Figura 1)*. Una plataforma de este estilo fomentaría la agregación de jugadores para formar equipos que compitan entre ellos.

Cada jugador podría ganar experiencia trabajando en equipo o jugando con otros jugadores que encuentre, lo que haría que aumente el compañerismo, además de venir acompañado de un aumento de su nivel de juego. El compañerismo es un elemento fundamental para combatir la toxicidad en los videojuegos.

La creciente variedad de videojuegos multijugador en línea, sobre todo aquellos que son gratuitos, han fomentado la aparición de comportamientos tóxicos que arruinan la experiencia de juego. La toxicidad puede ser encontrada de varias maneras, y podemos dividirla en: insultar a compañeros, morir voluntariamente y abandonar la partida. Estas tres categorías de toxicidad acaban en el mismo resultado, una ventaja para el equipo contrario y una mala experiencia de juego para los aliados. Desde el punto de vista de los desarrolladores, la toxicidad es un elemento nefasto para la comunidad, puesto que crea un ambiente negativo del cual los jugadores optarán por dejar de jugar a dicho juego. Existen métodos en contra de la toxicidad como los sistemas de reportes, sin embargo, no son del todo eficaces para erradicar la toxicidad en videojuegos (Leal 2018). Por ello, fomentar el compañerismo y un sentimiento de competición sano es fundamental para erradicar comportamientos tóxicos dentro de la comunidad.

La plataforma puede obtener un alcance muy amplio, ya que la misión principal es reunir jugadores de distintos juegos competitivos para que encuentren compañeros de juego y, en general, amigos. Además, se puede extender de manera que aquellos jugadores que estén interesados en buscar compañeros de equipo puedan encontrar a los compañeros perfectos. Una vez un determinado grupo de jugadores hayan completado la plantilla se podrá ofrecer la posibilidad de realizar partidos de práctica contra otros equipos ya formados, afianzando así la compenetración entre jugadores formando, de esta manera, una plantilla sólida.

1.2. Objetivos

A continuación enumeramos los objetivos propuestos para el desarrollo del proyecto.

- Realizar un estudio de necesidades y soluciones para el contexto determinado.

- Diseñar un sistema informático especialmente orientado a la utilización de arquitecturas y metodologías de desarrollo web.

- Implementar un prototipo de aplicación web usando tecnologías JEE en el lado del servidor.

1.3. Metodología

En este apartado definiremos la secuencia de actividades realizadas para el desarrollo del proyecto utilizando una metodología ágil, estilo SCRUM, como se explica en (Sutherland 2021).

Una metodología ágil, como SCRUM, requiere de un número suficiente de integrantes en el equipo para tomar todos los roles necesarios para implementar SCRUM, como podría ser el *scrum-master y el equipo de desarrollo*. Sin embargo, para el desarrollo de este TFG solo habrá una persona, lo que plantea algunos inconvenientes para aplicarla tal y como está definida. Por ejemplo, existen ciertos elementos contenidos en la metodología SCRUM que dejan de tener sentido al ser un equipo de un solo integrante, como pueden ser los *daily-meetings*, puesto que las reuniones e introspecciones se realizan de forma personal y unitaria. Habiendo considerado esto, nuestra propuesta es una metodología de tipo incremental, donde aplicaremos algunas técnicas empleadas en SCRUM aunque, sin embargo, la metodología resultante no puede ser considerada como SCRUM.

Para el análisis de requisitos utilizaremos historias de usuario y descomposición en tareas. Además utilizaremos metodologías para priorización de historias como el método *MoSCoW*, que junto a una estimación de tiempo, obtendremos una estimación inicial basada en puntos de historia.

A continuación mostramos algunas de las actividades básicas que habrá que realizar para el desarrollo del proyecto.

- Estudio de los requisitos del sistema a desarrollar mediante historias de usuario y su posterior descomposición en tareas.

- Recopilación bibliográfica sobre la temática y tecnologías relacionadas.

- Realizar una priorización de historias de usuario y posteriormente una estimación temporal y de costes del desarrollo.

- Diseño del sistema utilizando una metodología de orientación a objetos, haciendo uso de patrones de diseño y arquitecturas basadas en técnicas de Ingeniería del Software.

- Implementación y pruebas del prototipo del sistema.

1.4. Estructura del documento

Las siguientes partes del documento se han organizado de la siguiente forma: en el apartado 2 de análisis estudiaremos la problemática principal y la forma en la que

abordaremos el desarrollo del sistema. Además, también realizaremos una selección de tecnologías, aportando pros y contras de algunas de las tecnologías más populares para el desarrollo de aplicaciones web. También mostraremos la división del proyecto en las diferentes historias de usuario con sus respectivas estimaciones y un estudio de costes del proyecto, analizando la viabilidad del mismo. Por último, trataremos de mostrar el dominio del problema mostrando las relaciones entre las diferentes entidades.

En el apartado 3 de diseño comenzaremos mostrando y analizando el diagrama entidad-relación de nuestro sistema. También mostraremos y explicaremos el diagrama de clases final del proyecto, viendo las interrelaciones fundamentales entre clases, haciendo hincapié sobre las clases frontera de los sistemas, tanto del servidor como de la aplicación cliente. Además, mostraremos ejemplos de comunicación entre los elementos constituyentes de ambas aplicaciones mediante el uso de diagramas de secuencia. A continuación, detallaremos el diseño de la interfaz de nuestras aplicaciones, tanto en el servidor con la implementación de un API REST, como en la aplicación cliente, mostrando un storyboard indicando las principales pantallas, así como su evolución a lo largo del tiempo. Por último, detallaremos el plan de pruebas que planteamos a la hora de realizar los diferentes testeos para el sistema completo.

En el apartado 4 de implementación estudiaremos la arquitectura general del sistema, mostrando de forma gráfica los frameworks y librerías utilizadas para llevar a cabo el desarrollo del mismo. A su vez, comentaremos diferentes aspectos de implementación claves para dejar constancia de la metodología utilizada para su implementación, así como mostrar la manera en la que el sistema fue implementado para facilitar un posible futuro mantenimiento.

En el apartado 8 de pruebas indicaremos los resultados obtenidos dado el plan de pruebas establecido al final del apartado 3.

En el apartado 9 de conclusiones, justificamos el nivel de cumplimiento de los objetivos inicialmente planteados, además de mostrar impresiones personales sobre el enfoque utilizado y las diferentes tecnologías utilizadas. También comentamos ligeramente sobre el estado actual del proyecto, así como algunos puntos relevantes del mismo. Para finalizar comentamos algunas de las mejoras y trabajos futuros que no se han llevado a cabo pero podrían realizarse en un futuro, aportando valor al proyecto.

Para finalizar, hemos aportado algunos apéndices mostrando el manual de instalación del proyecto, un manual de usuario para describir su funcionamiento de manera sencilla y una descripción de los contenidos suministrados.

2. Análisis

2.1. Análisis preliminar

En este apartado vamos a detallar la descripción de la problemática de la que partimos para, posteriormente, ofrecer una descripción del sistema para el que se realizará el desarrollo.

La problemática principal que pretendemos abordar es la creación de una aplicación que permita ser un punto de encuentro y unión para jugadores de diversos juegos competitivos de actualidad. El objetivo principal del sistema es reunir jugadores individuales para que busquen otros jugadores compatibles y formen un equipo con el que puedan jugar y mejorar.

Sin embargo, esta idea inicial, se centra en la idea de formar equipos competitivos completos, lo que nos lleva a pensar, el porqué es necesario dicha restricción, cuando el mismo sistema puede funcionar para que jugadores de forma casual, encuentren otros jugadores para encontrar un compañero de juego.

Figura 3. Portada de Rainbow Six. (Ubisoft)

La problemática de encontrar compañeros de juego, sobretodo en juegos competitivos como pueden ser *League of Legends*, *VALORANT*, *Rainbow Six* (Figura 3), etc, reside en el hecho de que para encontrar alguien con quien jugar y establecer una amistad deben darse ciertas condiciones que en un escenario normal en una partida en cualquiera de dichos juegos no suelen darse. Estas

condiciones suelen ser que dos o más jugadores actúen de manera amigable, interactuando por el chat sin importar el estado de la partida, y con intenciones de encontrar algún compañero, por lo que posteriormente al finalizar la partida, podrían decidir colaborar. Esta casuística, sin embargo, es algo poco común, por lo que es preferible la existencia de un lugar que sirva de punto de encuentro entre jugadores con intereses comunes en el que encontrar compañeros de juego. Consideramos que el sistema propuesto, aunque su objetivo sea muy específico, puede resultar muy interesante en el contexto de los juegos multiusuario.

2.1.1. Soluciones existentes

Una vez vista la problemática a tratar del sistema y valorando la solución, debemos plantearnos cómo debe de presentarse el sistema para que sea efectivo. Para ello, realizaremos una pequeña investigación de mercado, donde buscaremos aplicaciones que tengan el mismo objetivo.

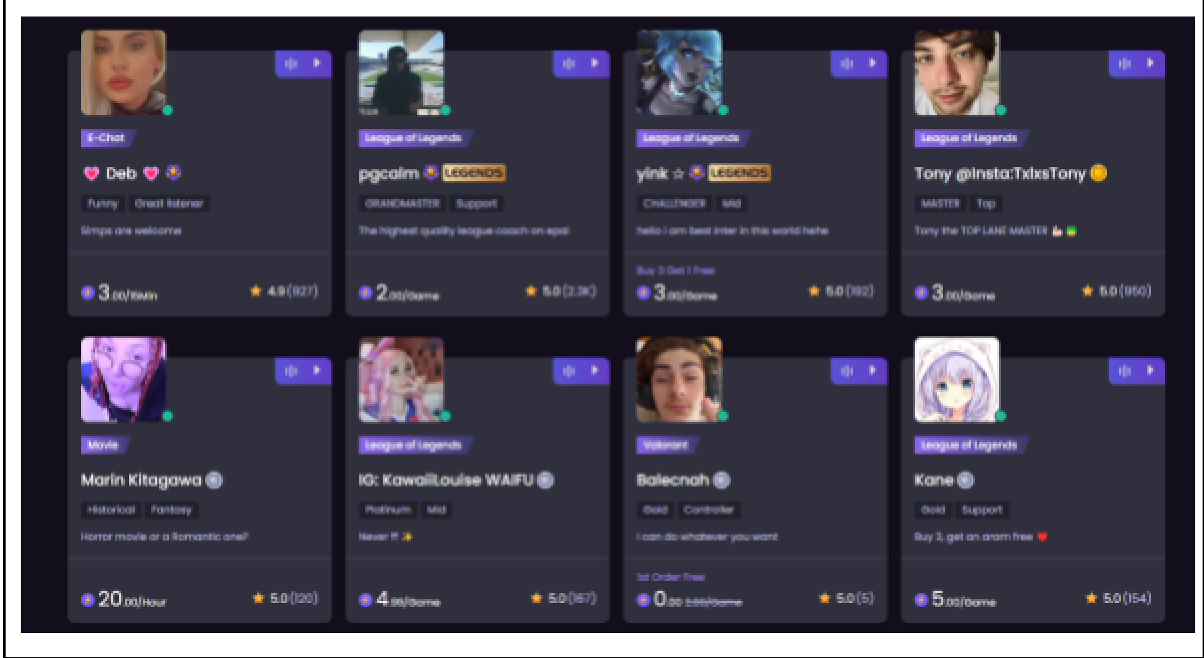

Figura 4. Sistema de selección de compañeros en E-Pal. (Fuente: https://www.epal.gg/, 2022)

Dentro de la casuística concreta, encontrar compañeros de juego, podemos encontrar pocas fuentes al respecto, ya que no existe un mercado amplio que lo rodee. Además, las pocas opciones que se ofrecen suelen ser de pago como es el caso de *E-Pal*[1], (Figura 4). Este sitio ofrece una solución de forma que una persona se presta a ser un compañero de juego, como si fuese un catálogo y, dependiendo de las preferencias de la persona que presta el servicio, esta puede cobrar por ofertar sus servicios.

[1] E-Pal es una plataforma para poder contratar compañeros de juego donde escoges un jugador/a en concreto, pagas una cuota y juegas con él/ella.

En nuestro caso, no es la solución que tenemos en mente, puesto que la aplicación mencionada ofrece una solución temporal a la problemática mencionada.

Si abstraemos el problema un paso más atrás, y nos centramos en la problemática social de encontrar compañeros, ya sea en forma de amistad o romántica. Podemos encontrar aplicaciones como *Tinder*[2], una aplicación en la que nosotros establecemos nuestros gustos y preferencias, y su algoritmo nos mostrará una serie de personas que pueden coincidir con nuestros gustos según nuestras preferencias, distancia, etc.

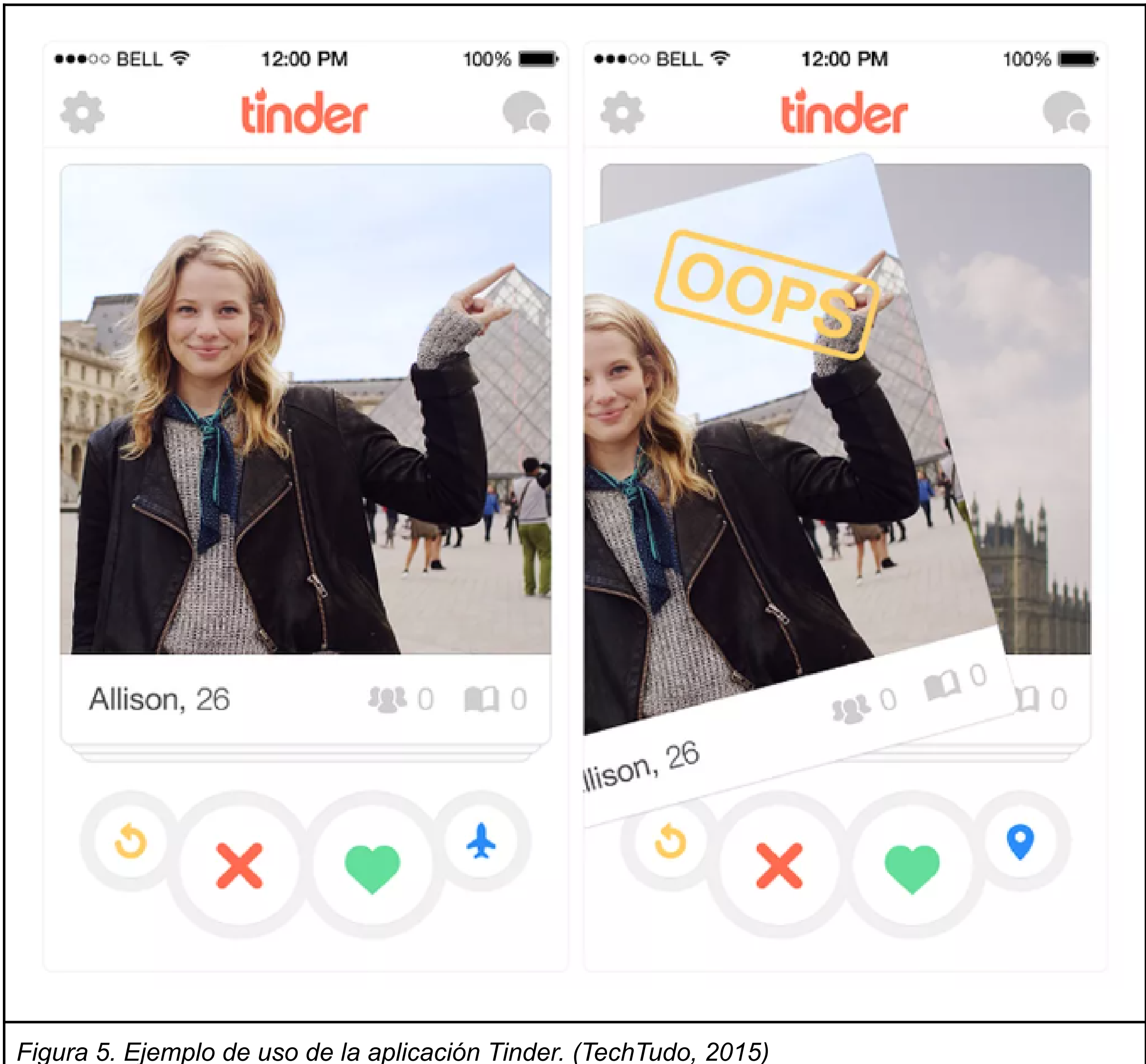

Figura 5. Ejemplo de uso de la aplicación Tinder. (TechTudo, 2015)

Además, en Tinder, el sistema de selección de compañeros es simple y sencillo, como se observa en la Figura 5, se ofrece una serie de tarjetas con la información justa y necesaria para saber lo que estás buscando. Citando a Igor Ochoa sobre el sistema de selección de Tinder:

[2] https://tinder.com/es-ES

> *"Por su simplicidad esta funcionalidad se la han ido apropiando otras aplicaciones de dating[3], pero también herramientas de todo tipo. Incluso hay plataformas para buscar puestos de trabajo o candidatos que la han incorporado. No obstante, también ha recibido bastantes críticas, ya que según algunos fomenta la superficialidad, los juicios y el tratar a las personas de una forma muy material"* (Ochoa 2019).

Esto nos lleva directamente a pensar que la solución debe residir en una aplicación móvil, lugar donde las redes sociales toman fuerza y podemos aprovechar la simplicidad de controlar una pantalla táctil para obtener una mayor atracción del público objetivo.

Con esto dicho, nuestro sistema debe de ser capaz de mostrar tanto jugadores como equipos de una forma condensada, para que un usuario pueda captar la información rápidamente y elegir si le interesa o no para ser un compañero de juego. Mostrando un jugador o equipo detrás de otro, creando una dinámica donde el usuario sienta que tiene muchas opciones donde elegir. De forma análoga, para que dos jugadores se pongan en contacto, ambos deben de indicar que están interesados en el otro de forma mutua, lo que en *Tinder* y demás aplicaciones similares se considera un *"like"*. Una vez que dos likes sean mutuos, se creará un *"match"*, es decir, una coincidencia entre usuarios, lo que abrirá una conversación entre ambos para que puedan conversar. De esta forma, consideramos que se crea un vínculo mayor que al buscar compañeros de equipo en una partida dentro del propio juego, aunque este disponga de un sistema de amigos dentro del propio juego. Por el contrario, como lo plantea nuestro sistema, ambos jugadores dan por asegurado que el otro está buscando compañeros para jugar juntos.

2.1.2. Selección de tecnologías

Una vez hemos ahondado en la problemática a tratar, y visto diferentes aspectos positivos que aportan valor a nuestro sistema, hablaremos sobre la propuesta de implementación de nuestro sistema. La primera decisión sobre la implementación es la creación de una aplicación web, en lugar de crear una aplicación de escritorio (Internet Ya 2020). La decisión de escoger una solución web sobre una de escritorio es debida a que ofrece varias ventajas. Una solución web nos permite utilizarla desde cualquier lugar, sin necesidad de actualizaciones ni incompatibilidades con versiones anteriores. La información se encuentra centralizada en un servidor común a todos los clientes. Además, esto no nos limita a la hora de escoger sistema operativo lo que es una gran ventaja puesto que nos permitirá ampliar su uso a dispositivos móviles, de forma inmediata, sin necesidad de implementar aplicaciones nativas.

[3] Las aplicaciones de dating son aplicaciones de citas online, donde personas buscan conocer a otras personas.

La utilización de una solución web sin embargo nos plantea algunas desventajas que debemos tener en cuenta a la hora de la implementación. La más evidente es la obligatoriedad de una conexión de red disponible. Además, esta dependencia de una conexión a internet crea unos tiempos de respuesta más lentos, debido a que la conexión en ocasiones puede ser inestable. Sin embargo, estos problemas se minimizan con tecnologías de carga asíncrona como es AJAX (Wikipedia 2022).

Para resumir un poco los motivos de peso a la hora de escoger una solución web sobre una de escritorio, nos encontramos con una mejor portabilidad y facilidades de instalación. Además, una aplicación web no necesita un rendimiento equiparable a una aplicación de escritorio y por último no requiere de software específico y librerías de terceros más allá del navegador.

2.1.2.1. Tecnologías para back-end

Una vez definida la solución web como opción predilecta, tendremos que escoger dentro del abanico de opciones que ofrece el desarrollo web tanto para el desarrollo en el cliente o front-end y para el desarrollo en el servidor o back-end.

Para el desarrollo de la parte del servidor, utilizaremos Spring (Wikipedia 2022), un framework open source que simplifica el proceso de desarrollo de aplicaciones en Java. Spring es un *framework* modulable que nos permite trabajar de forma rápida y sencilla gracias a los servicios que nos permite tales como la inversión de control (IoC[4]), la programación orientada a objetos o el uso de la arquitectura modelo vista controlador. La primera versión de Spring se remonta al 2002, escrita por Rod Johnson, lanzado bajo la licencia Apache 2.0 en junio de 2003. La versión actual es la 5.3.20 (Spring 2021).

Además de las ventajas que ofrece Spring facilitando el desarrollo mediante los diversos módulos que proporciona, es un framework con mucho soporte por parte de la comunidad debido a su uso extenso por diferentes usuarios a lo largo del mundo. Esto facilita la detección de errores y soluciones por parte del desarrollador que encuentra soporte de manera fácil y sencilla. Su popularidad también crea una compatibilidad entre muchos entornos de desarrollo y otros sistemas necesarios para el desarrollo como pueden ser los sistemas gestores de bases de datos.

Además, otro de los grandes motivos para haber escogido Spring frente a otras soluciones es debido a la formación previa durante el grado, lo que me permitirá profundizar su uso, además de acelerar el tiempo necesario para la fase de desarrollo al no necesitar un aprendizaje previo en profundidad.

[4] La inversión de control nos permite configurar los componentes y administrar el ciclo de vida de los objetos, llevado a cabo a través de la inyección de dependencias.

2.1.2.2. Tecnologías para front-end

Sin embargo, si nos movemos a la toma de decisión de un framework para el desarrollo en el cliente, nos encontramos con un amplio abanico de posibilidades, donde la mayoría suelen contener los servicios necesarios para un correcto desarrollo de manera sencilla. Para reducir el abanico de posibilidades a un tamaño manejable, comentaremos las principales similitudes y diferencias entre dos de los framework más utilizados en el desarrollo en el cliente, siendo Angular, y React (Delgado 2020).

2.1.2.2.1. Alcance del framework

Comencemos con lo más elemental, puesto que la palabra framework para referirse a React, no es la más correcta. React es una librería de Javascript para el desarrollo de interfaz de usuario, mientras que Angular si que es un framework de Javascript de código abierto para el desarrollo web y móvil.

El hecho de que React sea una librería, significa que su autosuficiencia como framework de desarrollo está limitado al uso de otras librerías de terceros como pueden ser Redux, React Router, etc para optimizar la gestión del estado, así como gestionar el enrutamiento. También para elementos básicos como el enlazado de datos, validaciones de formularios o inyección de dependencias requiere la instalación de módulos adicionales.

Angular en cambio, es un framework completo para el desarrollo de software, puesto que no requiere de librerías adicionales para un completo desarrollo, estando todas las funcionalidades mencionadas anteriormente integradas en Angular de forma nativa.

2.1.2.2.2. Curva de aprendizaje

React, al ser una librería, su aprendizaje es más liviano y minimalista. Este viene reducido si se posee un conocimiento previo de Javascript. Sin embargo, este aprendizaje liviano dura poco tiempo, puesto que al necesitar de librerías de terceros, se necesita de más tiempo de aprendizaje para configurar un proyecto correctamente, además de aprender a utilizar las librerías a utilizar.

Angular por el contrario, es un framework muy amplio y el aprendizaje de todas las librerías asociadas es un proceso largo y tedioso, siendo más extenso que React. Su sintaxis es más compleja, aunque permite un desarrollo más escalable de forma nativa. Angular utiliza como lenguaje de programación predeterminado Typescript, siendo similar a Javascript, pero tiene elementos que el usuario tendrá que aprender.

2.1.2.2.3. Comunidad

React es la librería de Javascript más popular en todo el mundo, y tiene una comunidad de desarrolladores que la apoya y ofrece soporte al público.

La documentación de React es algo escasa, sin embargo, el apoyo de la comunidad en diversos foros será capaz de solucionar la mayoría de problemas que tengas.

Por otro lado, el público es algo más escéptico con Angular, puesto que al ser un framework más complejo de utilizar, su comunidad es menor que la de React. Sin embargo, bajo la tutela de Google, su popularidad ha ido creciendo en los últimos años, además de tener un soporte a largo plazo y actualizaciones constantes.

2.1.2.2.4. Lenguaje de programación

React utiliza el lenguaje de programación Javascript, aunque permite también el uso de Typescript, que combinado con JSX script, siendo este último una extensión de sintaxis, para crear un código en Javascript similar a HTML.

Angular por otro lado, puede utilizar tanto Javascript como Typescript, siendo este último más compacto que Javascript, teniendo una detección de errores tipográficos detectables más fácilmente y acelera en gran medida el proceso de refactorización.

2.1.2.3. Conclusión

Pese a las diferencias, para el proyecto que vamos a desarrollar, Angular va a desempeñar un papel muy importante dentro del desarrollo de la aplicación puesto que, a pesar de la complejidad inicial, una vez comprendida la filosofía y modelo de desarrollo, Angular ofrece soluciones integradas en el propio *framework*, sin la necesidad de requerir librerías adicionales cuando se necesitan nuevas funcionalidades. El ejemplo más práctico es la posibilidad de desarrollar una aplicación para dispositivos móviles en Angular de forma nativa, sin necesidad de instalar una librería de terceros como en el caso de React.

2.2. Propuesta de solución

En apartados anteriores hemos abarcado muchas consideraciones de análisis. En este apartado intentaremos resumir lo comentado anteriormente, ofreciendo una visión general del sistema que se pretende desarrollar.

Dada la problemática mencionada anteriormente, nuestro sistema consistirá en una aplicación donde un usuario pueda registrarse como jugador, establecer unas preferencias, y encontrar otros jugadores que sean parecidos. Para ello dispondrá de un sistema de *likes* y *matching*, donde cuando ambos estén interesados mutuamente el sistema los podrá en contacto mediante un chat.

La aplicación tiene el propósito de unir jugadores para que puedan encontrar compañeros de juego. Además, se ofrecerá la posibilidad de crear un equipo desde cero, o entrar en otro ya creado, donde se buscarán jugadores que se ajusten a las preferencias del equipo.

De forma complementaria, los equipos podrán entrar en torneos o partidas para probar su destreza jugando en partidos de prueba que se podrán gestionar desde la aplicación.

Para el desarrollo de esta aplicación, utilizaremos Spring como framework de desarrollo en el servidor junto a H2 como gestor de base de datos relacional, un gestor integrado completamente en Spring con un fácil uso.

Para el desarrollo de la aplicación cliente utilizaremos Angular junto con Material UI Design[5] para la creación de interfaz de usuario, debido a su gran compatibilidad y facilidad de uso. Además, *Material* contiene componentes creados directamente de la librería que facilitan el desarrollo de la interfaz.

2.3. Historias de usuario

Las historias de usuario son unos de los componentes centrales de un programa ágil. Son una explicación general e informal de una función de software escrita desde la perspectiva del usuario final, cuyo propósito es mostrar en un lenguaje común como proporcionará una función software valor al cliente (Rehkopf 2022).

El esquema que siguen las siguientes historias de usuario se expresan en una frase con la siguiente estructura: “Como [perfil], [quiero] [para].” Esto nos permite detallar en pocas palabras la funcionalidad que queremos lograr, vista desde el punto de vista del usuario, para posteriormente desgranarla en pequeñas tareas, que junto a unos ciertos criterios de aceptación, ayudarán a reflejar lo realmente importante para el usuario y, por tanto, lo importante a implementar.

Como hemos mencionado anteriormente, posteriormente habría que añadir los criterios de aceptación y división en subtareas. También será de utilidad utilizar un sistema de puntuación a cada una de las historias de usuario, puesto que la idea es completarlas en cierto periodo de tiempo. Esta puntuación vendrá determinada por unos puntos de esfuerzo, que contarán la dificultad necesaria para realizar dicha historia de usuario, así como una priorización, establecida mediante una escala de importancia desde el punto de vista del usuario.

Para este sistema de estimación, utilizaremos el método MoSCoW, técnica que nos sirve para determinar la priorización dentro de proyectos con limitaciones de tiempo

[5] https://material.angular.io/

como se explica en (Simões 2020). El objetivo principal de esta técnica es dividir nuestras historias en cuatro categorías:

- Must have (Debe): Características absolutamente clave y que sin ellas el proyecto sería un fracaso.
- Should have (Debería): Aspectos críticos, pero no imprescindibles.
- Could have (Podría): Aportan valor al proyecto, pero no son críticas.
- Won't have (No se van a hacer): Características sin mérito de inversión, no aportan beneficio.

Por sí solo, el método MoSCoW es muy útil para estimar la prioridad de las historias de usuario desde el punto de vista del valor de producto. Sin embargo, en nuestro caso, tendremos en cuenta un segundo parámetro que llamaremos puntos de esfuerzo, incluidos en muchas historias de usuario, que vendrá medido por una estimación del tiempo necesario (en horas) para completar la historia de usuario.

Aquí presentamos la lista completa de historias de usuario creadas para la aplicación.

1) Como usuario/a, quiero poder registrarme en la aplicación, para poder tener un perfil y hacer uso de las funcionalidades que esta ofrece.
    - Priorización: Must have
    - Esfuerzo: 20h.
    - Subtareas:
        1) Creación de entidades para la transferencia de datos (DTO)
        2) Creación de entidad para almacenar en la base de datos (database)
        3) Creación de operación para añadir usuario en el servidor (back-end)
        4) Gestión de errores para evitar usuarios repetidos en el servidor (back-end)
        5) Creación de interfaz de usuario en el cliente (front-end)
        6) Sistema de validación de formularios síncrona para evitar errores en el cliente (front-end)
        7) Sistema de validación de formularios asíncrona para evitar restricciones de valores repetidos (front-end)
    - Criterios de aceptación:
        1) Ninguno de los campos de registro pueden estar vacíos, en caso contrario, se mostrará un error.
        2) El nombre de usuario debe de contener entre 5 y 16 caracteres, además de solo contener letras y números, en caso contrario, se mostrará un error.
        3) El correo electrónico debe de ser un correo electrónico válido, en caso contrario, se mostrará un error.

4) El nombre de usuario no puede repetirse con otros usuarios, en caso contrario, se mostrará un error.
5) El correo electrónico no puede repetirse con otros usuarios, en caso contrario, se mostrará un error.
6) La contraseña debe de contener como mínimo 8 caracteres, en caso contrario, se mostrará un error.
7) El nombre completo debe de contener entre 8 y 25 caracteres, y solo puede contener letras, en caso contrario, se mostrará un error.
8) No se puede completar el proceso de registro hasta que se hayan completado todos los campos correctamente.

2) Como usuario/a, quiero poder iniciar sesión con mis credenciales creadas previamente para entrar en la aplicación.
   - Priorización: Must have
   - Esfuerzo: 25h.
   - Subtareas:
     1) Creación de entidades para la transferencia de datos (DTO)
     2) Creación de operación para iniciar sesión en el servidor (back-end)
     3) Creación de interfaz de usuario en el cliente (front-end)
     4) Creación de sistema de generación y comprobación de tokens en el servidor (back-end)
     5) Implementación de inicio de sesión mediante token en el cliente (front-end)
     6) Creación de funcionalidad para refrescar el token para evitar salirse de la sesión (back-end)
     7) Implementación de funcionalidad de refrescar el token en el cliente (front-end)
     8) Creación de servicio de seguridad para proteger las rutas (back-end)
     9) Creación de seguridad de datos protegidos para acceder a datos privados (back-end)
   - Criterios de aceptación:
     1) El nombre de usuario no puede estar vacío, en caso contrario, se mostrará un error.
     2) La contraseña no puede estar vacía, en caso contrario, se mostrará un error.
     3) Si la contraseña para el nombre de usuario introducido es correcta, se redirigirá a la pantalla principal, en caso contrario, se mostrará un error.

3) Como usuario/a, quiero poder eliminar mi cuenta para darme de baja completamente de los servicios que ofrece la aplicación.

- Priorización: Should have
- Esfuerzo: 12h.
- Subtareas:
    1) Creación de la operación para eliminar un usuario del servidor (back-end)
    2) Comprobación de seguridad de la operación (back-end)
    3) Implementación de la funcionalidad de eliminar usuario en el cliente (front-end)
    4) Sistema de confirmación de borrado en el cliente (front-end)
    5) Actualizar asociaciones correspondientes en la base de datos (database)
- Criterios de aceptación:
    1) El usuario debe de estar registrado y haber iniciado sesión.
    2) El usuario debe de confirmar la eliminación de la cuenta, en caso contrario, no se eliminará.
    3) Si el usuario confirma la eliminación de la cuenta, se cerrará su sesión.
    4) El usuario con todos los datos asociados deben de eliminarse completamente del sistema.

4) Como usuario/a, quiero poder modificar mi información personal como es mi nombre de usuario, nombre completo, correo electrónico, fecha de nacimiento y género para reflejar y actualizar mi perfil personal en cualquier momento.
    - Priorización: Should have
    - Esfuerzo: 12h.
    - Subtareas:
        1) Creación de la operación para modificar un usuario en el servidor (back-end)
        2) Comprobación de seguridad de la operación (back-end)
        3) Implementación de la funcionalidad de modificar usuario en el cliente (front-end)
        4) Gestión de la validación de formularios para modificación de un usuario en el cliente (front-end)
        5) Actualización de los valores correspondientes en la base de datos (database)
    - Criterios de aceptación:
        1) Ninguno de los campos de modificación pueden estar vacíos, en caso contrario, se mostrará un error.
        2) El nombre de usuario debe de contener entre 5 y 16 caracteres, además de solo contener letras y números, en caso contrario, se mostrará un error.
        3) El correo electrónico debe de ser un correo electrónico válido, en caso contrario, se mostrará un error.

4) El nombre de usuario no puede repetirse con otros usuarios, en caso contrario, se mostrará un error.
5) El correo electrónico no puede repetirse con otros usuarios, en caso contrario, se mostrará un error.
6) El nombre completo debe de contener entre 8 y 25 caracteres, y solo puede contener letras, en caso contrario, se mostrará un error.

5) Como usuario/a, quiero poder dar *like* a jugadores y equipos para conseguir un posible match con un jugador o equipo que me parece interesante.
   - Priorización: Must have
   - Esfuerzo: 20h.
   - Subtareas:
     1) Creación de las entidades usuario, jugador y equipo
     2) Creación de las entidades para la transferencia de datos (DTO)
     3) Creación de las funcionalidades de dar like a una entidad en el servidor (back-end)
     4) Almacenamiento de las entidades junto a los likes dados y recibidos en la base de datos (database)
     5) Creación de la funcionalidad de dar like en el cliente (front-end)
     6) Creación de la interfaz de dar like en el cliente (front-end)
   - Criterios de aceptación:
     1) El usuario debe de estar registrado y haber iniciado sesión.
     2) La información relacionada con el jugador o equipo debe de estar visible y resumida.
     3) El usuario debe de clicar en el botón correspondiente para dar like al jugador o equipo.
     4) El usuario puede clicar en otro botón para no dar like al jugador o equipo y pasar al siguiente.

6) Como jugador/a, quiero poder contactar con otros jugadores interesados en buscar compañeros para poder conocer otros jugadores sin necesidad de entrar en un equipo.
   - Priorización: Must have
   - Esfuerzo: 20h.
   - Subtareas:
     1) Creación de entidades para la transferencia de datos (DTO)
     2) Creación de entidades para la gestión de mensajería.
     3) Creación de controlador específico de mensajería (back-end)
     4) Implementación de funcionalidad de enviar mensaje (back-end)
     5) Implementación de funcionalidad de ver mensajes (back-end)
     6) Implementación de sistema de salas para gestionar múltiples chats (back-end)

7) Gestión de entidades de mensajería almacenadas en la base de datos (database)
8) Implementación de sistema de mensajería en el cliente (front-end)
9) Creación de la interfaz de usuario asociada a la mensajería en el cliente (front-end)
10) Creación del sistema de notificaciones en el cliente (front-end)
11) Validación de formularios para el envío de mensajes (front-end)
12) Creación de sistema de contactos en el cliente (front-end)

- Criterios de aceptación:
  1) El usuario debe de estar registrado y haber iniciado sesión.
  2) El usuario debe de tener un match con al menos un jugador.
  3) El usuario puede seleccionar al jugador con el que quiere conversar.
  4) El usuario puede visualizar su historial de mensajes con el jugador.
  5) El usuario puede enviar nuevos mensajes al jugador.
  6) El jugador debe recibir los mensajes.

7) Como jugador/a, quiero poder contactar con equipos interesados en buscar jugadores para entrar en su plantilla y formar parte de su equipo.
   - Priorización: Could have
   - Esfuerzo: 25h.
   - Subtareas:
     1) Reestructuración del sistema de mensajería en el servidor para soportar salas grupales (back-end)
     2) Reestructuración del sistema de mensajería para el almacenamiento de los mensajes (database)
     3) Reestructuración del sistema de mensajería en el cliente para distinción entre chats de jugadores y de equipos (front-end)
     4) Reestructuración del sistema de contactos en el cliente (front-end)
   - Criterios de aceptación:
     1) El usuario debe de estar registrado y haber iniciado sesión.
     2) El usuario debe de tener un match con al menos un equipo.
     3) El usuario puede seleccionar al equipo con el que quiere conversar.
     4) El usuario puede visualizar su historial de mensajes con el equipo.
     5) El usuario puede enviar nuevos mensajes al equipo.
     6) Todos los integrantes del equipo deben recibir los mensajes.

8) Como jugador/a, quiero poder buscar los equipos según mis preferencias para buscar solo aquellos en los que estoy interesado/a

- Priorización: Should have
- Esfuerzo: 18h.
- Subtareas:
    1) Gestión de las preferencias entre jugadores y equipos en el servidor (back-end)
    2) Creación de funcionalidad para obtener equipos respecto a las restricciones impuestas (back-end)
    3) Creación de búsquedas personalizadas para obtener equipos respecto a las restricciones impuestas (database)
    4) Creación de funcionalidad para modificar preferencias en el servidor (back-end)
    5) Creación de la interfaz mediante formularios para visualizar y modificar las preferencias en el cliente (front-end)
- Criterios de aceptación:
    1) Las preferencias del jugador deben representar los intereses del jugador.
    2) El jugador puede cambiar su preferencia en cualquier momento.
    3) El sistema debe de encontrar jugadores con el mismo juego que el usuario.
    4) El sistema debe de encontrar jugadores con un rol diferente que el usuario.
    5) El sistema debe de encontrar jugadores con un rango similar al del usuario.
    6) El usuario visualizará solo jugadores que cumplan las preferencias establecidas en su perfil, en caso de no haber ninguno disponible, se le avisará al usuario.

9) Como jugador/a, quiero poder buscar jugadores en lugar de equipos para buscar compañeros para jugar.
    - Priorización: Should have
    - Esfuerzo: 15h.
    - Subtareas:
        1) Creación de funcionalidad para obtener jugadores para un jugador, respecto a las restricciones impuestas (back-end)
        2) Creación de búsquedas personalizadas para obtener jugadores respecto a las restricciones impuestas (database)
        3) Creación de funcionalidad para modificar preferencias de búsqueda en el cliente (front-end)
        4) Creación de la interfaz mediante formularios para visualizar y modificar las preferencias de búsqueda en el cliente (front-end)
    - Criterios de aceptación:
        1) Las preferencias del jugador deben representar los intereses del jugador.
        2) El jugador puede cambiar su preferencia en cualquier momento.

3) El sistema debe de encontrar equipos con el mismo juego que el usuario.
4) El sistema debe de encontrar equipos cuyos roles disponibles contengan el rol del usuario.
5) El sistema debe de encontrar equipos con un rango similar al del usuario.
6) El usuario visualizará solo equipos que cumplan las preferencias establecidas en su perfil, en caso de no haber ninguno disponible, se le avisará al usuario.

10) Como jugador/a, quiero poder crear un equipo para formar una plantilla desde cero.
- Priorización: Should have
- Esfuerzo: 12h.
- Subtareas:
  1) Creación de entidades para la transferencia de datos (DTO)
  2) Creación de entidad para almacenar en la base de datos (database)
  3) Creación de operación para añadir equipo en el servidor (back-end)
  4) Gestión de errores para evitar equipos repetidos en el servidor (back-end)
  5) Creación de interfaz de usuario en el cliente (front-end)
  6) Sistema de validación de formularios síncrona para evitar errores en el cliente (front-end)
  7) Sistema de validación de formularios asíncrona para evitar restricciones de valores repetidos (front-end)
- Criterios de aceptación:
  1) El usuario debe de estar registrado y haber iniciado sesión.
  2) El usuario no puede estar en ningún equipo para crear uno.
  3) El nombre del equipo debe de contener entre 5 y 16 caracteres, y solo puede contener números y letras.
  4) El sistema actualiza sus preferencias para buscar jugadores.

11) Como jugador/a, quiero comunicarme con mis compañeros/as de equipo.
- Priorización: Could have
- Esfuerzo: 25h.
- Subtareas:
  1) Reestructuración del sistema de mensajería en el servidor para soportar salas grupales (back-end)
  2) Reestructuración del sistema de mensajería para el almacenamiento de los mensajes (database)
  3) Reestructuración del sistema de mensajería en el cliente para distinción entre chats de jugadores y de equipos (front-end)

4) Reestructuración del sistema de contactos en el cliente (front-end)
5) Creación de sistema de invitación para formar parte del equipo (back-end)

- Criterios de aceptación:
  1) El usuario debe de estar registrado y haber iniciado sesión.
  2) El usuario debe de estar en un equipo.
  3) El usuario puede seleccionar para comunicarse con su equipo.
  4) El usuario puede visualizar su historial de mensajes con el equipo.
  5) El usuario puede enviar nuevos mensajes al equipo.
  6) Todos los integrantes del equipo deben recibir los mensajes.

12) Como jugador/a, quiero poder dejar un equipo para buscar otro que me interese más.
- Priorización: Should have
- Esfuerzo: 12h.
- Subtareas:
  1) Creación de la funcionalidad para eliminar un jugador de un equipo (back-end)
  2) Implementación de la funcionalidad para abandonar un equipo en el cliente (front-end)
  3) Implementación de la interfaz para abandonar un equipo en el cliente (front-end)
- Criterios de aceptación:
  1) El usuario debe de estar en un equipo.

13) Como miembro de un equipo, quiero poder contactar con jugadores interesados en buscar equipo para entrar en la plantilla.
- Priorización: Could have
- Esfuerzo: 25h.
- Subtareas:
  1) Reestructuración del sistema de mensajería en el servidor para soportar salas grupales (back-end)
  2) Reestructuración del sistema de mensajería para el almacenamiento de los mensajes (database)
  3) Reestructuración del sistema de mensajería en el cliente para distinción entre chats de jugadores y de equipos (front-end)
  4) Reestructuración del sistema de contactos en el cliente (front-end)
  5) Creación de sistema de invitación para formar parte del equipo (back-end)
  6) Creación de la interfaz de invitación para formar parte del equipo (front-end)

- Criterios de aceptación:
    1) El usuario debe de estar registrado y haber iniciado sesión.
    2) El usuario debe de estar en un equipo.
    3) El usuario debe de tener un match con al menos un jugador.
    4) El usuario puede seleccionar al jugador con el que quiere conversar.
    5) El usuario puede visualizar su historial de mensajes con el jugador.
    6) El usuario puede enviar nuevos mensajes al jugador.
    7) El jugador debe recibir los mensajes.

14) Como miembro de un equipo, quiero poder filtrar a los jugadores según el juego al que pertenecen para buscar solo aquellos en los que estoy interesado.
    - Priorización: Should have
    - Esfuerzo: 12h.
    - Subtareas:
        1) Gestión de las preferencias entre jugadores y equipos en el servidor (back-end)
        2) Creación de funcionalidad para obtener equipos respecto a las restricciones impuestas (back-end)
        3) Creación de búsquedas personalizadas para obtener equipos respecto a las restricciones impuestas (database)
        4) Creación de funcionalidad para modificar preferencias en el servidor (back-end)
        5) Creación de la interfaz mediante formularios para visualizar y modificar las preferencias en el cliente (front-end)
    - Criterios de aceptación:
        1) El usuario debe de estar registrado y haber iniciado sesión.
        2) El usuario debe de estar en un equipo.
        3) Las preferencias del equipo deben representar los intereses del equipo.
        4) El sistema debe de encontrar jugadores con el mismo juego que el equipo.
        5) El sistema debe de encontrar jugadores con un rol contenido en los roles disponibles del equipo.
        6) El sistema debe de encontrar jugadores con un rango similar al del equipo.
        7) El usuario visualizará solo jugadores que cumplan las preferencias establecidas por su equipo, en caso de no haber ninguno disponible, se le avisará al usuario.

15) Como fundador de un equipo, quiero buscar un emparejamiento contra otro equipo para practicar un enfrentamiento entre equipos.

- Priorización: Won’t have
- Esfuerzo: 20h.
- Subtareas:
    1) Creación de entidades necesarias para los enfrentamientos
    2) Creación de las entidades necesarias para la transferencia de datos (DTO)
    3) Creación de sistema para gestionar enfrentamientos entre equipos en el servidor (back-end)
    4) Creación de funcionalidad para buscar enfrentamientos en el cliente (front-end)
    5) Creación de la interfaz de usuario para la búsqueda de enfrentamientos en el cliente (front-end)
    6) Creación de interfaz de usuario para la aceptación de enfrentamientos entre equipos en el cliente (front-end)
    7) Almacenamiento de resultados de los enfrentamientos en la base de datos (database)
    8) Creación de historial de enfrentamientos en el servidor (back-end)
    9) Creación de interfaz de enfrentamientos en el cliente (front-end)
- Criterios de aceptación:
    1) El usuario debe de estar registrado y haber iniciado sesión.
    2) El usuario debe de estar en un equipo.
    3) El usuario debe ser fundador del equipo.
    4) El usuario debe de poder seleccionar de una lista de equipos similares un contrincante.
    5) El sistema debe seleccionar equipos con el mismo juego.
    6) El sistema debe seleccionar equipos con un rango similar.
    7) El equipo seleccionado recibirá una invitación de enfrentamiento.
    8) Si el equipo seleccionado acepta la invitación, ambos equipos recibirán la información necesaria para realizar el enfrentamiento.

16) Como fundador de un equipo, quiero entrar en un torneo para competir contra otros equipos.
    - Priorización: Won’t have
    - Esfuerzo: 25h.
    - Subtareas:
        1) Creación de las entidades necesarias para los torneos, formados por diversos enfrentamientos.
        2) Creación de las entidades necesarias para la transferencia de datos (DTO)
        3) Creación del sistema para gestionar torneos en el servidor (back-end)

4) Creación de la funcionalidad para entrar o crear torneos en el cliente (front-end)
5) Creación de la interfaz para visualizar, entrar, crear torneos en el cliente (front-end)
6) Almacenamiento de la información relacionada con los torneos en la base de datos (database)

- Criterios de aceptación:
    1) El usuario debe de estar registrado y haber iniciado sesión.
    2) El usuario debe de estar en un equipo.
    3) El usuario debe ser fundador del equipo.
    4) El usuario debe de poder seleccionar de una lista de torneos disponibles.
    5) El sistema debe seleccionar torneos con el mismo juego.
    6) El sistema debe seleccionar torneos con un rango similar.
    7) El torneo debe de indicar la fecha y hora de cada enfrentamiento
    8) El equipo recibirá una notificación de enfrentamiento junto a los datos del contrincante.
    9) Ambos equipos notificarán el resultado de la partida.

17) Como fundador de un equipo, quiero poder agregar jugadores al equipo para completar la plantilla.

- Priorización: Could have
- Esfuerzo: 20h.
- Subtareas:
    1) Creación de entidades necesarias para la gestión de invitaciones
    2) Creación de entidades necesarias para la transferencia de datos (DTO)
    3) Creación de sistema de invitación para formar parte del equipo (back-end)
    4) Creación de la interfaz de invitación para formar parte del equipo (front-end)
    5) Almacenamiento de las invitaciones en la base de datos (database)
    6) Gestión de invitaciones, control para evitar invitaciones repetidas en el servidor (back-end)
    7) Creación de la interfaz en el cliente para obtener listado de invitaciones en el cliente (front-end)
- Criterios de aceptación:
    1) El usuario debe de estar registrado y haber iniciado sesión.
    2) El usuario debe de estar en un equipo.
    3) El usuario debe ser fundador del equipo.
    4) El usuario debe de tener un match con al menos un jugador.

5) El usuario puede seleccionar al jugador con el que quiere conversar.
6) El usuario puede visualizar su historial de mensajes con el jugador.
7) El usuario puede enviar nuevos mensajes al jugador.
8) El jugador debe recibir los mensajes.
9) El usuario puede enviar una invitación al otro jugador para entrar en su equipo, en caso de que el equipo esté lleno, esta opción no estará activa.
10) El jugador puede aceptar o rechazar la invitación, en caso de que acepte, formará parte del equipo.

18) Como fundador de un equipo, quiero poder eliminar miembros del equipo para poder añadir nuevos miembros.
    - Priorización: Should have
    - Esfuerzo: 12h.
    - Subtareas:
        1) Creación de la funcionalidad para eliminar un jugador de un equipo (back-end)
        2) Implementación de la funcionalidad para eliminar un jugador de un equipo en el cliente (front-end)
        3) Implementación de la interfaz para eliminar un jugador de un equipo en el cliente (front-end)
    - Criterios de aceptación:
        1) El usuario debe de estar registrado y haber iniciado sesión.
        2) El usuario debe de estar en un equipo.
        3) El usuario debe ser fundador del equipo.
        4) El usuario debe de ver la lista de miembros del equipo.
        5) El miembro del equipo eliminado dejará de tener equipo
        6) El usuario no puede expulsarse a sí mismo.

19) Como fundador de un equipo, quiero poder eliminar el equipo para crear uno desde cero, o dejar de estar en un equipo.
    - Priorización: Should have
    - Esfuerzo: 12h.
    - Subtareas:
        1) Creación de la funcionalidad para eliminar un equipo (back-end)
        2) Implementación de la funcionalidad para eliminar un equipo en el cliente (front-end)
        3) Implementación de la interfaz para eliminar un equipo en el cliente (front-end)
    - Criterios de aceptación:
        1) El usuario debe de estar registrado y haber iniciado sesión.
        2) El usuario debe de estar en un equipo.

3) El usuario debe ser fundador del equipo.
4) Todos los miembros del equipo, incluido el fundador, dejarán de tener equipo.

Esta comprendería la lista completa de historias de usuario que tienen como objetivo ser implementadas en la aplicación final, para proporcionar la mayor cantidad posible de funcionalidades al usuario y facilidades de uso en la aplicación. Sin embargo, no todas las historias de usuario son igual de prioritarias a la hora de la implementación, por lo que se ha determinado una priorización de cada una de ellas, indicándose en la pila de producto.

2.4. Planificación inicial de tareas

Nuestra descomposición previa del proyecto en historias de usuario nos dejó con un total de 19 historias de usuario, las cuales suman un total de 342 horas de trabajo para completar el desarrollo del producto.

Para organizar el avance del proyecto a lo largo del tiempo, dividiremos la carga de trabajo total en 5 iteraciones, de unas 70 horas aproximadamente. A continuación definiremos los objetivos para cumplir en cada iteración.

Cada una de las iteraciones llevará asociado un número de puntos de historia, que calcularemos a partir de la estimación obtenida del sistema *MoSCoW* y la estimación de tiempo de la iteración. Este número refleja de forma numérica la cantidad de valor de producto que estamos añadiendo con cada historia de usuario.

Los puntos de historia por cada iteración se calculará a partir de la suma de los números de historia de cada una de las historias de usuario, donde multiplicamos el número de horas por un valor fijo asociado a la priorización de la historia de usuario.

Must have	Should have	Could have	Won’t have
1.3	1.2	1.1	1

Tabla 1. Multiplicadores de los puntos de usuario dependiendo de la prioridad.

Como se puede apreciar en la tabla 1, dependiendo de la prioridad de la historia de usuario, los puntos de historia serán mayores o menores.

*Ejemplo: Para una historia de usuario con una estimación de 25 horas y priorizada con “Must have”, el número de puntos de historia será de 25 * 1.3, es decir, 32.5 puntos de historia.*

2.4.1. Propuesta inicial de iteraciones

2.4.1.1. Iteración 1

En la primera iteración, el objetivo principal es establecer el esqueleto básico de una aplicación CRUD[6], que contenga las funcionalidades básicas de registro, marcado por la historia de usuario 1, crear un sistema de inicio de sesión, explicado en la historia de usuario 2, además de permitir borrar y modificar el usuario como se menciona en las historias de usuario 3 y 4.

Esta iteración tiene una duración estimada de 69 horas y una estimación de valor de 87.3 puntos de historia. Al ser la primera iteración, se requerirá un esfuerzo mayor, puesto que no partimos de ninguna base. Además, existirán varios roles a cubrir en el desarrollo de la aplicación, que mencionaremos a continuación junto a las tareas asociadas a cada rol.

- Analista: Su función será fundamentalmente la gestión de la documentación técnica, además de coordinar el trabajo de diseñadores y programadores. La duración estimada de su trabajo será de 6 horas.
- Diseñador de software: Su trabajo consistirá en determinar la estructura básica y arquitectura del sistema. Además se encargará de la seguridad del sistema, cuyas tareas son la 2.4, 2.6, 2.8, 2.9, 3.2 y 4.2. La duración estimada de su trabajo será de 13 horas.
- Diseñador de interfaz: Se encargará del diseño de interfaz gráfica, encargándose de las tareas 1.5, 2,3, 3.3, 3.4 y 4.3. La duración estimada de su trabajo será de 13 horas.
- Programador de back-end: Se encargará de la implementación en el lado del servidor, cuyas tareas asociadas son la 1.1, 1.2, 1.3, 1.4, 2.1, 2.2, 3.1, 3.5, 4.1 y 4.5. La duración estimada de su trabajo será de 18 horas.
- Programador de front-end: Se encargará de la implementación en el lado del cliente, encargado de las tareas 1.5, 1.6, 1.7, 2.3, 2.5, 2.7 y 4.4. La duración estimada de su trabajo será de 18 horas.

2.4.1.2. Iteración 2

En la segunda iteración, partimos de una base sólida, por lo que el objetivo será crear las funcionalidades básicas del sistema, como buscar jugadores y equipos en función de las preferencias del usuario, objetivos incluidos en las historias de usuario 8, 9 y 14. Además, se incluirá la funcionalidad para que un jugador pueda crear un nuevo equipo, como indica la historia de usuario 10.

[6] En informática, CRUD es el acrónimo de “Crear, Leer, Actualizar y Borrar”.

Esta segunda iteración tiene una duración estimada de 57 horas, junto a una estimación de valor de 68.4 puntos de historia. Los roles necesarios para completar la iteración seguirán siendo iguales que la primera iteración.

- Analista: Su función será fundamentalmente la gestión de la documentación técnica, además de coordinar el trabajo de diseñadores y programadores. La duración estimada de su trabajo será de 6 horas.
- Diseñador de software: Su trabajo consistirá en organizar la correcta gestión de la base de datos, además de ayudar en tareas de gestión al programador de back-end. Las tareas asociadas son la 8.3, 9.2, 10.2, 14.1 y 14.3. La duración estimada de su trabajo será de 11 horas.
- Diseñador de interfaz: Se encargará del diseño de interfaz gráfica, encargándose de las tareas 9.4, 10.5 y 14.5. La duración estimada de su trabajo será de 11 horas.
- Programador de back-end: Se encargará de la implementación en el lado del servidor, cuyas tareas asociadas son la 8.1, 8.2, 8.4, 9.1, 10.1, 10.3, 10.4, 14.2 y 14.4. La duración estimada de su trabajo será de 15 horas.
- Programador de front-end: Se encargará de la implementación en el lado del cliente, encargado de las tareas 8.5, 9.3, 10.6, 10.7 y 14.5. La duración estimada de su trabajo será de 15 horas.

2.4.1.3. Iteración 3

En esta tercera iteración, se pretende cubrir las funcionalidades principales del sistema, como es el sistema de dar likes, explicado en la historia de usuario 5, y para complementar la funcionalidad, se implementará el sistema de mensajería entre jugadores, como se indica en la historia de usuario 6. Por último, se avanzará en el apartado de los equipos, incluyendo funcionalidades de borrado para el fundador del equipo, permitiendo eliminar tanto miembros como el propio equipo, como se indican en las historias de usuario 18 y 19.

Al finalizar esta iteración, el sistema habría abarcado la misión principal del sistema como se indica en el análisis, puesto que los jugadores ya tendrían disponibilidad para encontrar otros jugadores con quien jugar e incluso contactar con ellos en caso de match.

Esta iteración tiene una duración estimada de 64 horas, con un total de 80.8 puntos de historia. Los roles necesarios para completar la iteración seguirán siendo iguales que la primera iteración.

- Analista: Su función será fundamentalmente la gestión de la documentación técnica, además de coordinar el trabajo de diseñadores y programadores. La duración estimada de su trabajo será de 6 horas.

- Diseñador de software: Su trabajo consistirá en organizar la correcta gestión de la base de datos, además de ayudar en tareas de gestión al programador de back-end. Las tareas asociadas son la 5.1, 5.2, 5.4, 6.1, 6.2 y 6.7. La duración estimada de su trabajo será de 12 horas.
- Diseñador de interfaz: Se encargará del diseño de interfaz gráfica, encargándose de las tareas 5.6, 5.9, 18.2 y 19.3. La duración estimada de su trabajo será de 12 horas.
- Programador de back-end: Se encargará de la implementación en el lado del servidor, cuyas tareas asociadas son la 5.3, 6.3, 6.4, 6.5, 6.6, 18.1, y 19.1. La duración estimada de su trabajo será de 17 horas.
- Programador de front-end: Se encargará de la implementación en el lado del cliente, encargado de las tareas 5.5, 6.8, 6.10, 6.11, 6.12, 18.3 y 19.2. La duración estimada de su trabajo será de 17 horas.

2.4.1.4. Iteración 4

En esta cuarta iteración, el objetivo es expandir las funcionalidades creadas para los equipos, permitiendo el contacto entre jugadores y equipos, como indican la historias de usuario 7 y 13, además de incluir la posibilidad de incluir miembros al equipo mediante un sistema de invitaciones, gestionado por el fundador del equipo, como se indica en la historia de usuario 17. Además, los jugadores podrán dejar el equipo como indica la historia de usuario 12.

Esta iteración tiene una duración estimada de 82 horas, con un total de 90.2 puntos de historia. Los roles necesarios para completar la iteración seguirán siendo iguales que la primera iteración.

- Analista: Su función será fundamentalmente la gestión de la documentación técnica, además de coordinar el trabajo de diseñadores y programadores. La duración estimada de su trabajo será de 6 horas.
- Diseñador de software: Su trabajo consistirá en organizar la correcta gestión de la base de datos, además de ayudar en tareas de gestión al programador de back-end. Las tareas asociadas son la 7.1, 7.2, 13.1, 13.2, 17.1, 17.2 y 17.6. La duración estimada de su trabajo será de 20 horas.
- Diseñador de interfaz: Se encargará del diseño de interfaz gráfica, encargándose de las tareas 7.4, 12.3, 13.3, 17.4 y 17,7. La duración estimada de su trabajo será de 17 horas.
- Programador de back-end: Se encargará de la implementación en el lado del servidor, cuyas tareas asociadas son la 7.1, 7.2, 12.1, 13.1, 13.5, 17.3 y 17.5. La duración estimada de su trabajo será de 20 horas.
- Programador de front-end: Se encargará de la implementación en el lado del cliente, encargado de las tareas 7.3, 7.4, 12.2, 13.3, 13.4, 13.4, 13.6 y 17.7. La duración estimada de su trabajo será de 19 horas.

2.4.1.5. Iteración 5

En la última iteración, partimos de un sistema en un estado muy maduro y completo, y el principal objetivo es continuar puliendo el sistema, añadiendo un chat grupal de equipo, como se explica en la historia de usuario 11, además de incluir funcionalidades relacionadas con partidos amistosos y torneos organizados en la aplicación, mencionados en las historias de usuario 15 y 16.

Esta iteración tiene una duración estimada de 70 horas, con un total de 72.5 puntos de historia. Los roles necesarios para completar la iteración seguirán siendo iguales que la primera iteración.

- Analista: Su función será fundamentalmente la gestión de la documentación técnica, además de coordinar el trabajo de diseñadores y programadores. La duración estimada de su trabajo será de 6 horas.
- Diseñador de software: Su trabajo consistirá en organizar la correcta gestión de la base de datos, además de ayudar en tareas de gestión al programador de back-end. Las tareas asociadas son la 11.1, 11.2, 15.1, 15.2, 15.7, 16.1 y 16.6. La duración estimada de su trabajo será de 17 horas.
- Diseñador de interfaz: Se encargará del diseño de interfaz gráfica, encargándose de las tareas 11,4, 15.5, 15.6, 15.9 y 16.5. La duración estimada de su trabajo será de 14 horas.
- Programador de back-end: Se encargará de la implementación en el lado del servidor, cuyas tareas asociadas son la 11.1, 11.2, 11.5, 15.3, 15.8, 16.2 y 16.3. La duración estimada de su trabajo será de 18 horas.
- Programador de front-end: Se encargará de la implementación en el lado del cliente, encargado de las tareas 11.3, 11.4, 15.4, 15.9 y 16.4. La duración estimada de su trabajo será de 15 horas.

2.4.2. Evolución de la planificación inicial

Una vez hemos detallado y definido los objetivos y estimaciones de cada una de las 5 iteraciones que componen el proyecto, procederemos a mostrar de forma gráfica la planificación inicial del proyecto mediante una gráfica *burn down*.

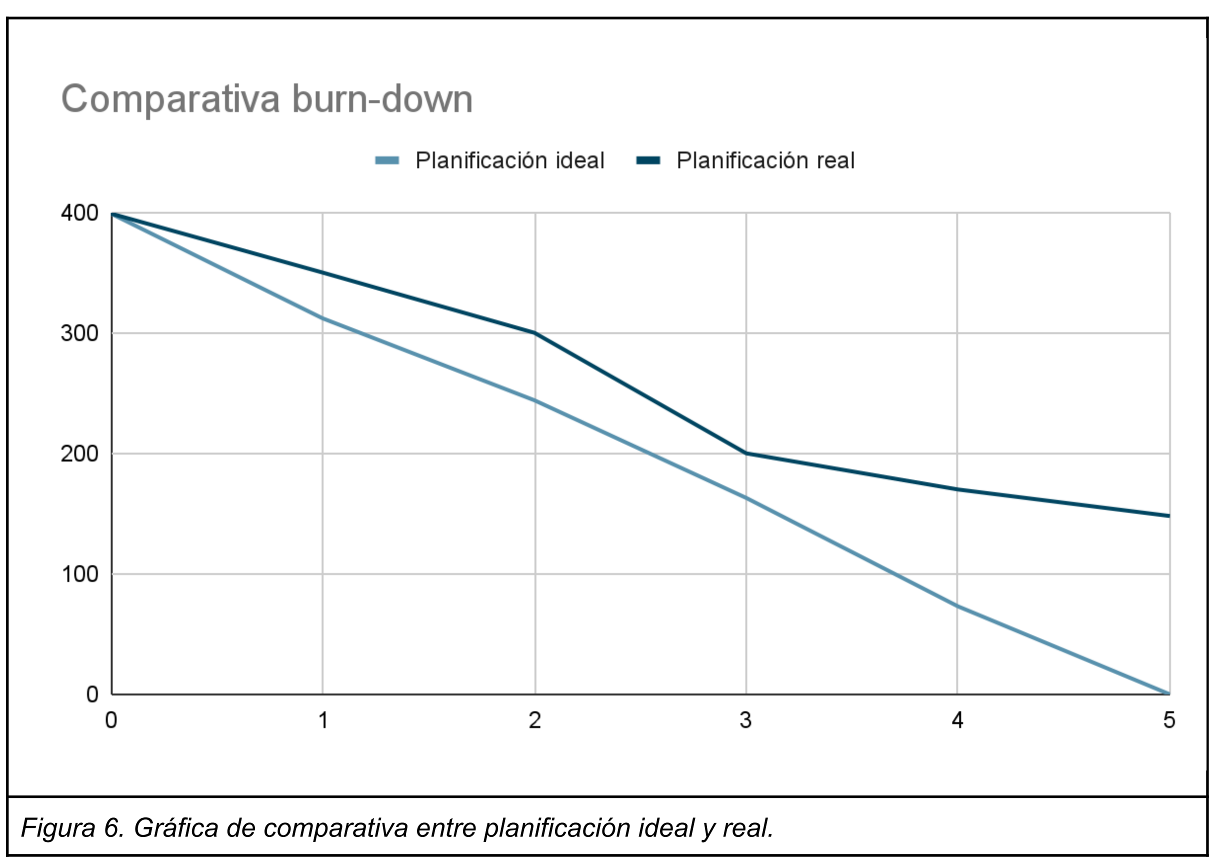

Figura 6. Gráfica de comparativa entre planificación ideal y real.

Como podemos observar en la comparativa entre la planificación ideal y la real (Figura 6), al finalizar el primer sprint o iteración el proyecto se vió retrasado. Este retraso se debe a una subestimación del tiempo que llevaría finalizar las historias de usuario del primer sprint.

Esta subestimación provocó que la finalización de las historias previstas para la primera iteración ocurriese al terminar el segundo sprint. Una vez acabado el segundo sprint, el proyecto se vio acelerado, obteniendo las historias de usuario deseadas para el segundo sprint y parte de las historias previstas para el tercer sprint.

Por último, el avance final del proyecto se vió gravemente afectado nuevamente por las historias de usuario previstas para la tercera iteración. Sobre todo la historia de usuario 6, relacionada con el envío de mensajes a jugadores, creó un cuello de botella que duraría hasta la finalización del quinto y último sprint. Como consecuencia de esto, las historias de usuario previstas para el cuarto y quinto sprint se quedaron en la pila de producto, sin ser implementadas. Sin embargo,

gracias a la priorización las historias de usuario fueron descartadas aquellas que tenían una importancia mínima.

El cumplimiento de las historias de usuario esenciales fue satisfactorio, ya que, como hemos mencionado en la tercera iteración, las historias de usuario relacionadas con los jugadores quedarían completamente finalizadas. Por el contrario, la visión de ampliar el ámbito del proyecto a la creación de equipos, permitiendo así no solo encontrar compañeros de juego, si no también encontrar jugadores ambiciosos con los que compartir metas, se vió obligada a quedarse en la pila de producto.

2.5. Estudio de viabilidad

El estudio de la viabilidad es la medida de beneficio de una organización obtenido gracias a la explotación de un sistema informático. Analizar la viabilidad de un proyecto puede servirnos como guía para realizar de forma correcta la toma de decisiones antes de comenzar la etapa de planificación.

Esta estimación nos aproxima si las necesidades del usuario detectadas en la fase de análisis son posibles de satisfacer con las tecnologías actuales, tanto en software como en hardware.

A continuación desglosamos los costes del proyecto según su categoría (*software*, *hardware*, nóminas). También tendremos en cuenta que el desarrollo del proyecto software tendrá lugar en 4 meses, sin embargo, para una estimación correcta donde todas las historias de usuario sean completas, realizaremos un segundo estudio de viabilidad con un periodo de desarrollo de 1 año. Finalmente realizaremos una comparativa entre ambas estimaciones.

2.5.1. Estimación de costes software

En este apartado abordaremos el *software* utilizado, incluyendo licencias. Sin embargo, la mayoría del software utilizado es gratuito, lo que nos reducirá los gastos.

Como hemos comentado anteriormente, definiremos dos estimaciones de coste, siendo el coste del estado actual del proyecto como un desarrollo de 4 meses, y la estimación del proyecto completado, en 1 año.

Software	Precio por unidad de tiempo	Precio total (4 meses)	Precio total (1 año)
Visual Studio Code	0€	0€	0€
Figma (Free version)	0€	0€	0€
Postman	0€	0€	0€
IntelliJ IDEA Ultimate	499€ / año	166,3€	499€
Google Chrome	0€	0€	0€
Google Documents	0€	0€	0€
GitHub	0€	0€	0€
Visual Paradigm 16.3	0€	0€	0€
Total		166,3€	499€

Tabla 2. Tabla de costes software.

Como podemos observar en la Tabla 2, cabe destacar que el software *IntelliJ IDEA Ultimate*, en el caso del desarrollo, el coste fue gratuito puesto que la Universidad de Jaén proporciona licencias de uso gratuitas para estudiantes.

2.5.2. Estimación de costes hardware

Para el cálculo de los costes *hardware*, hemos utilizado como parámetro el costo, dividido por su tiempo de vida útil, de esta manera podemos obtener el precio del producto dado el tiempo de uso que le vamos a someter durante el desarrollo del proyecto.

Hardware	Precio por unidad de tiempo	Precio total (4 meses)	Precio total (1 año)
PC Custom	1522€ / 20 años	25.3€	76€
Monitor BenQ GW2780	168.99€ / 15 años	3.6€	11€
Teclado Logi G415	79.99€ / 7 años	3.83€	11.5€
Raton Logi G502 Hero	66.99€ / 5 años	4.46€	13.4€
Conexión a internet	30€ / mes[7]	120€	360€
Total		157.19€	471.9€

Tabla 3. Tabla de costes hardware.

Como se observa en la tabla 3 de costes hardware, hemos incluido la conexión a internet como un gasto hardware, ya que es un elemento indispensable para la estimación de costes.

[7] El precio de conexión a internet fue obtenido como ejemplo de la compañía Jazztel, cuya oferta es de 29.95€ / mes (Marzo de 2022).

2.5.3. Estimación de costes humanos

En este apartado seleccionamos el número de personal necesario para el desarrollo del producto, los asalariados que necesitaremos para el desarrollo del mismo son:

- Un analista *(Fuente: https://es.indeed.com/career/analista-programador/salaries)*
- Un diseñador software *(Fuente: https://es.indeed.com/career/ingeniero-de-software/salaries)*
- Un diseñador de interfaces *(Fuente: https://es.indeed.com/career/dise%C3%B1ador-web/salaries)*
- Un programador de back-end *(Fuente: https://es.indeed.com/career/desarrollador-java/salaries)*
- Por último un programador de front-end *(Fuente: https://es.indeed.com/career/programador-front-end/salaries)*

Cargo	Salario / hora	Nº total de horas (4 meses)	Coste total (4 meses)	Coste total (1 año)
Analista	13.94€	30	418.2€	836.4€
Diseñador software	18.09€	73	1320.57€	2641.14€
Diseñador UI	9.79€	67	655.93€	1311.86€
Programador de back-end	14.89€	88	1310.32€	2620.64€
Programador de front-end	16.82€	84	1412.88€	2825.76€
Total			5117.9€	10235.8€

Tabla 4. Tabla de costes humanos.

Como podemos ver en la tabla 3, hemos calculado los costes humanos, por lo que ahora tenemos todos los datos necesarios para calcular el coste total del proyecto.

2.5.4. Estimación de coste total

Como hemos mencionado anteriormente, el coste total será la suma de los costes software, hardware y asalariados. Calcularemos el proyecto dado 4 meses de trabajo, y con 1 año de trabajo, siendo esta última estimación el resultado final para que el proyecto se finalice al completo, de forma que ninguna historia de usuario quedase en la pila de producto.

Tipo de coste	Coste (4 meses)	Coste (1 año)
Software	166.3€	499€
Hardware	157.19€	471.9€
Mano de obra	5117.9€	10235.8€
Total	5441.39€	11206.7€

Tabla 5. Tabla de coste total del proyecto.

Como podemos observar en la tabla 5, el coste total del proyecto asciende a 5441.39€ teniendo en cuenta un desarrollo de 4 meses, y 11206.7€ si el desarrollo se extendiese a 1 año. Sin embargo, estos valores no representan el coste total del proyecto, puesto que esa cifra significa el coste de la producción del mismo, a esta cifra hay que añadir el margen de beneficio que se espera obtener con el proyecto.

Es interesante comparar estas dos estimaciones, puesto que a la hora de asignar un margen de beneficio, debe de estar en consecuencia al costo total. Para el desarrollo de 4 meses, un margen de beneficio del 20%, nos dejaría con un total de 1088.27€ de beneficio.

En el caso del desarrollo de un año, estableceremos el margen de beneficio en un 15%, ya que el costo de producción es bastante mayor. Lo que dejaría un beneficio de 1681€.

2.6. Modelo de dominio

El modelo de dominio en la resolución de problemas e ingeniería del software es un modelo conceptual que aúna los temas relacionados con un problema específico (Wikipedia 2020). Este puede utilizarse para captar y expresar las entidades del sistema como paso previo al diseño de un sistema (EcuRed contributors 2019).

El modelo de dominio se suele aplicar a la programación orientada a objetos, puesto que el funcionamiento del software que creemos imita en cierto sentido a la realidad, tratando de alcanzar una primera versión del sistema. La función del modelo de dominio es facilitarnos un vocabulario y conceptos clave del dominio del problema que vamos a tratar de resolver.

En él vamos a introducir las entidades detectadas, sus relaciones entre sí, roles que desempeñan algunas de las clases, etc.

Las entidades que hemos detectado a partir de las historias de usuario son:

- Usuario: Es la entidad básica de la que derivan tanto jugador como equipo. Un usuario es una generalización de ambas entidades, puesto que ambas pueden dar y recibir likes[8].

- Like: No conforma una entidad como tal, pero podría considerarse una entidad menor, ya que conforma una pieza clave en el dominio. Se encarga de almacenar la información de la entidad que recibe y que da el like.

- Match[9]: Es otra de las entidades menores del dominio, pero que también debería mencionarse, puesto que forma un elemento principal en la solución. Un match se produce cuando dos usuarios se dan like mutuamente, por lo que existe una reciprocidad, y establece la condición necesaria para que se produzca un match. Cuando un match se produce es cuando dos usuarios entran en contacto.

- Jugador: Es un usuario, el cual tiene una connotación específica como jugador, que tendrá unas preferencias para buscar tanto equipos como jugadores, además de tener información personal sobre el perfil que contiene. Un jugador puede pertenecer o no a un equipo. Si tiene un equipo, además un jugador puede ser el fundador si fue él mismo quien lo creó.

- Equipo: Es una agregación de jugadores que, como mínimo tendrá 1 jugador, ya que en caso contrario no existiría el equipo, y como máximo tendrá 5 jugadores, siendo éste el número de jugadores estándar para los juegos competitivos de actualidad. Todo equipo tiene un fundador y unas preferencias de equipo, creadas a partir de las preferencias de los jugadores que lo forman.

- Preferencias: Cada jugador tiene unas preferencias, que podrá modificar en cualquier momento, y que dictaminan el juego preferido del jugador, además de contener el rol que desempeña en dicho juego, su rango y, en caso de ser una jugadora, se le dará la oportunidad de preferir solo jugadoras femeninas.

---

[8] Like es una palabra inglesa que en español puede traducirse como la expresión “me gusta”. A lo largo del documento se referirá a este concepto como like, debido a la familiaridad con el término entre usuarios habituales en redes sociales.
[9] Match es una palabra inglesa que en español puede traducirse como “pareja”, de igual forma que con el término like, es habitual entre usuarios de redes sociales y aplicaciones de citas online.

- Preferencias de equipo: Cada equipo contiene unas preferencias de equipo, que se obtienen a partir de las preferencias de los miembros del mismo. Dichas preferencias contendrán el juego, la lista de roles que ocupan los miembros del equipo, y el rango promedio del equipo. Estas preferencias ayudarán a indicar qué jugadores son aptos para el equipo.

- Juego: Dentro de las preferencias, cada jugador podrá escoger el juego que más le guste. Un juego agrega una serie de roles y rangos específicos para cada juego.

- Rol: Un rol es la representación de la tarea que desempeña un cierto jugador en un juego. En varios juegos, solo un jugador puede desempeñar un rol, aunque no es la norma general.

- Rango: El rango es un identificador asociado a un juego que dictamina el nivel de conocimiento y desempeño de un jugador en el juego Cuanto más rango tenga un jugador, más experiencia y conocimiento tendrá sobre el juego. Los rangos ayudan a balancear el juego, ya que un jugador querrá enfrentarse a jugadores de su nivel para que sea una partida justa.

- Partida: Son parejas de equipos completos y se podrán repetir en caso de querer revancha. Estos equipos disputarán una partida y almacenarán el resultado de la misma.

- Torneo: Es una agregación de partidas, ya que un torneo es una serie de partidas donde los equipos se distribuyen en las partidas según el tipo de torneo (liga, eliminación directa, doble eliminación, etc).

- Mensaje: Esta entidad almacena un mensaje de texto que envía un usuario a otro usuario. Es utilizado para cuando dos usuarios están en contacto y son medio de comunicación entre ambos dentro del dominio.

- Sala: Un usuario puede estar en contacto con más de un usuario, por lo que deben existir diversas conversaciones para cada uno de los usuarios que tengan varios contactos.

A continuación mostraremos el diagrama del modelo del dominio, junto a las relaciones con sus respectivas cardinalidades, además de dejar algunas notas explicando detalles de algunas relaciones que deben de ser remarcadas.

Es un diagrama simple pero contiene los elementos esenciales que trataremos en el problema mencionado anteriormente.

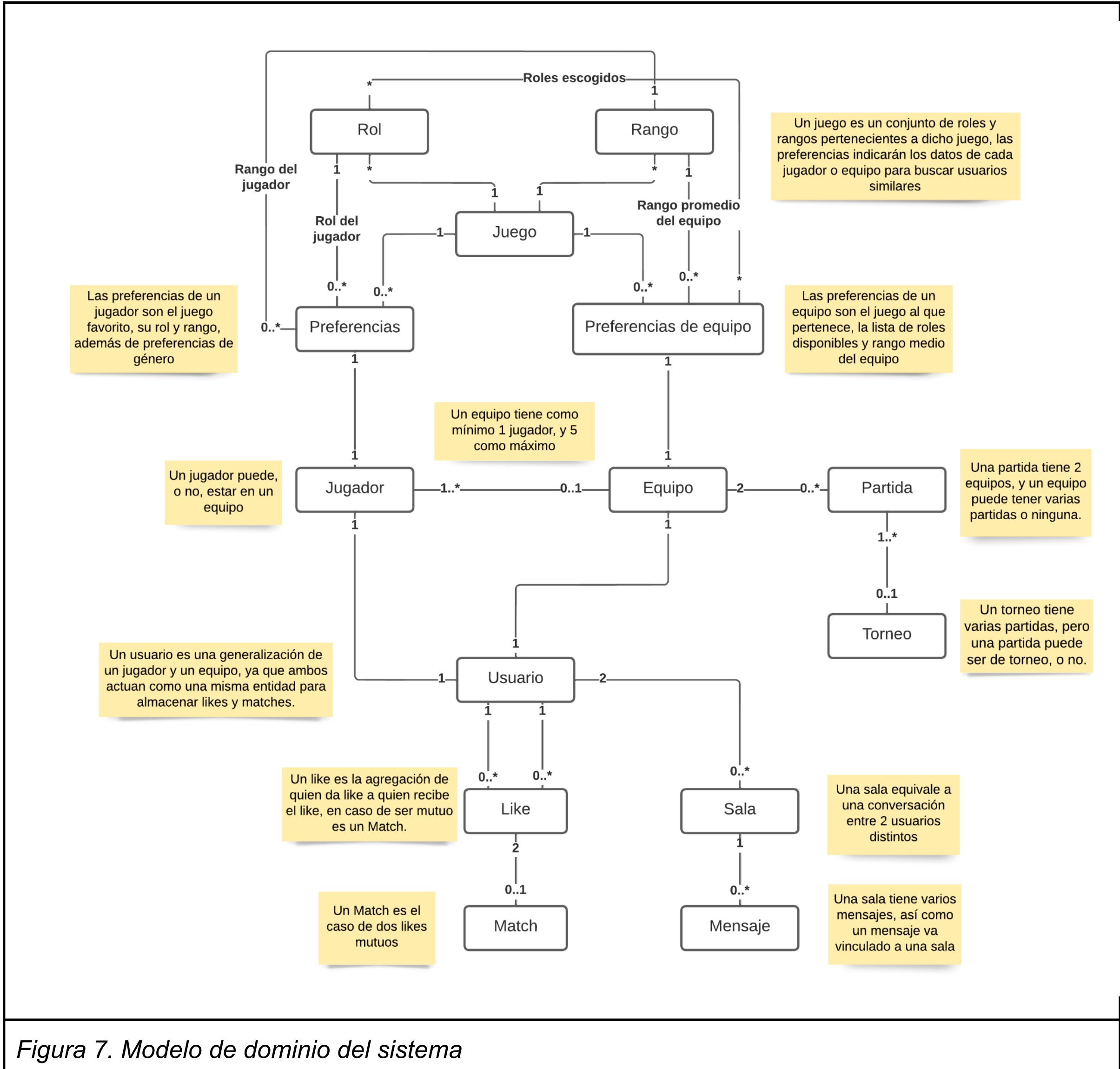

Figura 7. Modelo de dominio del sistema

En la Figura 7 podemos observar algunos detalles como la distinción entre preferencias de equipo y preferencias del jugador, debido principalmente a que un equipo dispone de una lista de roles escogidos, y se buscarán jugadores acorde a los roles disponibles de dicho juego.

También cabe destacar como tanto jugador y equipo, se generalizan en la entidad usuario, que almacena la información relacionada con los likes, los matches, y demás información compartida entre ambas entidades jugador y equipo.

3. Diseño

El diseño es el proceso de aplicar técnicas y principios con el propósito de conseguir un dispositivo, proceso o sistema en suficiente detalle como para permitir su realización física. Si aplicamos este concepto al software, se convierte en aquella actividad en la que se toman decisiones importantes, frecuentemente de tipo estructural o arquitectónico (Gomaa 2011).

Esta definición comparte aspectos con la programación, como la abstracción de la representación de la información o las secuencias de procesamiento. El diseño de software trata de construir representaciones bien definidas de los programas, centrándose en las interrelaciones de los componentes a mayor nivel, y en las operaciones lógicas en niveles inferiores.

En este apartado desarrollaremos diferentes acercamientos al diseño, como son los diagramas de entidad-relación, diagramas de clases y diagramas de secuencia. Para el diseño del sistema, utilizaremos una metodología orientada a objetos, mediante la cual diferentes clases se relacionan entre sí, creando asociaciones de diversos tipos. Representaremos de forma gráfica el modelado del sistema utilizando UML[10], siendo el lenguaje de modelado de sistemas más conocido y utilizado. Este nos permitirá representar de forma visual tanto procesos a bajo nivel en los diagramas de secuencia, como relaciones estructurales en el diagrama de clases (Pooley and Stevens 2008).

3.1. Diagrama entidad-relación

El modelo entidad-relación fue creado por Peter Chen en 1976. Es una herramienta para el modelado de datos, que nos facilita la representación de las entidades en una base de datos relacional (Carte, Jasperson, and Cornelius 2006).

Su base teórica está basada en una percepción del mundo real, que consta de una colección de objetos, llamados entidades, que se relacionan entre sí.

- Entidad: Representación de un objeto del mundo real, con existencia propia, lo que indica que es diferente al resto de objetos, aun siendo del mismo tipo.
- Atributos: Los objetos tienen una serie de características que definen a una entidad. Para cada tipo de entidad, existe un conjunto de atributos asociados, y cada entidad tiene valores específicos asociados para cada uno de sus atributos, de forma que tengan una identidad única.
- Relaciones: Algunos objetos están vinculados entre sí, lo que definen dependencias entre entidades, estas relaciones pueden ser de distintos tipos.

[10] Lenguaje unificado del modelado, o UML por sus siglas en inglés, es un lenguaje gráfico para representar y construir sistemas.

Por lo general estas relaciones se resumen en uno a uno, uno a muchos, muchos a muchos.

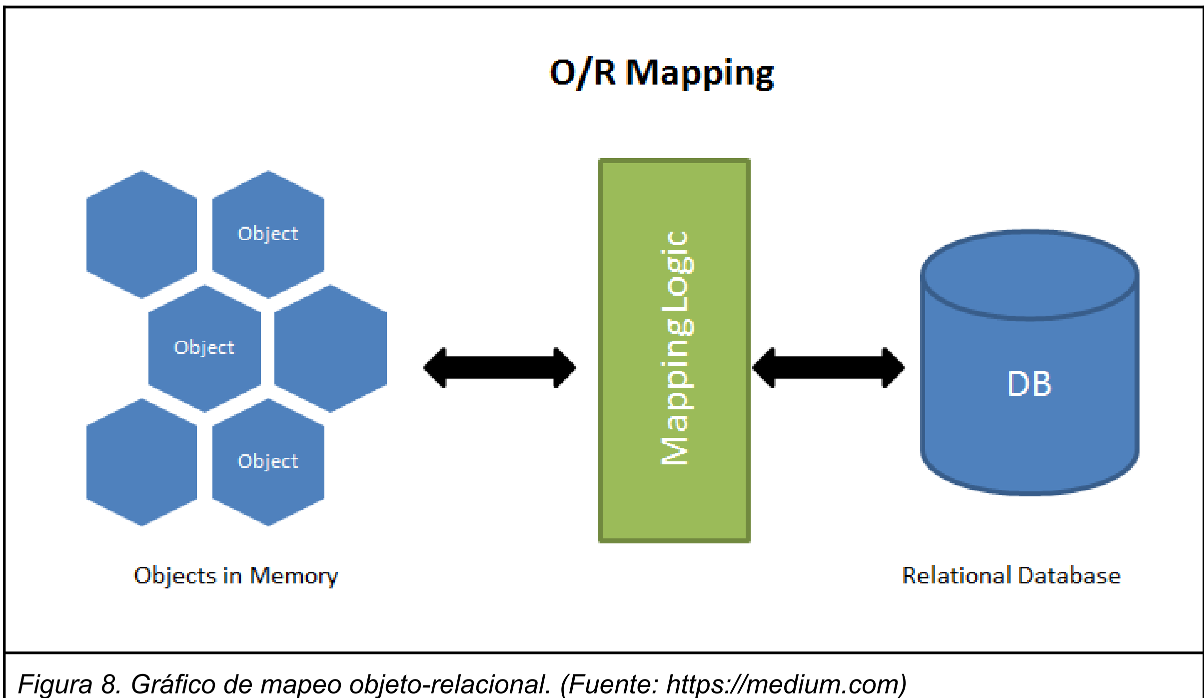

Figura 8. Gráfico de mapeo objeto-relacional. (Fuente: https://medium.com)

En nuestro caso, para la creación de las tablas y relaciones finales, haremos uso de un ORM[11], en la Figura 8 podemos ver un ejemplo gráfico del funcionamiento de un ORM. Concretamente haremos uso de la especificación de JPA[12], que proporciona persistencia en una base de datos relacional a una aplicación orientada a objetos. Para ello, JPA hace uso de unas entidades de persistencia, cuyo estado se almacena de manera asociada a una tabla en una base de datos relacional. Estas instancias corresponden a un registro o fila dentro de una tabla. Para crear los metadatos, JPA hace uso de especificaciones directamente sobre el fichero de la clase, usando anotaciones de Java.

Además JPA nos proporciona el acceso a las tablas sin hacer uso de código SQL[13], puesto que la generación de este es automática por parte de JPA. Sin embargo, para algunas consultas complejas que comentaremos más adelante, también sería posible crear consultas en SQL de forma manual.

Cabe destacar que al utilizar JPA, integrado en Spring, las clases que contengan las anotaciones específicas de JPA, es decir, aquellas clases que requieran de persistencia, se crearán automáticamente en la base de datos. De igual forma, las

[11] Mapeo objeto-relacional, o ORM por sus siglas en inglés, es una técnica de programación que convierte objetos de la base de datos, en objetos del lenguaje de programación utilizado, y viceversa.
[12] Java Persistence API, es una API de persistencia desarrollada para la plataforma Java EE.
[13] Lenguaje de consulta estructurada, o SQL por sus siglas en inglés, lenguaje utilizado para administrar y recuperar información en sistemas de gestión de bases de datos.

relaciones del diagrama entidad-relación se reflejarán automáticamente a partir de anotaciones específicas en los atributos de cada una de las clases.

JPA realiza el proceso de normalización del modelo entidad-relación a tercera forma normal, de forma automática. Para que un esquema de una base de datos esté en tercera forma normal, implica que está en segunda forma normal[14], y que además no existen dependencias transitivas entre atributos no clave.

[14] La segunda forma normal asegura que los atributos no clave dependen de toda la clave principal.

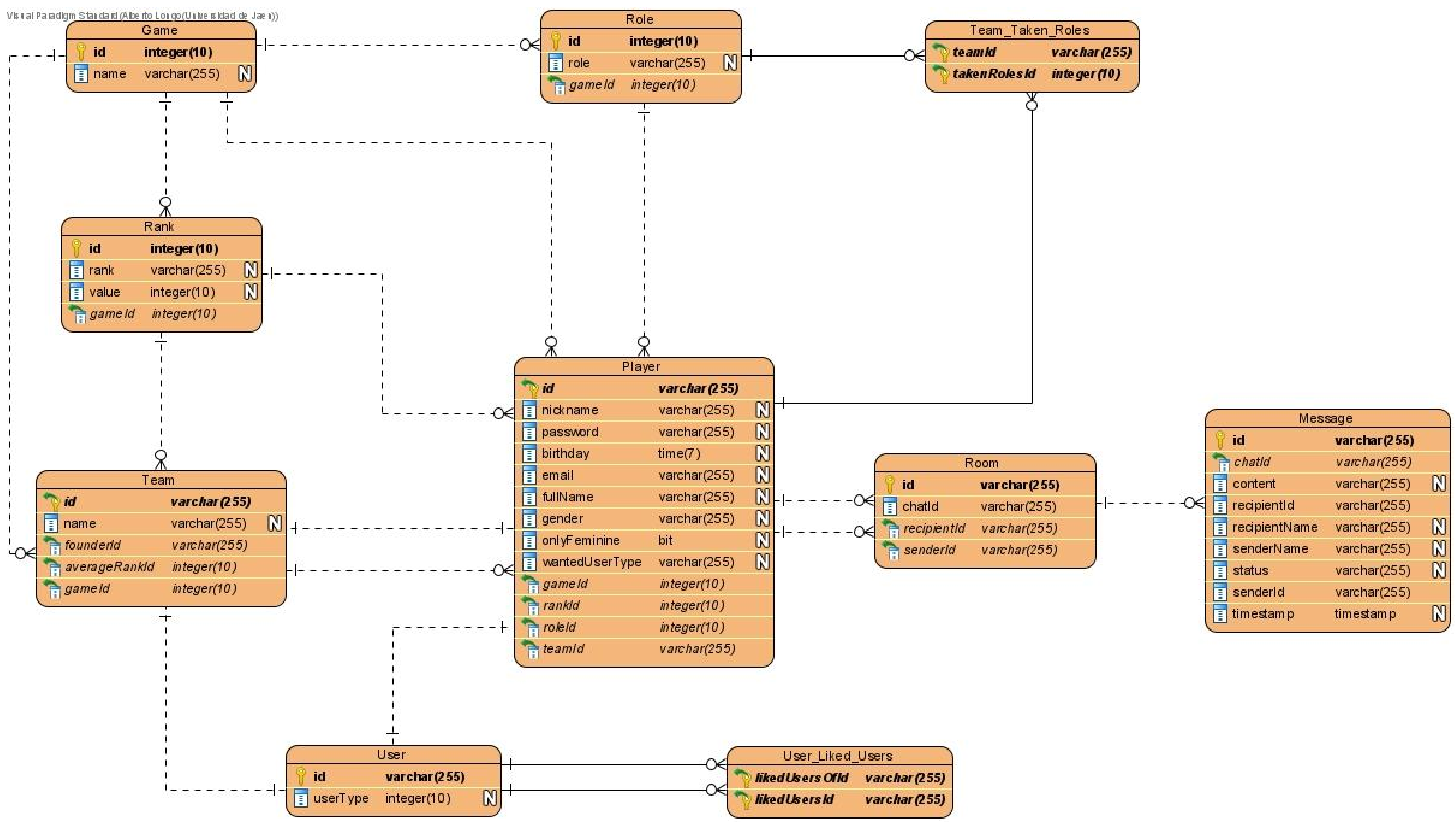

Figura 9. Diagrama entidad-relación.

Como se puede observar en el diagrama entidad-relación de la Figura 9, revisando el esquema presente podemos afirmar que se encuentra en tercera forma normal.

3.2. Diagrama de clases

Una vez definido el diagrama entidad-relación, pasaremos a definir el diagrama de clases del sistema, cuyo objetivo es mostrar una visión general del sistema en su conjunto, mostrando las diferentes clases que hay en el sistema, y su relación entre ellas.

En nuestro sistema, implementaremos una arquitectura empresarial orientada a servicios, la cual se basa en la existencia de un servidor o backend, donde se implementará la lógica de negocio, y que a su vez, exportará una serie de servicios, que utilizaremos desde nuestro cliente. Debido a esto, y a la complejidad del sistema, separaremos la definición de los diagramas en dos partes, el servidor y el cliente.

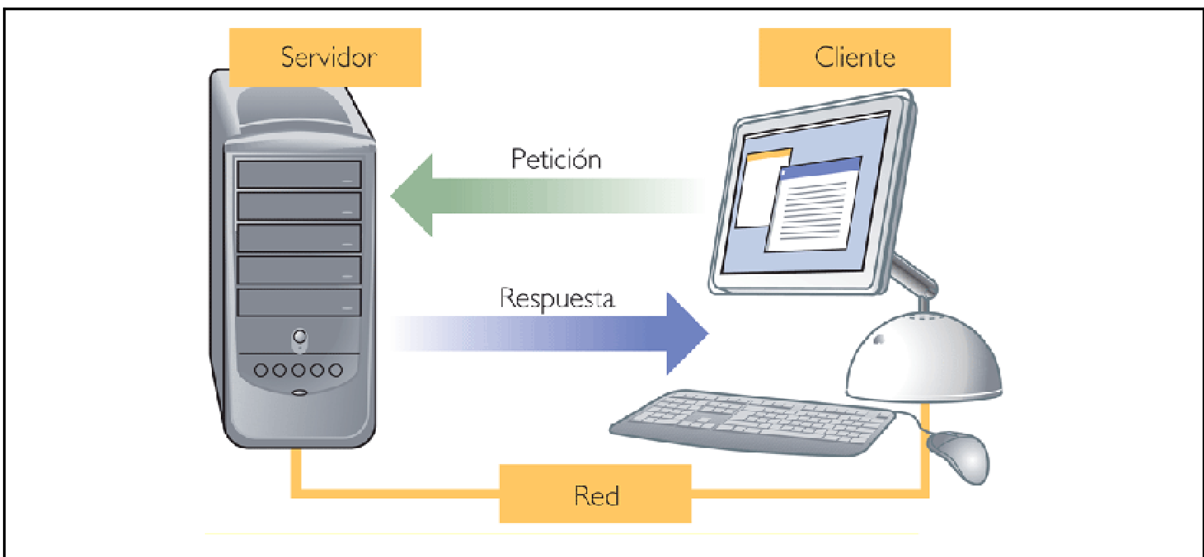

Figura 10. Ejemplo gráfico de arquitectura cliente servidor. (Fuente: https://laurmolina7821.wordpress.com)

En la Figura 10 podemos observar un ejemplo de arquitectura cliente servidor, donde las tareas relacionadas con la lógica de negocio se reparten entre los distintos proveedores, o servidores, que proporcionan una serie de servicios a los demandantes, los clientes. Un cliente realiza peticiones al servidor, el cual le envía la respuesta de vuelta.

3.2.1. Servidor

Como hemos mencionado anteriormente, la parte del sistema relacionada con la implementación de la lógica de negocio y proveer de servicios será el servidor, implementado sobre Spring, concretamente sobre Spring Boot, un empaquetado de servicios para crear aplicaciones sobre Java de manera rápida. Además, esta aplicación la implementaremos sobre el patrón de arquitectura REST[15].

Los servicios web REST están sustituyendo rápidamente a otros servicios tradicionales basados en SOAP[16], sin perder potencial frente a opciones más novedosas como GraphQL[17]. Este éxito se debe a su gran sencillez teórica, sobre todo en comparación con servicios web basados en SOAP, además de tener una buena integración con el protocolo HTTP. Sin embargo, REST puede proporcionar algunos problemas ya que implementar un API restful correctamente no es sencillo, tiene problemas para esquemas más complejos que un CRUD, además de que el protocolo HTTP[18] impone muchas limitaciones (Santos 2017).

REST es un patrón de arquitectura que integra el protocolo HTTP y los servidores web tradicionales, está orientado a recursos a diferencia de SOAP, que está orientado a operaciones. Es más ligero y proporciona una mejor adaptación a dispositivos (clientes) con capacidades limitadas como los móviles. Además este patrón define una serie de principios para aplicar correctamente la filosofía restful.

- El servicio debe de ir codificado como una serie de recursos. Las URLs no pueden contener referencias a operaciones.
- La operación a realizar sobre el recurso establecido se define mediante los métodos estándar HTTP (PUT, GET, POST, etc).
- El resultado de cada operación se indicará utilizando los códigos de respuesta estándar HTTP (200 OK, 400 Bad Request, etc).
- El servidor no debe de contener estado de la conversación con el cliente (*stateless*). De forma alternativa, puede implementarse el estado de la conversación mediante hipertexto, o HATEOAS[19].

[15] Transferencia de estado Representacional, o REST por sus siglas en inglés.
[16] SOAP (Simple Object Access Protocol) es un protocolo que define cómo dos objetos pueden comunicarse por medio de intercambio de datos XML.
[17] GraphQL es un lenguaje de consulta y manipulación de datos para APIs.
[18] Protocolo de transferencia de hipertexto, o HTTP por sus siglas en inglés.
[19] Hypertext As The Engine Of Application State, principio REST que usa URLs en las respuestas para guiar la conversación.

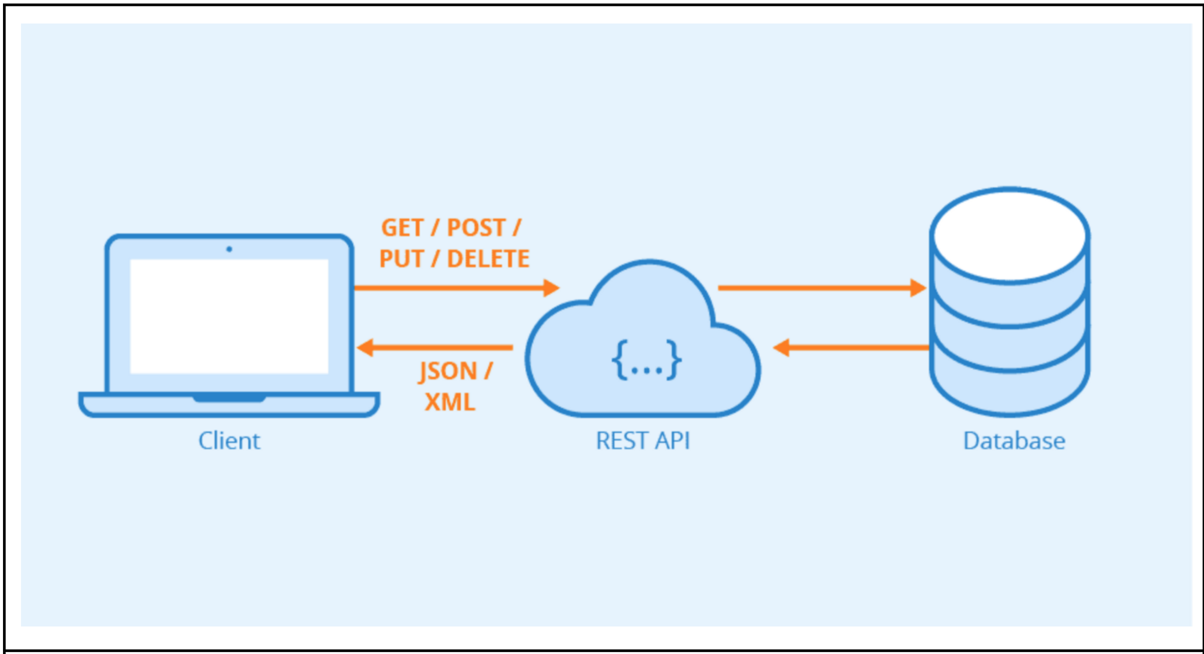

Figura 11. Arquitectura REST. (Fuente: https://www.seobility.net, Autor: Seobility, Licencia: CC BY-SA 4.0)

En la Figura 11 podemos observar como es el esquema básico de una arquitectura REST. Además de la arquitectura REST, también se hará uso del patrón DAO, este pretende independizar la aplicación de la forma de acceder a la base de datos. Para ello se centraliza el código relativo al acceso al repositorio en las clases llamadas DAO.

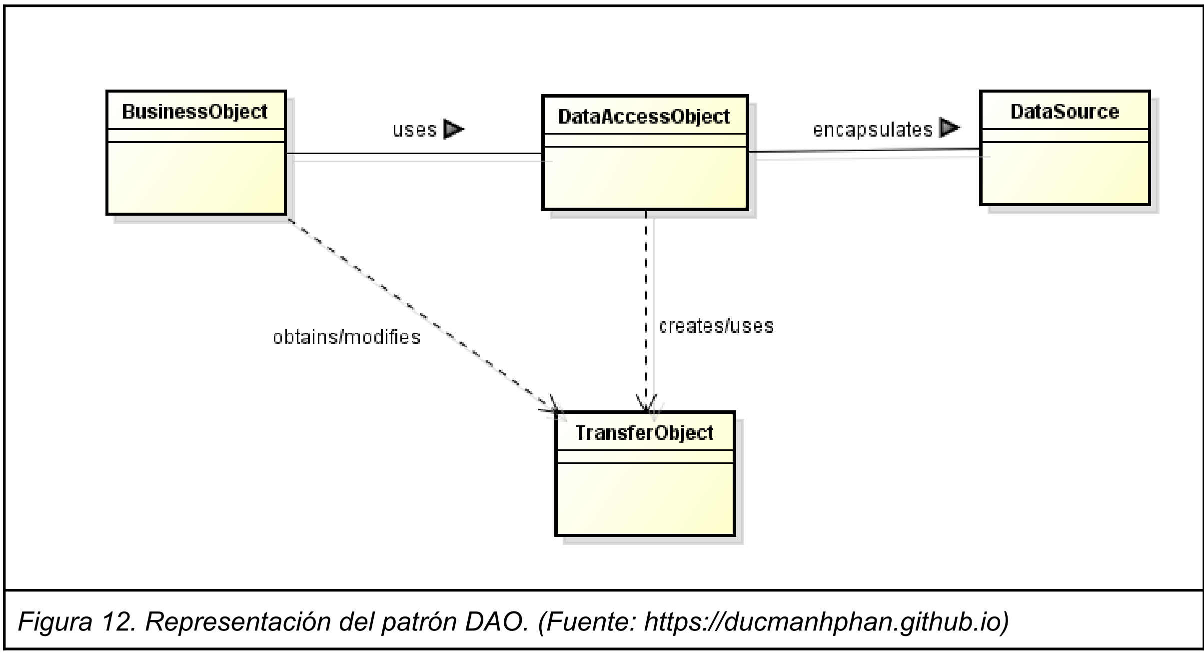

Figura 12. Representación del patrón DAO. (Fuente: https://ducmanhphan.github.io)

Nuestro servidor hará uso del patrón DAO para cada una de las entidades principales que requieran de persistencia en la base de datos, en la Figura 12, podemos observar como es la implementación del patrón DAO de forma gráfica. Las

entidades principales para las que se necesita un DAO son Usuario, Jugador, Equipo, Juego, Mensaje y Sala. Los DAO relativos a cada una de las entidades serán los encargados de gestionar el acceso a la base de datos, conteniendo el código necesario para para implementar las operaciones CRUD básicas, u otras específicas en caso de ser necesario.

Además, para el envío de información por medio de peticiones HTTP, utilizaremos unas clases secundarias o auxiliares llamadas DTOs[20]. Estas clases son POJOs[21] que solo incluyen campos simples, que encapsulan la información necesaria de la entidad que representa. Esto lo utilizaremos para enviar información al cliente de forma simple.

[20] Data Transfer Object, es un objeto intermedio utilizado para el transporte de datos.
[21] Plain Old Java Object, es una clase simple que no es dependiente de ningún framework en especial.

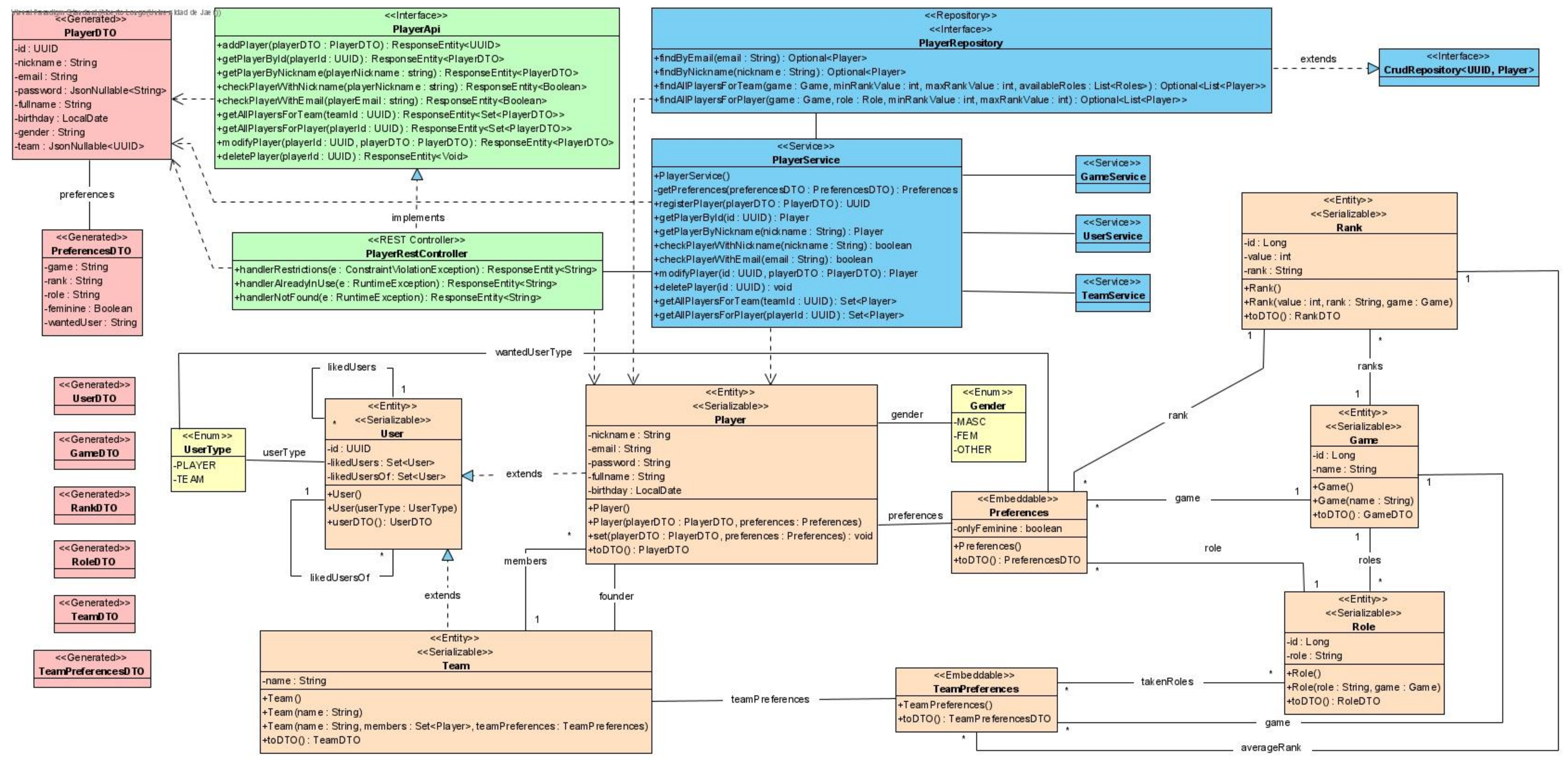

Figura 13. Diagrama de clases de la aplicación servidor.

En el diagrama de clases visible en la Figura 13, podemos visualizar una pequeña parte del sistema, que ejemplifica de forma completa el funcionamiento e interconexión de los elementos que lo forman. En este podemos visualizar en color verde las clases frontera, aquellas que representan la interfaz de usuario. En este caso pusimos el caso de ejemplo de la clase controladora del Jugador, conteniendo la funcionalidad de dicha entidad. En color rojo podemos encontrar las clases DTO, que nos servirán para el envío de información al cliente, como hemos mencionado anteriormente. En color azul podemos encontrar el servicio que aporta la funcionalidad de la lógica de negocio relacionado con los jugadores, además del repositorio que se encarga de abstraer el almacenamiento de datos en la base de datos de la misma entidad. La clase *PlayerService*, hace uso de otros servicios como son *TeamService*, *GameService*, *UserService*, sin embargo, con el fin de simplificar el diagrama para maximizar el entendimiento del esqueleto básico, no hemos ahondado en su funcionalidad. En un color naranja podemos encontrar las clases relacionadas con el modelo de la aplicación, y las conexiones entre ellas, como podemos observar, el modelo está cohesionado entre sí. Finalmente de un color amarillo encontramos algunos enumerables utilizados en el modelo, sin embargo, estos elementos no son entidades propias.

Para el resto de clases REST, la funcionalidad es similar, puesto que cada controlador REST integra la lógica de control, y en cada uno de ellos se inyectan los servicios necesarios para realizar las acciones, ya que estos son los que implementan la lógica de negocio. Posteriormente, cada uno de los servicios hace referencia a los repositorios necesarios de cada una de las entidades existentes en el sistema.

3.2.2. Cliente

El desarrollo para la parte del sistema relacionada con el cliente ha sido construido en Angular, un *framework* para aplicaciones web desarrollado en Typescript. Este framework utiliza la arquitectura MVC[22] y está basado en una serie de clases llamadas “Componentes”, entre cuyas principales características está el enlazado de datos.

[22] Modelo-Vista-Controlador es un patrón de arquitectura que se basa en la separación de la vista de la lógica de negocio.

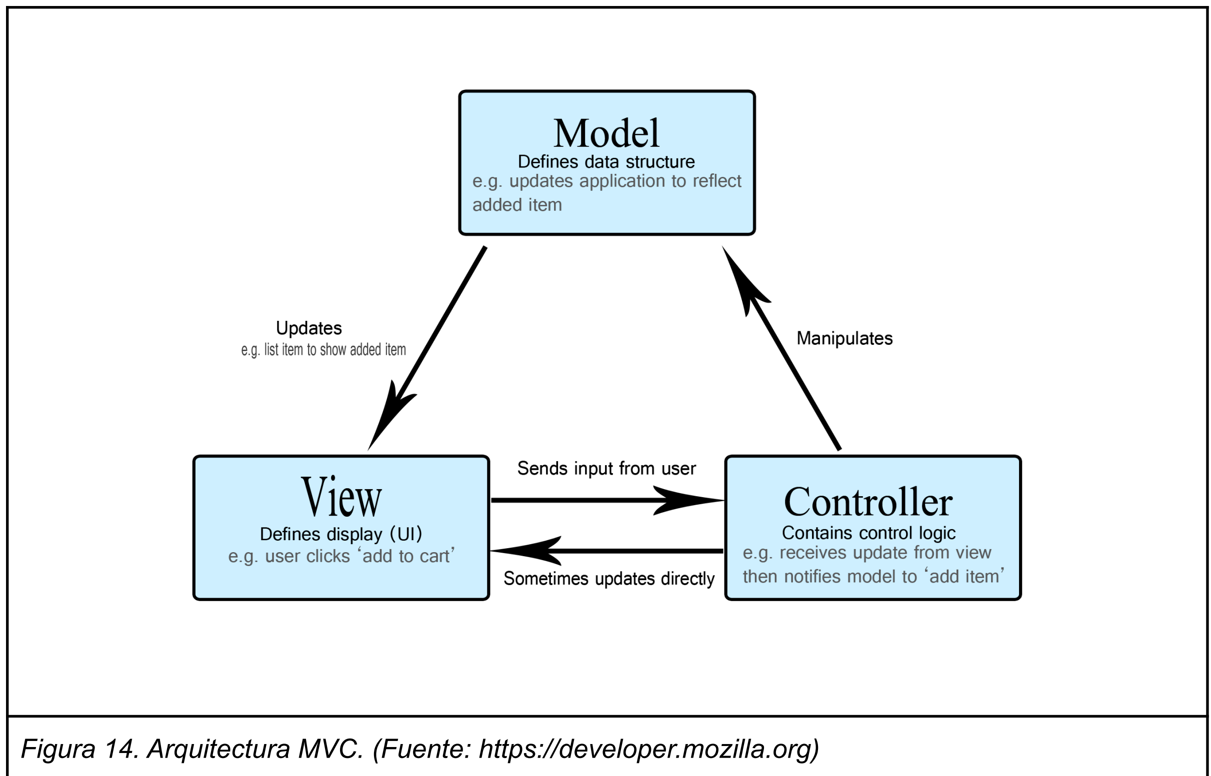

Figura 14. Arquitectura MVC. (Fuente: https://developer.mozilla.org)

En la Figura 14, podemos observar que el patrón MVC o Modelo-Vista-Controlador es un patrón de diseño de software muy comúnmente utilizado para implementar aplicaciones con un alto uso de interfaces gráficas. Puesto que enfatiza una separación entre la lógica de negocio y su visualización, permitiendo así definir diferentes vistas para una misma aplicación. Esta separación de preocupaciones proporciona una mejora del mantenimiento general de la aplicación, además de establecer unas mejores divisiones del trabajo. Cada una de las partes del patrón MVC se encarga de una parte diferente.

- Modelo: Se encarga de manejar los datos y lógica de negocio.
- Vista: Se encarga de la presentación en la interfaz de usuario.
- Controlador: Se encarga de enlazar tanto el modelo como la vista mediante un sistema de enrutamiento y comandos.

Además, existen otros arquetipos basados en MVC como MVVM (Modelo, Vista, Vista-Modelo), MVP (Modelo, Vista, Presentador), etc. El framework Angular, no implementa puramente el patrón MVC, ya que en el *pipeline*[23] de desarrollo de una aplicación en Angular, el modelo y la vista, se relacionan bastante entre sí.

[23] El pipeline es un proceso de actividades o procesos a seguir para conseguir un objetivo.

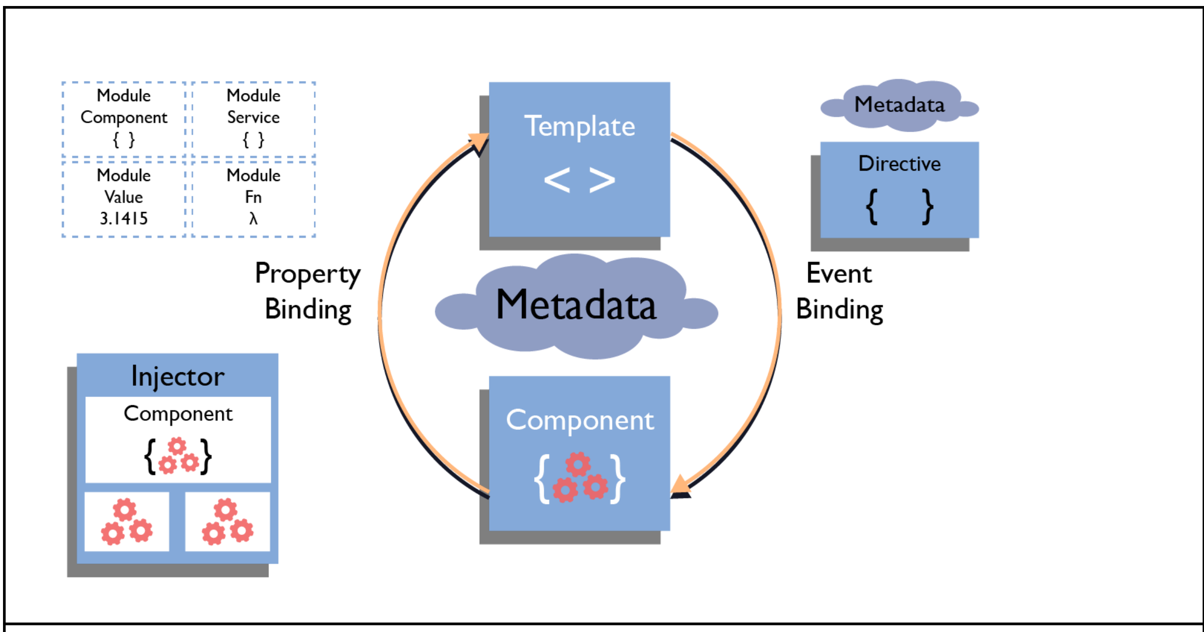

Figura 15. Arquitectura interna de Angular. (Fuente: https://ngchallenges.gitbook.io)

En la Figura 15 podemos observar el funcionamiento interno de una aplicación en Angular. Esto se debe principalmente a la funcionalidad de Angular de enlazar los datos doblemente, ya que la forma de sincronización de los datos entre la vista y el modelo-vista es totalmente dependiente, es decir, en la vista podemos modificar el modelo, y en el modelo podemos modificar la vista.

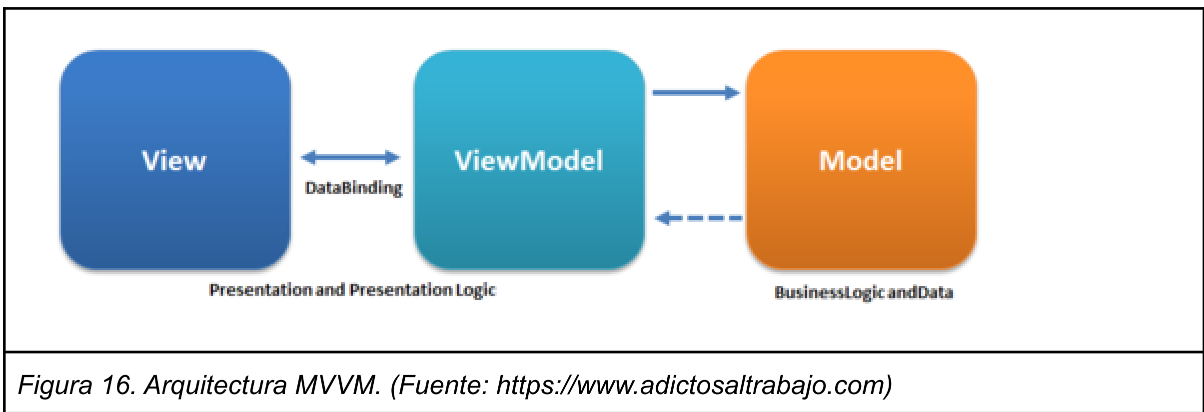

Figura 16. Arquitectura MVVM. (Fuente: https://www.adictosaltrabajo.com)

Esto nos da una pista de qué Angular no implementa puramente MVC, puesto que esta dependencia que se produce no existe en el patrón original MVC. Este caso se suele conocer como MVVM, o MVW[24]. En la Figura 16 podemos observar la estructura del patrón MVVM, siendo bastante similar al patrón MVC.

En el diagrama de clases del cliente, se muestran las clases relacionadas con la vista, encargándose de la recogida de los datos por parte del usuario, además de los diferentes controladores y estructura que sigue la aplicación en el cliente.

[24] Modelo-vista-whatever, es el arquetipo MVC donde el controlador no está muy bien definido, ya que tiene dependencias con el modelo y la vista.

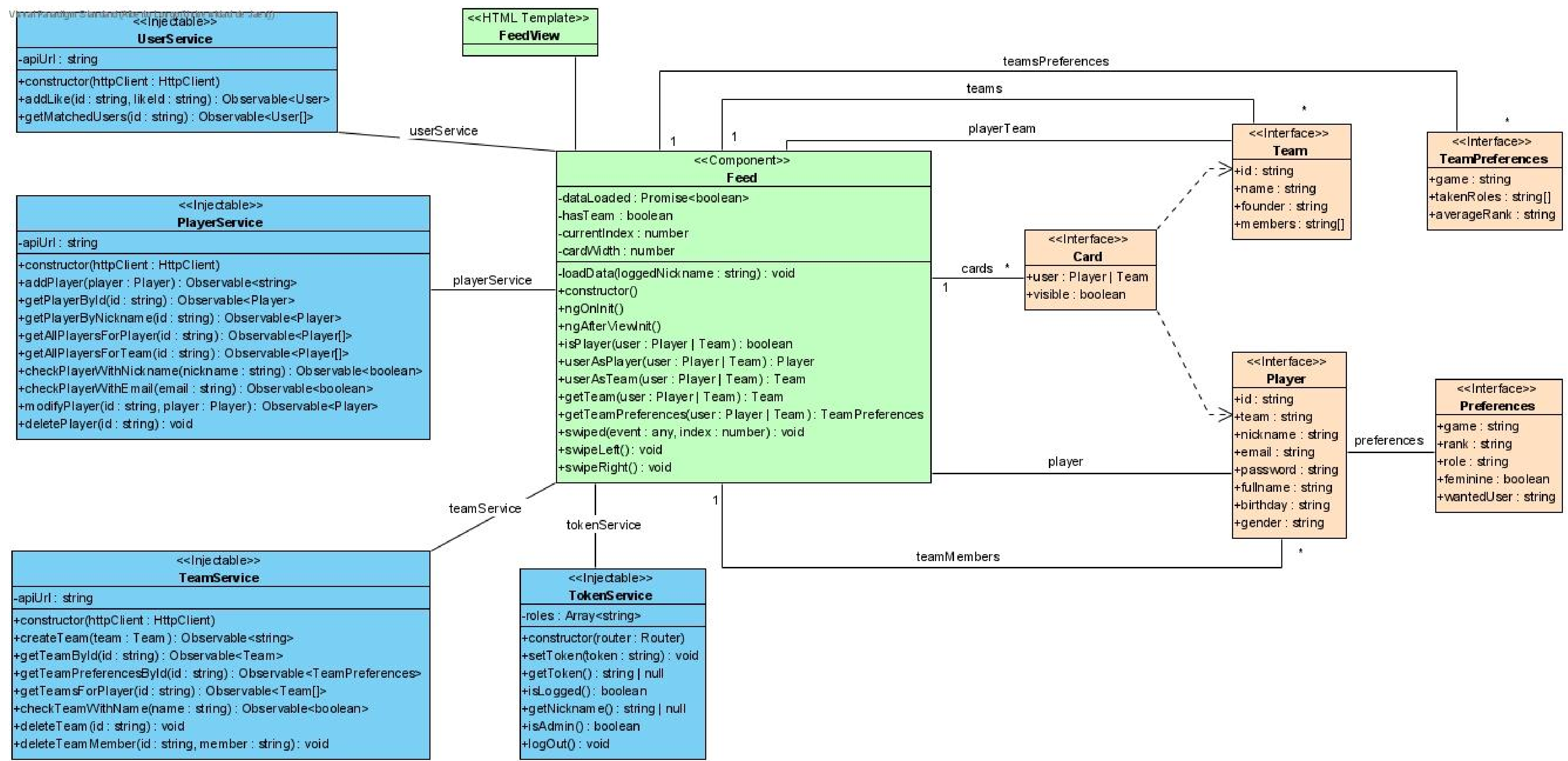

Figura 17. Diagrama de clases de la aplicación cliente.

En la Figura 17 mostramos el diagrama de clases de la aplicación cliente, en la cual seguiremos la misma filosofía que hemos seguido para el diagrama de clases del servidor, ya que, debido a su extensión, el diagrama de clases completo dificultará la visualización de la arquitectura presente dentro del mismo. Por ello, para ejemplificar la arquitectura presente en la aplicación cliente, mostraremos la estructura de la vista *feed*, es aquella vista donde el usuario visualizará una serie de tarjetas, y podrá moverlas para dar like o no a los usuarios, ya sean jugadores o equipos.

En el diagrama de clases podemos ver en verde la vista *feed*, que se encargará de recoger los datos de entrada del usuario, además la vista tiene asociado el componente *Feed*, el cual hará de *view-model* dentro de la arquitectura MVVM, este componente tendrá la lógica asociada a los elementos de la vista, además de servir de controlador entre la vista y el modelo. A su vez este componente se comunica con algunos servicios que guardarán la lógica necesaria para comunicarse con el servidor. Estos servicios, visibles en azul, serán los encargados de enviar y recibir las peticiones y respuestas HTTP al servidor, enviando y recogiendo la información proveniente del mismo. En un color anaranjado, podemos ver el modelo, con los datos necesarios para la visualización de la vista.

3.3. Diagramas de secuencia

En este apartado, incluimos y explicamos algunos de los diagramas de secuencia más relevantes del proyecto, explicando los aspectos más relevantes de cada uno para comprender el funcionamiento del mismo.

Este apartado tiene como objetivo explicar el funcionamiento interno del diseño propuesto, con el objetivo de facilitar un futuro mantenimiento. Se han escogido una serie de funcionalidades que consideran más representativas del proyecto, que ofrecen una visión extensa del proyecto, de tal forma que el resto de funcionalidades puedan ser deducidas sin mayor complicación a partir de los diagramas presentados.

Los diagramas de secuencia contendrán elementos de la aplicación servidor y de la aplicación cliente. Para facilitar una mejor comprensión del diagrama y de la conexión entre ambas aplicaciones, hemos representado las clases de la aplicación cliente de un color rojizo, mientras que las clases relacionadas con el servidor, están representadas de un color verde.

3.3.1. Login

El primer de los diagramas de secuencia que vamos a explicar, es el proceso de iniciar sesión en la aplicación. Este refleja el proceso completo de transmisión de información entre el cliente y el servidor, además de verse reflejado el envío del token, que será la llave para poder estar autorizado a lo largo de la aplicación

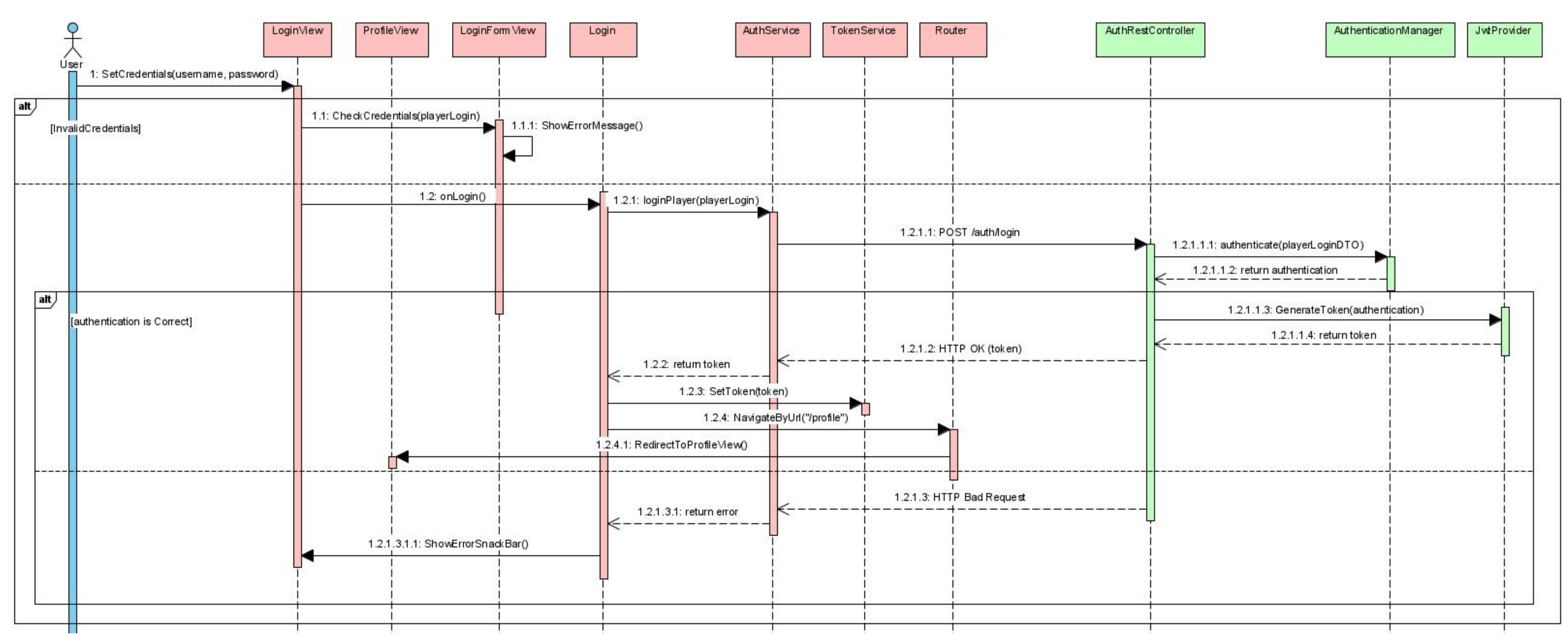

Figura 18. Diagrama de secuencia del caso de uso: iniciar sesión.

En la Figura 18 podemos ver el diagrama relacionado con el caso de uso iniciar sesión. En este, un usuario, desde la página de login, introducirá sus datos de acceso, si los ha introducido incorrectamente, o faltan por rellenar algún campo, no se realizará ninguna petición al servidor, y se mostrará por pantalla un mensaje de error correspondiente al campo en cuestión. En caso de haber introducido los datos de inicio de sesión correctamente, la vista llamará al servicio de autenticación, el cual enviará una petición HTTP al servidor, a través del endpoint /auth/login. Una vez la información haya llegado al servidor satisfactoriamente, este comprobará los credenciales a través del *AuthenticationManager*, en caso de ser incorrecto, se devolverá un Bad Request con un mensaje de error, y saldrá un *SnackBar*[25] en el cliente conteniendo un mensaje de error. Si, por el contrario, es satisfactorio, la autenticación llamará al *JwtProvider* para generar un token único y devolverlo al cliente. Finalmente el cliente guardará el token y redirigirá al usuario a su perfil personal, terminando la acción de iniciar sesión.

Como se puede observar, es un caso de uso muy completo, ya que contiene todos los elementos mencionados anteriormente, tanto las diferentes vistas, y su división en pequeños componentes, así como la comunicación con el servidor a través de un servicio en el cliente. Por último, el servidor realiza la lógica de negocio, y devuelve la respuesta al cliente junto a la información necesaria para continuar.

3.3.2. Modificar perfil de jugador

Para el diagrama de secuencia “modificar el perfil del jugador” podemos tener varios casos de uso, ya que la funcionalidad está repartida de tal forma que puedes modificar diferentes tipo de información del jugador, como su información relacionada con la cuenta, datos personales como puede ser el nombre completo o su fecha de nacimiento, y también la posibilidad de modificar sus preferencias.

En este diagrama, mostraremos el caso más representativo que es modificar los datos de cuenta, como son el nombre del usuario o el email.

[25] Un SnackBar es un elemento de interfaz que proporciona información emergente al usuario sin entorpecer la experiencia de uso, estos no necesitan de interacción del usuario para desaparecer y suelen aparecer en la parte inferior de la pantalla.

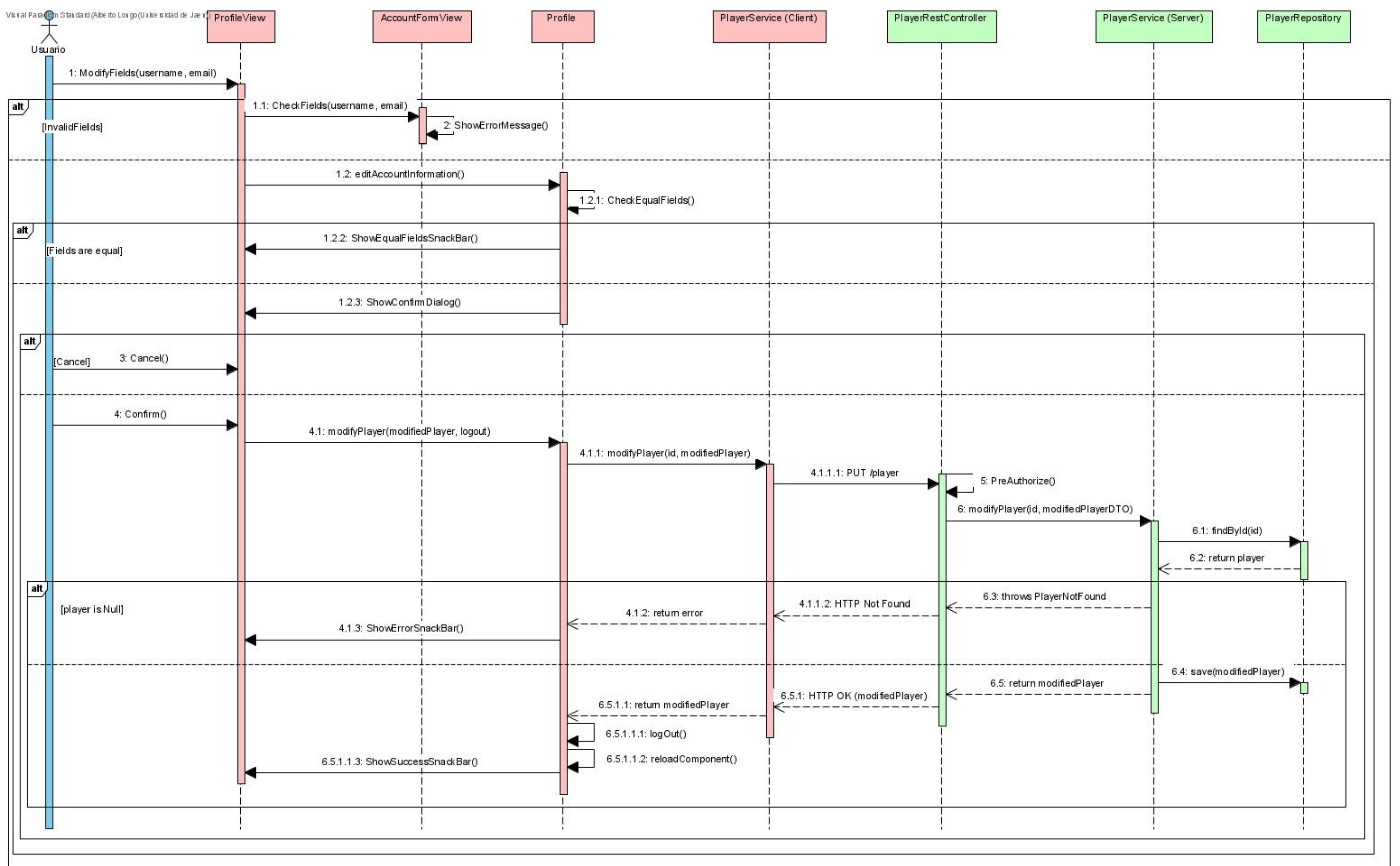

Figura 19. Diagrama de secuencia del caso de uso: modificar perfil.

En la Figura 19 mostramos el diagrama de secuencia para modificar el perfil del jugador. Una vez un usuario registrado y logueado quiera modificar su perfil, este abrirá los paneles desplegables y modificará los campos correspondientes, en el caso del diagrama, los campos de nombre de usuario y correo electrónico. Una vez que los haya modificado, puede ocurrir que estos sean inválidos, ya sea porque están en uso, o porque no están formados correctamente, como que el correo no sea válido. Cuando los campos estén correctamente rellenos, el jugador clica en guardar y confirmará en la ventana emergente. Una vez haya confirmado, el perfil llamará a *PlayerService* para enviar la petición al servidor, esta llegará y será procesada. Una vez en el servidor, puede ocurrir que el usuario enviado no exista, lo que devolverá un error al cliente, que se encargará de mostrar un *SnackBar* con el mensaje correspondiente. Si por el caso contrario, el usuario existe, se modificará y se enviará al cliente de nuevo esta vez conteniendo un mensaje de confirmación. El cliente recargará el componente, cerrará la sesión del jugador y mostrará un *SnackBar* de confirmación.

El motivo por el que el cliente cierra la sesión del jugador se debe a que el jugador ha modificado información sensible sobre su cuenta, y elementos como el token de sesión guardan información obsoleta, por lo que para evitar errores de seguridad, el sistema cerrará su sesión y tendrá que iniciar sesión de nuevo. En otro caso de uso donde el usuario no modifique información sobre su cuenta, como son las preferencias, el sistema no cerrará la sesión, simplemente recargará el componente para recargar la información presente en pantalla.

3.3.3. Dar like a otro jugador o equipo

El caso de uso “dar like” se efectúa de la misma manera tanto para equipos como para jugadores, puesto que para el sistema, ambas entidades son tratadas como un mismo tipo de entidad, usuarios.

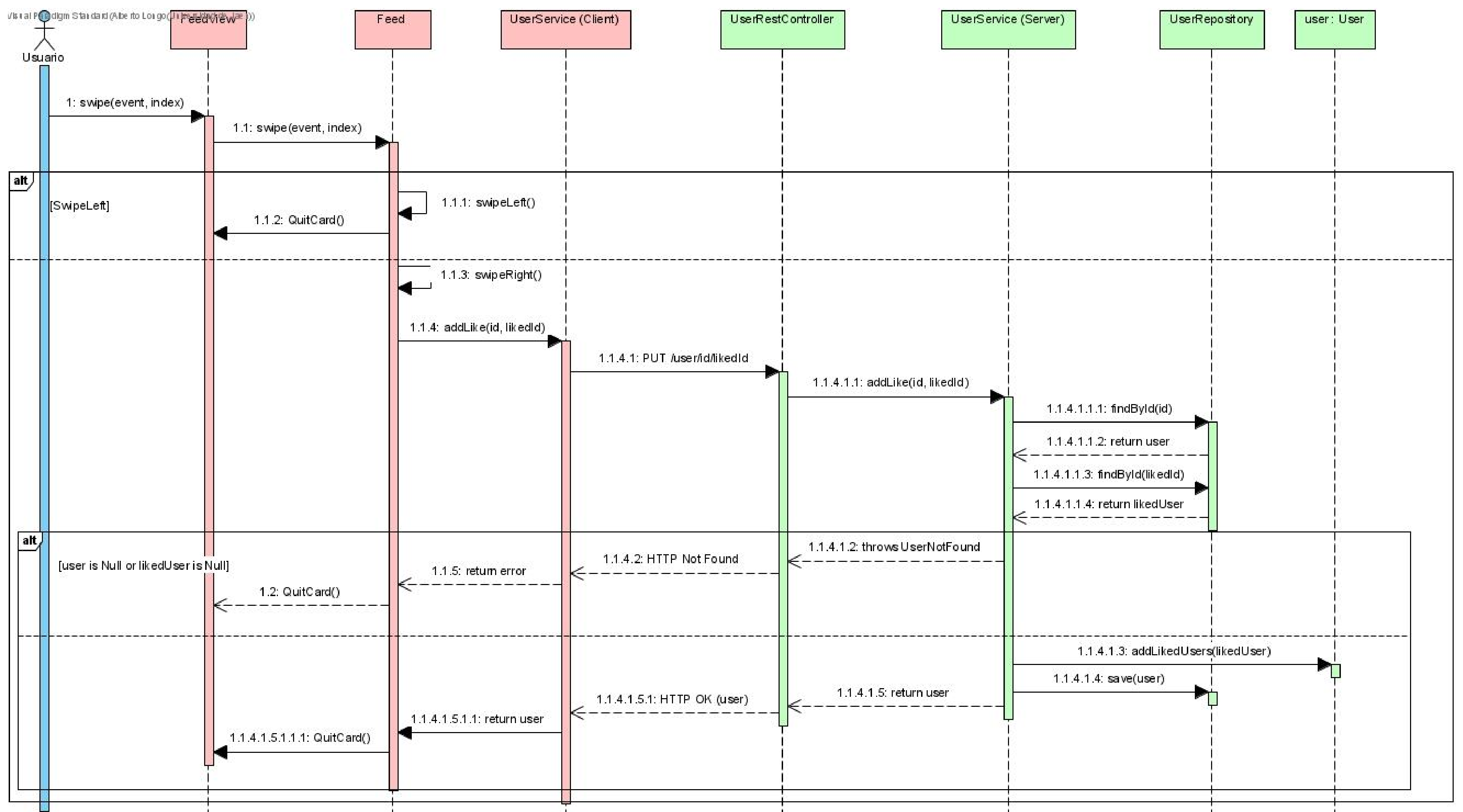

Figura 20. Diagrama de secuencia del caso de uso: dar like.

En este caso, como podemos observar en la Figura 20, el usuario para interactuar entra en la pestaña principal, y va deslizando las diferentes tarjetas por la pantalla o pulsando sobre los botones que hay sobre cada una de las tarjetas. Cada una de estas tarjetas representan un usuario, ya sea un jugador o equipo. Para dar like el usuario debe de deslizar la tarjeta a la derecha, en caso contrario, la tarjeta simplemente desaparece. En caso de que el usuario haya deslizado hacia la derecha, el sistema enviará una petición al servidor, para procesar el like y almacenarlo en la base de datos. Puede darse el caso de que los identificadores proporcionados para dar like, correspondan a jugadores inexistentes, sin embargo, esto no crea ningún error grave en la aplicación, ya que en ambos casos, la tarjeta simplemente desaparece.

Es un diagrama sencillo pero mantiene el proceso de comunicación entre cliente y servidor. Al ser una acción importante, y que se realizará mucho, la sencillez del caso de uso es beneficioso, ya que en caso de gestionar múltiples peticiones al mismo tiempo, el sistema no se sobrecargan con tanta facilidad, al ser peticiones ligeras.

3.3.4. Enviar un mensaje a un match

Como hemos mencionado anteriormente, el sistema de mensajería entre dos usuarios está habilitado una vez que exista un match entre ambos.

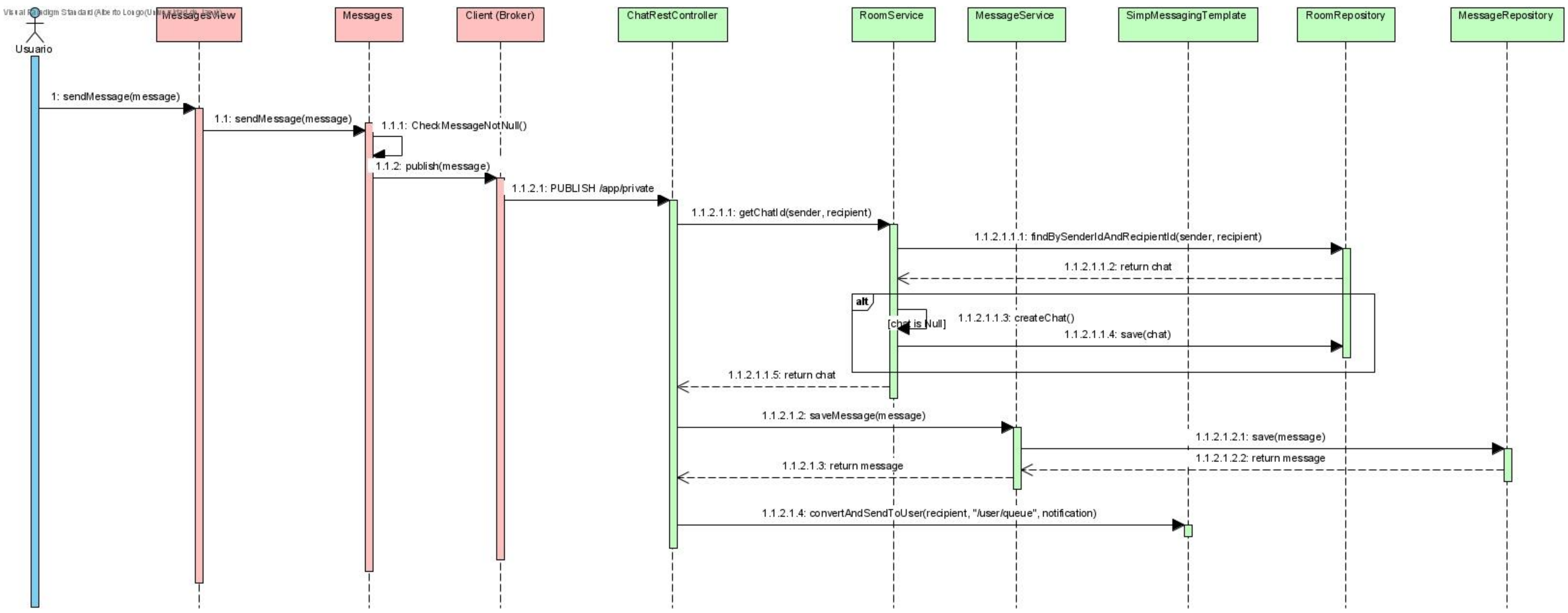

Figura 21. Diagrama de secuencia del caso de uso: enviar un mensaje a otro jugador.

En la Figura 21 podemos ver el diagrama de secuencia para enviar un mensaje. Una vez el usuario ha escrito el mensaje en el chat correspondiente, y le pulse al botón de enviar, la vista enviará al servicio de mensajería un mensaje a la ruta del servidor, donde se recogerán todos los mensajes. El sistema de mensajería funciona mediante STOMP[26], un protocolo similar a como funciona HTTP, pero funciona sobre el protocolo TCP[27]. Una vez el servidor haya recibido el mensaje, este busca dentro del mensaje la información relacionada con el chat al que pertenece, y una vez obtenido o creado el chat en caso de que no exista, se guardará el mensaje en la base de datos. Por último, el servidor envía el mensaje en formato de notificación a una cola específica para el usuario receptor del mensaje.

El usuario receptor en el cliente, estará suscrito a un evento que será lanzado una vez el servidor envíe un mensaje a su cola, por lo que verá una notificación en caso de no estar en el chat del que proviene el mensaje, o simplemente visualizará el mensaje en el chat abierto.

3.4. Diseño de la interfaz

Una vez determinadas las historias de usuario más básicas, podemos empezar a definir el diseño de la interfaz. En este proyecto, aplicaremos un diseño *Mobile-First*, debido principalmente a que el flujo de usuario está optimizado para un dispositivo móvil. Aplicar *Mobile-First* trata, en pocas palabras, de crear una experiencia de usuario completa para un dispositivo móvil: menos texto, fuentes más grandes, etc (Xia 2017). Además, nos proporciona ciertas ventajas sobre un diseño web adaptable, *Responsive Design,* debido a la necesidad de eliminar cualquier información superflua (W3C 2008, 3.1).

Desarrollar la interfaz de usuario centrándonos en la experiencia de usuario en un dispositivo móvil, requiere de más esfuerzo, debido a que la experiencia móvil posee más restricciones de diseño que la experiencia de escritorio. Sin embargo, desarrollar la interfaz en móvil nos ayudará posteriormente a trasladar la experiencia a escritorio sin necesidad de rediseñar la interfaz.

3.4.1. Estética

Como se afirma en (Blog Option 2020), *“Las personas tienden a creer que las cosas que se ven mejor, funcionan mejor, incluso si en realidad son menos efectivas o eficientes”*. Esta frase define el efecto de la usabilidad estética, que a través de estudios se logró percibir que una interfaz poco funcional, pero estéticamente

[26] Simple (o Streaming) Text Oriented Protocol es un protocolo de transferencia de mensajes de texto simple.
[27] Protocolo de control de transmisión es uno de los protocolos fundamentales en Internet que funciona sobre la capa de transporte en el modelo OSI.

agradable, genera una respuesta positiva en el cerebro, incluso hace creer que el diseño funciona mejor.

Al construir una interfaz de usuario, el objetivo principal que todo diseñador debe aspirar a alcanzar es lograr una buena experiencia de usuario *UX Design*. Y para conseguir una buena experiencia, existen dos pilares fundamentales para crear una interfaz exitosa: usabilidad y estética. Es de suma importancia encontrar un punto de equilibrio entre ambos pilares, ya que abusar de uno puede perjudicar al otro (Garret 2011).

La función de la estética hace que los usuarios tengan una experiencia más relajada y agradable, tienden a encontrar facilidades de uso, incluso en situaciones donde no lo sea.

En este punto hablaremos de cómo lograr una estética aceptable, tratando de analizar algunos puntos clave para lograr un resultado cohesivo entre sí y fundamentado en buenas prácticas de diseño.

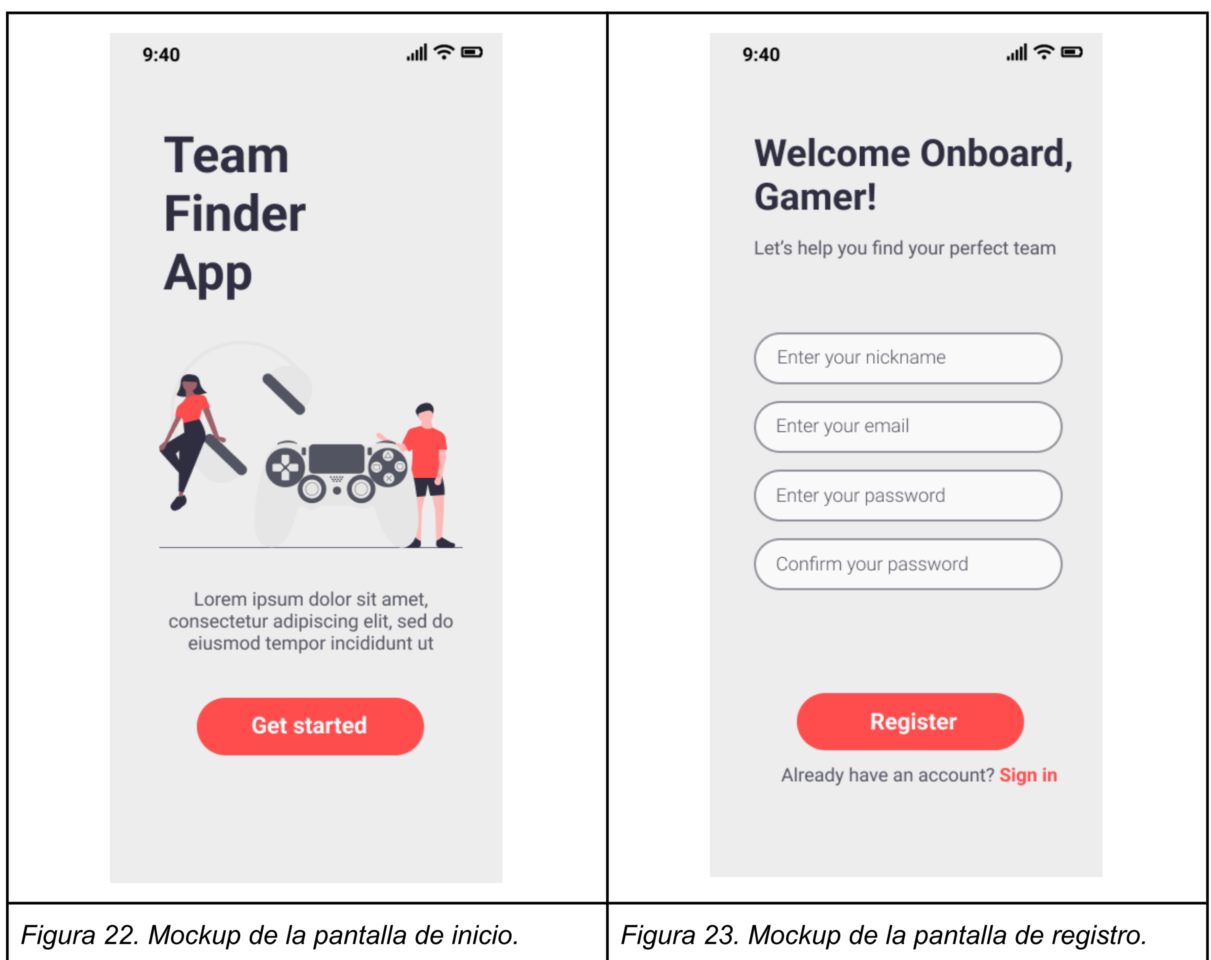

Figura 22. Mockup de la pantalla de inicio. *Figura 23. Mockup de la pantalla de registro.*

Uno de los principales puntos a destacar cuando queremos crear una interfaz de usuario estéticamente agradable, es el uso de una paleta de colores adecuada.

Como podemos observar en la Figura 22, observamos que el color predominante es el azul marino oscuro, este será nuestro color principal, y de él podremos obtener diferentes colores para utilizar distintas tonalidades del mismo, a los que llamaremos colores neutros (Busquets 2018).

Además, es importante seleccionar un color de acento, este puede encontrarse de diferentes formas a partir del color principal, en este caso, seleccioné este rojo para nuestro color de acento, su funcionalidad principal es añadir énfasis a los diferentes elementos de la interfaz. En la Figura 22 puede observarse como el botón destaca sobre el resto de la pantalla.

Debemos seguir ciertas reglas que harán que nuestra interfaz sea más accesible y estética. Como por ejemplo, no debemos de utilizar colores excesivamente saturados, al igual que utilizar el negro o blanco puro, puesto que esto hace que la interfaz sea menos legible y el usuario sienta cansancion en la vista mas rápidamente. Además, utilizar un negro puro hará que domine sobre el resto de colores, dejando inutilizado el objetivo de nuestro color primario.

Un elemento primordial en cualquier interfaz de usuario son los botones, y deben de estar diseñados acorde a la marca, y al estilo utilizado en la interfaz, para que mantenga una coherencia de estilo. Los botones deben de ser elementos que destaquen en la pantalla, en pocas palabras, tienen que parecer un botón real, y que el usuario se sienta atraído por presionarlo.

Utilizar colores que destaquen ayuda a llamar la atención, los bordes redondeados hacen que tenga más forma de botón como vemos en la Figura 23. Para lograr un diseño de botón completo, también debemos de ayudar a la accesibilidad haciendo visible cuando un botón está activo o está desactivado, si está presionado, cuando pasamos el ratón por encima, etc.

En caso de que nuestra aplicación requiera más de un tipo de botón según el contexto, deberemos de diseñar un estilo de botón secundario e incluso terciario, haciendo que destaque menos que el primario pero sin embargo que conserve las características mencionadas anteriormente.

3.4.2. Storyboard / Diagrama de navegación

El nombre de storyboard hace referencia al conjunto de ilustraciones, generalmente en películas de animación, para representar de manera rápida y sencilla la evolución de los actos a lo largo del tiempo.

Sin embargo, cuando lo utilizamos en el ámbito del desarrollo de aplicaciones software, el objetivo sigue siendo parecido, crear un prototipo visual rápido y barato de construir que nos permita obtener una primera impresión del producto, a la vez

que determinar las distintas vistas que tendrá la aplicación final así como la navegación entre ellas.

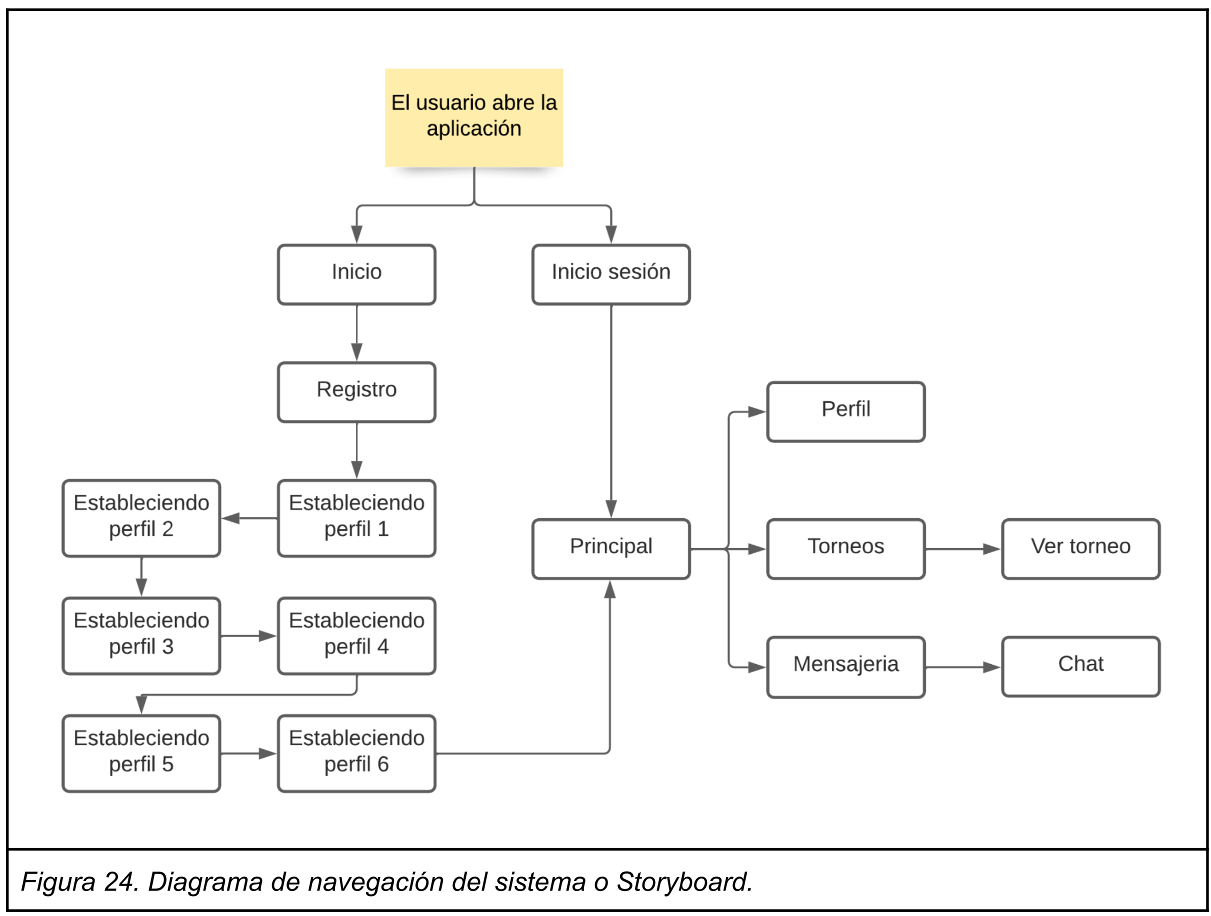

Figura 24. Diagrama de navegación del sistema o Storyboard.

Como podemos observar en la Figura 24, la navegación principal de la aplicación toma dos vertientes al comenzar la aplicación, dependiendo de si el usuario está registrado previamente o no.

En el caso de no estar registrado, recorrerá las diferentes pantallas de registro y establecerá los datos principales del perfil de usuario. En caso contrario, el usuario podrá iniciar sesión y pasará directamente a la pantalla principal de la aplicación, teniendo acceso al resto de opciones como son acceder a su perfil, mirar la lista de torneos y ver más información sobre alguno en concreto, o entrar a la pestaña de mensajería donde estarán todos los chats que tiene iniciado con otros equipos / jugadores.

Para facilitar la visualización del diagrama de navegación las pantallas o vistas no están incluidas en el propio diagrama. A continuación mostraremos las vistas más relevantes contenidas en el diagrama de navegación.

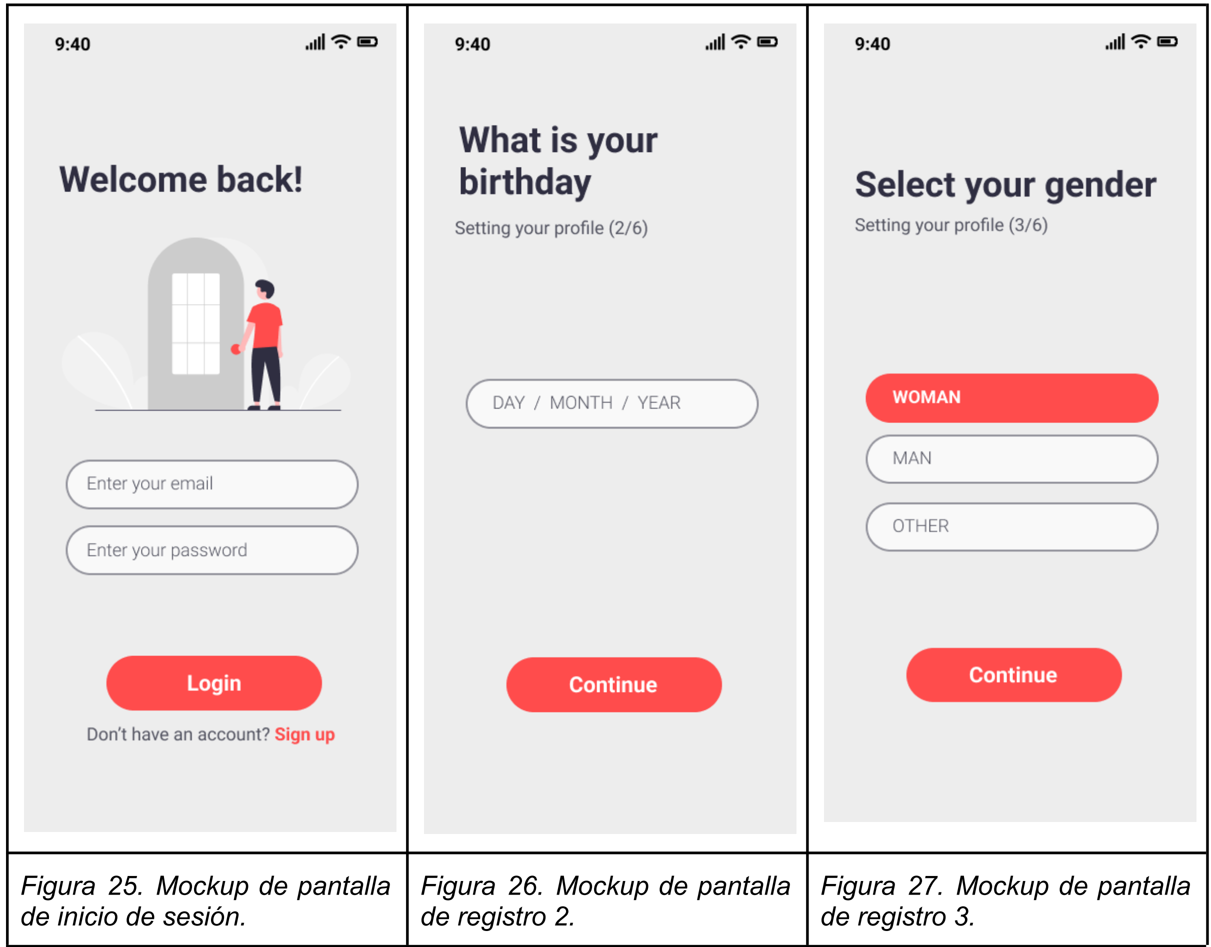

Figura 25. Mockup de pantalla de inicio de sesión.	*Figura 26. Mockup de pantalla de registro 2.*	*Figura 27. Mockup de pantalla de registro 3.*

En la Figura 25 podemos observar la pantalla de inicio de sesión, donde solo requiere los datos básicos del usuario para poder entrar.

En la Figura 26 encontramos la pantalla correspondiente a “Estableciendo perfil 2” en nuestro diagrama de navegación, donde solo introducimos un dato individual a través de un campo de texto.

En la Figura 27, sería la pantalla “Estableciendo perfil 3”, y en este caso el tipo de entrada sería múltiple, donde el usuario escoge una de las opciones, y podemos observar como la opción seleccionada queda destacada utilizando nuestro color de acento.

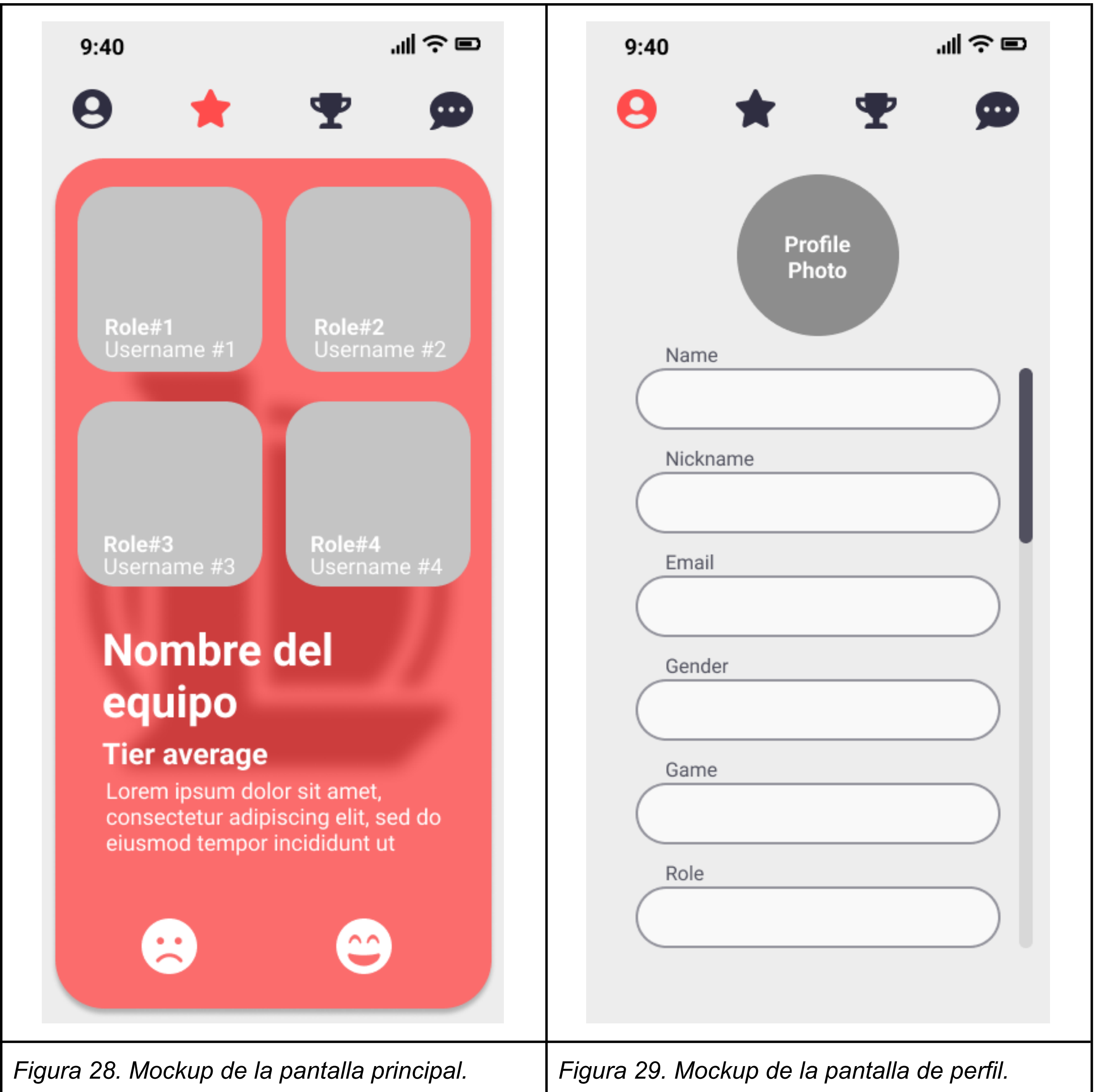

Figura 28. Mockup de la pantalla principal.

Figura 29. Mockup de la pantalla de perfil.

Una vez estemos registrados entraremos en la ventana principal de la aplicación o *"feed"*.

Como podemos observar en la Figura 28 encontramos en esta ventana de *feed* un menú superior con las 4 opciones mencionadas en el diagrama de navegación (perfil, principal, torneos y mensajes), estas opciones carecen de texto ya que gracias a la iconografía es intuitivo saber para qué sirve cada botón.

Se puede apreciar además una gran tarjeta, conteniendo todos los datos de un equipo ofrecido al jugador. La manera de aceptar o declinar la oferta del equipo será deslizando la tarjeta fuera de la pantalla, o clicando sobre los botones abajo de la pantalla.

En la Figura 29, se puede observar el apartado del perfil, donde tendremos los datos del usuario en cada uno de los campos de texto, además de la posibilidad de añadir una imagen de perfil.

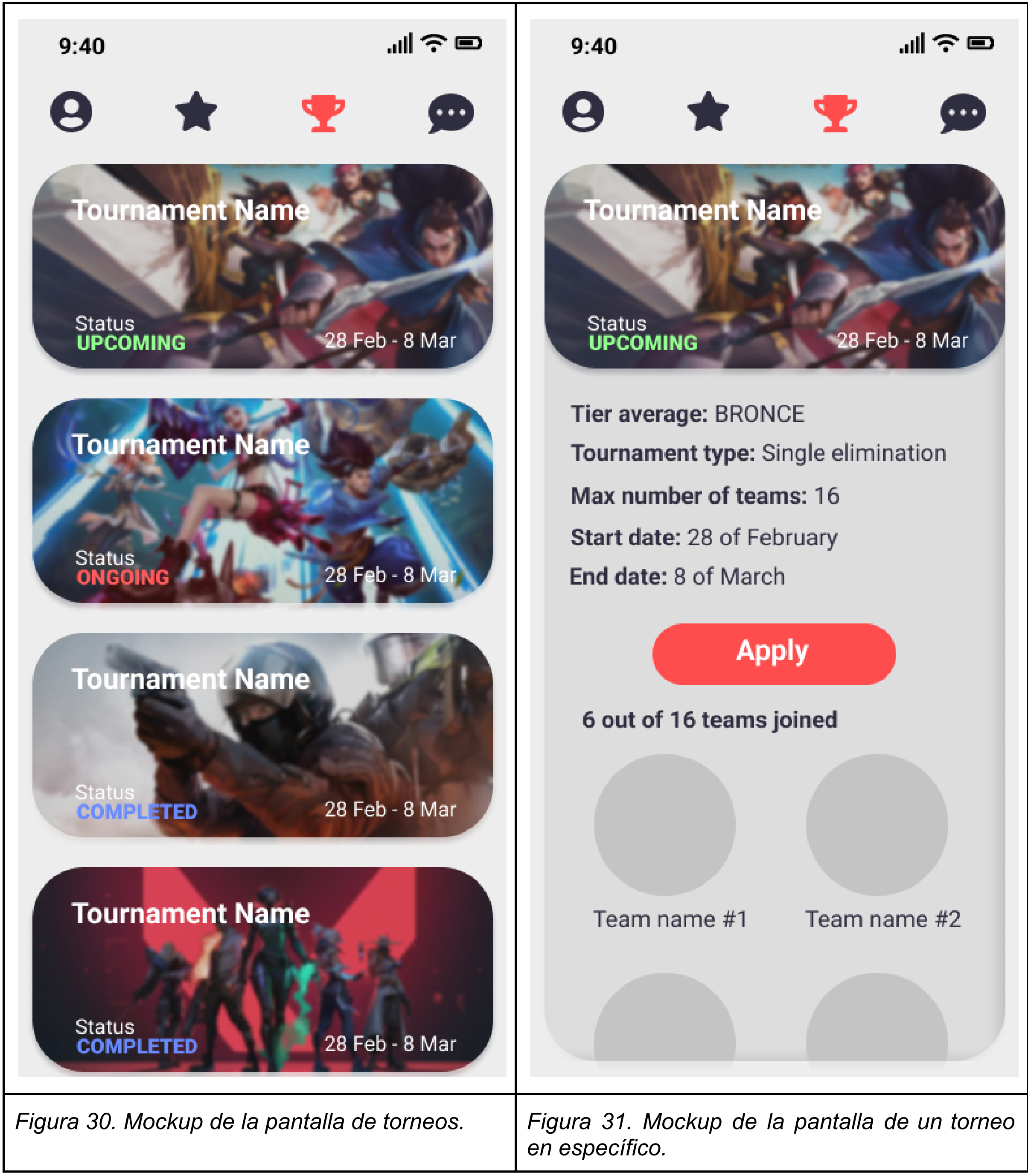

Figura 30. Mockup de la pantalla de torneos.

Figura 31. Mockup de la pantalla de un torneo en específico.

En las Figuras 30 y 31 podemos observar el apartado de Torneos. La Figura 30 muestra la lista de torneos disponibles, en curso y completados, además de indicar el juego al que pertenece añadiendo a la tarjeta una imagen de fondo del juego.

En la Figura 30 cabe destacar la importancia de los colores para cada uno de los estados del torneo, ya que los 3 colores entran dentro de una categoría de colores

llamados semánticos. Estos colores están presentes en todos los diseños, ya que psicológicamente envían mensajes dependiendo del color, p.e. El rojo indica peligro, o alerta, el azul indica información, y el verde, éxito y seguridad.

En nuestro caso podemos ver que la asignación de colores cobra sentido en función del matiz del estado del juego, p.e. El estado de disponible o *upcoming* en verde nos indica que estamos a tiempo para apuntarnos, sin embargo, el color rojo de *ongoing* o en curso, nos advierte que ya el torneo está empezado, y de igual forma con el azul, indicándonos que el torneo ya terminó hace tiempo.

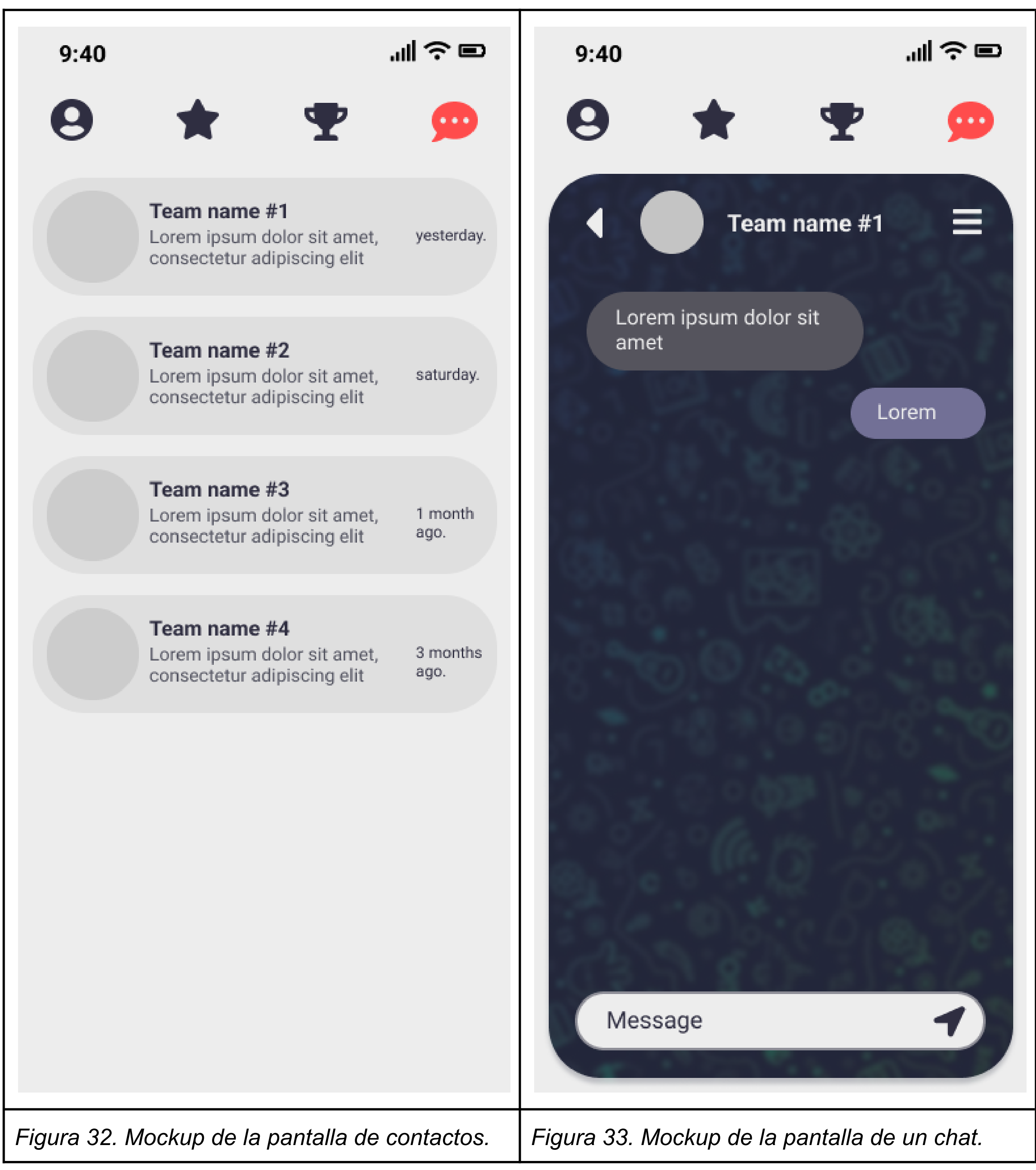

Figura 32. Mockup de la pantalla de contactos.

Figura 33. Mockup de la pantalla de un chat.

Por último, en las Figuras 32 y 33, podemos ver las ventanas del sistema de mensajería, y un chat abierto. Cabe destacar que mantenemos la coherencia a lo largo de todas las ventanas manteniendo los componentes de cada una de las vistas redondeados, dando forma similar a un *“squircle”*. Esto crea una identidad de marca que nos permite reutilizar dicha forma a lo largo de la aplicación (Caballero 2017).

3.4.2.1. Evolución del diseño de la interfaz

Durante el desarrollo de la aplicación, la estética y su diseño ha ido evolucionando desde el diseño presentado anteriormente, manteniendo todos los conceptos clave y fundamentales que proporcionaban una identidad de marca a nuestra aplicación.

Como hemos mencionado anteriormente, para el desarrollo de la interfaz del cliente, hemos aprovechado la librería de componentes de interfaz Material Design, que nos proporciona una amplia gama de componentes reutilizables que aportan una mejor experiencia de usuario y sobre todo un diseño coherente.

A continuación mostraremos algunas comparaciones entre el diseño preliminar presentado en el storyboard y el resultado final del diseño.

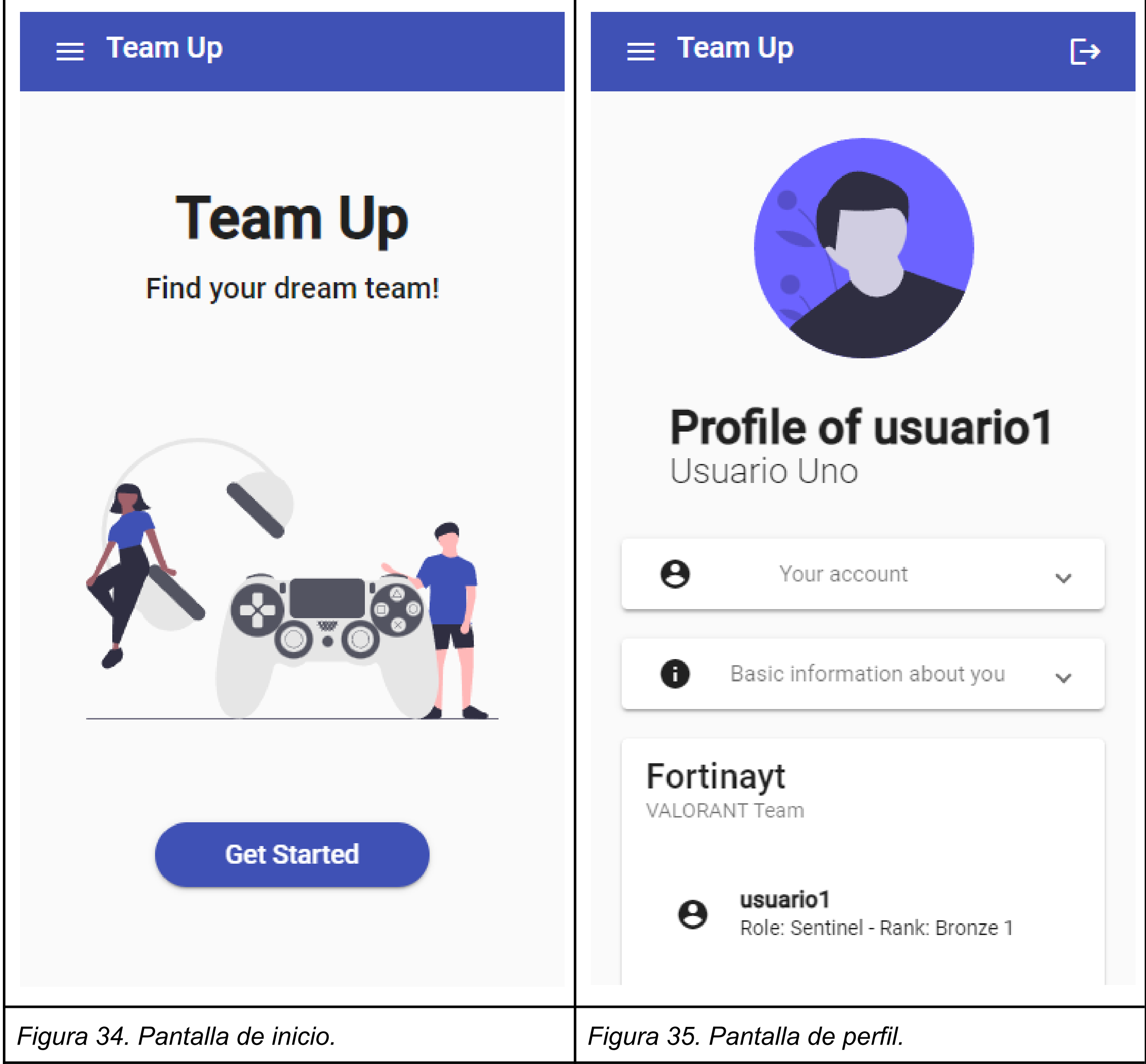

Figura 34. Pantalla de inicio. *Figura 35. Pantalla de perfil.*

Como podemos observar en la Figura 34, la pantalla principal se ha mantenido invariable respecto al diseño preliminar, sin embargo, se puede apreciar como hubo un cambio respecto al color insignia o color principal pasa a ser de una tonalidad añil.

También hubo un cambio en el color de acento, pasando a ser de un color rosado, que contrasta correctamente con el índigo y se queda reservado para elementos más concretos dentro de la aplicación, como puede ser el marcar los hipervínculos dentro de la misma.

En la Figura 35 también podemos observar como el perfil ha variado ligeramente respecto al diseño inicial, ya que debido a la cantidad de información necesaria para mostrar dentro del perfil de usuario, fue necesario organizarla en compartimentos contraíbles, separando la información por su dominio.

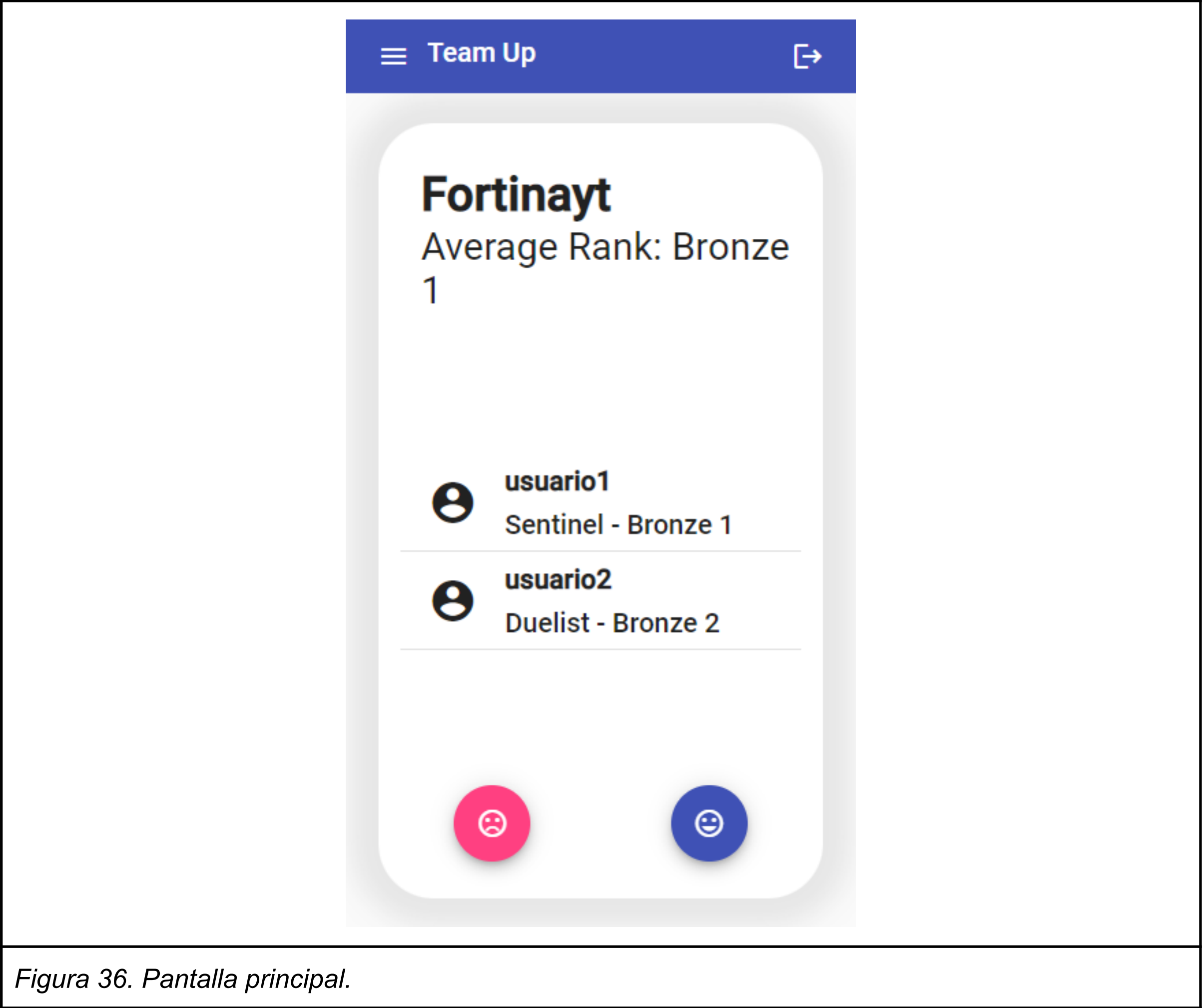

Figura 36. Pantalla principal.

Otro elemento importante que ha cambiado respecto al diseño inicial, fue la disposición de elementos dentro de cada tarjeta. Como se puede observar en la Figura 28, la predisposición inicial constaba de 4 cuadrados con la información dentro de cada uno de los recuadros. Sin embargo, esta predisposición dejaba información muy relevante como eran los rangos de cada uno de los integrantes del equipo al margen. Además, elementos como la descripción del equipo, fue un elemento finalmente no incluido dentro del diseño final de la aplicación, por lo que también tuvo que reestructurarse. El resto de elementos han sufrido modificaciones menores, como hemos podido analizar en el caso de la Figura 36.

A pesar de los cambios sobre el diseño a medida que avanzaba el desarrollo, se han mantenido todos los elementos clave que consideramos que aportan el mayor valor a la estética del producto.

3.4.3. API Rest

En el apartado anterior hemos definido la interfaz entre el usuario, y nuestra capa de presentación, también llamada *Front-end*. Es la capa que interactúa con los usuarios y recoge la información de entrada por parte de los mismos. De igual forma, debemos de definir una interfaz entre nuestra capa de presentación, el *front-end*, y nuestra capa de procesamiento de datos, el *back-end* (Wikipedia 2022).

Esta tarea la realizaremos mediante el uso de una API, una interfaz de programación de aplicaciones que nos ofrece el uso de una serie de métodos, subrutinas, etc como capa de abstracción para el uso de la aplicación (Wikipedia 2022). Para el desarrollo de nuestra API, utilizaremos REST, que se define como un estilo de arquitectura, no como un protocolo ni un estándar.

Para que una API en cuestión se considere RESTful, se deben cumplir ciertas precondiciones que se indican en (Red Hat 2020) como:

- Codificación de nuestro servicio como una serie de recursos accesibles mediante URLs. La cual nunca debe incluir referencias a operaciones sobre el recurso.
- Implementación de las operaciones sobre los recursos mediante los métodos estándar HTTP: PUT, GET, POST, DELETE
- Utilización de códigos de respuesta estandarizados HTTP para indicar el resultado de la petición: 200 para OK, etc.
- Implementación de un servidor stateless (sin estado), que no guarde el estado de la conversación con el cliente. Alternativamente, se puede implementar el estado de la conversación con el cliente mediante el principio HATEOAS.

A pesar de que un API REST debe cumplir todos estos principios, resulta más sencilla su implementación que optar por otros protocolos como SOAP[28] (W3C 2007), que es mucho más restrictivo que implementar REST; lo que hace a las APIs REST rápidas y ligeras, resultando ideales para el desarrollo de aplicaciones para dispositivos móviles, como es nuestro caso.

3.4.3.1. Documentación de la API Rest

Para la documentación de nuestra API Rest, haremos uso de Swagger[29], un framework que nos ofrece una serie de reglas, especificaciones y herramientas que ayudan a documentar una API de forma que sea comprensible para todo el mundo. Una de las mayores ventajas de Swagger es la documentación de nuestra API

[28] Intercambio de mensajes en formato XML.
[29] https://swagger.io/

utilizando el estándar OpenAPI 3.0[30], pudiendo crearla mediante un fichero .yaml[31] o .json[32].

La documentación en OpenAPI, nos será de utilidad para generar el código del servidor directamente a partir del mismo fichero de especificación. Ya que nos generará de forma automática las entidades que hayamos documentado, junto a los diferentes endpoints para cada una de las entidades.

Además, Swagger ofrece una herramienta llamada Swagger UI[33], siendo una de las herramientas más atractivas de la plataforma, facilitando la navegación por la documentación, organizándose de manera que sea legible a aquellos usuarios que quiera revisar la documentación de la API.

Swagger UI utiliza el fichero .yaml o .json mencionado previamente y lo muestra de forma interactiva, ordenando y categorizando las operaciones, siendo cada una de ellas expansible, mostrando de forma completa los parámetros, con sus respectivos ejemplos. Comenzaremos definiendo los métodos de acceso para uno de nuestros recursos más importantes en la aplicación, siendo los jugadores.

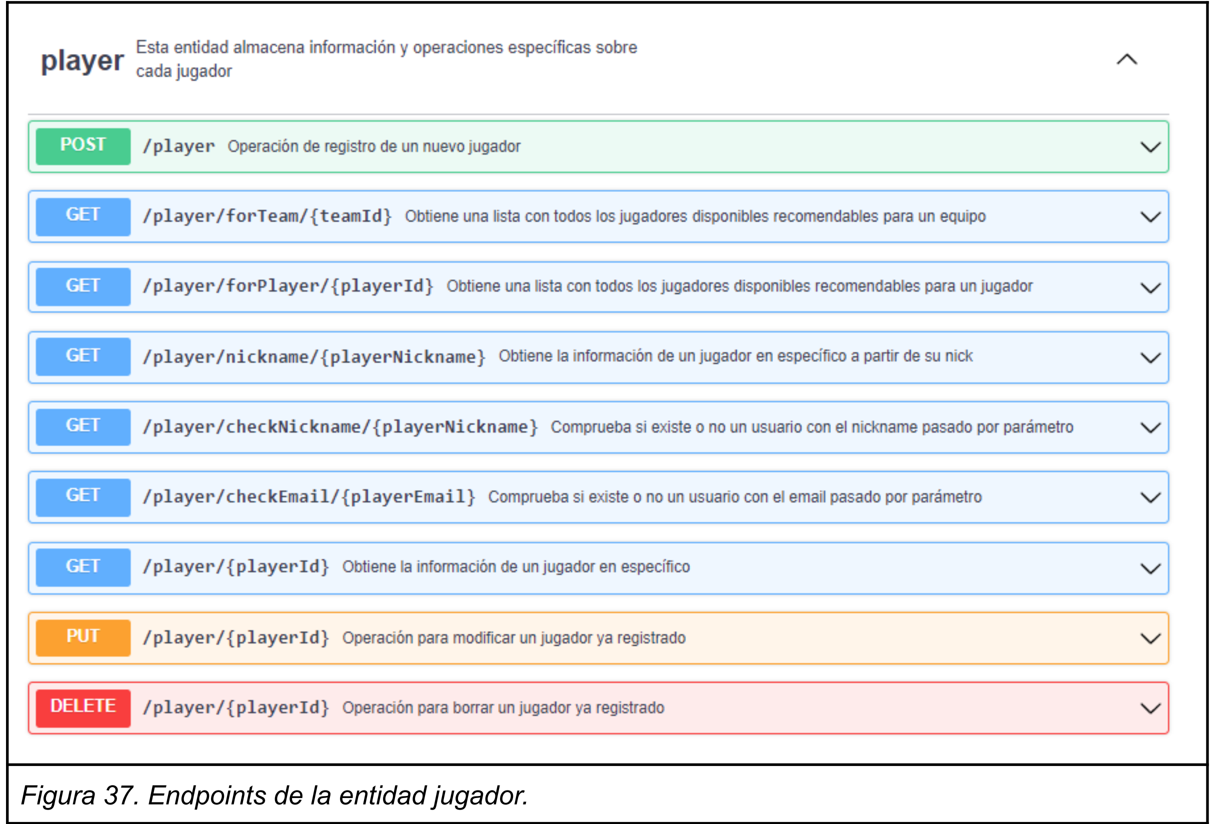

Figura 37. Endpoints de la entidad jugador.

[30] https://swagger.io/resources/open-api/
[31] YAML es un formato de serialización de datos legible por humanos propuesto por Clark Evans.
[32] JSON (acrónimo de JavaScript Object Notation) es un formato de texto sencillo para el intercambio de datos.
[33] https://swagger.io/tools/swagger-ui/

Como podemos observar en la Figura 37, el primer de nuestros endpoints es el método POST a la URL “*/player*”. En esta operación la funcionalidad consiste en crear un nuevo jugador e introducirlo en el sistema. Para ello, no será necesario ningún parámetro tanto en la cabecera como en la URI.

Figura 37.a. Información específica de envío al endpoint POST de la entidad jugador.

La información enviada en la petición irá dentro del cuerpo de la misma, conteniendo los valores de nombre de usuario, correo electrónico, nombre completo, fecha de nacimiento, género, juego de preferencia, rango y rol. En la Figura 37.a puede observarse los valores mencionados anteriormente.

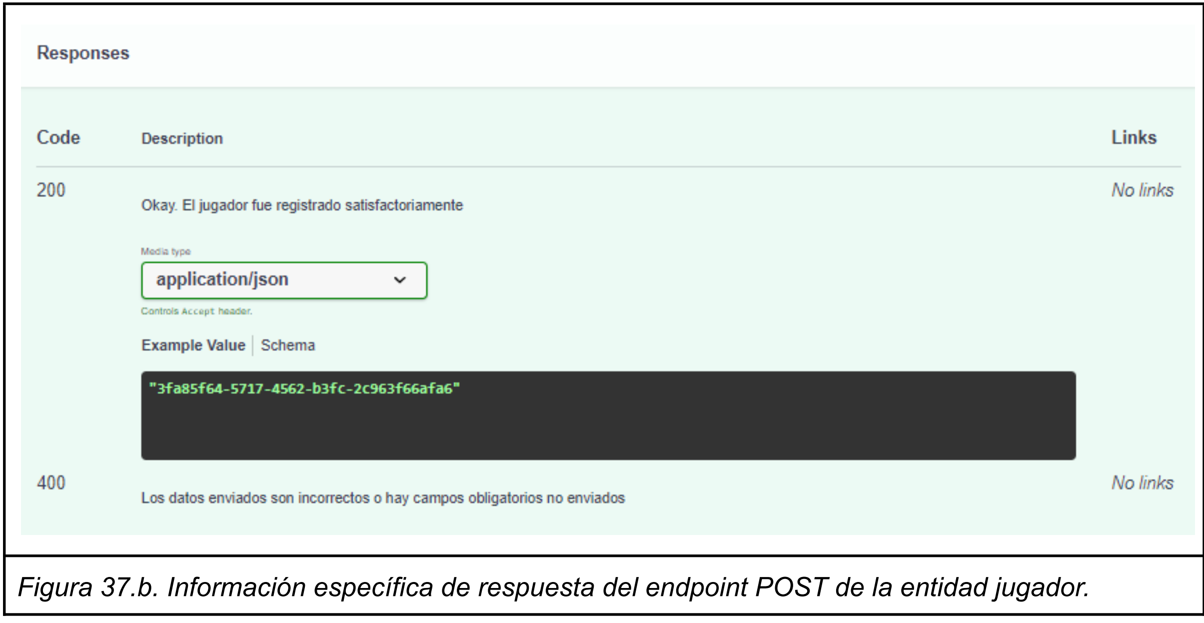

Figura 37.b. Información específica de respuesta del endpoint POST de la entidad jugador.

En la Figura 37.b se ven los valores de respuesta en dicha petición. Puede ser un código 200, indicando que el registro fue realizado satisfactoriamente, además de devolver el identificador generado asociado al jugador recientemente creado. En

caso de que los datos enviados no sean correctos, ya sea porque faltan campos por rellenar o sean incorrectos, el mensaje de respuesta será un código 400.

En la Figura 37 también podemos observar el resto de endpoints que tiene, las dos siguientes operaciones sirven para obtener una lista de jugadores adecuados para el equipo o jugador pasado por parámetro en la URI.

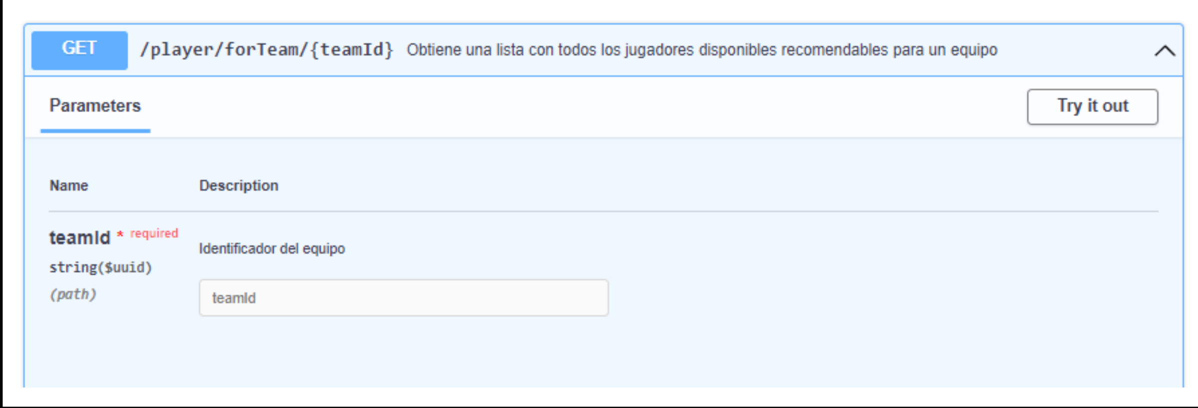

Figura 37.c. Endpoint de la entidad jugador que obtiene una lista de jugadores recomendados para un equipo.

En la Figura 37.c podemos ver el caso de la operación mencionada anteriormente para obtener una lista de jugadores que vayan acordes con el equipo pasado por parámetro.

Para recuperar un usuario podemos hacerlo de dos formas diferentes, a través de su identificador, o a partir de su nombre de usuario, puesto que este último es único. Estas operaciones vienen representadas por los endpoints GET /player/{playerId} y GET /player/nickname/{playerNickname}. También existen endpoints que nos permiten comprobar la existencia de un usuario con un email o nombre de usuario pasado por parámetro.

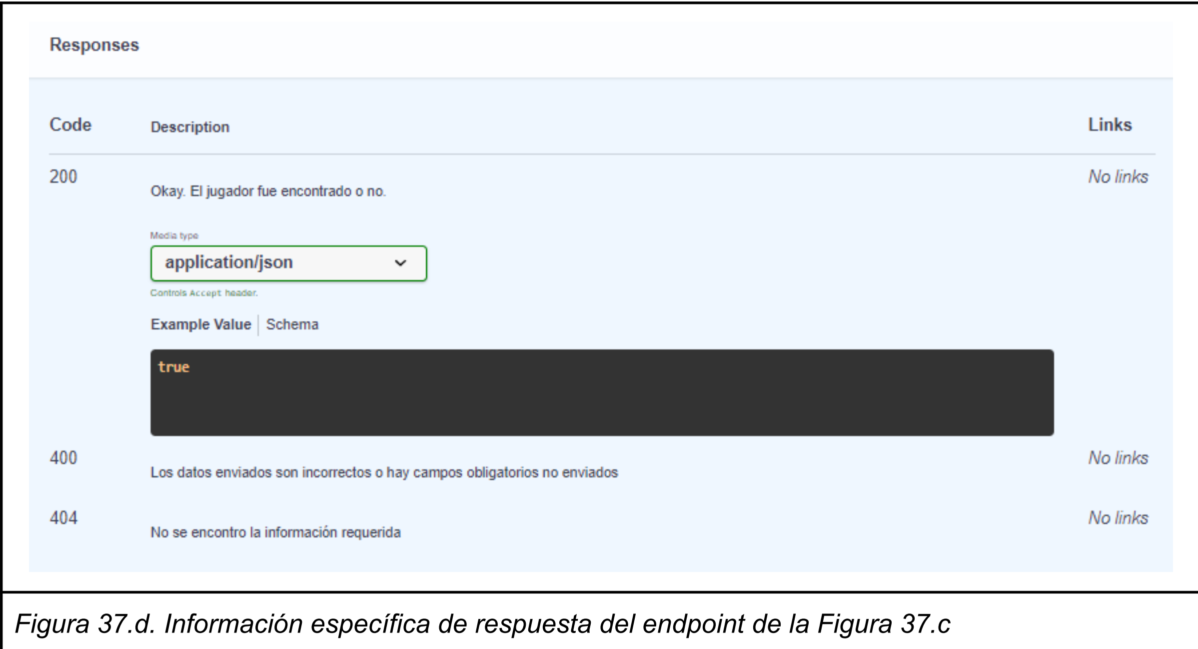

Figura 37.d. Información específica de respuesta del endpoint de la Figura 37.c

Como se puede observar en la Figura 37.d, para estos endpoints el valor devuelvo sería un valor booleano indicando la existencia de un jugador con dicho nombre de usuario o correo electrónico.

En el caso del método PUT, además del parámetro en la URI, será necesario introducir en el cuerpo del mensaje un JSON con los valores asociados a cada jugador, modificados. En caso de ser satisfactorio la respuesta será 200, junto a los valores del jugador modificado dentro del cuerpo de la respuesta. En caso contrario, los códigos serán 400 o 404, dependiendo si el error es debido a introducir valores incorrectos o no se encontró el jugador, respectivamente.

Figura 37.e. Endpoint de la entidad jugador que elimina un jugador a partir de su identificador.

Por último, en el método DELETE, ocurre de igual manera que con el método GET, y se puede observar en la Figura 37.e, ya que solo necesitamos el parámetro dentro de la URI para realizar la petición.

Para las operaciones PUT y DELETE, en caso de no introducir los parámetros necesarios o introducirlos incorrectamente, el mensaje de respuesta será 400, y en el caso de introducir un identificador válido pero no existente, el mensaje de respuesta será 404, indicando que no se encontró la información requerida.

A continuación definiremos las funciones relacionadas con los equipos, recurso que agrupa a una serie de jugadores en una sola entidad.

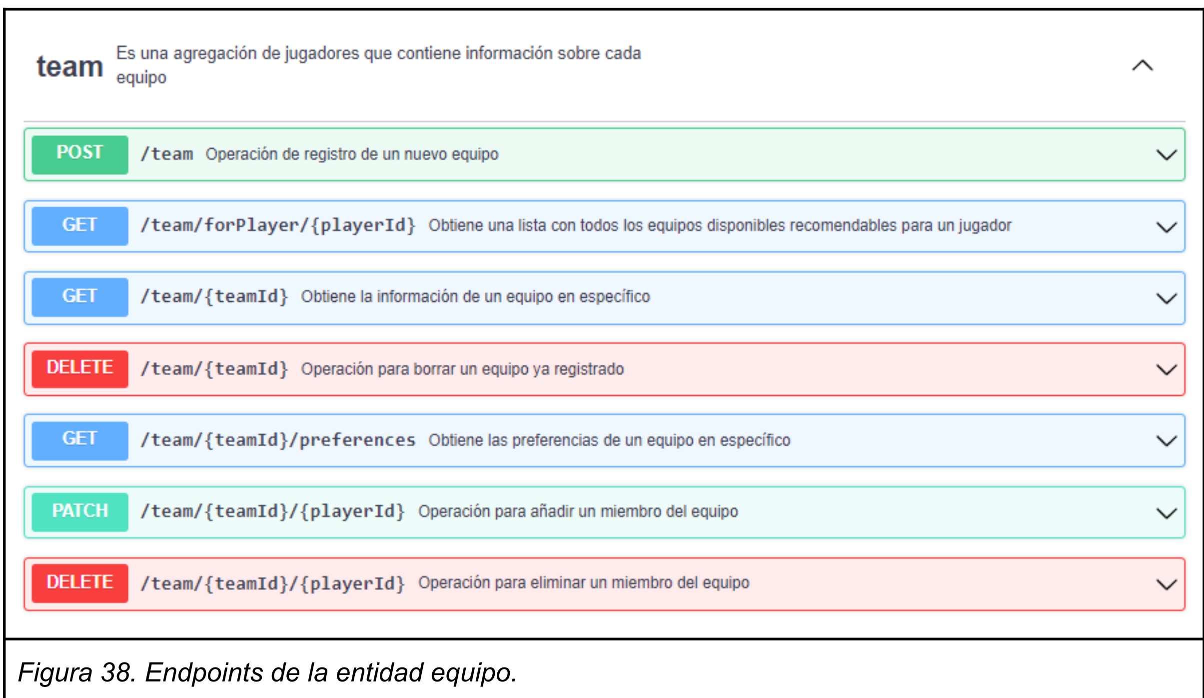

Figura 38. Endpoints de la entidad equipo.

Las operaciones POST, GET /team/forPlayer/{playerId}, GET /team/{teamId} y DELETE /team/{teamId} son similares a las mencionadas anteriormente en el caso del recurso jugador. Sin embargo, mencionaremos las principales diferencias entre ambos. Ya que los datos que almacena un equipo, no son los mismos que un jugador; un equipo almacena una lista de identificadores de los jugadores miembros del equipo, junto a un identificador separado que indicará el fundador del equipo, además del nombre del equipo.

Figura 38.a. Información asociada al endpoint POST de la entidad equipo para su registro.

En la Figura 38.a se muestran los datos que almacena un equipo, como se explicó anteriormente.

Para el caso de la operación DELETE /team/{teamId}/{playerId}, eliminará el miembro del equipo con el identificador pasado por parámetro. En el caso de ser eliminado, devolverá el equipo con las modificaciones apropiadas, en caso contrario devolverá el código de error correspondiente.

El método GET /team/{teamId}/preferences, devuelve las preferencias del equipo, generadas a partir de las preferencias de los jugadores, y contiene la siguiente información.

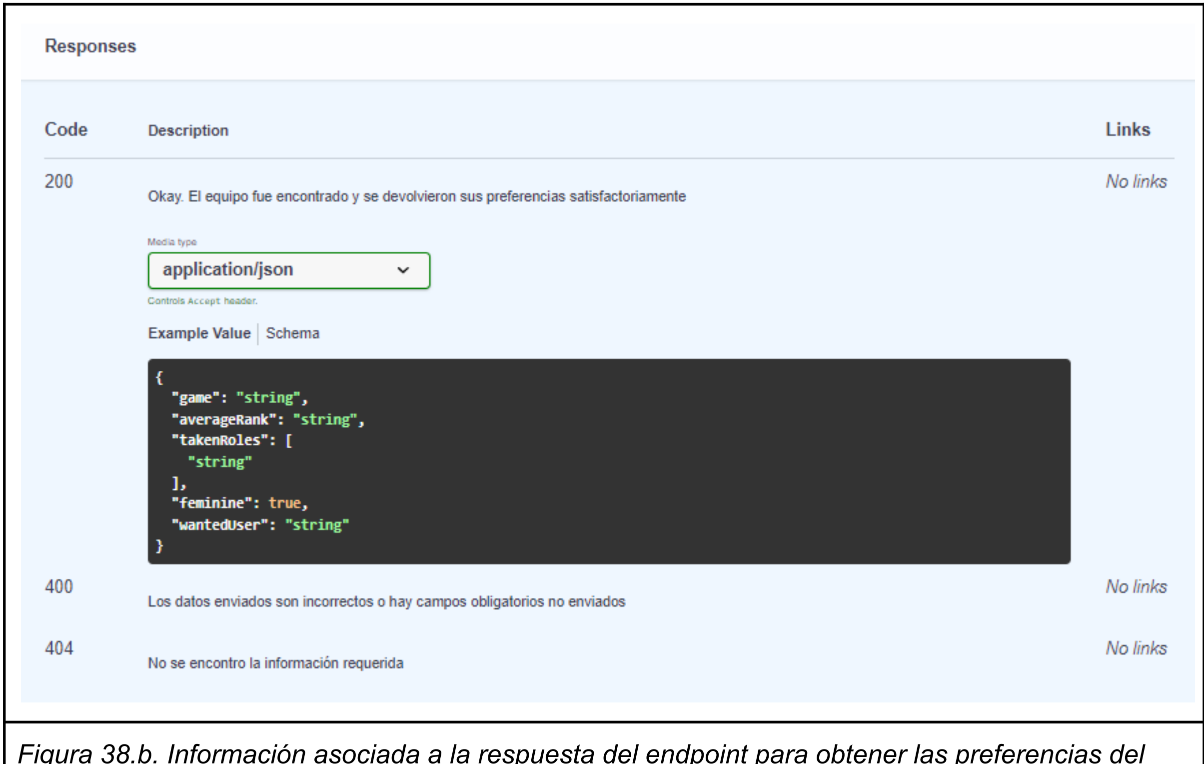

Figura 38.b. Información asociada a la respuesta del endpoint para obtener las preferencias del equipo.

En la Figura 38.b, se pueden observar los datos que se devuelven. Son similares a las preferencias de un jugador, sin embargo, devuelve un rango promedio del equipo, además de una lista de roles que ya fueron escogidos por los miembros del equipo.

El método PATCH hace referencia a una modificación parcial del recurso, ya que la acción que realiza es añadir un miembro al equipo. Funciona de forma similar al método DELETE correspondiente a eliminar un miembro del equipo, ya que este se indicará a través de la URI, y en caso de ser satisfactorio, devolverá el equipo con las modificaciones aplicadas.

En el caso de la entidad usuario, encontramos los siguientes métodos.

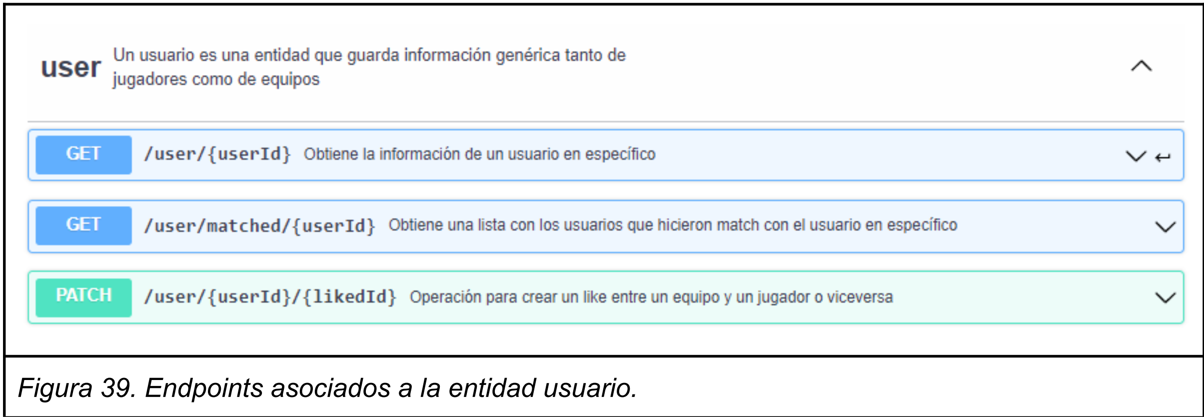

Figura 39. Endpoints asociados a la entidad usuario.

En la Figura 39 podemos ver los métodos asociados a la entidad usuario, la cual tiene endpoints para obtener tanto la información de un usuario específico, como para obtener una lista de usuarios que hicieron match con el usuario pasado por parámetro.

La funcionalidad para dar like, reside en el endpoint PATCH /user/{userId}/{likedId}.

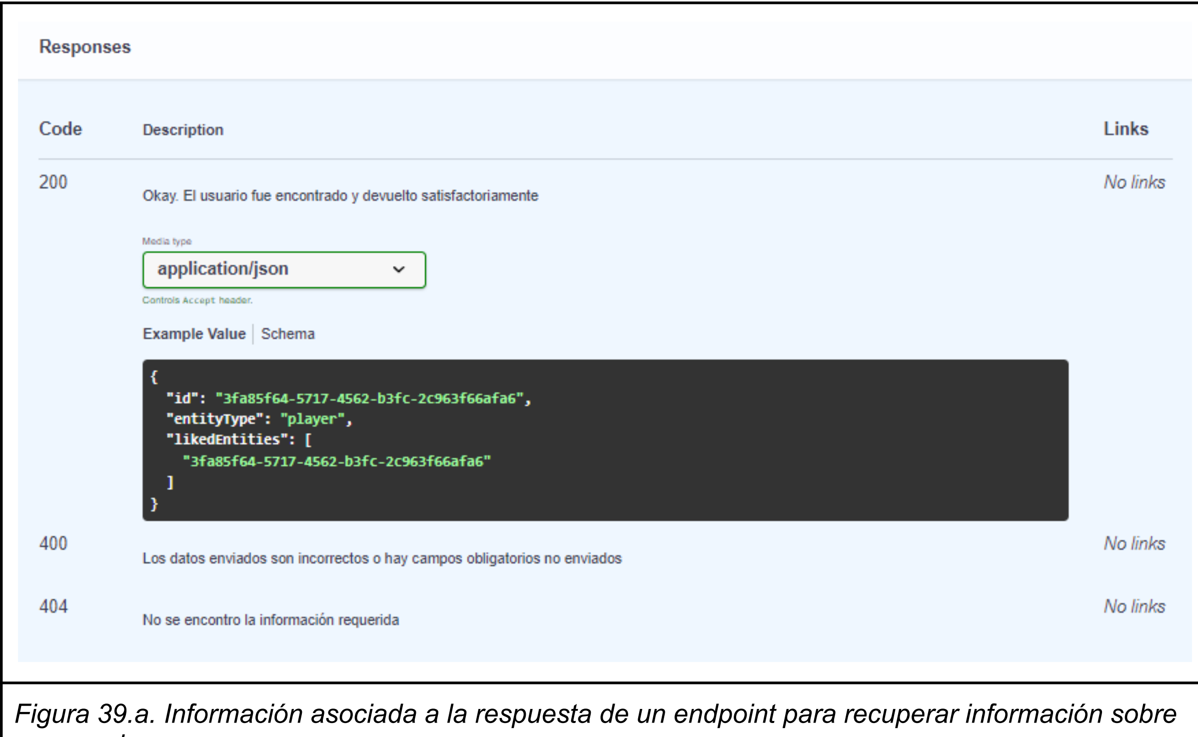

Figura 39.a. Información asociada a la respuesta de un endpoint para recuperar información sobre un usuario.

En la Figura 39.a podemos observar la estructura de un usuario, que guarda el tipo de entidad que es, ya sea un jugador o un equipo, además de una lista de identificadores de los usuarios que le dio like.

También existen una serie de endpoints para acceder a la entidad juego.

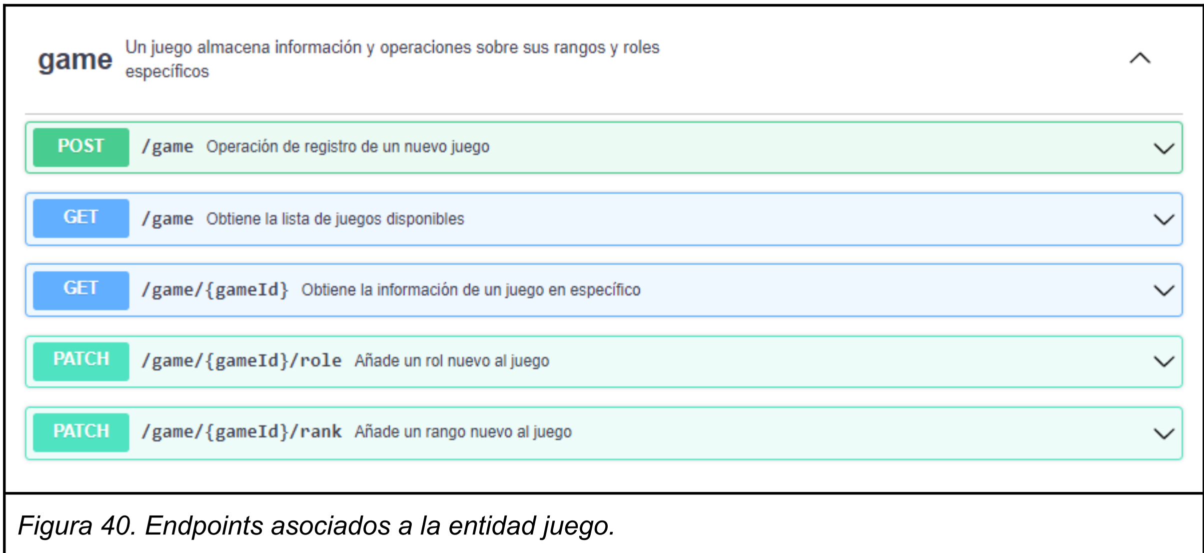

Figura 40. Endpoints asociados a la entidad juego.

Como se puede observar en la Figura 40, hay un método POST para crear nuevos juegos, dos métodos PATCH para añadir tanto roles como rangos, y dos métodos GET para acceder a la información.

Los métodos POST y PATCH son métodos no accesibles y protegidos, puesto que desde el cliente no se permite la funcionalidad de crear y modificar juegos.

Figura 40.a. Información asociada a la respuesta de un endpoint para obtener información sobre un juego.

La estructura que tiene un juego es la visible en la Figura 40.a, contiene un nombre, además de una lista de roles y rangos, los roles sólo almacenan el nombre, mientras que los rangos además del nombre, tienen asociado un valor que permite comparar qué rango es más alto que otro.

3.5. Plan de pruebas

En este apartado vamos a describir el proceso de testeo que realizaremos durante el desarrollo del sistema. Para nuestro plan de pruebas, nos centraremos en testear las funcionalidades definidas mediante las historias de usuario y el correcto funcionamiento de las mismas, tanto en la aplicación servidor como en el cliente.

El tipo de pruebas que realizaremos para nuestro proyecto serán pruebas de caja negra. Las pruebas de caja negra es una técnica para probar software en la cual la funcionalidad se verifica sin tomar en cuenta detalles de implementación, así como la estructura interna del código. En este tipo de pruebas nos enfocaremos solamente en las entradas y las salidas del sistema, sin entrar en detalles sobre el funcionamiento interno del software (Terrera 2017).

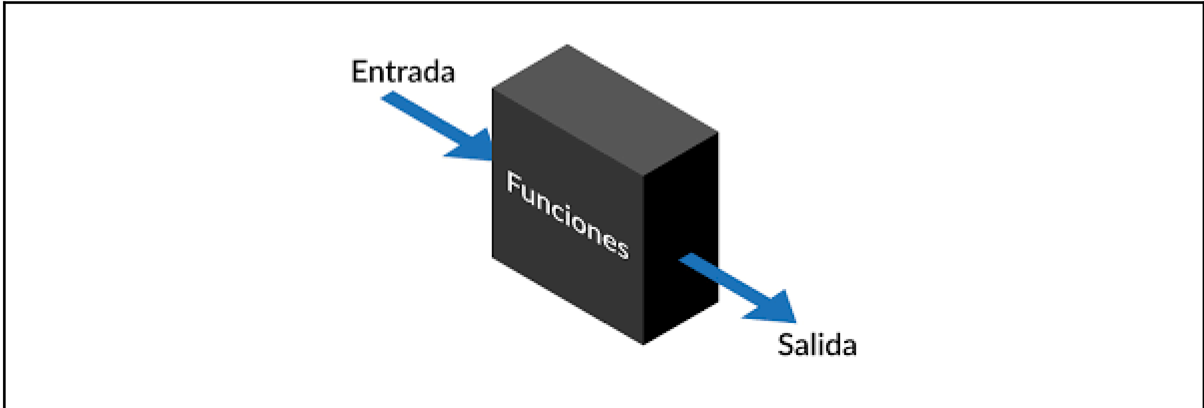

Figura 41. Representación gráfica de una prueba de caja negra. (Fuente: http://contenidos.sucerman.com)

Estas pruebas se realizan conjuntamente con el desarrollo del sistema, para ir comprobando de forma asidua el correcto funcionamiento de nuestro sistema. De esta manera, podemos partir de una base de nuestro sistema que funciona correctamente a medida que avanzamos con el desarrollo del mismo.

3.5.1. Plan de pruebas en la aplicación servidor

Para testear la aplicación servidor, realizaremos pruebas para cada uno de los endpoints de nuestra aplicación Rest. Esto nos permitirá interactuar con la interfaz que ofrece el sistema y comprobar que los resultados obtenidos son los correctos o esperados.

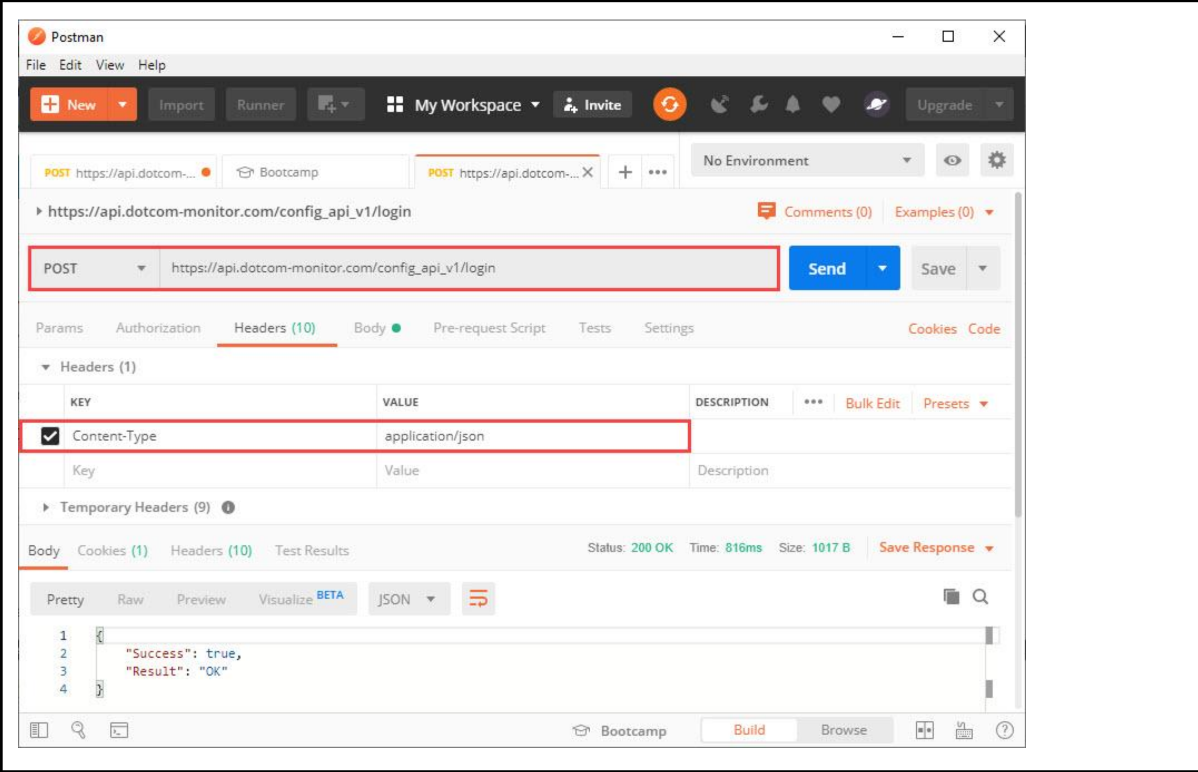

Figura 42. Ejemplo de funcionamiento de la aplicación Postman. (Fuente: https://www.dotcom-monitor.com)

Estas pruebas las realizaremos mediante piezas de software externas para probar endpoints como puede ser Postman[34], en la Figura 42 podemos observar un ejemplo de funcionamiento de dicha aplicación. En nuestra aplicación servidor, tenemos un total de 24 endpoints que tendremos que comprobar manualmente mediante el envío de peticiones HTTP. Para realizar las pruebas, comprobaremos tanto los comportamientos esperados, así como situaciones anómalas, definidas mediante los diferentes códigos de estado de respuesta.

[34] https://www.postman.com/

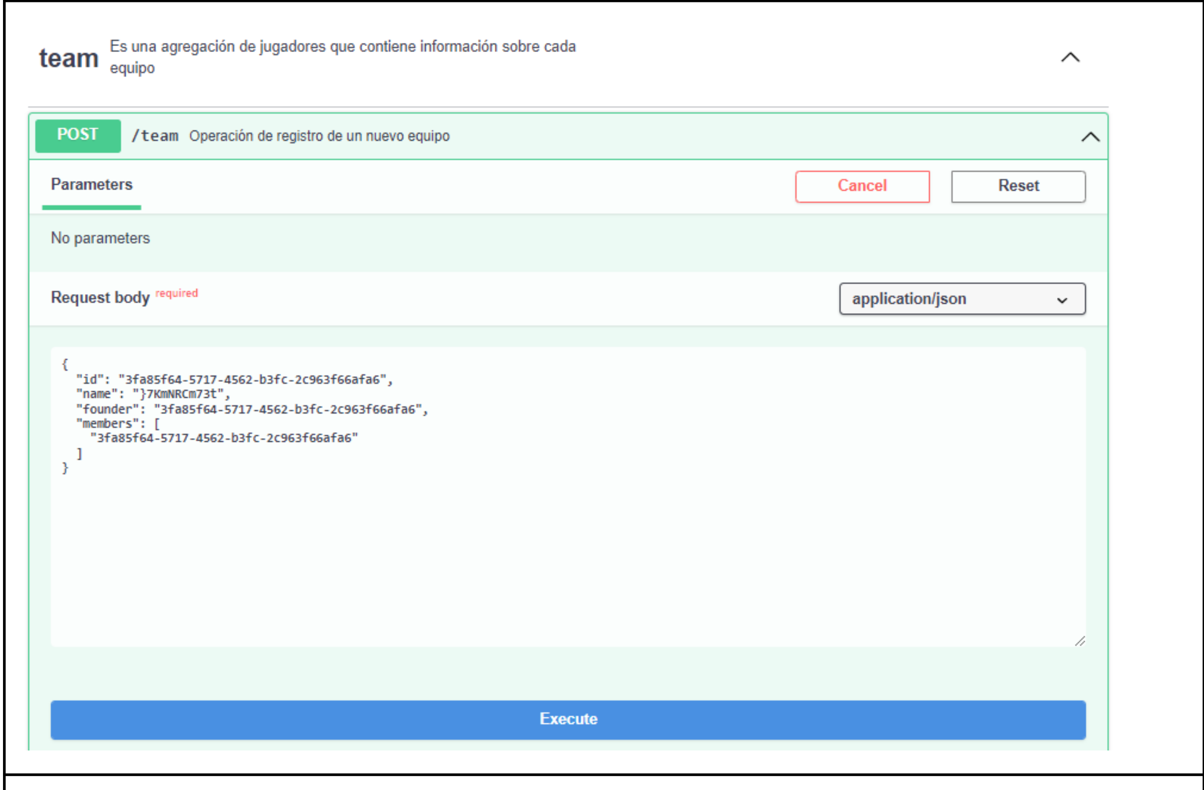

Figura 43. Ejemplo de funcionamiento de la interfaz del editor Swagger para el testeo de endpoints.

Como hemos mencionado anteriormente, para la documentación de nuestra API Rest, utilizaremos Swagger, sobre la especificación OpenAPI 3.0, la cual nos permitirá también realizar pruebas de testeo de igual manera que nos ofrece Postman, de forma integrada en la documentación de nuestra aplicación. En la Figura 43 podemos observar un ejemplo de funcionamiento del testeo que ofrece.

3.5.2. Plan de pruebas en la aplicación cliente

Para testear nuestra aplicación cliente, nos centraremos en realizar pruebas sobre la interfaz de usuario creada en la aplicación. Al terminar cada historia de usuario definida en cada una de las iteraciones propuestas, realizaremos una serie de pruebas donde probaremos las funcionalidades manualmente comprobando cada uno de los criterios de aceptación establecidos para cada una de las historias de usuario.

Estas pruebas se realizan al finalizar cada iteración y también al finalizar cada una de las historias de usuario, trabajando así sobre una base sólida para poder continuar con las siguientes historias de usuario.

4. Implementación

4.1. Arquitectura

En este apartado trataremos de mostrar una visión general del proyecto, mostrando aquellos elementos que lo constituyen, y la manera en la que se comunican entre ellos para formar el sistema completo.

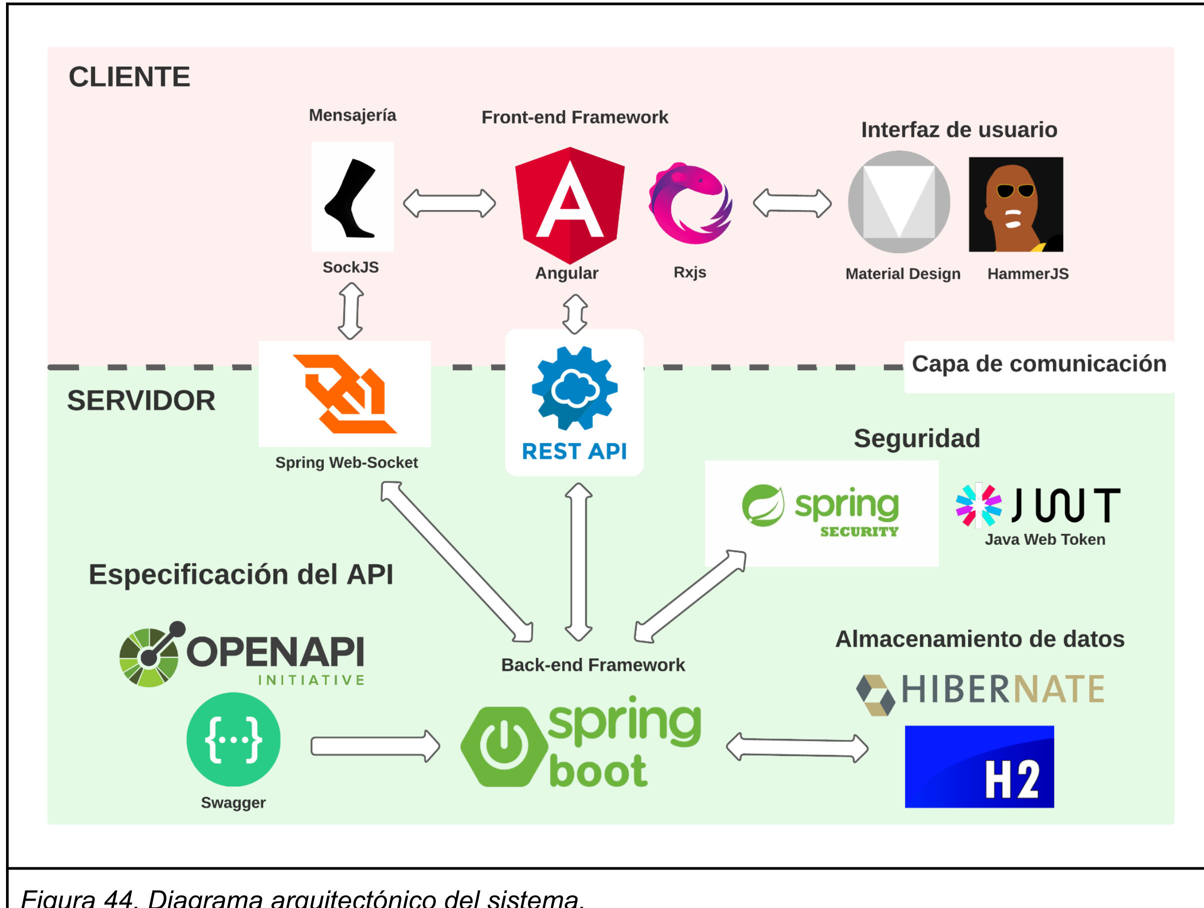

Figura 44. Diagrama arquitectónico del sistema.

Como podemos observar en la Figura 44, el diagrama arquitectónico se divide en dos grandes bloques, formando la arquitectura del cliente, y la arquitectura del servidor, dividida por una línea de comunicación, formando la interfaz entre cliente y servidor.

La arquitectura comienza en el servidor, específicamente en la documentación o especificación del API, creada mediante Swagger y OpenAPI, que juntos forman la especificación OpenAPI 3.0. OpenAPI define un lenguaje estándar, similar al lenguaje humano, para la especificación de APIs RESTful, sin la necesidad de acceder al código fuente. Proporciona además funcionalidades de interacción donde el usuario puede utilizar la especificación creada con un mínimo de lógica implementada. Esto nos servirá para acelerar el proceso de creación del API en

Spring, puesto que ofrece diversos plugins que facilitan la generación de código a partir de la especificación.

El código generado, y el framework a usar para el servidor, será Spring, más concretamente, Spring Boot, lo que nos permite crear aplicaciones autocontenidas, de forma que podamos enfocarnos únicamente en el desarrollo, delegando tareas como la configuración de dependencias, despliegue del proyecto, etc.

Spring contiene muchas funcionalidades de las que haremos uso en nuestra aplicación como puede ser la compatibilidad con el ORM Hibernate, creado para JPA, que nos simplificará el proceso del mapeo objeto-relacional y junto a una base de datos embebida como es H2, solventará el problema de la persistencia de datos en el lado del servidor.

Spring Boot contiene además varias dependencias adicionales de las que haremos uso, como puede ser Spring Security, que nos solventará el problema de la seguridad proporcionando una gran variedad de opciones para la autenticación, autorización, y demás características para la seguridad en aplicaciones empresariales. Acerca de la seguridad de nuestra aplicación, utilizaremos también un sistema de autenticación mediante web tokens. JWT es un estándar basado en JSON para la creación de tokens de acceso que permiten la propagación de identidad y privilegios.

De esta manera, podremos autorizar las diferentes peticiones HTTP provenientes del cliente, sabiendo de forma sencilla el usuario que las envía, y pudiendo así autorizar de forma correcta a los diferentes usuarios.

Para la validación de información utilizaremos tanto Bean Validation en Spring, como la validación de formularios que ofrece Angular, de esta manera, podremos limitar el envío de información maliciosa tanto en el cliente como en el servidor.

Angular es el framework que utilizaremos para el desarrollo en el cliente, basado en Typescript, es un marco de trabajo que nos proporciona muchas funcionalidades para el desarrollo de aplicaciones web. Angular se utiliza para la creación y mantenimiento de SPA[35]. Angular por dentro funciona con diferentes tecnologías como pueden ser HTML, CSS, Typescript, etc. Además, para el desarrollo de la aplicación en el cliente, utilizaremos Rxjs[36], una librería para programación reactiva que facilita la creación de código asíncrono, para ello, hace uso de los patrones Observer, Iterador y la programación funcional.

[35] Single-Page Application, o aplicaciones web de una sola página.
[36] https://rxjs.dev/

Sin embargo, para la creación de la interfaz gráfica en nuestra aplicación en el cliente, aprovecharemos el amplio catálogo de Material Design, una guía completa para el diseño visual e interactivo, diseñado para la visualización del sistema operativo Android, utilizado en numerosas aplicaciones web y en general, cualquier plataforma. Además de Material Design, utilizaremos HammerJS[37], para potenciar no solo nuestra interfaz de usuario, si no también la usabilidad por parte del usuario, añadiendo soporte a eventos táctiles para nuestra aplicación web, indispensables para una aplicación móvil.

Hasta ahora hemos tratado nuestro sistema explicando el envío de mensajes a partir de un API RESTful que implementa nuestro servidor, sin embargo, en nuestro sistema también existe la posibilidad de enviar mensajes mediante mensajería instantánea. Por lo que haremos uso de Spring WebSocket, una tecnología que hace posible abrir una sesión de comunicación entre el cliente y el servidor. Lo que permite enviar mensajes y recibir respuestas controladas por eventos sin tener que consultar al servidor para recibir una respuesta. En el lado del cliente, nos encontramos con SockJS[38], una librería de Javascript que nos permite unirnos al canal de comunicación creado por el servidor.

4.2. Detalles sobre implementación

Una vez mencionados los distintos componentes de la arquitectura del sistema, vamos a comentar aspectos metodológicos y técnicos sobre la implementación de los componentes mencionados. Diferenciaremos entre aspectos de implementación relacionados con el servidor y aquellos otros aspectos relacionados con el cliente para una mejor comprensión de la arquitectura del sistema.

4.3. Detalles de implementación en el servidor

En este apartado comentaremos aquellos aspectos técnicos que tienen que ver con el apartado del servidor.

4.3.1. Diagrama de paquetes

A continuación vamos a exponer mediante un diagrama de paquetes la estructura general en la que se organiza el servidor, para ofrecer una visión general y detallada de las clases que lo forman, proporcionando una guía sobre la estructura del sistema.

[37] https://hammerjs.github.io/
[38] https://github.com/sockjs/sockjs-client

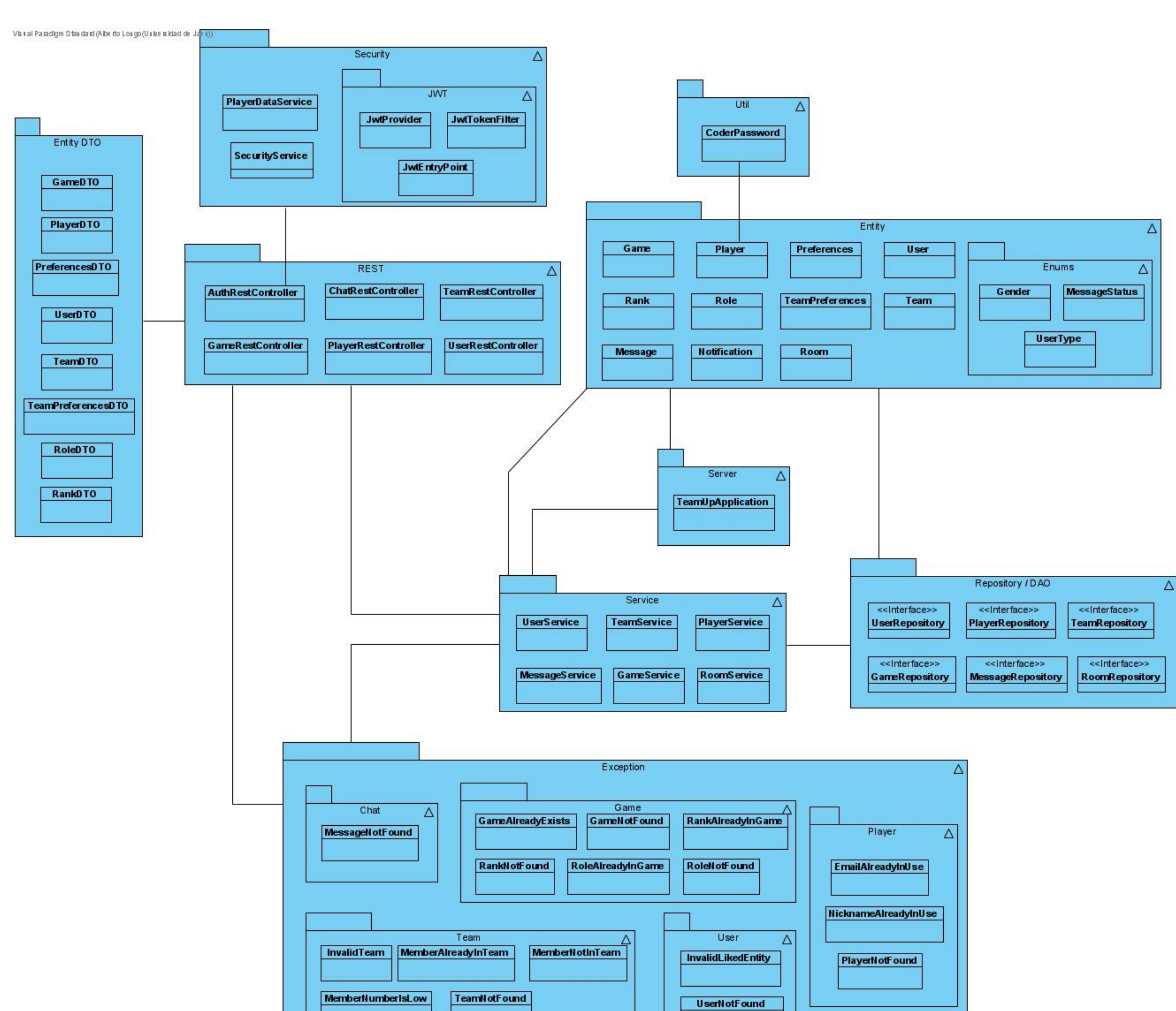

Figura 45. Diagrama de paquetes del servidor

En el diagrama de paquetes presente en la Figura 45 podemos visualizar cómo se interconectan los diferentes paquetes separados según su categoría. La conexión con la base de datos se establece en el paquete Repository / DAO, que encapsula el código relacionado con la persistencia de datos. Posteriormente tenemos los servicios, dentro del paquete Service, que encapsulan la lógica de negocio, por lo que hacen uso tanto de los repositorios mencionados anteriormente como de las entidades del dominio. Una vez definidas las funcionalidades de la lógica de negocio en los servicios, tendremos el *endpoint*[39] de la aplicación, siendo los servicios REST, encargados de utilizar la lógica de negocio de los servicios, y comunicar con los clientes mediante peticiones HTTP. A su vez, los servicios REST utilizan los DTOs para enviar la información necesaria a los clientes, en lugar de enviar la entidad completa. Si nos fijamos en el diagrama, veremos que no existe ningún DTO asociado al mensaje, esto se debe a que la clase que implementa los mensajes, es un POJO de por sí, de esta manera, no necesita utilizar un DTO adicional.

En el diagrama podemos apreciar un paquete que engloba una serie de excepciones para cada una de las entidades principales del sistema, esto es así para poder lanzar errores en la capa de la lógica de dominio, evitando la ambigüedad de excepciones genéricas.

Por último cabe mencionar que el apartado de seguridad de la aplicación queda englobado en el paquete Security del diagrama, aquí podemos ver como este engloba a su vez un paquete llamado JWT[40], encargado de contener las funcionalidades de seguridad mediante token[41].

[39] Es una interfaz expuesta por un comunicante o un canal de comunicación.
[40] JSON Web Token es un estándar abierto basado en JSON propuesto por IETF para la creación de tokens de acceso
[41] El token es una referencia (un identificador) que regresa a los datos sensibles a través de un sistema de tokenización.

4.3.2. Especificación del API Rest

Comenzaremos por la documentación de nuestra API Rest, creada con el editor de Swagger[42]. Esta especificación utilizará el estandar OpenAPI 3.0[43], siendo la evolución de la especificación anterior Swagger 2.0.

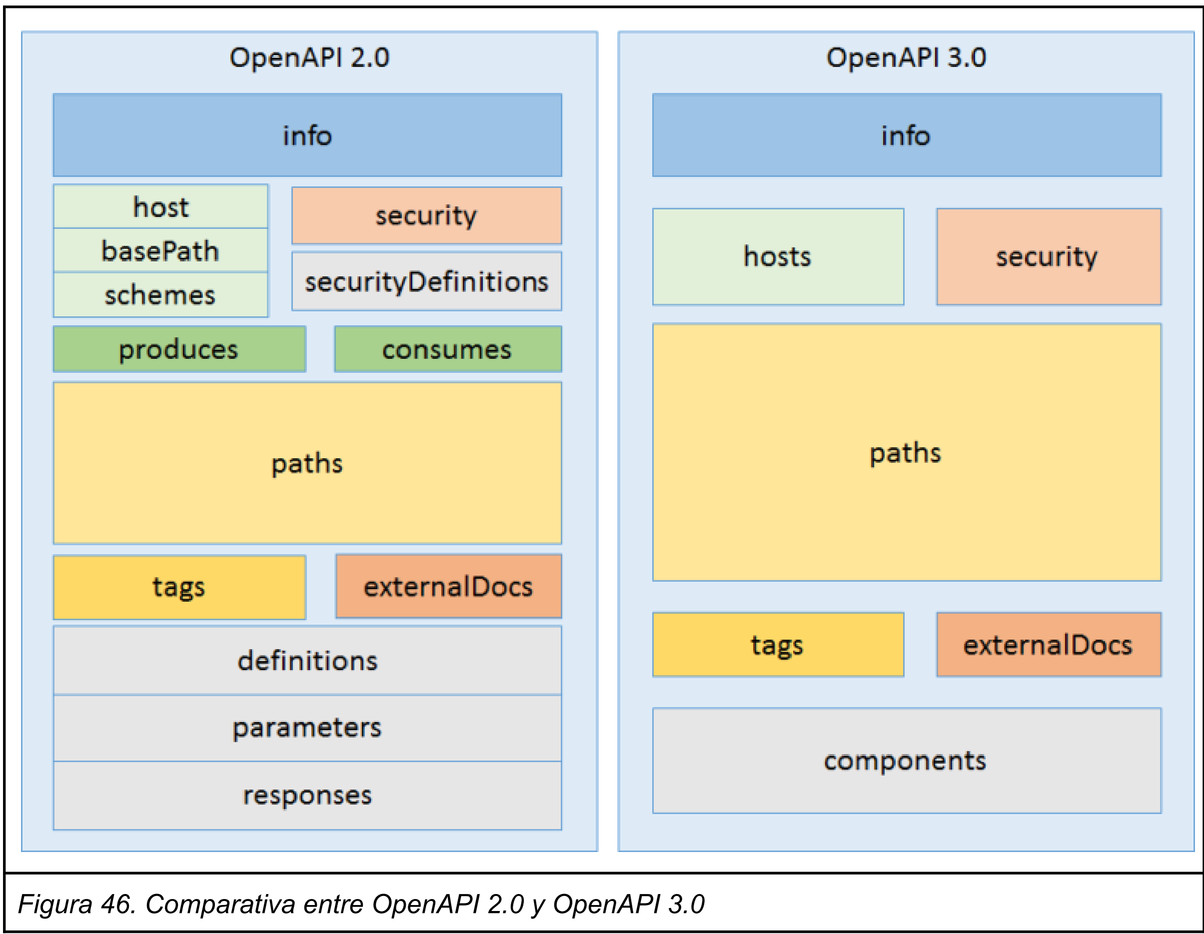

Figura 46. Comparativa entre OpenAPI 2.0 y OpenAPI 3.0

Observando la comparativa entre la versión 3.0 con la 2.0 de la Figura 46, la versión 3.0 incluye varias mejoras respecto a la anterior, aumentando la modularidad para reutilización de componentes, además de simplificar la estructura general del documento.

La primera parte de nuestro documento de especificación en Swagger contendrá información genérica del proyecto, como el nombre, descripción, contacto, y las diferentes entidades que tendrá la especificación.

[42] https://editor.swagger.io/
[43] https://swagger.io/specification/

Swagger Editor Supported by SMARTBEAR File ▾ Edit ▾ Insert ▾ Generate Server ▾ Generate Client ▾

```yaml
openapi: 3.0.0
info:
  title: Team Finder App
  description: >-
    Esta API se encarga de gestionar la información de la base de datos
    relacionada con la aplicacion Team Finder App.
  version: 1.0.0
  contact:
    email: alg00073@red.ujaen.es
tags:
  - name: game
    description: >-
      Un juego almacena información y operaciones sobre sus rangos y roles
      específicos
  - name: user
    description: >-
      Un usuario es una entidad que guarda información genérica tanto de
      jugadores como de equipos
  - name: player
    description: >-
      Esta entidad almacena información y operaciones específicas sobre cada
      jugador
  - name: team
    description: Es una agregación de jugadores que contiene información sobre cada equipo
```

Figura 47. Especificación inicial de la documentación del API.

Como podemos observar en la Figura 47, lo primero será definir la versión de la especificación que utilizaremos, en nuestro caso utilizaremos OpenAPI 3.0. Además, en la sección de tags, añadiremos las entidades que definiremos. Esto nos servirá para indicar a qué entidad pertenece cada uno de los endpoints que creemos.

```yaml
paths:
  /game:
    post:
      tags:
        - game
      summary: Operación de registro de un nuevo juego
      operationId: addGame
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/gameDTO'
        required: true
      responses:
        '200':
          description: Okay. El juego fue registrado satisfactoriamente
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/gameId'
        '400':
          $ref: '#/components/responses/BadRequest'
```

Figura 48. Definición de un endpoint POST para la entidad juego.

Aquí tenemos un ejemplo de endpoint, en este caso, para la entidad Juego. Podemos ver en la Figura 48 como estamos definiendo una petición de tipo POST, junto al nombre de la operación. Dentro del contenido, podemos indicar que se requiere un objeto tipo JSON con las características de un GameDTO. La palabra clave *$ref* nos permite hacer referencia a un componente definido al final de nuestro documento de especificación, para poder reutilizar la estructura en diferentes métodos.

Además de poder describir los elementos que deben de contener el cuerpo de la petición, podemos definir las distintas posibilidades de respuesta que tiene la petición. En este caso, podemos observar que puede obtener un código de respuesta 200 o OK, la cual indicará una petición satisfactoria, y devolverá el identificador de la entidad. También puede darse el caso de no ser satisfactorio, por lo que el servidor devolverá un código de respuesta 400 o *Bad Request*.

```
schemas:
  gameDTO:
    type: object
    properties:
      id:
        type: integer
        format: int64
        description: Identificador del juego
      name:
        type: string
        description: Nombre del juego
      roles:
        type: array
        description: Lista de nombres de los roles del juego
        items:
          $ref: '#/components/schemas/roleDTO'
        uniqueItems: true
      ranks:
        type: array
        description: Lista de nombres de los rangos del juego
        items:
          $ref: '#/components/schemas/rankDTO'
        uniqueItems: true
```

Figura 49. Componente reutilizable "GameDTO" que guarda la información sobre un juego.

En la Figura 49 podemos ver cómo es la estructura de un GameDTO. En la especificación podemos determinar el tipo, en este caso es un objeto, el cual tendrá distintas propiedades como: un identificador, que será un número entero de 64 bits, un nombre, que será una cadena de caracteres, un array de roles y un array de rangos, cuyos elementos son definidos por otro componente.

Figura 50. Interfaz que genera la especificación creada con OpenAPI 3.0

Este fichero de especificación nos generará una interfaz con la que podremos interactuar, con los elementos que hemos definido en la especificación. En la Figura 50 podemos ver como para la entidad Juego, tenemos distintos tipos de métodos disponibles junto a la ruta de acceso.

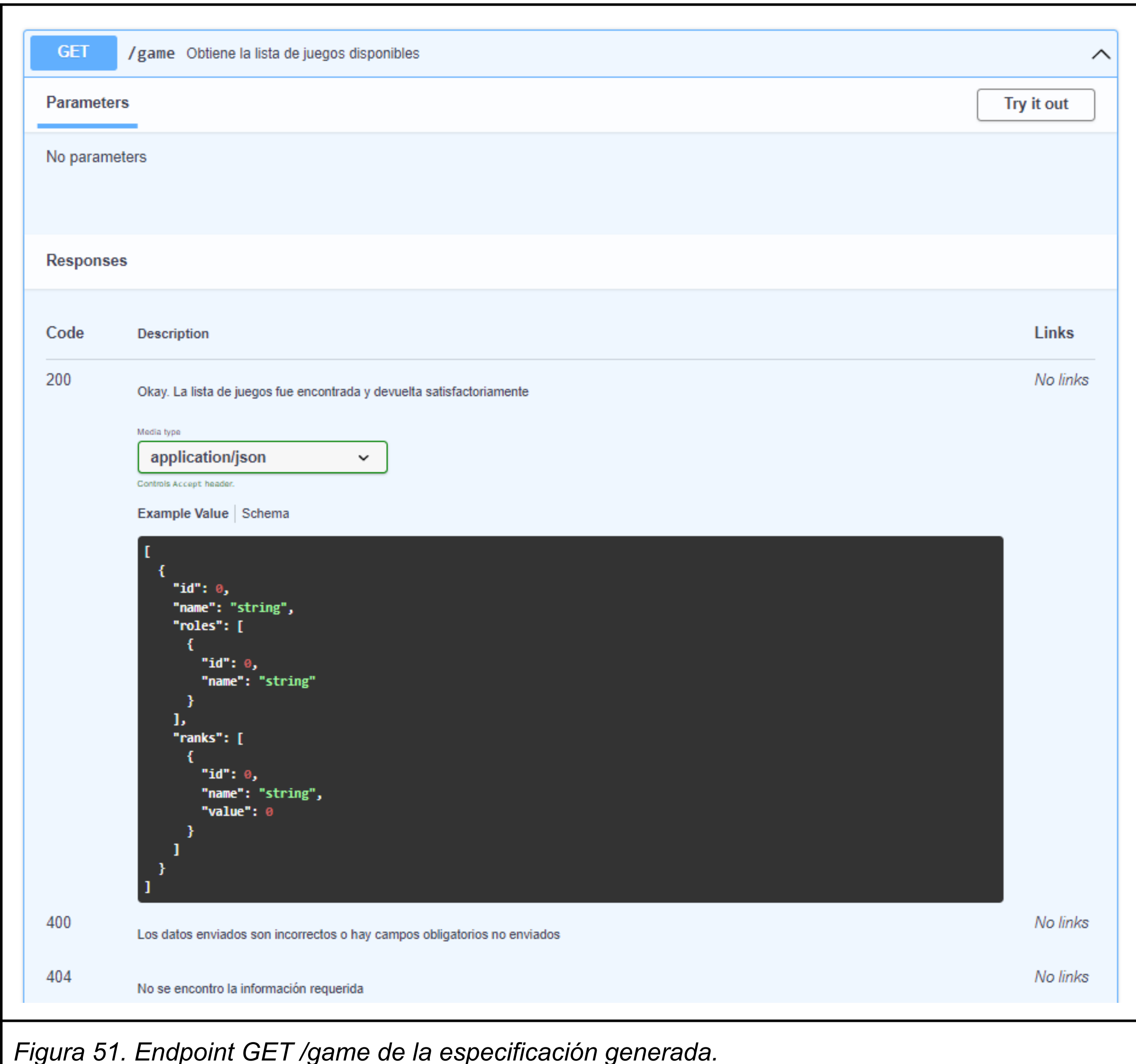

Figura 51. Endpoint GET /game de la especificación generada.

Si abrimos alguno de los métodos, nos aparecerá una descripción detallada sobre el funcionamiento del método, los parámetros necesarios, elementos a introducir dentro del cuerpo de la petición, así como las diferentes respuestas y elementos que devuelven.

En la Figura 51 podemos observar el método GET para obtener una lista de juegos disponibles, en este caso, no se requieren parámetros ni elementos dentro del cuerpo, y este devolverá un array con todos los juegos disponibles.

Finalmente, esta especificación la podremos descargar en formato YAML, para posteriormente generar código asociado a la documentación creada.

4.3.3. Generación del servidor a partir de la especificación

Aprovechando la documentación creada para el API Rest utilizando el estándar OpenAPI 3.0, podemos generar el código necesario para implementar el API a partir

de la documentación creada utilizando un plugin llamado OpenAPI Generator[44], que como su nombre indica, puede generar código a partir de la especificación utilizando OpenAPI. El plugin tiene soporte para muchos lenguajes de programación como C++ , C#, Java, PHP, Python, Ruby, etc.

En nuestro caso, al utilizar Java, generamos el código a partir de su plugin de Maven[45]. Para comenzar, tendremos que descargar el fichero .yaml de especificación de nuestra API. A continuación, añadiremos el plugin a nuestro archivo pom.xml, indicando la ruta donde hemos colocado nuestro archivo de especificación.

```
<plugin>
    <groupId>org.openapitools</groupId>
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <inputSpec>
                    ${project.basedir}/src/main/resources/static/api-v10.yml
                </inputSpec>
                <generatorName>spring</generatorName>
                <apiPackage>es.albertolongo.teamup.api</apiPackage>
                <modelPackage>es.albertolongo.teamup.model.dto</modelPackage>
                <supportingFilesToGenerate>
                    ApiUtil.java
                </supportingFilesToGenerate>
                <configOptions>
                    <sourceFolder>src/main/java/</sourceFolder>
                    <delegatePattern>true</delegatePattern>
                    <interfaceOnly>true</interfaceOnly>
                </configOptions>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Figura 52. Plugin Maven para generar el código del servidor asociado a la especificación.

[44] https://github.com/OpenAPITools/openapi-generator
[45] Maven es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl.

En la Figura 52 podemos observar que hemos incluido el archivo YAML en el apartado *inputSpec*. Además, como necesitamos un servidor basado en Spring, colocamos en el apartado *generatorName* el nombre de *spring*.

También debemos incluir la ruta de paquetes donde queremos que se genere la API, en nuestro caso, para el caso del API, indicaremos que se genere en la ruta “es.albertolongo.teamup.api”, mientras que las clases con el modelo, se generarán en la ruta “es.albertolongo.teamup.model.dto”.

Como estaremos generando un servidor en Spring sobre Spring Boot, también necesitaremos las dependencias de Spring Boot Starter Web y Spring Data JPA. Para las dependencias relacionadas con Spring Boot, utilizaremos la versión 2.6.4. Por último, también necesitaremos una serie de dependencias para permitir la correcta generación del código.

```
<!--DOCUMENTATION DEPENDENCIES-->
<dependency>
    <groupId>io.swagger</groupId>
    <artifactId>swagger-annotations</artifactId>
    <version>1.6.5</version>
</dependency>

<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.6.6</version>
</dependency>

<!--VALIDATION DEPENDENCIES-->
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
</dependency>

<dependency>
    <groupId>org.openapitools</groupId>
    <artifactId>jackson-databind-nullable</artifactId>
    <version>0.2.2</version>
</dependency>
```

Figura 53. Dependencias de Swagger y OpenAPI.

Como podemos observar en la Figura 53, las dependencias incluidas solventan los problemas de documentación del API, además de algunas dependencias para la validación de restricciones marcadas en la especificación.

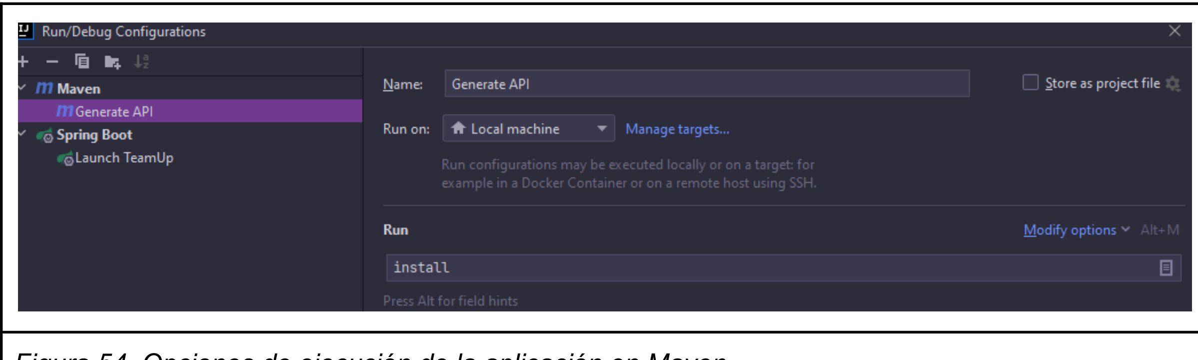

Figura 54. Opciones de ejecución de la aplicación en Maven.

Como podemos ver en la Figura 10, para generar el código solo resta ejecutar la aplicación con Maven indicando en las opciones *install*. Esto nos generará una serie de interfaces y de clases de las cuales podremos heredar para completar nuestra lógica de negocio.

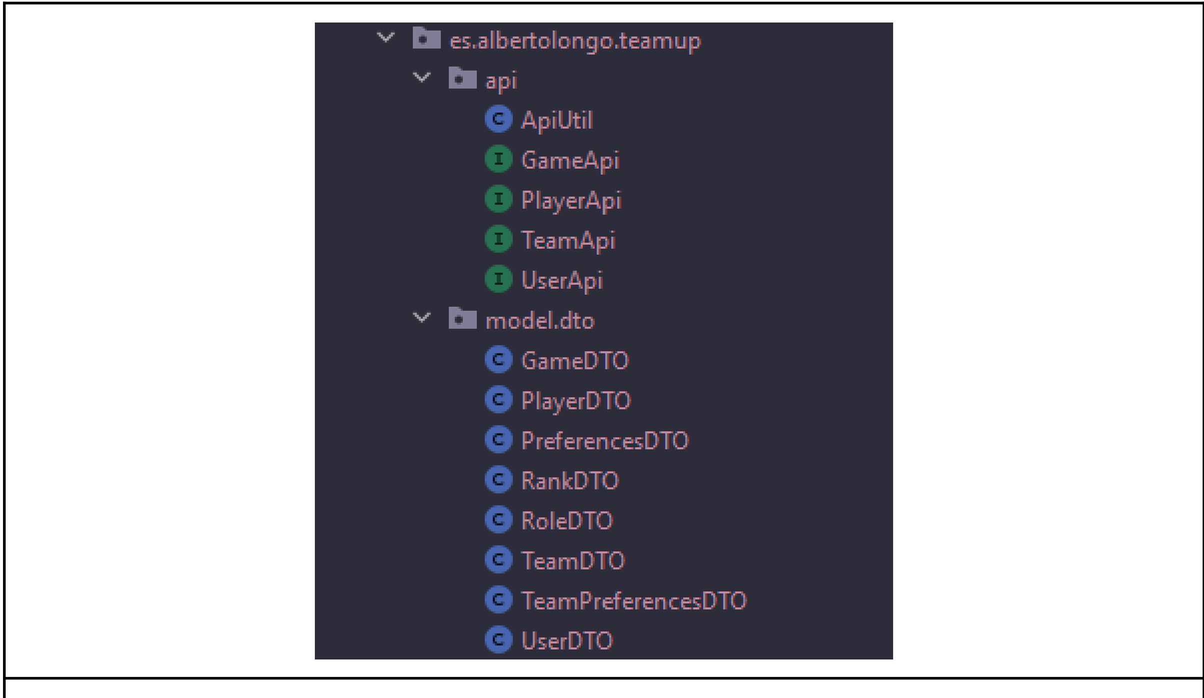

Figura 55. Archivos generados por la especificación OpenAPI utilizando el plugin generador.

Una vez generado el código, en la Figura 55 podemos ver como este nos implementa una serie de interfaces dentro del paquete api, que contienen el código relacionado con los diferentes endpoints, con la ruta asociada a cada uno de las entidades. Además también nos genera un paquete model, donde encontraremos los DTOs utilizados dentro de las interfaces.

4.3.4. Conexión del servidor a la base de datos

En este apartado comentaremos algunos aspectos de implementación de la base de datos en nuestro sistema. La base de datos que utilizamos en el proyecto es el sistema gestor de base de datos H2, la cual está embebida completamente en el ecosistema de Spring Boot, permitiéndonos así una fácil integración con el proyecto, sin necesidad de utilizar programas externos.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<!--DATABASE DEPENDENCIES-->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Figura 56. Dependencias de JPA y de la base de datos H2.

Como podemos observar en la Figura 56, necesitamos las dependencias del ORM utilizado por spring, siendo JPA, además de la dependencia de la base de datos H2. Una vez incluidas estas dependencias en nuestro proyecto, al arrancarlo se generará automáticamente un EntityManager, a través del cual nos permitirá acceder a los datos almacenados.

```
spring.datasource.url=jdbc:h2:mem:teamupdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=admin
spring.datasource.password=admin
#spring.jpa.show-sql=true
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
spring.h2.console.path=/h2
```

Figura 57. Archivo de configuración “application.properties” de Spring.

En la Figura 57 podemos ver el archivo de configuración de Spring, conteniendo la configuración necesaria para la base de datos. Algunas de las propiedades

indicadas en el fichero pueden ser omitidas, sin embargo, la URL, el nombre y la contraseña son fundamentales para el correcto funcionamiento de la base de datos.

Una vez creada la base de datos y configurada correctamente, ya está disponible para su uso a través de una inyección de dependencias del EntityManager. Sin embargo, nosotros haremos uso de una interfaz que nos proporciona JPA, la cual nos proporcionará de funcionalidades CRUD básicas, además de generar consultas simples como búsqueda por ciertos campos.

```
@Repository
public interface TeamRepository extends CrudRepository<Team, UUID> {

    Optional<Team> findByName(String name);

    @Query("SELECT t FROM Team t LEFT JOIN t.teamPreferences.takenRoles r WHERE " +
            "t.teamPreferences.game=:game AND " +
            "t.teamPreferences.averageRank.value>:minRankValue AND " +
            "t.teamPreferences.averageRank.value < :maxRankValue AND " +
            ":playerRole NOT IN r")
    Optional<List<Team>> findAllTeamsForPlayer(@Param("game") Game game,
                                               @Param("minRankValue") int minRankValue,
                                               @Param("maxRankValue") int maxRankValue,
                                               @Param("playerRole") Role playerRole);
}
```

Figura 58. Ejemplo de repositorio utilizando la interfaz CrudRepository de JPA.

En la Figura 58 podemos observar el repositorio que se encarga de almacenar y extraer los equipos, el cual solo necesita extender de la interfaz *CrudRepository*, proporcionada por JPA. Únicamente extendiendo la interfaz ya tendríamos toda la funcionalidad CRUD disponible además de varias consultas simples generadas dinámicamente. Estas consultas se crean a partir de una definición de consultas manuales a través de la cadena de caracteres que forma el nombre del método (Spring 2022). Además, también permite la creación de consultas complejas mediante el uso de anotaciones, insertando código SQL directamente en el código. En la Figura 14 podemos observar que el método *findAllTeamsForPlayer()* es una consulta compleja y no puede ser generado fácilmente por JPA. Este método nos devuelve una lista de equipos que sean adecuados según sus preferencias para el usuario que se indica por parámetro.

Esta interfaz de JPA nos ahorra tiempo a la hora de crear funcionalidades CRUD y consultas simples, permitiéndonos focalizar el tiempo de desarrollo en elementos no repetitivos.

4.3.5. Generación de los end-points

En este apartado vamos a abarcar como hemos estructurado el servidor para proveer de los diferentes servicios que ofrece.

4.3.5.1. Definición de los servicios

Una vez creados los repositorios para cada una de las entidades, crearemos los servicios, que se encargará de implementar funcionalidades relacionadas con la lógica de negocio, también convertirán aquellas excepciones genéricas en excepciones de lógica de negocio.

```
@Service
@Validated
public class PlayerService {

    @Autowired
    PlayerRepository playerRepository;
```

Figura 59. Ejemplo de una anotación de una clase servicio en Spring.

```
public Player getPlayerById(@NotNull UUID id) {

    Optional<Player> player = playerRepository.findById(id);

    if (!player.isPresent()) {
        throw new PlayerNotFound("Player not found");
    }

    return player.get();
}
```

Figura 60. Ejemplo de método de un servicio implementando la lógica de negocio.

Al utilizar Spring, cada uno de nuestros servicios, serán beans, por ello en la Figura 59, el servicio está anotado como Service, en esta misma figura podemos observar cómo se realiza una inyección de dependencias del repositorio de jugadores, para poder acceder a su funcionalidad y modificar los datos. En la Figura 60 podemos ver cómo se implementa la funcionalidad para obtener un jugador a partir de un identificador. En caso de no encontrarse, se lanzará una excepción acorde con un mensaje de error autoexplicativo.

4.3.5.2. Definición de los controladores REST

Una vez tengamos los diferentes servicios que nos proveen de funcionalidades relacionadas con la lógica de negocio, tendremos que establecer las clases frontera de nuestro sistema, que serán encargadas de implementar las interfaces generadas a partir de nuestra especificación.

```java
@RestController
@RequestMapping("/")
@CrossOrigin
public class PlayerRestController implements PlayerApi {

    @Autowired
    PlayerService playerService;
```

Figura 61. Ejemplo de anotación de un controlador REST en Spring.

```java
@Override
public ResponseEntity<UUID> addPlayer(PlayerDTO playerDTO) {
    UUID id = playerService.registerPlayer(playerDTO);
    return ResponseEntity.status(HttpStatus.OK).body(id);
}

@Override
public ResponseEntity<PlayerDTO> getPlayerById(UUID playerId) {
    Player player = playerService.getPlayerById(playerId);
    return ResponseEntity.status(HttpStatus.OK).body(player.toDTO());
}
```

Figura 62. Ejemplos de métodos de un controlador REST, implementando la interfaz creada por el generador OpenAPI.

En la Figura 61, podemos ver como nuestra clase *PlayerRestController*, implementa la funcionalidad generada en la interfaz *PlayerApi*. Además, podemos ver como mediante anotaciones de clase, hemos incluido la seguridad para el intercambio de recursos de origen cruzado o CORS[46] en la anotación *CrossOrigin*. También incluimos mediante inyección de dependencias nuestro servicio, para tener disponible las funcionalidades del mismo. En la Figura 62 podemos ver algunos de los métodos que implementan la interfaz PlayerApi, así como los códigos de respuesta enviados.

[46] Un mecanismo que utiliza cabeceras HTTP adicionales para permitir que un cliente obtenga permiso para acceder a recursos de un servidor, desde un origen distinto.

4.3.5.2.1. Gestión de excepciones

Como hemos podido observar en las figuras anteriores, en ningún momento se trataban con códigos de respuesta de error, y esto es así debido a que los códigos de respuesta de error son lanzados mediante la captura de las excepciones.

```
@ExceptionHandler({EmailAlreadyInUse.class, NicknameAlreadyInUse.class})
@ResponseStatus(HttpStatus.BAD_REQUEST)
public ResponseEntity<String> handlerAlreadyInUse(RuntimeException e) {
    return ResponseEntity.badRequest().body(e.getMessage());
}

@ExceptionHandler({PlayerNotFound.class, GameNotFound.class,
        RankNotFound.class, RoleNotFound.class})
@ResponseStatus(HttpStatus.NOT_FOUND)
public ResponseEntity<String> handlerNotFound(RuntimeException e) {
    return ResponseEntity.badRequest().body(e.getMessage());
}
```

Figura 63. Ejemplo de captura de excepciones y mapeado a códigos de respuesta dentro del controlador REST.

En nuestro controlador REST capturamos las excepciones de lógica de negocio, y devolvemos códigos de error relacionados como nos indica uno de los principios REST. En la Figura 63 podemos observar como para ciertas excepciones en la capa del dominio, se envían diferentes códigos de respuesta, por ejemplo, para las excepciones relacionadas con entidades no encontradas, el código de respuesta será 404 o *Not Found*.

4.3.6. Sistema de mensajería en el servidor

Para explicar el sistema de mensajería en el servidor lo trataremos en un punto aparte. Aunque a pesar de que guarda relación en muchos aspectos con el resto de endpoints, contiene detalles de implementación diferenciales que deben de ser mencionados.

De igual manera que el resto de entidades, tanto los mensajes como las salas están almacenadas en repositorios CRUD como hemos explicado anteriormente. También tienen servicios especializados para encargarse de encapsular los repositorios y ofrecer funciones relacionadas con la lógica de negocio.

```java
@Controller
public class ChatRestController {

    @Autowired
    private SimpMessagingTemplate messagingTemplate;

    @Autowired
    private MessageService chatMessageService;

    @Autowired
    private RoomService chatRoomService;
```

Figura 64. Controlador REST relacionado con el sistema de mensajería instantánea.

En la Figura 64 podemos ver el controlador del sistema de mensajería, el cual además de contener referencias a los distintos servicios creados, inyecta además la clase *SimpMessagingTemplate*, la cual implementa diferentes funciones relacionadas con la mensajería, proporcionada por Spring.

```java
@MessageMapping("/private")
public void processMessage(@Payload Message chatMessage) {
    var chatId :Optional<String> = chatRoomService
            .getChatId(chatMessage.getSenderId(), chatMessage.getRecipientId(), createIfNotExist: true);
    chatMessage.setChatId(chatId.get());

    Message saved = chatMessageService.save(chatMessage);
    System.out.println(saved.getRecipientId());
    messagingTemplate.convertAndSendToUser(
            chatMessage.getRecipientId(), destination: "/user/queue",
            new Notification(
                    saved.getId(),
                    saved.getSenderId(),
                    saved.getSenderName()));
}
```

Figura 65. Método dentro del controlador REST de mensajería encargado de recibir los mensajes y redireccionarlos.

En la Figura 65, podemos observar el endpoint principal del sistema de mensajería, que utiliza *WebSockets*. Para enviar mensajes a los diferentes usuarios hace uso del *SimpMessagingTemplate*, el cual envía los mensajes a los diferentes usuarios, dejándolos en su cola correspondiente privada. El resto de métodos de la clase controladora son endpoints HTTP como hemos explicado anteriormente.

4.4. Detalles de implementación en el cliente

En este apartado comentaremos los aspectos de implementación en el apartado del cliente más relevantes, con el objetivo de mejorar la comprensión general de cada uno de los subsistemas.

Para comenzar el desarrollo de un proyecto en Angular, haremos uso de Angular CLI[47], que contiene una serie de herramientas en línea de comandos que nos permiten empezar a desarrollar, añadir componentes y previsualizar la aplicación de forma rápida y sencilla.

4.4.1. Diagrama de paquetes

A continuación vamos a exponer mediante un diagrama de paquetes la estructura general en la que se organiza el cliente, para ofrecer una visión general y detallada de las clases que forman el sistema del cliente, así como la organización en carpetas que sigue, proporcionando una guía que sirva para orientar a aquellos que quieran trabajar sobre el proyecto.

[47] Angular CLI es una herramienta de interfaz de línea de comandos para gestionar y mantener un proyecto en Angular.

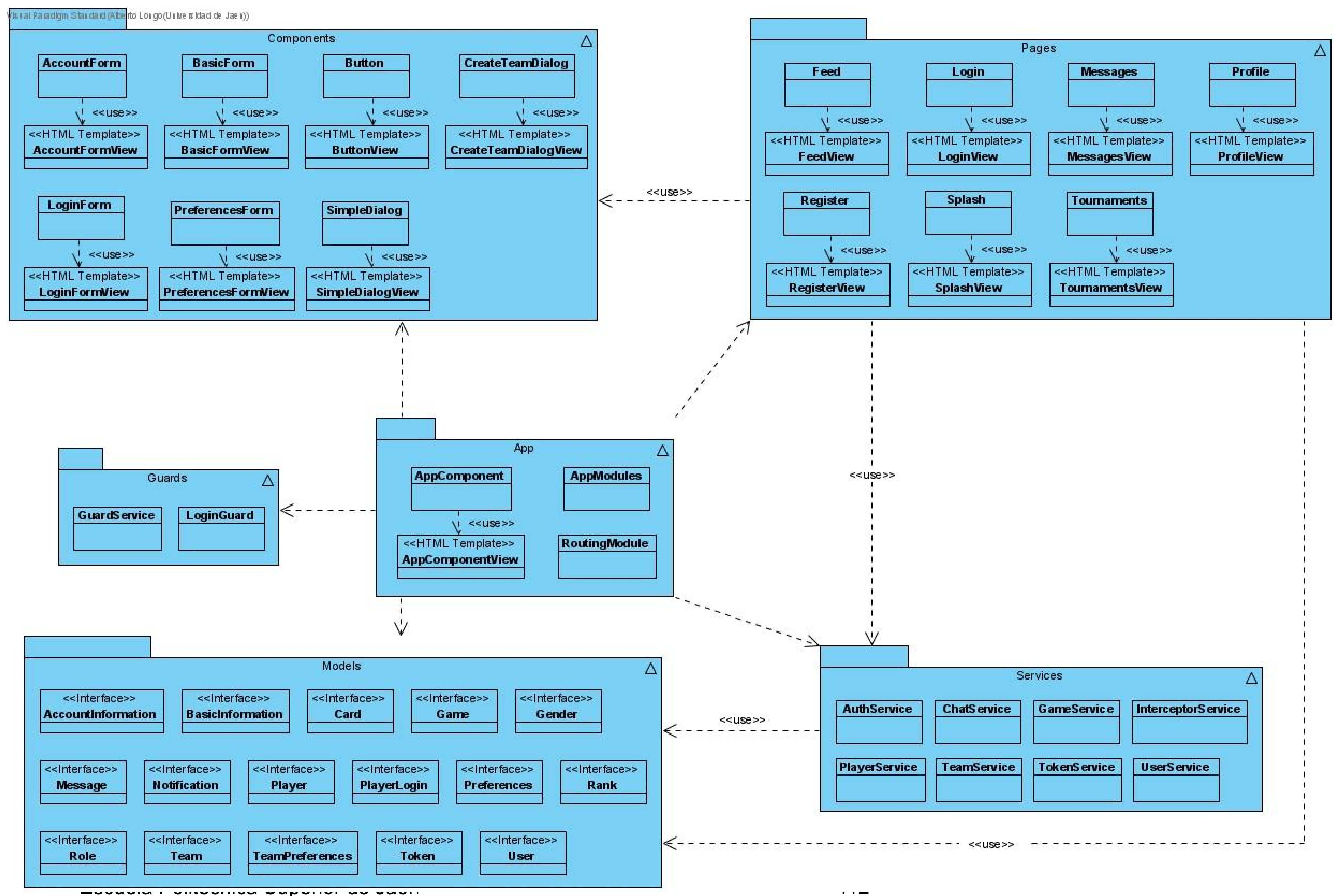

Figura 66. Diagrama de paquetes de la aplicación cliente.

Antes de nada, cabe destacar que la separación de los diferentes paquetes está hecha de tal manera que cada vista está almacenada junto a su controlador, agrupando el par vista controlador en el mismo paquete. Esto es una práctica habitual en proyectos que siguen esta arquitectura, puesto que de forma general, cada controlador se suele asociar únicamente con una sola vista, por lo que es común guardar los dos juntos, ya que representan la misma funcionalidad. Esta práctica hace que la visualización general del diagrama quede más limpia y fácil de entender.

El paquete Services, almacena clases encargadas de establecer la comunicación entre el cliente y el servidor, mediante el envío de peticiones HTTP hacia el servidor, que implementa el API Rest. Estas peticiones se encargan de transmitir y recibir la información proveniente del servidor, y son de carácter asíncrono, por lo que es un elemento a tener en cuenta para desarrollar el resto de funcionalidades en las vistas y controladores. Sin embargo, cabe destacar que en el envío de peticiones al servidor, estas deben estar protegidas, por lo que solo un cliente autorizado puede realizar el envío de peticiones relacionadas con su perfil. Esto se consigue mediante los sistemas de salvaguardas dentro del paquete *Guards*, que interceptan las peticiones enviadas al servidor y realizan una comprobación previa de autorización mediante el token activo.

También existe una copia del modelo existente en el servidor en la parte del cliente, puesto que al recoger la información de las diferentes entidades que existen en la aplicación, deberán de ser almacenadas y procesadas de la forma correcta. Este modelo está creado dentro del paquete *Models*, y contiene una serie de interfaces de acuerdo con los DTOs que llegarán desde el servidor.

4.4.2. Estructura de carpetas

En este apartado trataremos de mostrar la estructura de carpetas seguida en el proyecto de la aplicación cliente. La estructura que explicaremos a continuación se trata de la contenida dentro de la carpeta *src*[48] del proyecto, puesto que el resto de carpetas y archivos del proyecto son archivos relacionados con archivos de configuración global del proyecto así como los diferentes módulos de node[49].

[48] Acrónimo de *source*, en español significa código fuente.
[49] Node.js es un entorno en tiempo de ejecución multiplataforma basado en Javascript.

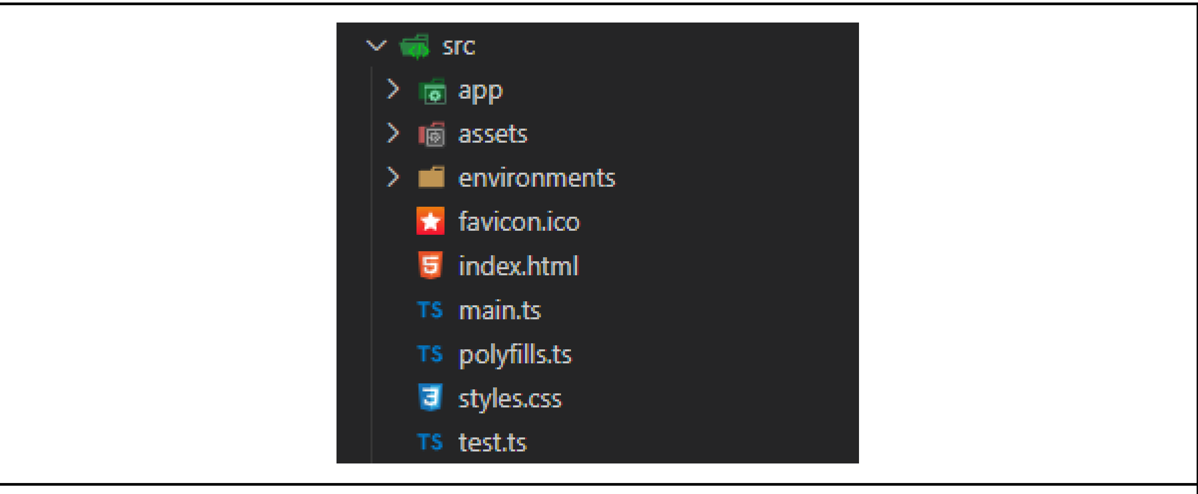

Figura 67. Directorio de carpetas raíz del proyecto de la aplicación cliente.

Dentro de la carpeta *src*, como podemos observar en la Figura 67, encontramos la carpeta app, la cual contendrá los componentes y servicios de la aplicación, junto a los diferentes servicios. También tenemos la carpeta *assets*, la cual guarda aquellos elementos gráficos utilizados en el proyecto, como las imágenes utilizadas en las vistas, iconos, etc.
La carpeta *enviroments* guarda elementos de configuración global del proyecto, que son las variables de entorno. A continuación tenemos diversos archivos como el archivo html principal, el icono de la aplicación, archivo de estilos globales, etc.
Ahora mostraremos la estructura de la carpeta donde se encuentra la mayor parte del desarrollo del proyecto: la carpeta *app*.

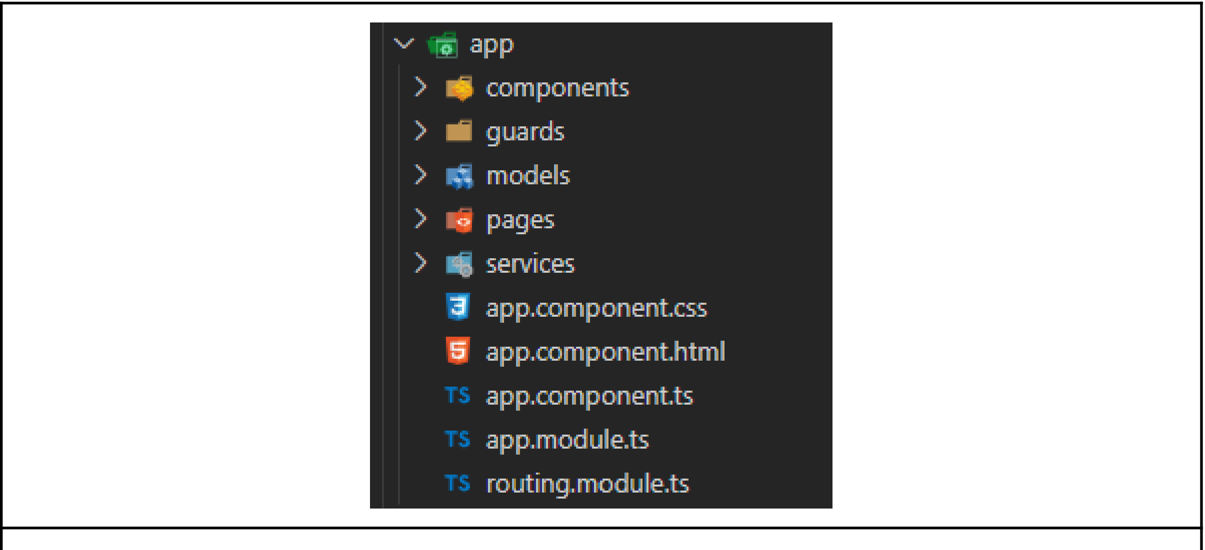

Figura 68. Directorio de carpetas en /app de la aplicación cliente.

En la Figura 68 podemos observar la estructura que seguimos para organizar el proyecto, esta divide los elementos en: componentes reutilizables, dentro de la carpeta *components*; los salvaguardas dentro de la carpeta *guards*; las interfaces con la que obtendremos y enviamos datos con el servidor, similares a las clases

DTO, dentro de la carpeta *models*; los componentes principales, que serán composiciones de componentes más pequeños, formando así las distintas vistas de la aplicación, dentro de la carpeta *pages* y, por último, dentro de la carpeta *services*, nos encontraremos con los servicios que nos permiten establecer comunicación con el servidor, enviando y recibiendo peticiones HTTP.

Además, tenemos el componente principal de la aplicación, que es el componente *app*, sobre el cual irán se montaran el resto de componentes. También existe un archivo *app.module.ts*, el cual guardará referencia a todos los componentes creados en la aplicación, además de incluir también los módulos utilizados de librerías externas como Material Design, etc.

Por último, nos encontramos con otro módulo, llamado *routing*, como se puede observar en la Figura 68. Este se encargará de gestionar el componente *router* de la aplicación, el cual activará uno u otro componente dependiendo de la ruta establecida en la URL, además de establecer datos de seguridad como los roles necesarios, así como establecer la salvaguarda utilizada en cada una de las vistas.

Debido al enrutamiento, hemos dividido los componentes creados en la aplicación en componentes reutilizables, dentro de la carpeta *components*, y componentes que definan la vista de cada una de las rutas dentro de la carpeta *pages*, aglutinando los componentes pequeños y guardando la lógica de dicha vista. De esta manera, al asignar componentes a las distintas rutas dentro del módulo *routing*, asignaremos una ruta diferente para cada uno de los componentes existentes dentro de la carpeta *pages*.

4.4.3. Estructura de un componente

A continuación, comentaremos la estructura que existe para cada componente, lo que permitirá clarificar la estructura general que sigue la aplicación. En Angular, un componente es un elemento que está organizado en un archivo HTML[50] que contiene los datos de visualización del componente, , un archivo de estilos CSS[51], con el que modificaremos su apariencia y, por último, un archivo de lógica en Typescript, que contendrá una clase con la implementación de las distintas acciones que se pueden realizar sobre la vista.

Como ejemplo, utilizaremos el componente relacionado con el perfil, puesto que contiene bastante complejidad, y nos permite explicar las funcionalidades existentes y cómo se integran dentro del componente.

---

[50] HTML, siglas en inglés de HyperText Markup Language, hace referencia al lenguaje de marcado para la elaboración de páginas web.
[51] CSS, en español “Hojas de estilo en cascada”, es un lenguaje para definir y crear la presentación de un documento escrito en un lenguaje de marcado (HTML o similar).

4.4.3.1. Estructura de la vista

Comenzaremos explicando la estructura de la vista, indicando algunas de las directivas comunes más utilizadas a lo largo del proyecto.

```
<div *ngIf="dataLoaded | async" class="container">
  <div class="header">
    <div>
      <img
        loading="lazy"
        *ngIf="player.gender === 'MASC'"
        src="assets/images/male_avatar.svg"
        class="avatar"
        alt=""
      />
      <img
        *ngIf="player.gender !== 'MASC'"
        src="assets/images/female_avatar.svg"
        class="avatar"
        alt=""
      />
    </div>
    <div>
      <h1 class="header-nickname">Profile of {{ player.nickname }}</h1>
      <h3 class="header-fullname">{{ player.fullname }}</h3>
    </div>
  </div>
```

Figura 69. Encabezado del fichero HTML de la vista perfil.

En la Figura 69 podemos observar el encabezado de nuestra vista perfil, el cual muestra una imagen u otra en función del género del jugador. A esta información se accede a través de la directiva *ngIf*, la cual nos permite incluir expresiones condicionales, y dentro de esta expresión accedemos a la información presente en el archivo de lógica asociado con la vista, en este caso, al género. También podemos observar como a continuación se muestra el nombre de usuario del jugador, al igual que su nombre completo. La sintaxis presente de la doble llave se llama interpolación de strings, y permite colocar expresiones dentro de la estructura de la vista, en este caso, para la expresión *{{ player.nickname }}* colocará el valor que contiene el campo *nickname* del jugador.

En la Figura 3 también se puede observar como al principio de nuestra vista existe un *div* que engloba a la totalidad del componente, y este a su vez contiene una directiva *ngIf*, la cual solo renderiza el elemento en caso de que la expresión sea verdadera. Esto nos permite realizar una carga asíncrona del componente, evitando

así cargar el componente hasta que toda la información necesaria para la visualización haya sido recogida del servidor.

```
<form [formGroup]="accountForm">
  <div class="form">
    <mat-form-field appearance="legacy">
      <mat-label>Your nickname</mat-label>
      <input
        matInput
        formControlName="nickname"
        required
        maxlength="16"
      />
      <mat-error *ngIf="getNicknameErrorMessage()">
        {{ nicknameErrorMessage }}
      </mat-error>
    </mat-form-field>
    <mat-form-field appearance="legacy">
      <mat-label>Your email</mat-label>
      <input matInput formControlName="email" required />
      <mat-error *ngIf="getEmailErrorMessage()">
        {{ emailErrorMessage }}
      </mat-error>
    </mat-form-field>
  </div>
</form>
```

Figura 70. Etiqueta form junto a su contenido dentro de la vista perfil.

En la Figura 70 podemos observar la estructura de un formulario dentro de nuestro componente perfil. En este caso, el formulario hace referencia a la información relacionada con la cuenta del jugador, como su nombre de usuario y su correo electrónico. Cada campo del formulario está envuelto en una etiqueta *mat-form-field*, que nos permite estilizar nuestro formulario, además de proporcionar utilidad de validación como podemos comprobar en los atributos dentro de la etiqueta *input*. En caso de que el formulario tenga un error, la etiqueta *mat-error* será visible y visualizará el error correspondiente.

```
<mat-action-row>
  <button
    [disabled]="accountForm.invalid"
    mat-button
    color="primary"
    (click)="editAccountInformation()"
  >
    <mat-icon>edit</mat-icon>
    Edit
  </button>
  <button mat-button color="primary" (click)="logOut()">
    <mat-icon>logout</mat-icon>
    Logout
  </button>
  <button mat-button color="warn" (click)="deletePlayer()">
    <mat-icon>delete</mat-icon>
    Delete
  </button>
</mat-action-row>
```

Figura 71. Botonera con ejemplos de enlazado de datos en Angular.

En la Figura 71, podemos ver la estructura de unos botones, los cuales podemos indicar si están deshabilitados o no con la directiva *disabled* entre corchetes. Esto habilita un *binding*, que realiza un flujo de información desde el modelo hacia la vista. También existen otros tipos de *binding*, en la cual la vista se comunica con el modelo, como es el caso de la directiva *click*, que junto a los paréntesis, marcan un evento, el cual dirigen la dirección desde la vista hacia el modelo. Si utilizamos los corchetes y los paréntesis sobre la misma directiva, se habilitará un doble *binding*.

Por lo tanto, en el caso de la directiva *disabled*, el valor guardado en la variable *accountForm.invalid* indicará si el botón está activado o no, y será el modelo es quien indica cuando cambia. Sin embargo, en la directiva *click*, cuando el usuario pulse sobre el botón, será la vista la encargada de enviar la información al modelo, llamando a la función correspondiente.

4.4.3.2. Estructura de los estilos

A continuación mostraremos brevemente la estructura del archivo de estilos de un componente.

```css
.header {
  margin: 32px 0px;
  display: flex;
  flex-direction: row;
  justify-content: center;
  align-items: center;
  flex-wrap: wrap;
  gap: 25px;
}

.header-nickname {
  margin: 0;
  font-size: 2rem;
  font-weight: bold;
}

.header-fullname {
  margin: 0;
  font-size: 1.5rem;
  font-weight: 100;
}
```

Figura 72. Archivo de estilos CSS utilizados para la vista perfil.

En la Figura 6 podemos observar algunas de las clases de estilos definidas en el fichero CSS del componente. Estas clases, junto a las propiedades que contienen, se utilizan para diseñar y dar estilo al contenido. En la Figura 6, podemos ver la primera clase *header*, que utilizaremos para estilizar el encabezado del perfil. En este caso, el objetivo principal de esta clase es colocar el contenido de manera centrada, aplicando además un margen y cierto espacio entre los elementos.

Para estilizar la aplicación, haremos uso de una librería externa de componentes llamada *Material Design*. Esta librería nos facilitará el trabajo de diseñar y construir componentes reutilizables tales como menús desplegables, además de proporcionarnos interfaces que unen los diseños web y de aplicaciones móviles, todo junto en una sola librería. Por ello, los ficheros de estilos creados de forma manual, los utilizaremos para adaptar principalmente los componentes que *Material* nos ofrece a nuestra visión de producto final.

4.4.3.3. Estructura de la lógica

En Angular, cada componente lleva asociado un fichero de lógica, el cual almacena los datos necesarios para visualizar el componente, además de implementar la lógica detrás de la interacción con la vista.

```
@Component({
  selector: 'app-profile',
  templateUrl: './profile.component.html',
  styleUrls: ['./profile.component.css'],
})
export class ProfileComponent implements OnInit {
  // Data
  player: Player;
  hasTeam: boolean = false;
  playerTeam: Team;
  teamPreferences: TeamPreferences;
  teamMembers: Player[] = [];
  selectedMember: Player;
```

Figura 73. Decorador de la clase que actúa como controlador de la vista perfil.

En la Figura 73 podemos observar cómo cada componente tiene asociada una clase específica. Esta clase en Angular viene acompañada de un decorador *Component* que contiene información sobre el componente, así como la localización de la vista, en la propiedad *templateUrl*, así como el archivo de estilos asociado en la propiedad *styleUrls*. También el decorador tiene una propiedad *selector*, que servirá para poder llamar al componente desde otros componentes, en este caso, su etiqueta será *<app-profile/>*.

A continuación podemos incluir toda la lógica que necesitemos dentro de nuestro componente.

```
deletePlayer(): void {
  const dialogRef = this.dialog.open(SimpleDialogComponent, {
    data: {
      primaryText: "You're about to delete your account",
      secondaryText: 'This action will be irreversible',
      tertiaryText: 'Are you sure you want to delete your account?',
      buttonText: 'Delete',
    },
  });

  dialogRef.afterClosed().subscribe((result) => {
    console.log(result);

    if (result == undefined) return;

    this.playerService.deletePlayer(this.player.id!).subscribe(() => {
      this.tokenService.logOut();
    });
  });
}
```

Figura 74. Método de la clase profile que se encarga de la lógica de borrar un jugador.

En la Figura 74 podemos ver un ejemplo de funcionamiento. En este caso, el método *deletePlayer()* será llamado desde un evento de botón en la vista que, cuando sea pulsado, ejecutará este fragmento de código. Cuando esto ocurre lo primero que realiza es crear un nuevo componente, de tipo *SimpleDialogComponent*, el cual abre una ventana emergente para pedirnos confirmación a la hora de eliminar el jugador de forma permanente. Dependiendo de lo que haya seleccionado, se borrará o no.

4.4.4. Validación en el cliente

Uno de los elementos más importantes a la hora de crear formularios dentro de la aplicación cliente, es la validación de los datos de entrada. La validación de los diferentes campos puede realizarse de dos formas distintas. Utilizaremos una validación síncrona cuando el cliente tiene toda la información necesaria para conocer si un campo es válido o no, por ejemplo, a través de una expresión regular. O, por el contrario, si no se conoce la información necesaria para validar el campo, como por ejemplo conocer si un nombre de usuario está repetido o no, será necesario realizar una petición al servidor, realizando una validación asíncrona.

```
ngOnInit(): void {
  this.form = this.builder.group({
    fullname: [
      this.data.fullname,
      [
        Validators.required,
        Validators.minLength(8),
        Validators.maxLength(25),
        Validators.pattern('^[A-Z][a-z]*(\\s[A-Z][a-z]*)?'),
      ],
    ],
    birthday: [this.data.birthday, [Validators.required]],
    gender: [this.data.gender.toLowerCase(), [Validators.required]],
  });
```

Figura 75. Método de inicialización que contiene la creación de un formulario y sus validaciones.

En la Figura 75 podemos observar la inicialización de un formulario que contiene el nombre completo, la fecha de cumpleaños y el género. En este caso, utiliza una validación síncrona, puesto que para cada campo las validaciones son directas Para el caso de la fecha de cumpleaños y el género, la única validación es que el campo debe de estar relleno al final, no puede quedar vacío, mientras que para el nombre completo, además se requiere un mínimo y un máximo de caracteres, además de satisfacer una expresión regular. Este tipo de validación no requiere del servidor para comprobar ningún campo.

```
ngOnInit(): void {
  this.form = this.builder.group(
    {
      nickname: [
        this.data.nickname,
        [
          Validators.required,
          Validators.minLength(5),
          Validators.maxLength(16),
          Validators.pattern('^[^0-9\\\\]\\w+$'),
        ],
        [this.nicknameValidator()],
      ],
      email: [
        this.data.email,
        [Validators.required, Validators.email],
        [this.emailValidator()],
      ],
      password: [
        '',
        [
          Validators.required,
          Validators.minLength(8),
          Validators.maxLength(25),
        ],
      ],
      confirmPassword: ['', Validators.required],
    },
    {
      validators: this.mustMatch('password', 'confirmPassword'),
    }
  );
```

Figura 76. Método de inicialización de un formulario que contiene validaciones asíncronas.

En la Figura 76 podemos observar la inicialización de un formulario más complejo: el formulario de registro, donde tenemos que incluir la información acerca de nuestra cuenta, como es el nombre de usuario, correo electrónico y contraseña. En nuestro sistema una de las restricciones es que tanto el nombre de usuario como el correo electrónico son únicos para cada jugador, por lo que no pueden repetirse. En la Figura 10 podemos observar como cada uno de los campos contienen validaciones síncronas como hemos explicado anteriormente. Sin embargo, a continuación de ellas, se indican las validaciones asíncronas. La función *nicknameValidator()* y *emailValidator()* son funciones asíncronas que nos devuelven un *Observable*, conteniendo un error de validación, o *null* en caso de no haber ningún error.

```
private nicknameValidator(): AsyncValidatorFn {
  return (control: AbstractControl): Observable<ValidationErrors | null> =>
    this.playerService
      .checkPlayerWithNickname(control.value)
      .pipe(
        map((response: boolean) => (response ? { nicknameUsed: true } : null))
      );
}

private emailValidator(): AsyncValidatorFn {
  return (control: AbstractControl): Observable<ValidationErrors | null> =>
    this.playerService
      .checkPlayerWithEmail(control.value)
      .pipe(
        map((response: boolean) => (response ? { emailUsed: true } : null))
      );
}
```

Figura 77. Métodos encargados de gestionar las validaciones asíncronas.

En la Figura 77 podemos observar la implementación de cada una de las validaciones asíncronas. Ambas son parecidas, puesto que la funcionalidad es similar: realizan una petición al servidor, comprobando el valor del formulario contenido dentro de la propiedad *control.value*. El resultado de la petición será un booleano, por lo que hacemos uso de la funcion *pipe* y *map* para transformar este resultado en un *Observable* conteniendo un error de validación o nulo en función del valor booleano recogido.

Además de las validaciones sincronas mencionadas anteriormente, también es posible añadir validaciones personalizadas, como se muestra en la Figura 10, donde al final añadimos una serie de validaciones que nos permitirán validar que las contraseñas coinciden.

```
mustMatch(controlName: string, matchingControlName: string) {
  return (form: FormGroup) => {
    const control = form.controls[controlName];
    const matchingControl = form.controls[matchingControlName];
    if (matchingControl.errors && !matchingControl.errors['mustMatch']) {
      return;
    }
    if (control.value !== matchingControl.value) {
      matchingControl.setErrors({ mustMatch: true });
    } else {
      matchingControl.setErrors(null);
    }
  };
}
```

Figura 78. Método encargado de gestionar una validación personalizada.

En la Figura 78 podemos observar la implementación de la validación correspondiente.

4.4.5. Interacción componente - servicio

Como hemos visto en casos anteriores, un componente puede llamar a las diferentes funcionalidades de los servicios que necesite. Esto se realizará mediante la inyección de los servicios dentro de los componentes.

```
@Injectable({
  providedIn: 'root',
})
export class AuthService {
  authURL = 'http://localhost:8080/auth/';

  constructor(private httpClient: HttpClient) {}
```

Figura 79. Decorador de una clase servicio dentro de la aplicación cliente.

Para ello, nuestros servicios tienen el decorador *Injectable*, como podemos observar en la Figura 79, lo que indica al compilador que esta clase puede ser inyectada mediante DI[52] en los componentes que sean necesarios. De esta manera podemos hacer uso de las funcionalidades de nuestros servicios desde los componentes en los que inyectamos el servicio.

[52] La inyección de dependencias es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos.

```
constructor(
  private playerService: PlayerService,
  private teamService: TeamService,
  private gameService: GameService,
  private dialog: MatDialog,
  private tokenService: TokenService,
  private builder: FormBuilder,
  private snackBar: MatSnackBar,
  private router: Router
) {
  // Setting max date.
  const currentDate = new Date();
  currentDate.setFullYear(currentDate.getFullYear() - 16);
  this.maxDate = currentDate;
}
```

Figura 80. Inicialización de un componente, junto a la inyección de servicios y otros componentes.

En la Figura 80 tenemos el constructor del componente perfil, en cuyos parámetros se encuentran varios servicios, además de algunas dependencias de *Material Design* y el router. Estos parámetros, al contener el decorador *Injectable*, una vez que los incluyamos en el constructor, estarán disponibles en el componente, donde podremos hacer uso de sus funcionalidades en cualquier parte de su código.

4.4.6. Autenticación

Como hemos comentado anteriormente, para la autenticación utilizaremos un token para autenticar las diferentes peticiones que enviaremos al servidor. Cuando estamos en la página de inicio de sesión y enviamos nuestras credenciales, haremos uso del servicio de autenticación en el cliente.

```
@Injectable({
  providedIn: 'root',
})
export class AuthService {
  authURL = 'http://localhost:8080/auth/';

  constructor(private httpClient: HttpClient) {}

  public loginPlayer(player: PlayerLogin): Observable<Token> {
    return this.httpClient.post<Token>(
      this.authURL + 'login',
      player,
      httpOptions
    );
  }
```

Figura 81. Servicio encargado de la autenticación dentro de la aplicación cliente.

En la Figura 81 podemos observar cómo enviamos al servidor nuestros credenciales. En caso de ser correctas, este nos devolverá un token, el cual nos servirá para autenticar nuestras futuras peticiones.

```
public setToken(token: string): void {
  window.localStorage.removeItem(TOKEN_KEY);
  window.localStorage.setItem(TOKEN_KEY, token);
}

public getToken(): string | null {
  return localStorage.getItem(TOKEN_KEY);
}
```

Figura 82. Métodos encargados de almacenar el token dentro del almacenamiento local del navegador.

Cuando obtengamos el token, lo almacenamos en el almacenamiento local del navegador, como se puede observar en la Figura 82. Para tenerlo disponible en cualquier momento de la sesión. Sin embargo, el token tiene una duración limitada, por lo que debemos de implementar también la funcionalidad de refrescar el token cuando esté caducado.

Esto es necesario en el momento que enviemos una petición al servidor con el token caducado, puesto que el servidor nos denegará la petición y ocurrirán errores en el cliente que no tendremos manera de solucionar. Para evitar este problema, hacemos uso de un interceptor.

Un interceptor básicamente es un servicio que se encarga de inspeccionar y modificar las peticiones que se dirigen del cliente al servidor y viceversa.

```
export class InterceptorService implements HttpInterceptor {
  constructor(
    private tokenService: TokenService,
    private authService: AuthService
  ) {}

  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    // Si no es una ruta protegida, simplemente pasamos
    if (!this.tokenService.isLogged()) return next.handle(req);

    let intReq = req;
    const token = this.tokenService.getToken();

    intReq = this.addToken(req, token!);

    return next.handle(intReq).pipe(
      catchError((error: HttpErrorResponse) => {
        // El token anterior ha caducado
        if (error.status === 401) {
          const token: Token = { token: this.tokenService.getToken()! };
          return this.authService.refreshToken(token).pipe(
            concatMap((data: any) => {
              console.log('Refreshing token');
              this.tokenService.setToken(data.token);
              intReq = this.addToken(req, data.token);
              return next.handle(intReq);
            })
          );
        }
```

Figura 83. Implementación del interceptor para comprobar la seguridad mediante token.

En la Figura 83 podemos observar nuestro servicio encargado de interceptar las peticiones implementando la interfaz *HttpInterceptor*. Esta se encargará de revisar la petición, en caso de obtener una respuesta con un estado 401, es decir, no autorizado, llamará al servicio de autenticación para refrescar el token, y posteriormente realizar la petición de nuevo. Esto automatiza el proceso de refrescar el token cada vez que este caduque y evite cerrar la sesión automáticamente al estar no autorizado.

Para finalizar con la autenticación, implementaremos lo que se llaman salvaguardas, o *guards* en inglés. Los *guards* en Angular son *middlewares* que se ejecutan antes de cambiar de ruta y determinan o no si se puede cargar dicha ruta. De forma similar al interceptor con las peticiones HTTP, este se ejecuta de forma que los

usuarios no autorizados no puedan ver una interfaz para la que no están autorizados.

```
export class GuardService implements CanActivate {
  realRol: string = '';

  constructor(private tokenService: TokenService, private router: Router) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean {
    // Protege de ir a cualquier ruta sin tener el rol adecuado
    const expectedRol = route.data['expectedRol'];
    this.realRol = this.tokenService.isAdmin() ? 'admin' : 'user';
    if (
      !this.tokenService.isLogged() ||
      expectedRol.indexOf(this.realRol) < 0
    ) {
      this.router.navigate(['/']);
      return false;
    }
    return true;
  }
}
```

Figura 84. Implementación de una salvaguarda para comprobar previamente los roles antes de redirigir la vista.

En la Figura 84 podemos encontrar una salvaguarda, la cual interpretará los cambios de ruta. En caso de que un usuario deje de estar autorizado, o no tenga un rol adecuado para dicha ruta, lo redirigirá a la página de inicio. Esto permite asegurar la aplicación contra usuarios que cambien la ruta manualmente para acceder a vistas las cuales no tienen autorización.

```
export class LoginGuard implements CanActivate {
  constructor(private tokenService: TokenService, private router: Router) {}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean {
    if (this.tokenService.isLogged()) {
      this.router.navigate(['/profile']);
      return false;
    }
    return true;
  }
}
```

Figura 85. Implementación del salvaguarda utilizado cuando el usuario está registrado.

De igual manera que en la Figura 84, en la Figura 85 hemos establecido otra salvaguarda, la cual redirige a cualquier usuario que entre en la aplicación a su página de perfil si había previamente iniciado sesión en lugar de permanecer en la página de inicio. Los *guards* pueden ser útiles para gestionar el redireccionamiento de la aplicación, además de proporcionar una capa de seguridad al cliente.

```
const routes: Routes = [
  {
    path: '',
    component: SplashComponent,
    canActivate: [LoginGuard],
  },
  {
    path: 'login',
    component: LoginComponent,
    canActivate: [LoginGuard],
  },
  {
    path: 'register',
    component: RegisterComponent,
    canActivate: [LoginGuard],
  },
  {
    path: 'profile',
    component: ProfileComponent,
    canActivate: [GuardService],
    data: { expectedRol: ['admin', 'user'] },
  },
  {
    path: 'feed',
    component: FeedComponent,
    canActivate: [GuardService],
    data: { expectedRol: ['admin', 'user'] },
  },
```

Figura 86. Fichero routing.module, que almacena las rutas de la aplicación junto a sus atributos.

Para finalizar, en la Figura 86 encontramos el fichero *routing.module* que establece las diferentes rutas y componentes que deben de activarse en dicha ruta. También, guarda información sobre los diferentes *guards* que queremos que se activen en cada momento, además de los roles que deben tener los usuarios para poder acceder a dichas rutas. Como se puede observar en la Figura 86, para la ruta por defecto, el inicio de sesión y el registro, se activará el salvaguarda que redirige a los usuarios que hayan iniciado sesión, además de ser rutas públicas, ya que cualquier usuario puede acceder a ellas sin haberla iniciado. Sin embargo, para la ruta del perfil y la página principal, está activa el *guard* para evitar usuarios no autorizados en dichas vistas, además de ser necesario alguno de los dos roles marcados, ya sea administrador o usuario.

5. Pruebas

En este apartado vamos a comentar los resultados en las pruebas realizadas al sistema. Como hemos mencionado anteriormente en el plan de pruebas, realizaremos pruebas de caja negra, las cuales nos permiten obtener resultados sin verificar el código interno. Además, dividiremos las pruebas realizadas en dos subapartados, uno correspondiente a los resultados en la aplicación servidor, y otro para los resultados en la aplicación cliente. De esta forma nos aseguramos que todas las funcionalidades quedan probadas y verificadas, ya que si probamos el resultado final únicamente, existe la posibilidad de dejarnos alguna funcionalidad sin probar.

5.1. Pruebas en la aplicación servidor

Como hemos comentado en el plan de pruebas para la aplicación servidor, realizaremos pruebas sobre cada uno de los endpoints de la aplicación Rest. Nuestro servidor contiene 31 endpoints diferentes, divididos para las diferentes entidades que contiene el sistema, además de contener endpoints específicos para la autenticación y para el envío de mensajes.

Para cada endpoint, se realizará una media de 5 pruebas, probando así no solo el correcto funcionamiento, si no la recepción de errores adecuada en caso de fallo en las diferentes peticiones. Por lo que se realizaron un total de 155 pruebas en total a lo largo del desarrollo. En la práctica, varias pruebas fueron realizadas múltiples veces, puesto que como hemos mencionado en el plan de pruebas, estas se realizaron a medida que avanzaba el desarrollo de la aplicación. La herramienta de pruebas Postman permite almacenar las diferentes peticiones con parámetros específicos en cada una de ellas para poder repetirlas fácilmente; esto simplifica el proceso de comprobación tras realizar cambios en el código.

A continuación mostraremos algunas pruebas realizadas para mostrar el resultado de las mismas.

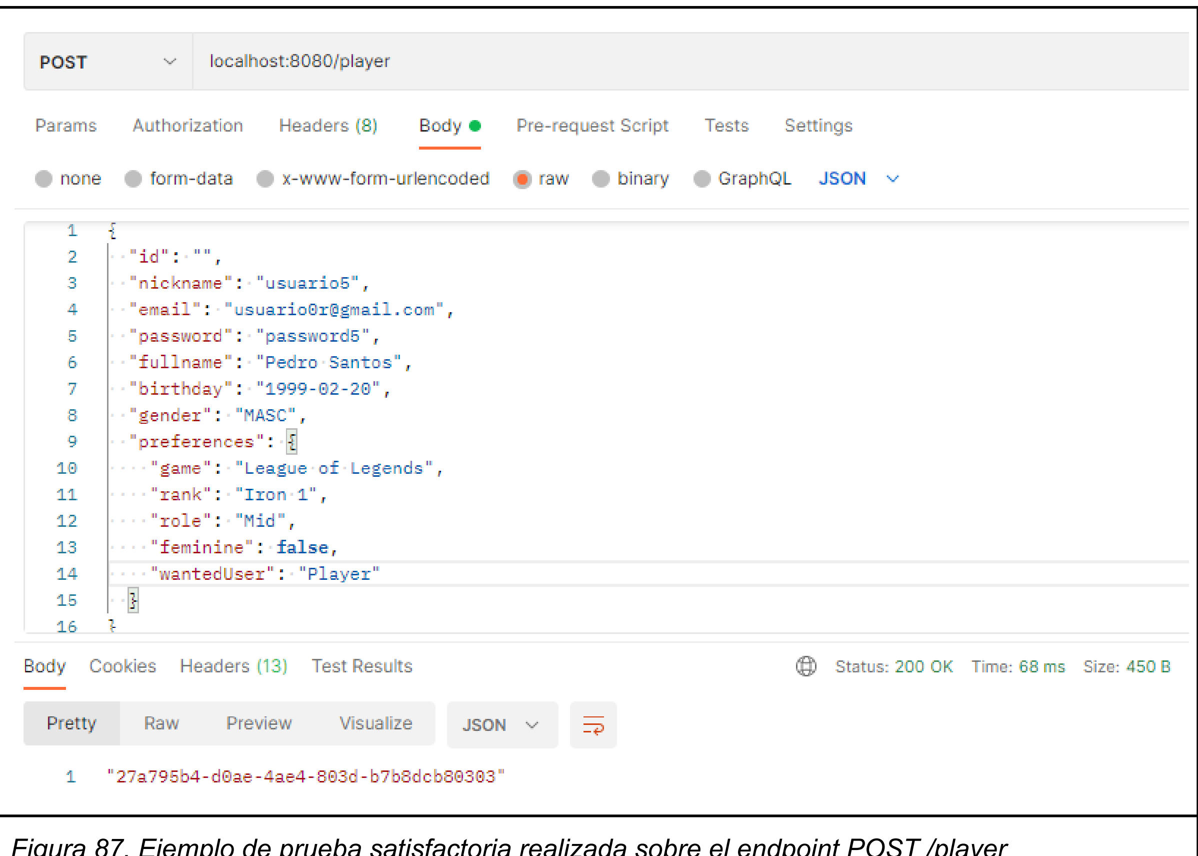

Figura 87. Ejemplo de prueba satisfactoria realizada sobre el endpoint POST /player

En la Figura 87 podemos observar una prueba realizada a uno de los endpoints, en este caso, un POST a la entidad jugador, lo que corresponde al método de registro. En esta prueba, enviamos todos los datos correctos, por lo que nos devuelve un código 200, por lo que la petición fue satisfactoria, además de devolvernos el UUID[53] correspondiente al jugador recién creado.

[53] Identificador único universal.

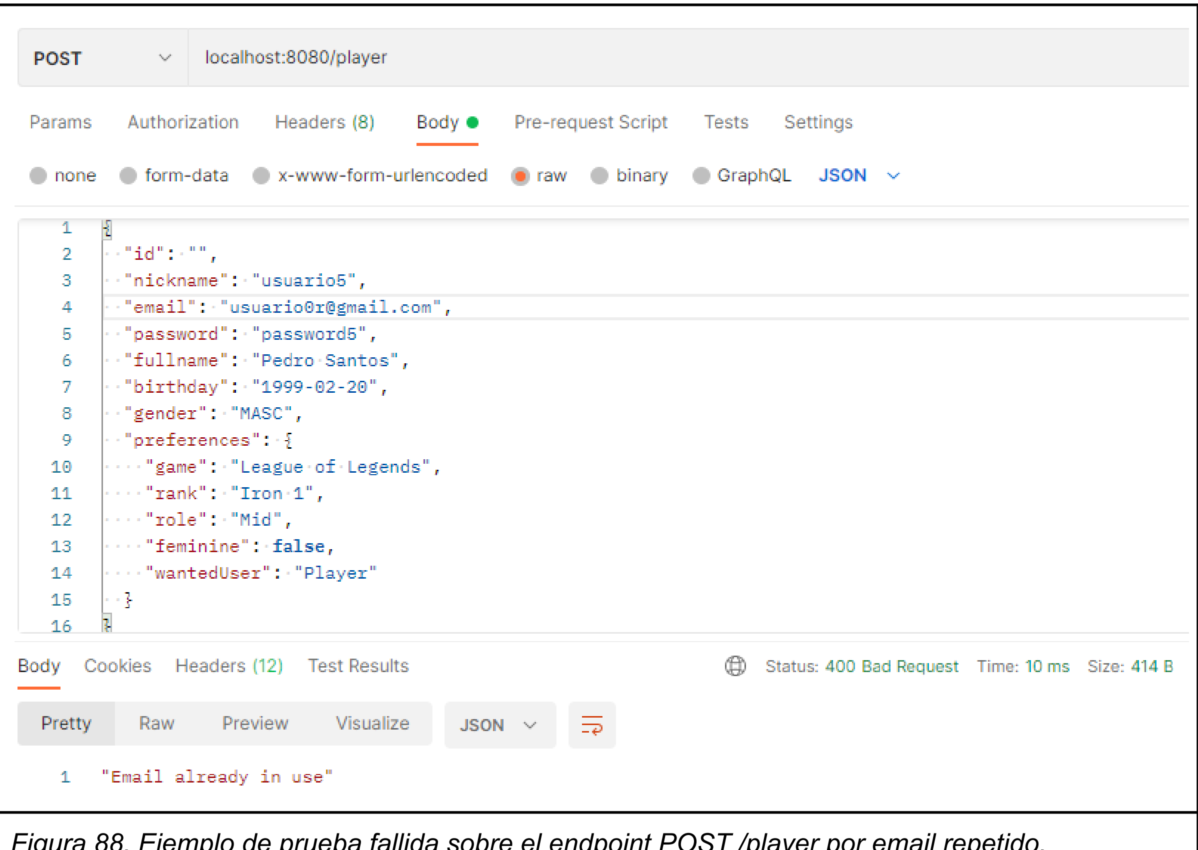

Figura 88. Ejemplo de prueba fallida sobre el endpoint POST /player por email repetido.

En la Figura 88, volvemos a realizar la misma operación, sin embargo, como podemos observar, el correo electrónico ya fue utilizado en la Figura 1, por lo que en este caso, si intentamos registrarnos, nos devolverá un código 400, o *Bad Request*, junto a un mensaje de error explicativo indicando que el correo ya está en uso, siendo este el resultado esperado.

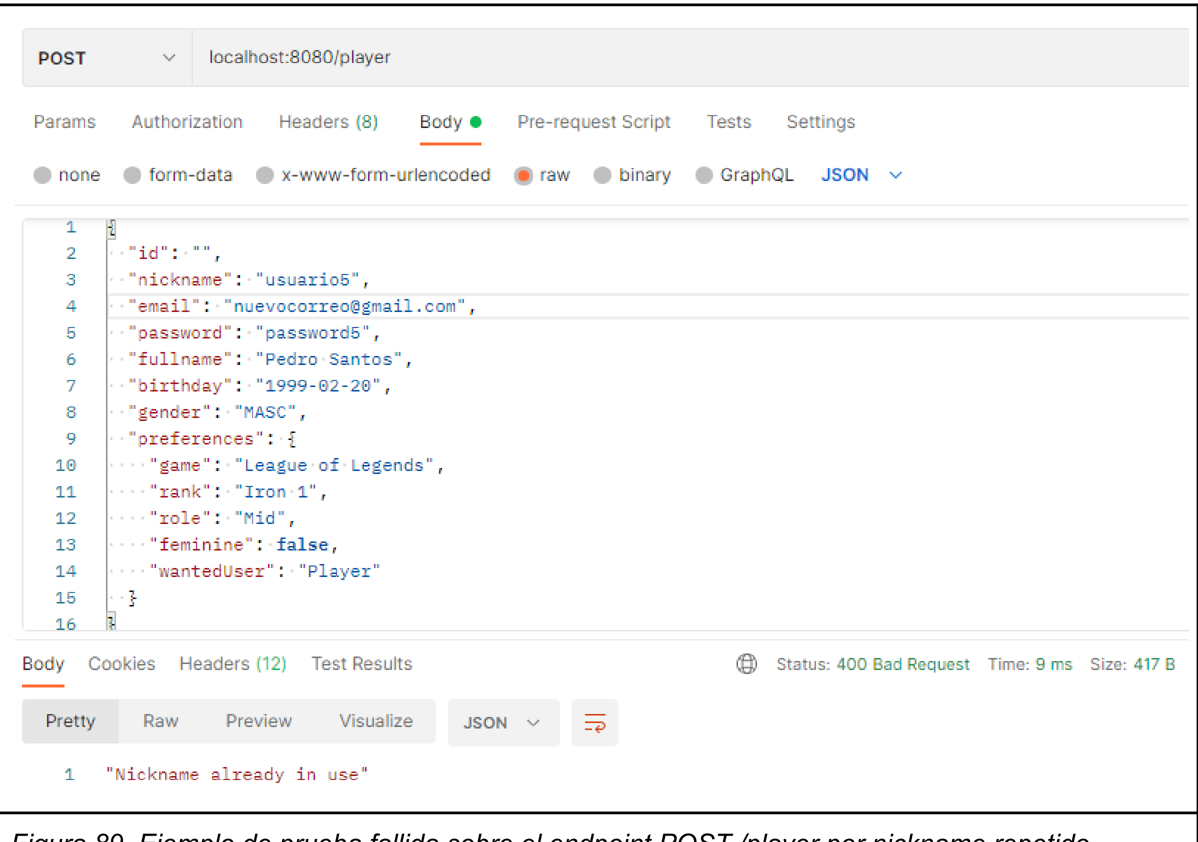

Figura 89. Ejemplo de prueba fallida sobre el endpoint POST /player por nickname repetido.

En la Figura 89, volvemos a repetir la misma operación, pero esta vez, cambiamos el correo por otro no utilizado anteriormente, el resultado sigue siendo un error de tipo *Bad Request*, debido a que el nombre de usuario sí que está repetido, como se nos indica en el mensaje de error asociado a la respuesta.

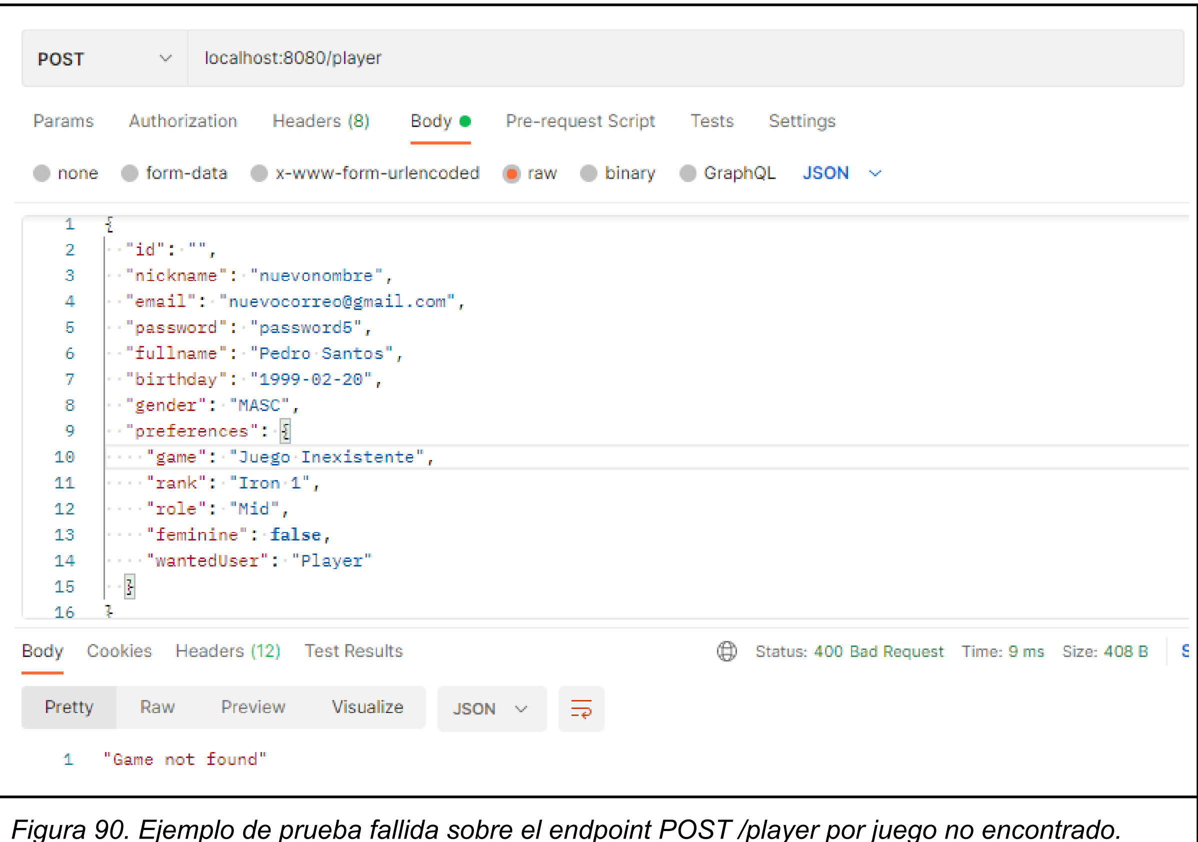

Figura 90. Ejemplo de prueba fallida sobre el endpoint POST /player por juego no encontrado.

En la Figura 90, volvemos a repetir una última vez la misma operación de registro, habiendo cambiado tanto el nombre de usuario como el correo electrónico, pero también cambiamos el juego de nuestras preferencias por uno no existente, lo que nos sigue devolviendo un error 400, junto a un mensaje de error que nos indica que dicho juego no existe.

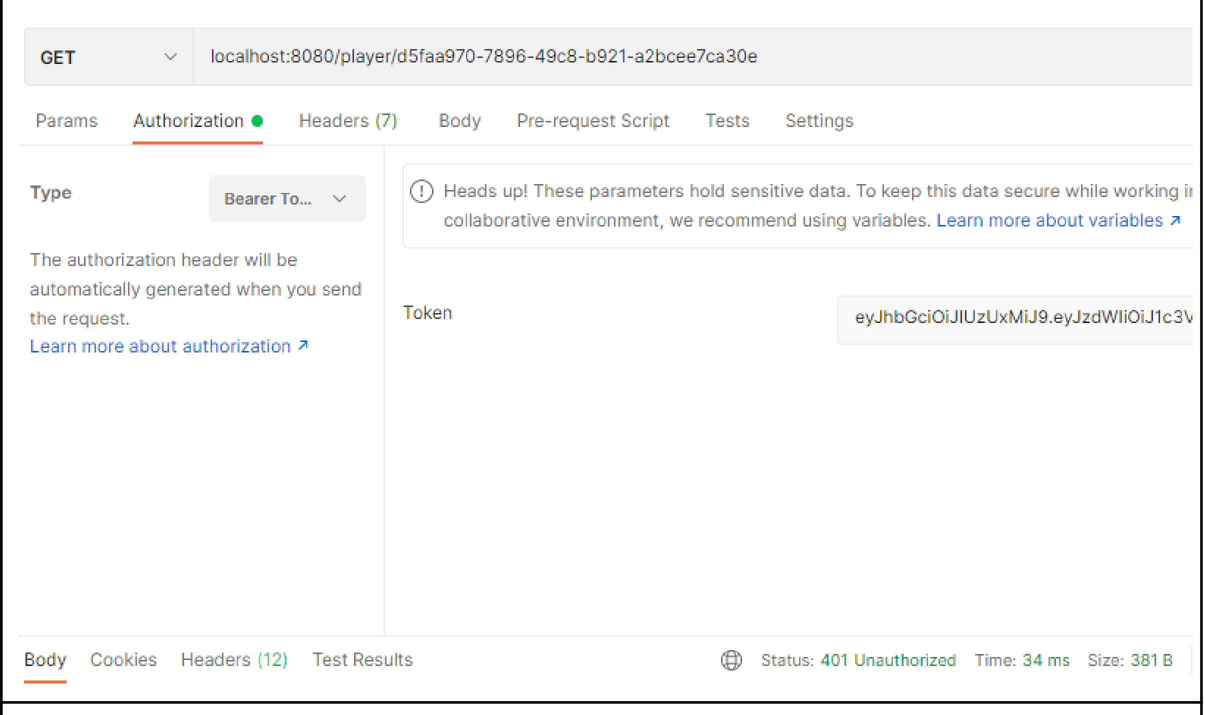

Figura 91. Ejemplo de prueba fallida realizada sobre el endpoint GET /player/id por estar no autenticado.

En la Figura 91, vamos a tratar de realizar una prueba sobre otro endpoint diferente al anterior, en este caso, sobre un GET de la entidad jugador, para recuperar la información a partir de su identificador. En este caso, es fallido, debido a que este endpoint, está protegido, y debemos de haber iniciado sesión previamente.

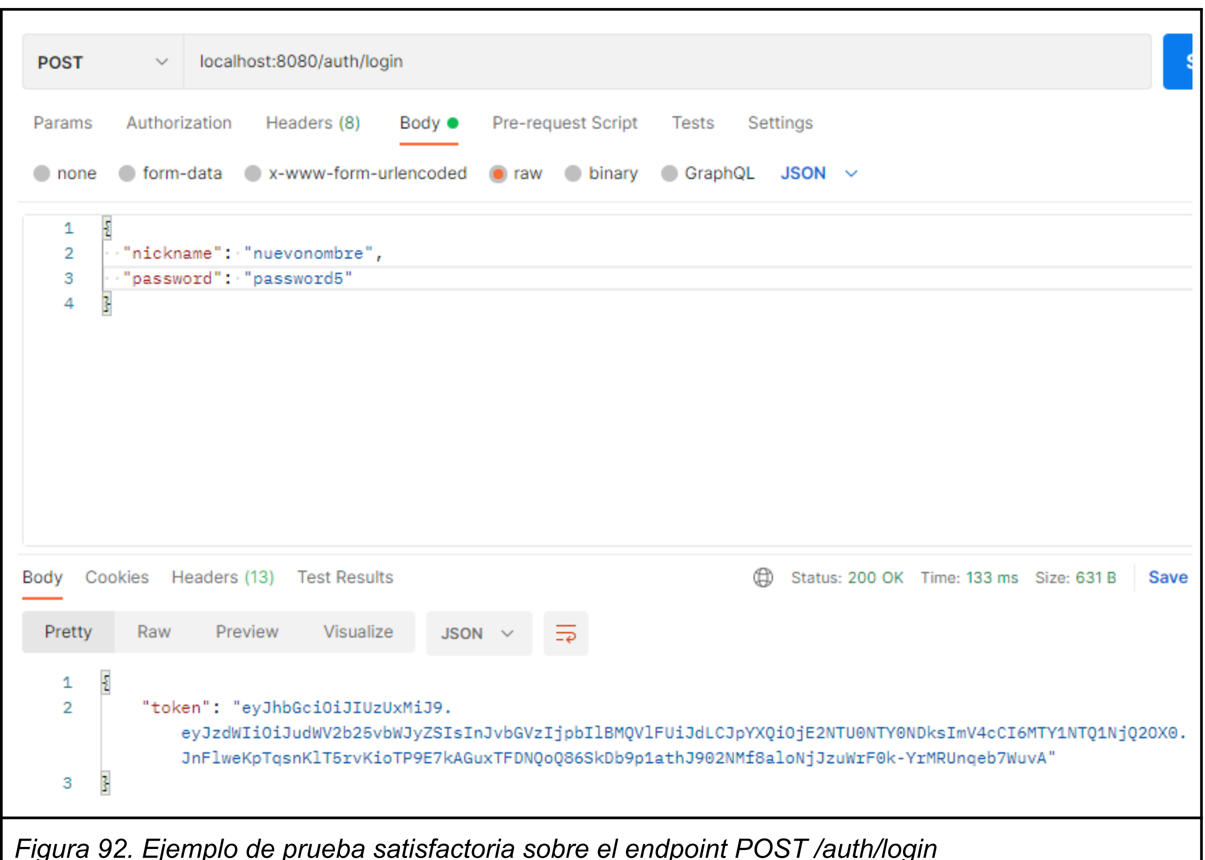

Figura 92. Ejemplo de prueba satisfactoria sobre el endpoint POST /auth/login

En la Figura 92, podemos observar como hemos hecho uso del endpoint de autenticación, para iniciar sesión en la aplicación. Este endpoint, en caso de ser satisfactorio, nos devolverá un token, el cual colocaremos en las cabeceras de nuestras peticiones, para confirmar al servidor que hemos iniciado sesión.

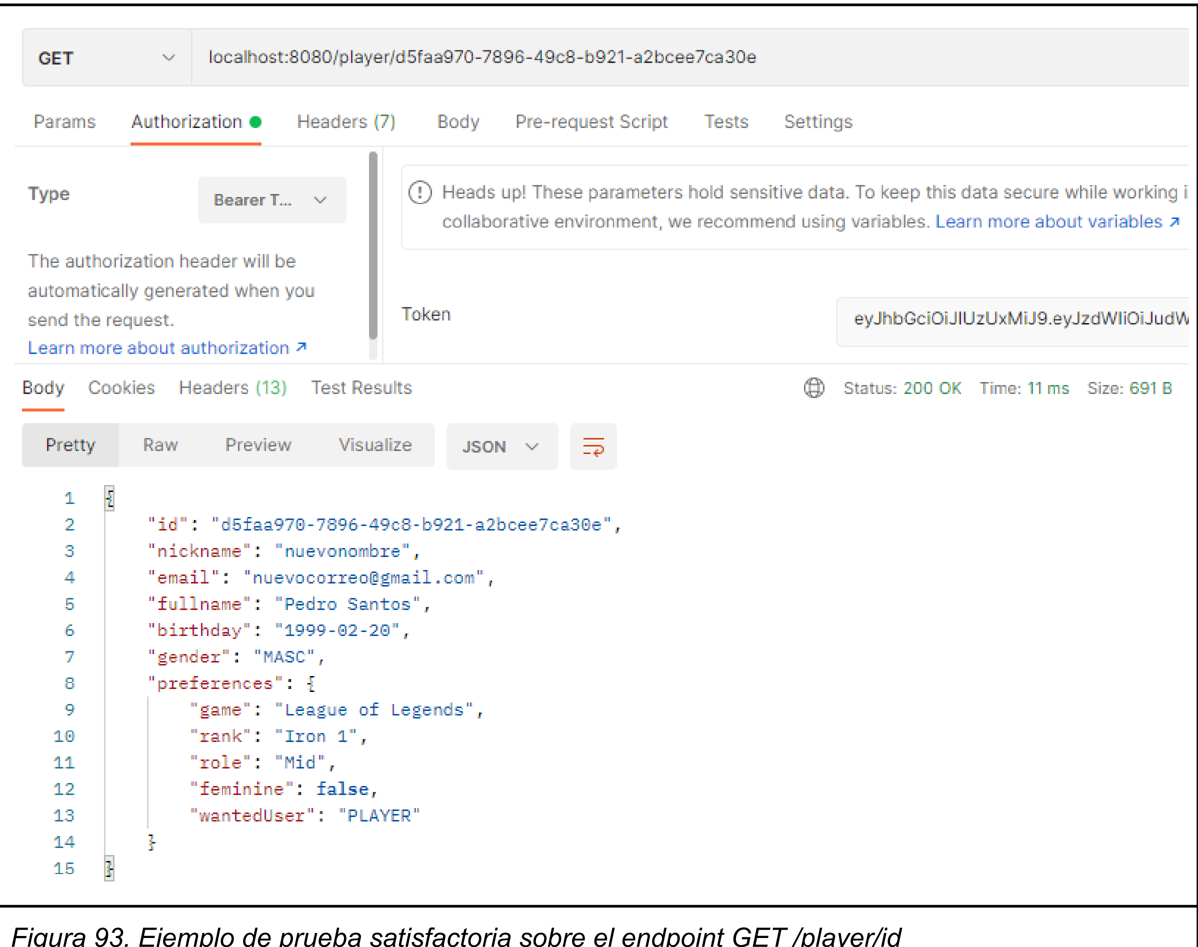

Figura 93. Ejemplo de prueba satisfactoria sobre el endpoint GET /player/id

En la Figura 93, hemos introducido nuestro token dentro del apartado de autenticación, y al realizar la petición, esta es satisfactoria y obtenemos la información relacionada con dicho identificador, sin la contraseña.

Sin embargo, el endpoint asociado a la Figura 93, fue problemático, debido a que a pesar de marcar el campo de la contraseña como *nullable*, esta seguía apareciendo al realizar el endpoint. La detección temprana de este fallo gracias al testeo continuo de endpoints, junto a la facilidad de repetir la prueba repetidamente, solventó el problema.

El resto de pruebas realizadas a los diferentes endpoints son similares a los mostrados previamente. Además, cabe destacar que todas las pruebas realizadas a los diferentes endpoints obtuvieron resultados satisfactorios en relación con el resultado esperado.

5.2. Pruebas en la aplicación cliente

Para el testeo de la aplicación cliente, como hemos mencionado en el plan de pruebas, hemos hecho uso de los criterios de aceptación definidos en cada una de las historias de usuario. Estas pruebas se han realizado de forma integrada con el

desarrollo de la aplicación, puesto que han permitido probar la interfaz a medida que se ha ido avanzando en las diferentes historias de usuario.Además, al finalizar cada iteración, se realizará una prueba global de los criterios de todas las historias de usuario que contiene dicha iteración. Esta forma de testear nos ha permitido obtener una base sólida sobre la que ir avanzando.

A continuación mostraremos los pasos seguidos para realizar una prueba de la interfaz dentro de la aplicación cliente. En este caso, mostraremos el caso de uso de registro como ejemplo.

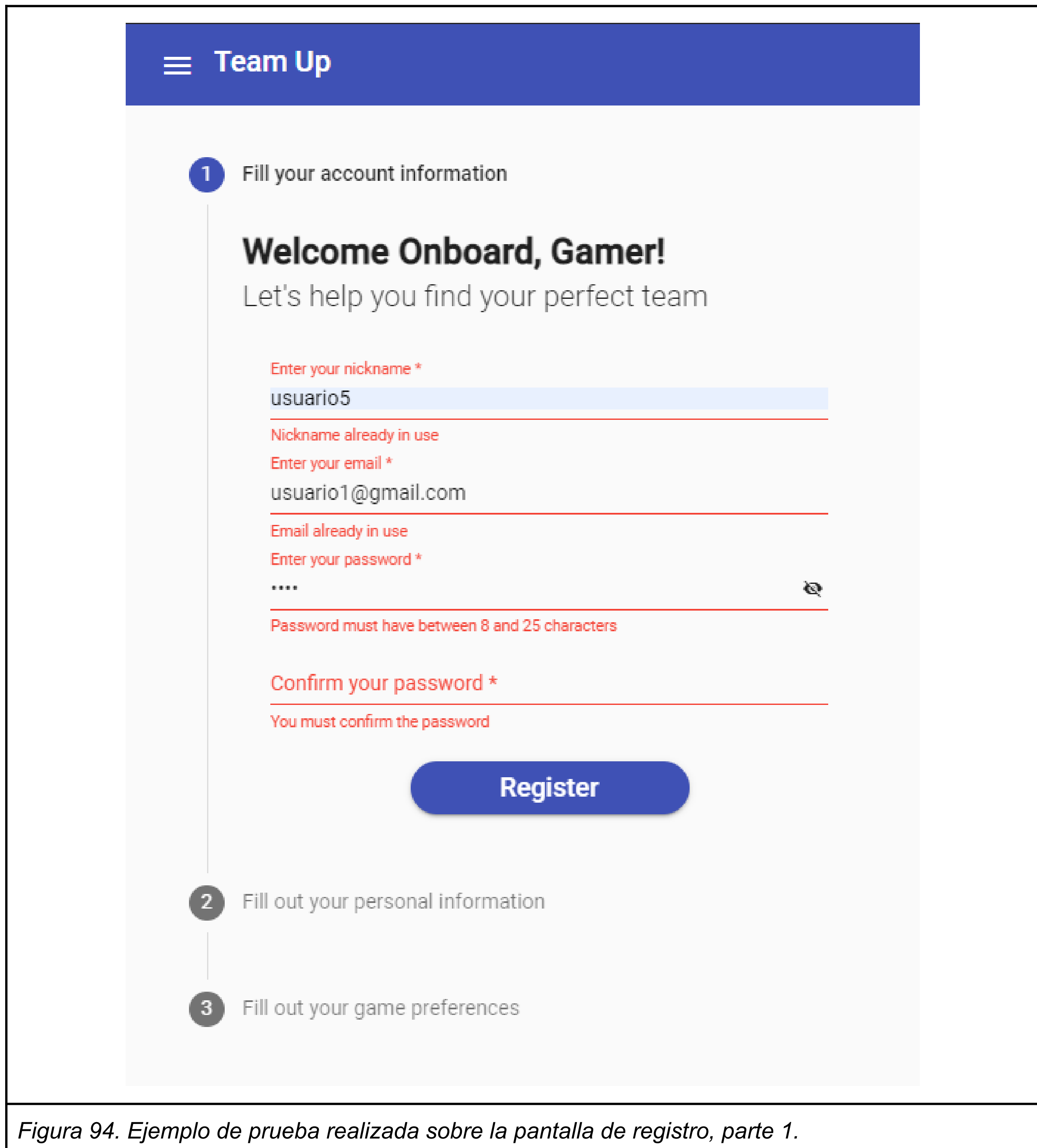

Figura 94. Ejemplo de prueba realizada sobre la pantalla de registro, parte 1.

En la Figura 94 podemos observar la pantalla de registro. En esta hemos rellenado los campos referentes al nombre de usuario, correo electrónico y contraseña. Como se puede observar, la validación de formularios funciona, puesto que obtenemos mensajes de error en todos los campos, incluyendo aquellas validaciones asíncronas como pueden ser el caso del nombre de usuario o correo electrónico.

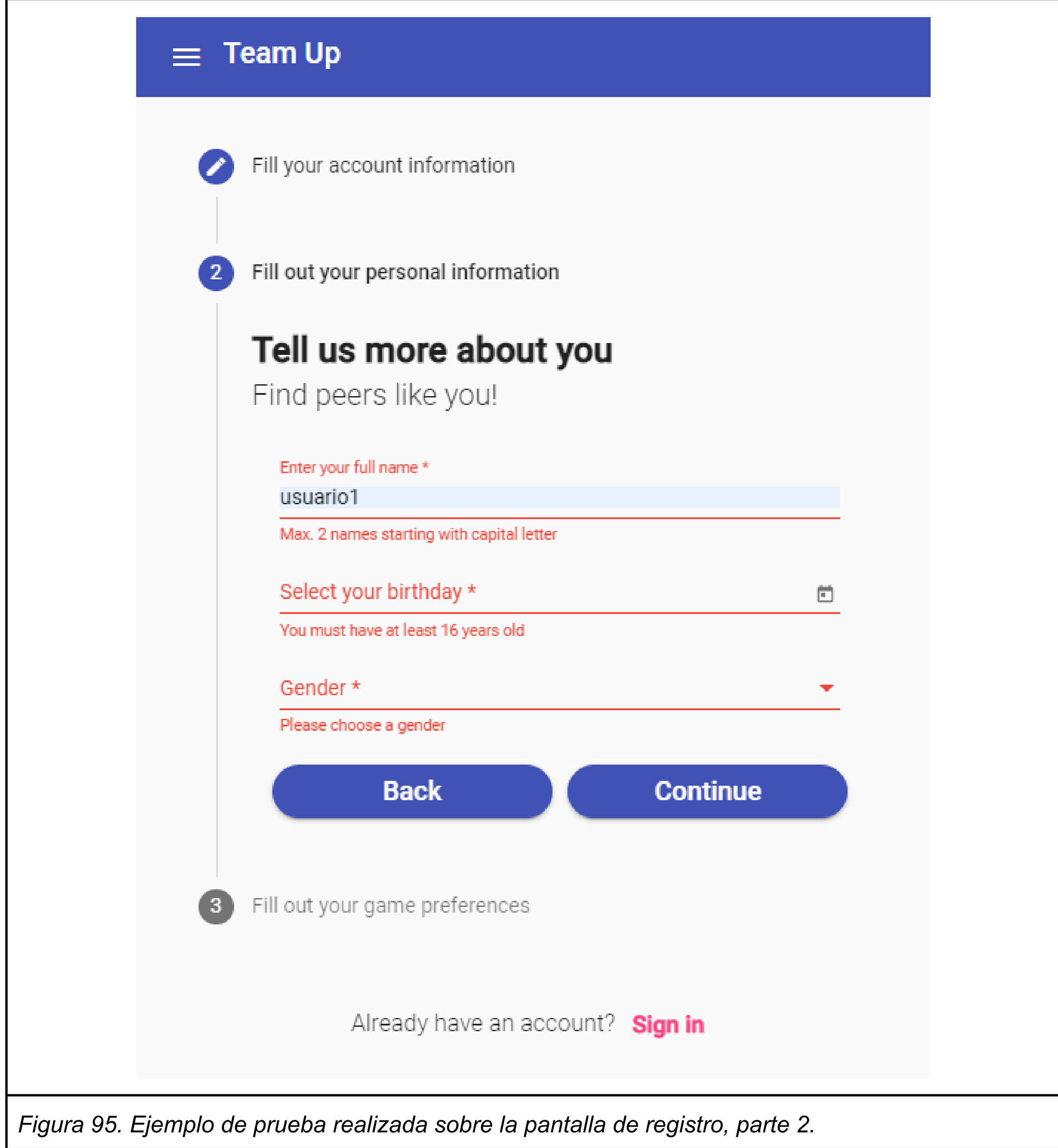

Figura 95. Ejemplo de prueba realizada sobre la pantalla de registro, parte 2.

En la Figura 95 podemos observar que ya hemos pasado a la siguiente sección. En este caso, nos pide información personal como nuestro nombre completo, fecha de nacimiento y género. Al incluir un nombre que contenga números y no empiece por mayúscula, nos aparecerá un error indicando que es incorrecto. Además, debemos de tener más de 16 años, así como seleccionar un género. Hasta que no completamos los pasos requeridos, no se permite avanzar en el registro.

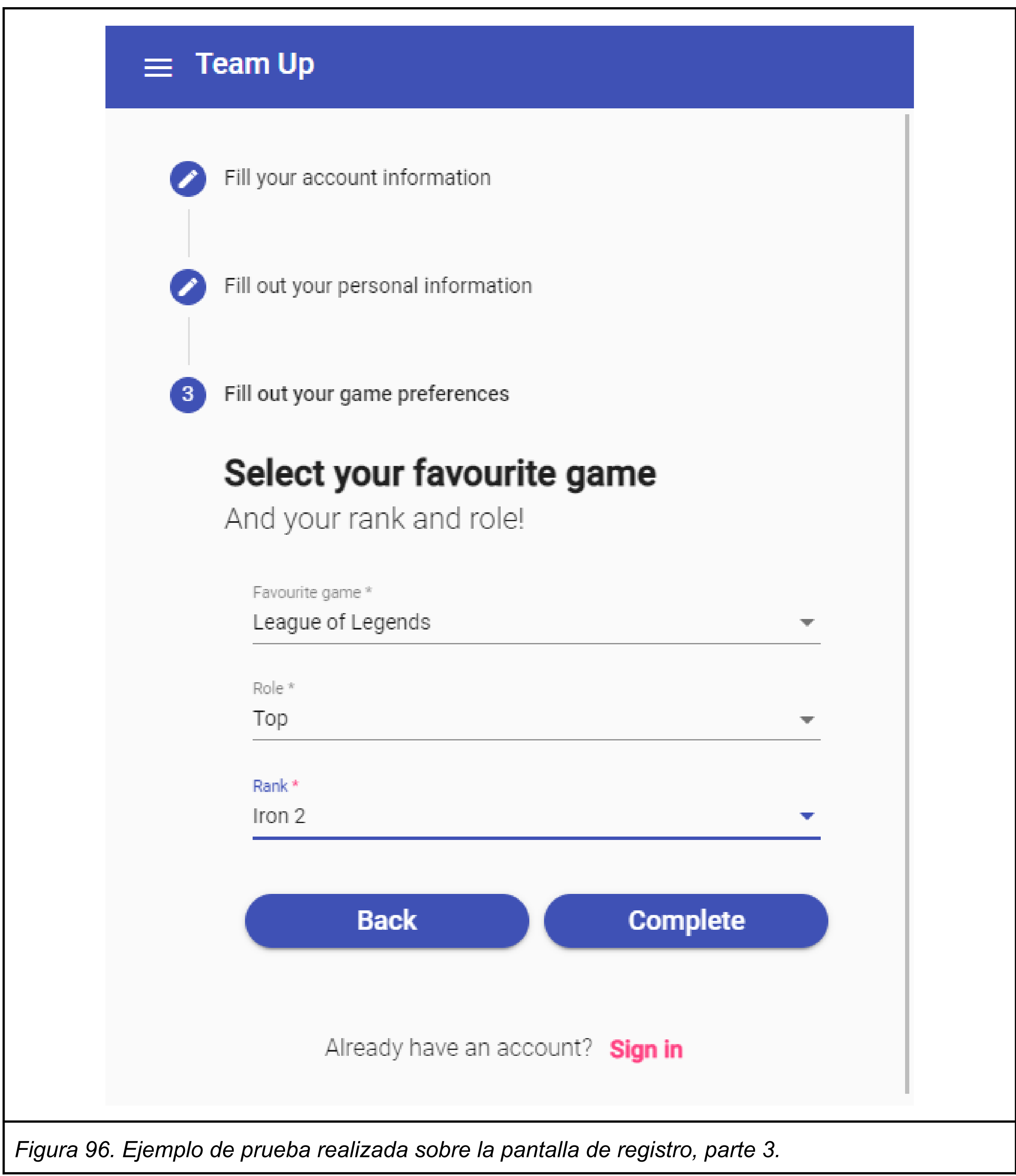

Figura 96. Ejemplo de prueba realizada sobre la pantalla de registro, parte 3.

En la Figura 96 podemos observar cómo una vez que hemos terminado de rellenar los campos correspondientes de forma correcta, podemos completar el registro.

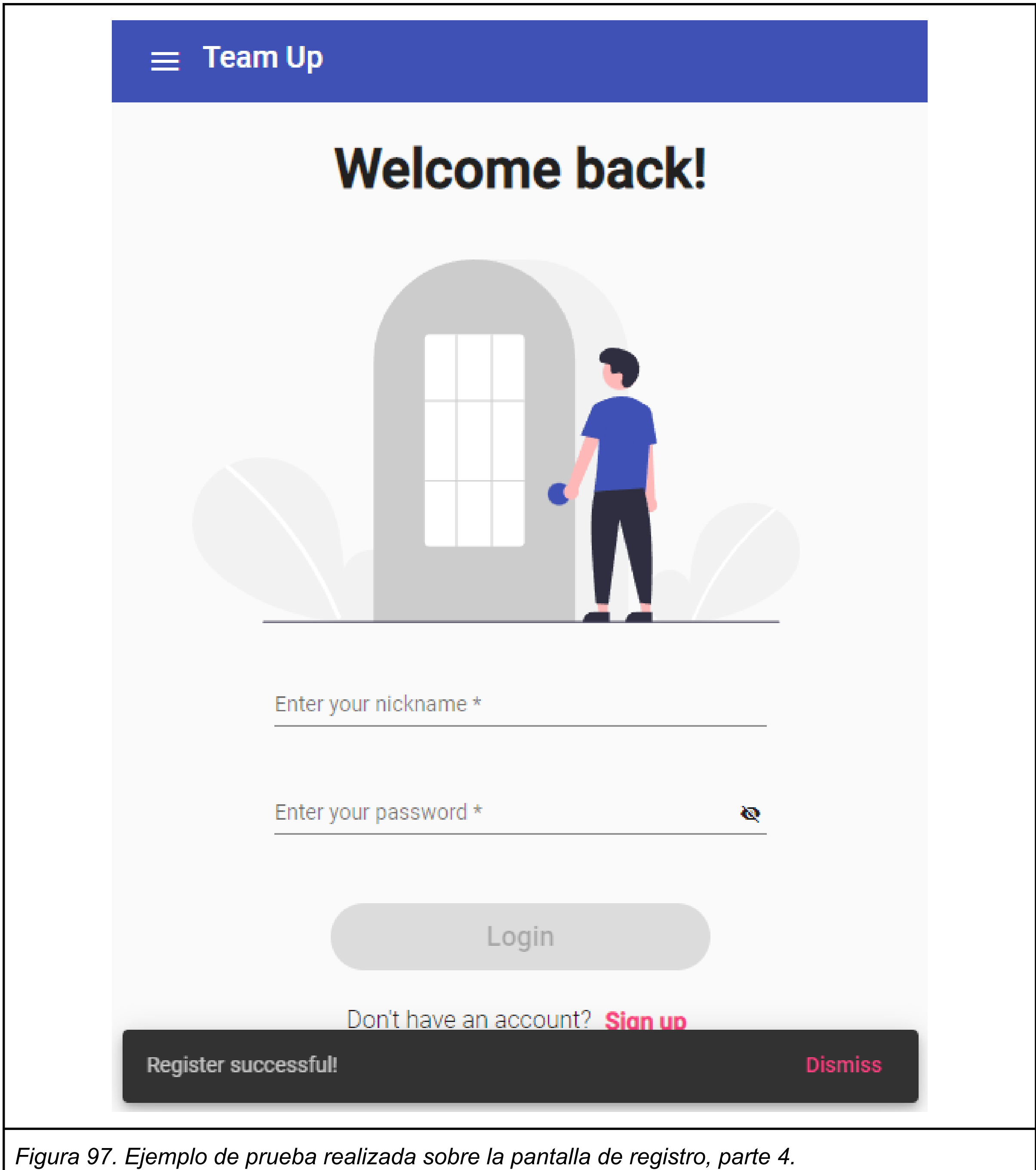

Figura 97. Ejemplo de prueba realizada sobre la pantalla de registro, parte 4.

En la Figura 97, podemos observar cómo una vez terminado el registro, nos redirigirá a la página de inicio de sesión, donde podremos ingresar nuestros credenciales recién creados, junto a un *snack bar* indicándonos que el registro fue satisfactorio. De esta manera, hemos terminado el proceso de registro de forma satisfactoria, habiendo probado todas las funcionalidades que ofrece.

Hay un total de 7 formularios y una media de 10 comprobaciones para cada formulario, probando las distintas validaciones de cada uno de los campos, tanto síncronas como asíncronas, así como una prueba para comprobar el correcto funcionamiento del formulario. Esto hace un total de 70 pruebas a realizar sobre los

diferentes formularios de la aplicación cliente. Estas comprobaciones fueron realizadas repetidas veces a lo largo del desarrollo, comprobando la funcionalidad a medida que se iba desarrollando. Esto permitió detectar diferentes errores a la hora de probar diferentes formularios complejos con varios pasos.

Alguno de los fallos más recurrentes a lo largo del desarrollo fue la implementación de las validaciones asíncronas, y el uso de este tipo de metodología de pruebas permitió detectar errores en una etapa muy temprana del desarrollo.

Las diferentes pruebas sobre el resto de funcionalidades se realizaron de forma similar a la mostrada anteriormente.

6. Conclusiones

Dada la problemática inicial y la propuesta establecida, se puede afirmar que el nivel de cumplimiento de los objetivos establecidos inicialmente se han completado de forma satisfactoria. El primero de los objetivos es realizar un estudio de necesidades y soluciones, y se ha llevado a cabo en su totalidad. En el Apartado 2 de análisis podemos observar cómo hemos realizado este estudio, analizando la problemática de crear un punto de encuentro entre jugadores de videojuegos, estudiando soluciones similares, así como realizar una selección de tecnologías para llevar a cabo la solución. De forma análoga, hemos realizado un estudio de soluciones, mediante una división del trabajo en historias de usuario, una planificación inicial priorizando aquellas que maximizan el valor, así como un estudio de viabilidad del proyecto. El segundo de los objetivos podemos encontrarlo reflejado principalmente en el Apartado 3 de diseño, donde hemos diseñado las características internas del sistema, centrándonos especialmente en metodologías para el desarrollo web. Siguiendo una arquitectura cliente servidor, dividimos el sistema en dos aplicaciones, cada una con su respectivo diseño y arquitectura. Cabe destacar un elemento fundamental del diseño siendo el diseño de la interfaz de ambas aplicaciones, tanto la interfaz gráfica que tendrá la aplicación cliente como el API que tiene el servidor. El último de los objetivos es la implementación de un prototipo de aplicación web, este apartado está reflejado tanto en el Apartado 3 de diseño, como en el Apartado 4 de implementación. En estos apartados hemos ido detallando todas sus características internas, así como la arquitectura del sistema, y el modo de comunicación entre ambas aplicaciones.

El uso de las tecnologías escogidas han tenido en mi opinión un papel instructivo, sobre todo a la hora de escoger un *framework* para el desarrollo del cliente, puesto que Angular es un *framework* del que partía sin conocimientos previos. Esto ha creado ciertos obstáculos, puesto que toda parte de desarrollo tuvo que ser aprendida a medida que se iba creando el sistema, sin embargo, creo que estos problemas fueron posibles de superar gracias a una metodología ágil, la cual permitió un desarrollo siguiendo un buen proceso de trabajo, dividiendo el trabajo en pequeños periodos de tiempo, facilitando así la evolución del mismo.

Además, el uso de Spring Boot como *framework* para el desarrollo del servidor, ha acelerado el proceso de desarrollo del mismo, puesto que gracias a su conocimiento previo, ha permitido que pueda dedicar más tiempo de desarrollo en otros elementos los cuales no estaba tan familiarizado, como puede ser la creación de un sistema de mensajería, o la propia implementación del cliente con Angular.

Respecto a la mantenibilidad del proyecto a largo plazo, el sistema se encuentra en un estado óptimo, puesto que la organización utilizada, junto a la separación de responsabilidades entre las distintas funcionalidades, tanto en la aplicación del servidor y cliente, han favorecido una evolución del proyecto escalable y mantenible.

Acerca de la planificación, cabe destacar que siguen habiendo historias de usuario en el tintero que no han podido realizarse debido a una subestimación del tiempo de desarrollo de otras. Principalmente mi falta de experiencia a la hora de planificar fue el ocasionante de esto. No obstante, la correcta priorización de historias de usuario, permitió focalizar el desarrollo del sistema de manera que aquellas historias de usuario con mayor prioridad fueron realizadas a tiempo, en el caso de no haber priorizado correctamente, el proyecto no podría haberse acabado a tiempo, quedando un resultado incompleto. Por lo que la planificación del proyecto fue crucial, ya que facilitó el desarrollo, priorizando en aquellas funcionalidades que maximizan el valor del producto final.

A continuación me gustaría destacar algunos de los puntos más interesantes sobre el desarrollo del proyecto. Sin duda, uno de los puntos que más me llamó la atención, fue la creación de un modelo de dominio extenso, junto a la creación de consultas complejas, como pueden ser para recoger jugadores o equipos recomendados para cierto jugador o equipo. Otro elemento importante que considero fundamental fue la creación de una interfaz de usuario responsiva a cualquier dispositivo de cualquier tamaño, siguiendo siempre un diseño *mobile-first*, donde priorizamos el desarrollo de la interfaz para un dispositivo móvil, minimizando así la cantidad de información superflua que pueda haber en pantalla.

También me gustaría mencionar que, sin duda, uno de los elementos que personalmente más me interesó y me gustó fue el aprendizaje desde cero de un framework completo de desarrollo en el cliente, como es Angular. Angular siempre fue conocido como un framework complejo y de difícil aprendizaje pero, sin embargo, permite una gran escalabilidad con los proyectos creados, además tener a disposición un gran abanico de funcionalidades integradas, lo que evita el uso de librerías de terceros.

Para finalizar comentaré algunas reflexiones personales acerca del uso de los diferentes conocimientos adquiridos durante el grado y su aportación al proyecto. Considero que la adquisición de requisitos en metodologías clásicas o las metodologías de desarrollo ágil y todas sus técnicas son necesarias para el desarrollo de cualquier proyecto actual que se precie. Finalizar el trabajo te hace reflexionar sobre cómo se unen todos los conocimientos adquiridos durante el grado, para crear un único producto, de calidad. Desde el análisis, pasando por el diseño, su correspondiente implementación, y finalizando en las pruebas, es un proceso amplio, pero necesario.

6.1. Mejoras y trabajos futuros

A continuación enumeramos algunas de las funcionalidades presentes en historias de usuario que se quedaron en el tintero. También mencionaremos algunas mejoras

que, aun sin estar presentes en las historias de usuario definidas al principio, aportarían una gran cantidad de valor al producto.

- Permitir la comunicación entre jugadores y equipos, creando un chat único donde los participantes puedan hablar entre ellos en un chat grupal.

- Creación de un sistema de invitaciones para invitar a jugadores a unirse a un equipo una vez hayan tenido match y hablado.

- Creación de un chat grupal para el equipo, donde puedan comentar elementos personales sobre el equipo sin necesidad de estar en un chat con otro jugador.

- Añadir mayor nivel de personalización de tu perfil, permitiendo la posibilidad de aumentar o disminuir la diferencia de rangos entre jugadores o equipos objetivo.

- Permitir la búsqueda de equipos completos para encontrar otros equipos completos de un rango similar para establecer contacto y añadir la posibilidad de crear partidas personalizadas.

- Sistema para organizar los diferentes chats en la pestaña de mensaje, pudiendo separar entre jugadores, equipos, favoritos, etc.

- Creación de un sistema de torneos donde equipos puedan unirse y participar en torneos con diferentes premios.

- Creación de un sistema de liga interno de la aplicación, donde los equipos que jueguen partidas tendrán un ranking mayor en la clasificación.

- Permitir partidas personalizadas entre jugadores individuales, a modo de 1 versus 1.

Dado el estado actual del sistema, el diseño y tecnología utilizadas facilitan la integración de las mejoras mencionadas anteriormente. Esto se debe gracias a la modularidad del proyecto, y de cómo el modelo de dominio se relaciona entre sí. La forma en la que está construida la base del sistema permite su expansión agregando funcionalidades de manera sencilla. Por ejemplo, si se desea añadir mayor personalización al perfil, esto sería fácil de integrar utilizando el sistema de preferencias ya existente en el sistema. También permite que los equipos completos puedan buscar otros equipos para practicar, ya que por la manera en la que se estructura la búsqueda de entidades afines para una entidad en particular resultaría sencillo extenderlo a los equipos.

Para concluir, el sistema está construido y documentado de tal manera que facilite su mantenimiento y se puedan agregar funcionalidades sin necesidad de tener que modificar el código completo. La metodología de trabajo utilizada y la arquitectura del sistema han fomentado un desarrollo sostenible que ha finalizado en la creación de mucho valor para el producto final.

7. Bibliografía

- Busquets, Cris. 2018. “Paletas de colores para UI: créala en solo 2 pasos.” uiFromMars. https://www.uifrommars.com/crear-paletas-de-color-diseno-ui/.
- Caballero, Lucía. 2017. “El secreto está en los bordes: por qué los diseños de Apple nos entran por los ojos.” El Diario. https://www.eldiario.es/hojaderouter/tecnologia/hardware/mac-apple-diseno-squircle-jony-ive_1_3577136.html.
- Carte, Traci A., Jon S. Jasperson, and Mark E. Cornelius. 2006. “Integrating ERD and UML Concepts When Teaching Data Modeling.” Integrating ERD and UML Concepts When Teaching Data Modeling. https://aisel.aisnet.org/jise/vol17/iss1/9/.
- Delgado, Luis. 2020. “Angular vs React: Cuál Elegir Para tu Aplicación.” freeCodeCamp. https://www.freecodecamp.org/espanol/news/angular-vs-react-cual-elegir-para-su-aplicacion/.
- EcuRed contributors. 2019. “Modelo de dominio.” EcuRed. https://www.ecured.cu/Modelo_de_dominio.
- Garret, Jesse J. 2011. *The Elements of User Experience: User-centered Design for the Web and Beyond*. N.p.: New Riders.
- Gomaa, Hassan. 2011. *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. N.p.: Cambridge University Press.
- Internet Ya. 2020. “Aplicaciones Web Vs Software de Escritorio - INTERNET YA.” INTERNET YA - Soluciones Web. https://www.internetya.co/aplicaciones-web-vs-escritorio-2/.

- Leal, María P. 2018. “La toxicidad en los videojuegos: ¿podemos acabar con ella?” Mundo Deportivo. https://www.mundodeportivo.com/urbantecno/videojuegos/como-terminar-toxicidad-videojuegos.
- Ochoa, Igor. 2019. “Swipe Surge: La gran invención de Tinder.” Igor Ochoa. https://igorochoa.net/2019/10/09/swipe-surge-tinder/.
- Pooley, Rob, and Perdita Stevens. 2008. *Usare UML. Ingegneria del software con oggetti e componenti*. N.p.: Pearson.
- Red Hat. 2020. “Temas Integración ¿Qué es una API de REST?” Red Hat. https://www.redhat.com/es/topics/api/what-is-a-rest-api#rest.
- Rehkopf, Max. 2022. “Historias de usuario | Ejemplos y plantilla.” Atlassian. https://www.atlassian.com/es/agile/project-management/user-stories.
- Santos, Wendell. 2017. “Which API Types and Architectural Styles are Most Used?” ProgrammableWeb. https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26.
- Simões, Chiyana. 2020. “MoSCoW. ¿Qué es y cómo priorizar en el desarrollo de tu aplicación?” ITDO. https://www.itdo.com/blog/moscow-que-es-y-como-priorizar-en-el-desarrollo-de-tu-aplicacion/.
- Spring. 2021. “Spring Framework.” Spring. https://spring.io/projects/spring-framework#support.
- Spring. 2022. “Spring Data JPA - Reference Documentation.” Spring. https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.

- Sutherland, J. J. 2021. *Scrum: El revolucionario método para trabajar el doble en la mitad de tiempo*. Translated by Victoria E. Gordo del Rey. N.p.: Ariel.
- Terrera, Gustavo. 2017. “Pruebas de Caja Negra y un enfoque práctico.” Testing Baires. https://testingbaires.com/pruebas-caja-negra-enfoque-practico/.
- W3C. 2007. “SOAP Version 1.2 Part 0: Primer (Second Edition).” W3C. https://www.w3.org/TR/2007/REC-soap12-part0-20070427/.
- W3C. 2008. “Mobile Web Best Practices 1.0.” W3C. https://www.w3.org/TR/mobile-bp/#OneWeb.
- Wikipedia. 2020. “Modelo de dominio.” Wikipedia. https://es.wikipedia.org/wiki/Modelo_de_dominio.
- Wikipedia. 2022. “Interfaz de programación de aplicaciones.” Wikipedia. https://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones.
- Wikipedia. 2022. “Spring Framework.” Wikipedia. https://es.wikipedia.org/wiki/Spring_Framework.
- Wikipedia. 2022. “Front end y back end.” Wikipedia. https://es.wikipedia.org/wiki/Front_end_y_back_end.
- Wikipedia. 2022. “AJAX.” Wikipedia. https://es.wikipedia.org/wiki/AJAX.
- Xia, Vincent. 2017. “What is Mobile First Design? Why It's Important & How To Make It?” Medium. https://medium.com/@Vincentxia77/what-is-mobile-first-design-why-its-important-how-to-make-it-7d3cf2e29d00.

Apéndice 1. Descripción de contenidos suministrados.

Dentro del material suministrado se encuentran dos carpetas principales: *build* y *src*.

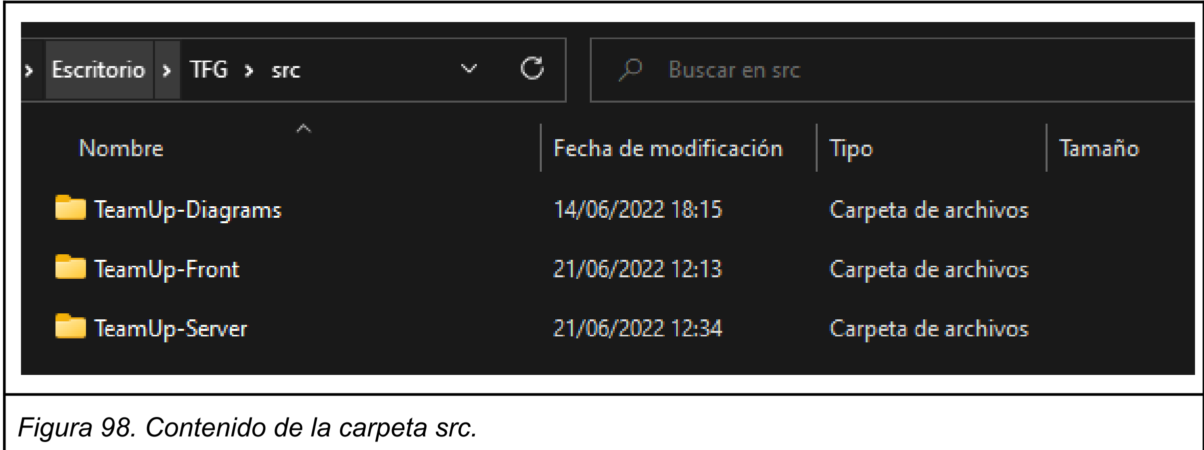

Figura 98. Contenido de la carpeta src.

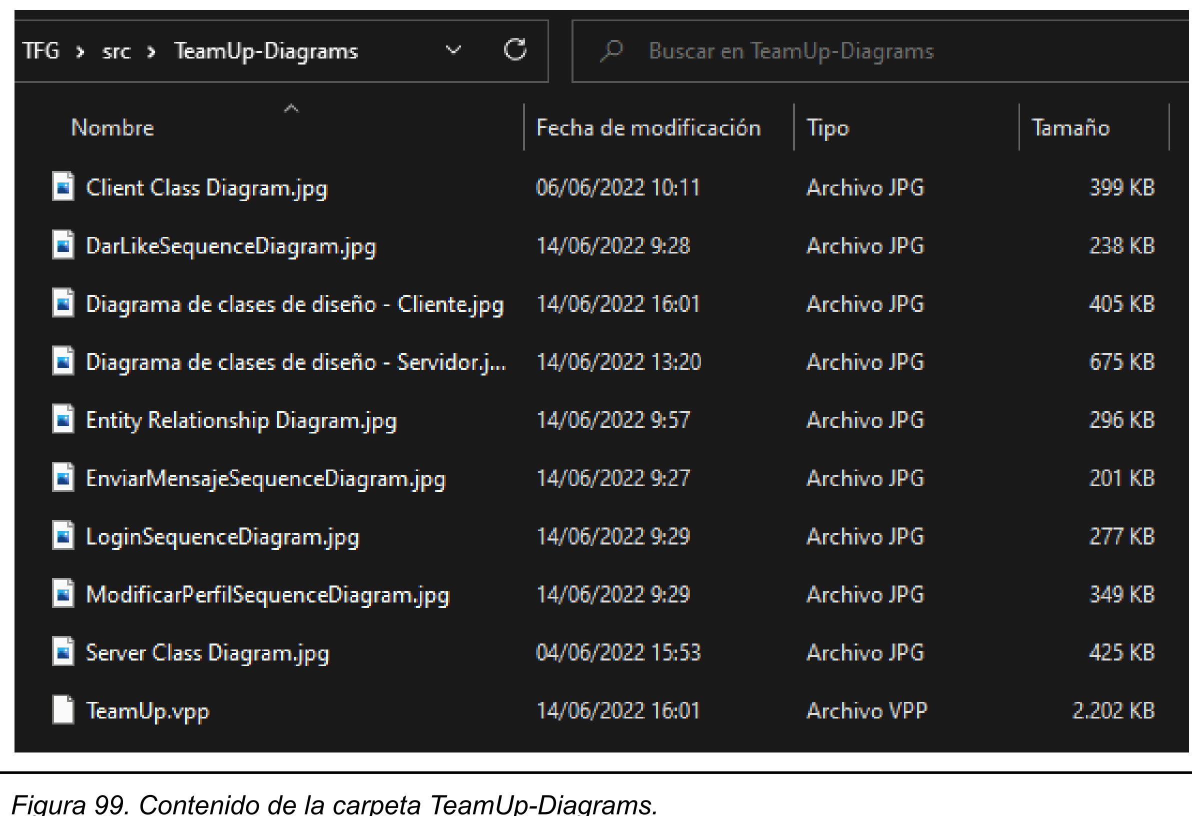

Figura 99. Contenido de la carpeta TeamUp-Diagrams.

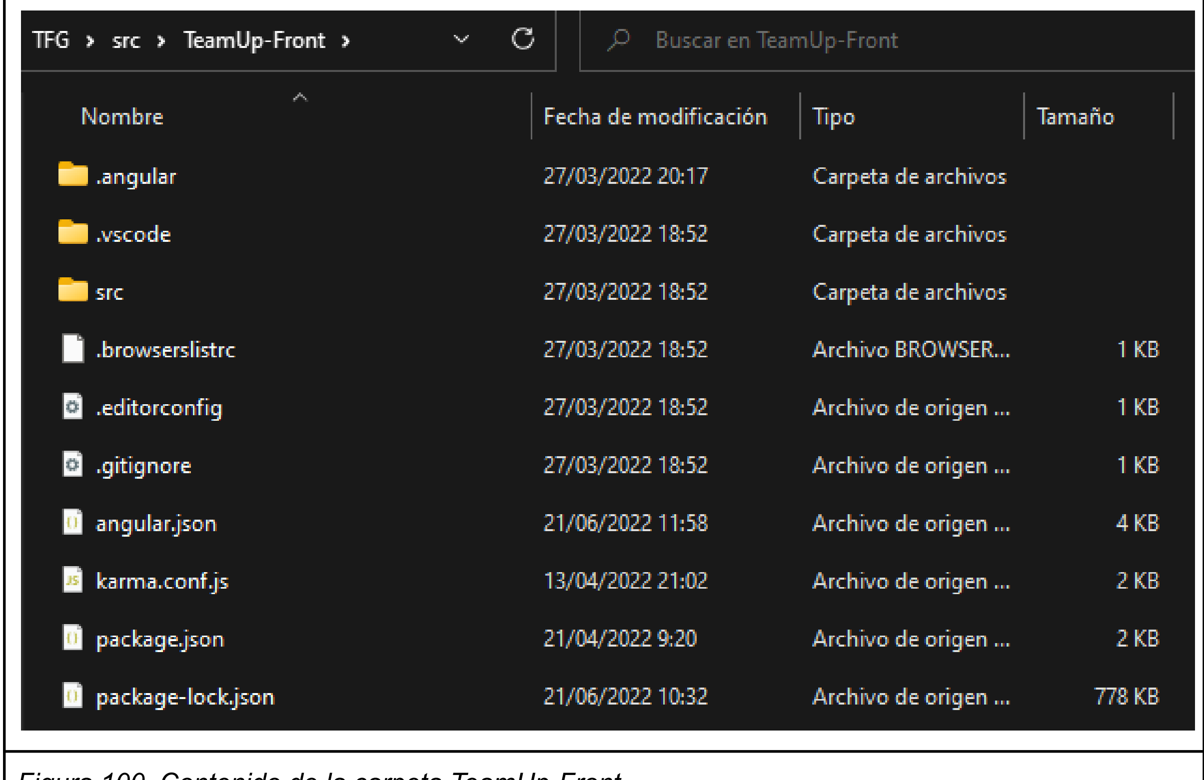

Figura 100. Contenido de la carpeta TeamUp-Front.

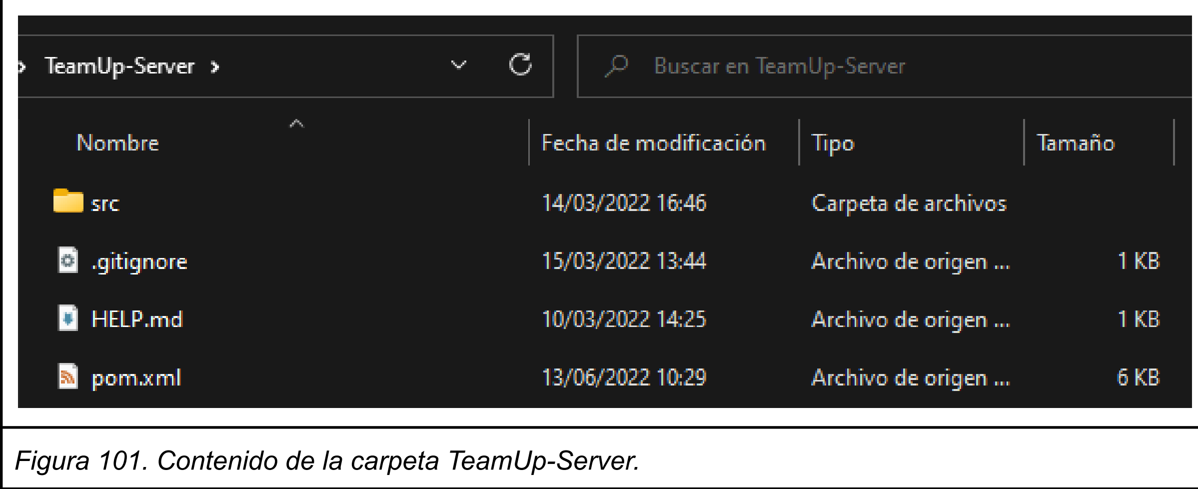

Figura 101. Contenido de la carpeta TeamUp-Server.

La carpeta *src*, Figura 98, contiene el proyecto con código fuente de la aplicación servidor en la carpeta *TeamUp-Server*, visible en la Figura 101, el proyecto con el código fuente de la aplicación cliente en la carpeta *TeamUp-Client* como se ve en la Figura 100, y por último contiene la carpeta *TeamUp-Diagrams* como se observa en la Figura 99, con los archivos fuente de Visual Paradigm en formato *.vpp*, con los diagramas expuestos en la memoria, junto a una serie de archivos en formato *.jpg* conteniendo las imágenes de los diferentes diagramas.

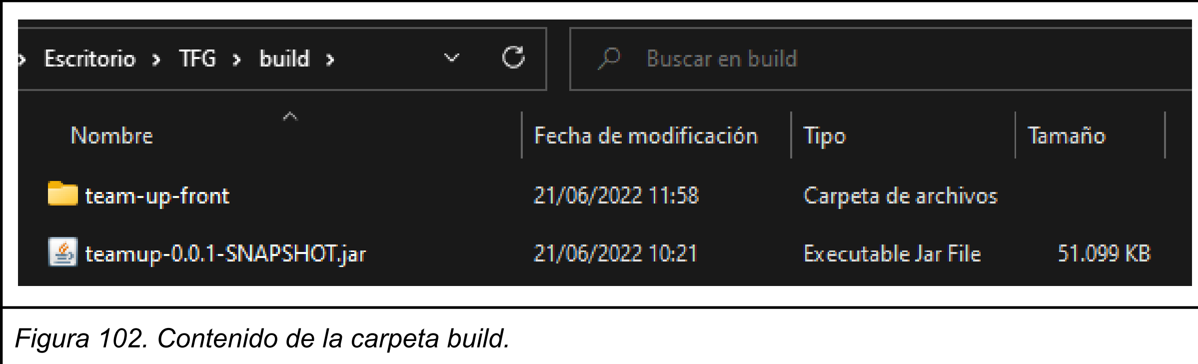

Figura 102. Contenido de la carpeta build.

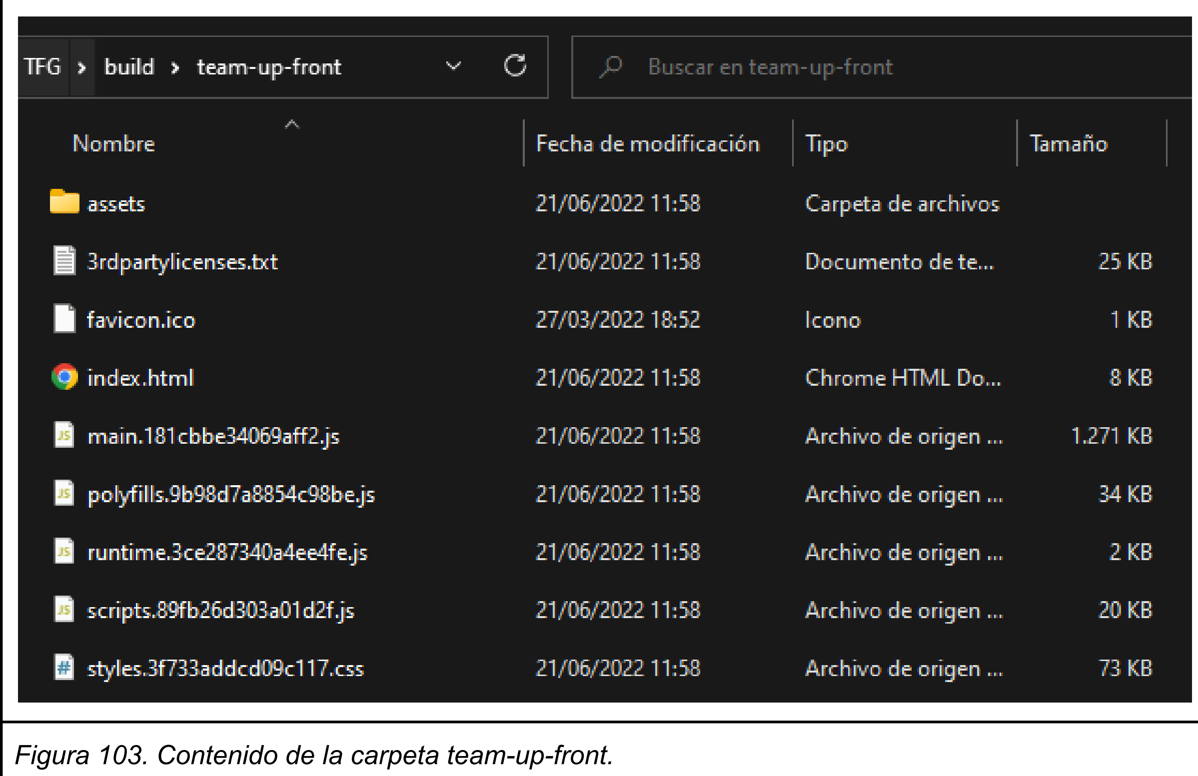

Figura 103. Contenido de la carpeta team-up-front.

La carpeta *build*, Figura 102, contiene los archivos necesarios para ejecutar el servidor y el cliente, esta contiene una carpeta llamada *team-up-front*, Figura 103, que contiene los archivos para ejecutar la aplicación cliente, y además está un archivo llamado *teamup-0.0.1-SNAPSHOT.jar* que contiene el ejecutable del servidor.

Para construir la aplicación a partir de los ficheros fuente, véase Apéndice 2.

Apéndice 2. Manual de Instalación del sistema

El sistema está compuesto por dos aplicaciones, una de ellas es el servidor, y la otra es el cliente. Para instalar y ejecutar el sistema debemos de ejecutar por separado cada una de las aplicaciones de forma independiente.

Instalación del Servidor

```
PS C:\Users\alber\Desktop\TFG\build> java -jar .\teamup-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.4)

2022-06-21 20:47:11.056  INFO 1276 --- [           main] e.albertolongo.teamup.Te
amUpApplication  : Starting TeamUpApplication v0.0.1-SNAPSHOT using Java 17.0.2 o
n PC-Alberto with PID 1276 (C:\Users\alber\Desktop\TFG\build\teamup-0.0.1-SNAPSHO
T.jar started by alber in C:\Users\alber\Desktop\TFG\build)
```

Figura 104. Lanzamiento del servidor en local a partir del archivo .jar

Para iniciar el servidor utilizando los archivos ejecutables, ejecutamos el archivo *teamup-0.0.1-SNAPSHOT.jar* dentro de la carpeta *build,* utilizando el comando *"java -jar ruta_del_archivo"*, como se puede observar en la Figura 104.

A continuación explicaremos cómo ejecutar el servidor a partir de los ficheros fuente. Partiremos del proyecto que contiene el código fuente, con los archivos de la especificación sin generar. Comenzaremos por abrir el proyecto con el IDE[54] de preferencia. En nuestro caso, utilizaremos *IntelliJ IDEA*. Posteriormente, debemos de generar las interfaces generadas por la especificación OpenAPI.

Para ello, debemos asegurarnos de que en la carpeta */src/main/resources/static* se encuentre el archivo de especificación. Una vez tengamos el archivo de especificación dentro de la carpeta, comprobaremos en el archivo de configuración pom.xml que la ruta de entrada para el plugin generador como puede observarse en la Figura 52 sea correcta.

Una vez tengamos esta configuración hecha, ejecutaremos la aplicación con Maven con el comando *install*, como se observa en la Figura 54. Una vez ejecutada la aplicación, nos habrá generado los archivos necesarios para poder ejecutar el servidor. Existen dos maneras de ejecutar el servidor, comenzaremos explicando la

[54] Entorno de desarrollo integrado

manera de lanzarlo desde el propio IDE. Al usar Spring Boot, lanzaremos el proyecto pulsando en el botón correspondiente en el IDE utilizado.

Sin embargo, otra manera que podemos realizar para ejecutar el servidor una vez hayamos generado el código es dirigirnos a la carpeta */target* del proyecto y ejecutar el archivo *teamup-0.0.1-SNAPSHOT.jar* mediante la consola con el comando "java -jar ruta_del_archivo".

De cualquier forma, en este momento el servidor junto a la base de datos ya estarían operativos. El servidor se lanza en *localhost* sobre el puerto 8080.

Instalación del Cliente

Para iniciar la aplicación cliente, primero será necesario instalar *http-server* mediante *node*, para ello usaremos el comando *"npm install http-server"* en una consola.

```
PS C:\Users\alber\Desktop\TFG\build> http-server .\team-up-front\ -p 4200
Starting up http-server, serving .\team-up-front\

http-server version: 14.1.1

http-server settings:
CORS: disabled
Cache: 3600 seconds
Connection Timeout: 120 seconds
Directory Listings: visible
AutoIndex: visible
Serve GZIP Files: false
Serve Brotli Files: false
Default File Extension: none

Available on:
  http://25.79.150.4:4200
  http://192.168.1.89:4200
  http://169.254.230.63:4200
  http://127.0.0.1:4200
Hit CTRL-C to stop the server
```

Figura 105. Lanzamiento de la aplicación cliente en local.

A continuación, nos situamos dentro de la carpeta *build*, y para iniciar el cliente ejecutaremos el comando *"http-server ./team-up-front -p 4200"*, como se observa en la Figura 105. El comando *http-server* por defecto lanza la aplicación sobre el puerto 8080, pero ya que nuestro servidor ocupa dicho puerto, lo cambiaremos por el puerto 4200, ya que el servidor admitirá peticiones desde dicho puerto.

A continuación explicaremos cómo ejecutar la aplicación cliente utilizando los ficheros fuente. Partiremos como en el caso del servidor con el código fuente del

proyecto. De igual manera, lo abriremos con nuestro IDE de preferencia, en nuestro caso, lo abriremos con Visual Studio Code, siendo el IDE recomendado para ejecutar el proyecto. Una vez abierto el proyecto, abriremos una nueva terminal, y ejecutaremos el comando *“npm install”*. Este comando descarga e instala las dependencias necesarias indicadas en el archivo *package.json*. Una vez tengamos el proyecto junto a sus dependencias, ejecutaremos el comando *“ng serve”*.

En este momento la aplicación cliente estaría operativa en *localhost* sobre el puerto 4200.

Apéndice 3. Manual de Usuario

Una vez ambas aplicaciones se encuentren operativas, cuando entremos en la aplicación, nos encontraremos en la pantalla de inicio como se muestra en la Figura 34. En esta pantalla únicamente podemos avanzar a la pantalla de inicio de sesión.

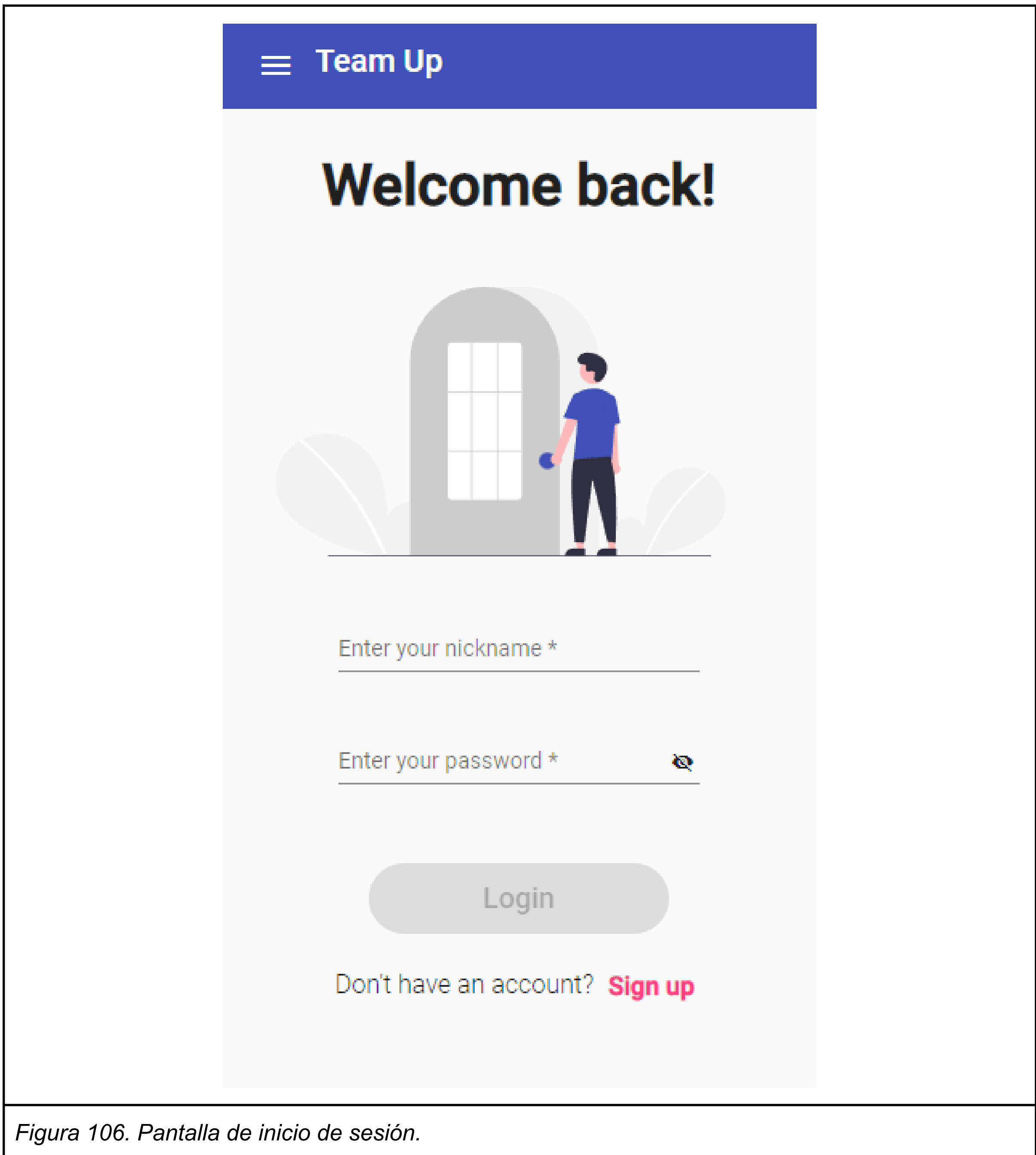

Figura 106. Pantalla de inicio de sesión.

Como se puede observar en la Figura 106, en la pantalla de inicio de sesión podemos ingresar nuestras credenciales e iniciar sesión o dirigirnos a la página de registro.

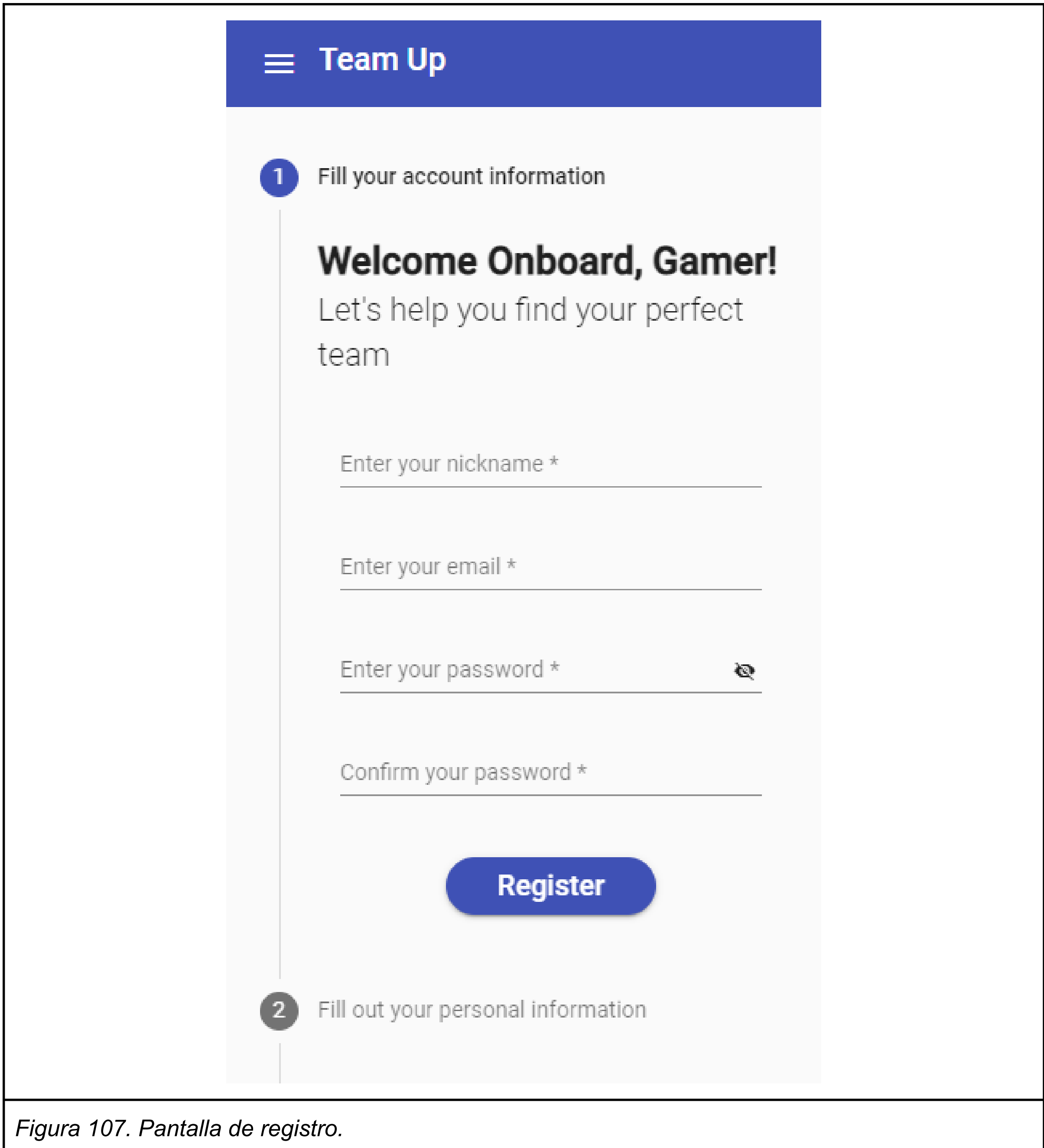

Figura 107. Pantalla de registro.

En la pantalla de registro, Figura 107, podemos rellenar los datos de los diferentes campos para registrarnos o volver a la pantalla de inicio de sesión. Una vez nos registremos, volveremos a la pantalla de inicio de sesión igualmente para rellenar nuestros credenciales.

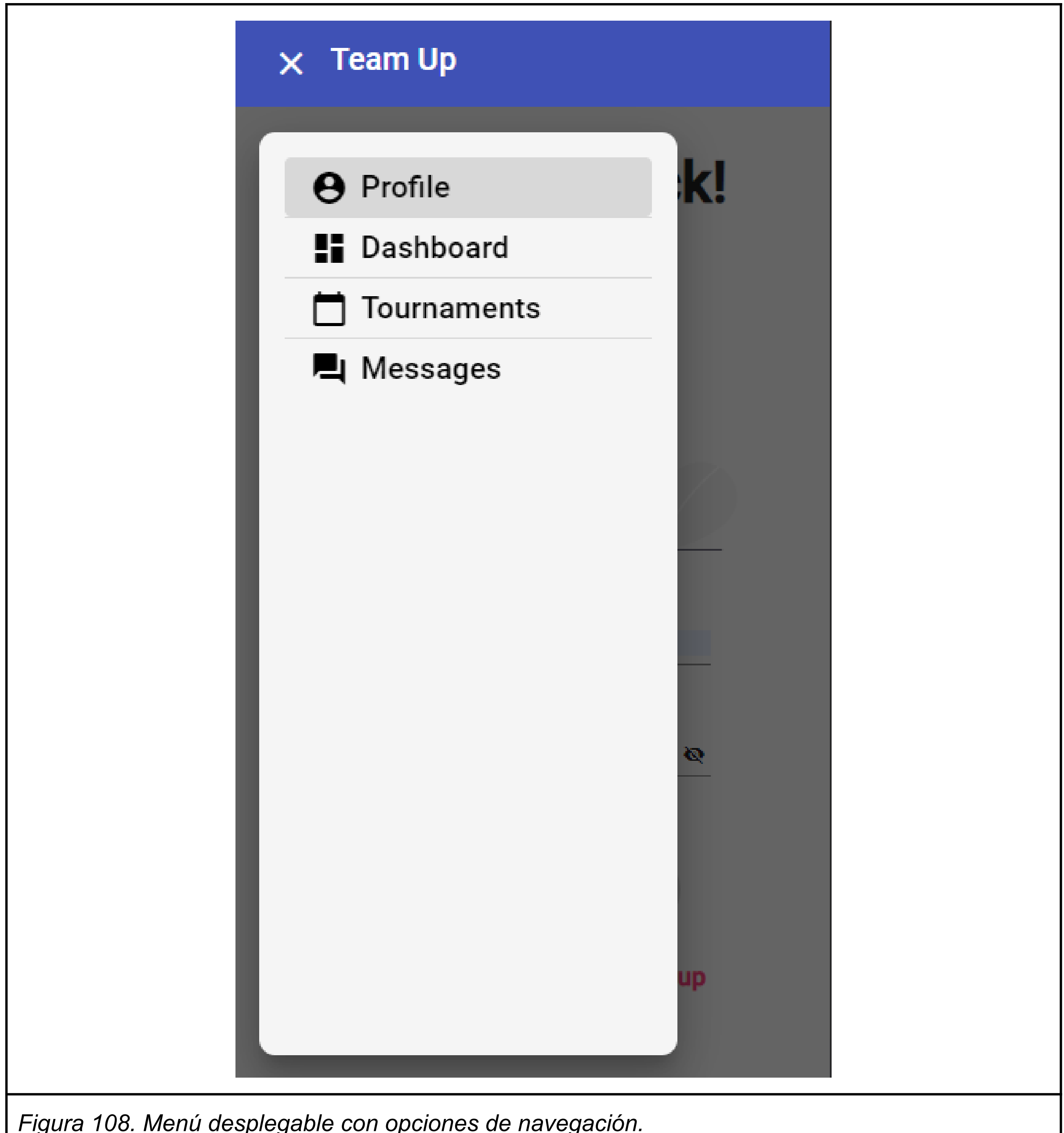

Figura 108. Menú desplegable con opciones de navegación.

En la Figura 108 se observa el menú desplegable con distintas opciones para navegar en la aplicación en caso de no haber iniciado sesión. Si pulsamos sobre cualquier opción nos redirigirá a la pantalla de inicio, pero, si estamos logueados, nos dirigirá a la pestaña en cuestión mientras permanezcamos conectados.

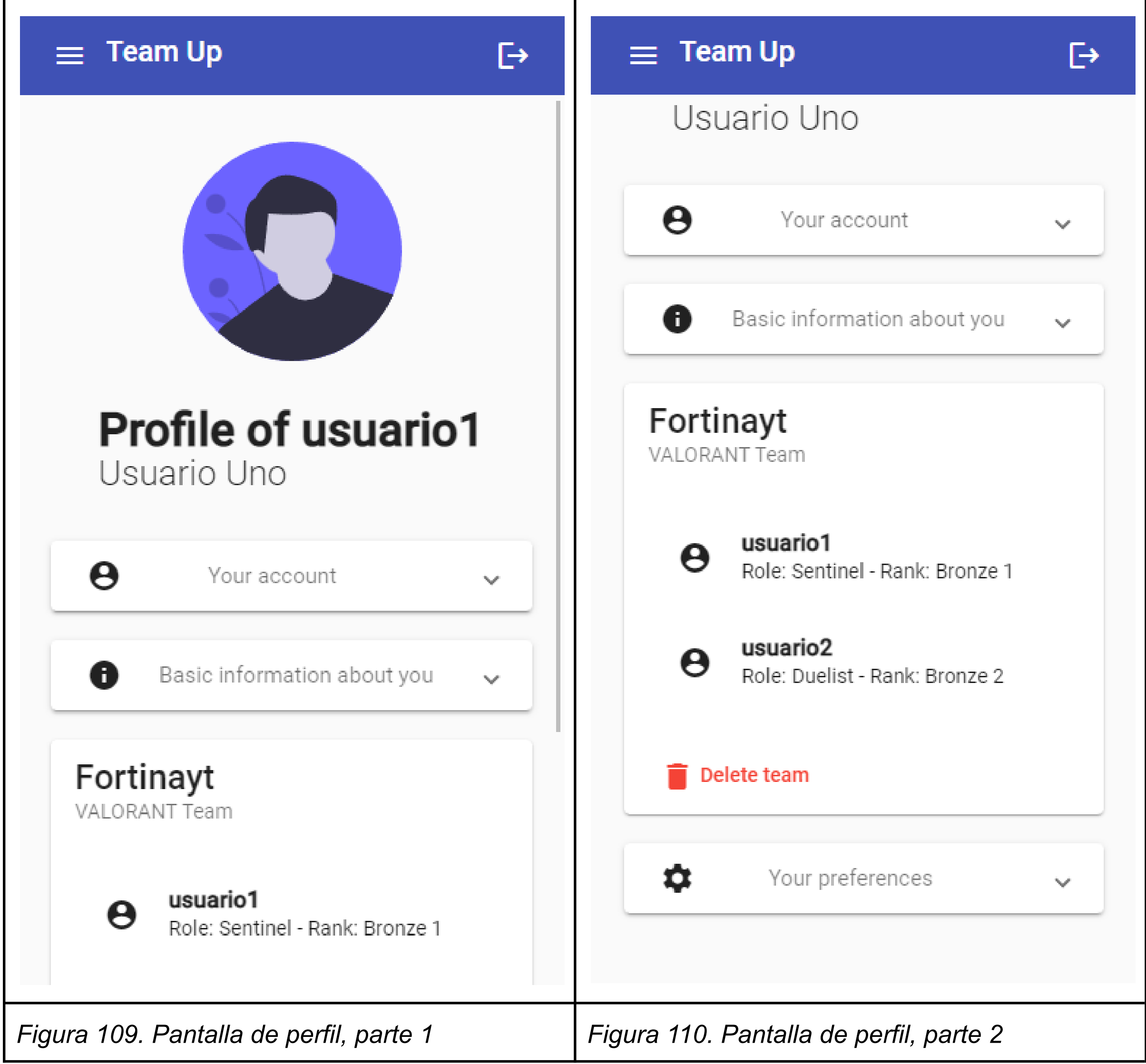

Figura 109. Pantalla de perfil, parte 1

Figura 110. Pantalla de perfil, parte 2

En las Figuras 109 y 110, podemos observar el perfil del usuario, una vez hayamos iniciado sesión, en esta parte podemos modificar los campos del jugador en los menús expandibles como “Your account” o “Basic information about you”. Además de dejar el equipo, modificar tus preferencias. Por último, también en cualquier momento podrás abandonar la sesión con el botón que hay en la parte superior derecha.

Cabe destacar que si modificamos algunos valores como el nombre de usuario o el correo electrónico, la aplicación te avisará que son acciones las cuales debes de cerrar sesión tras realizarlas, puesto que son datos sensibles.

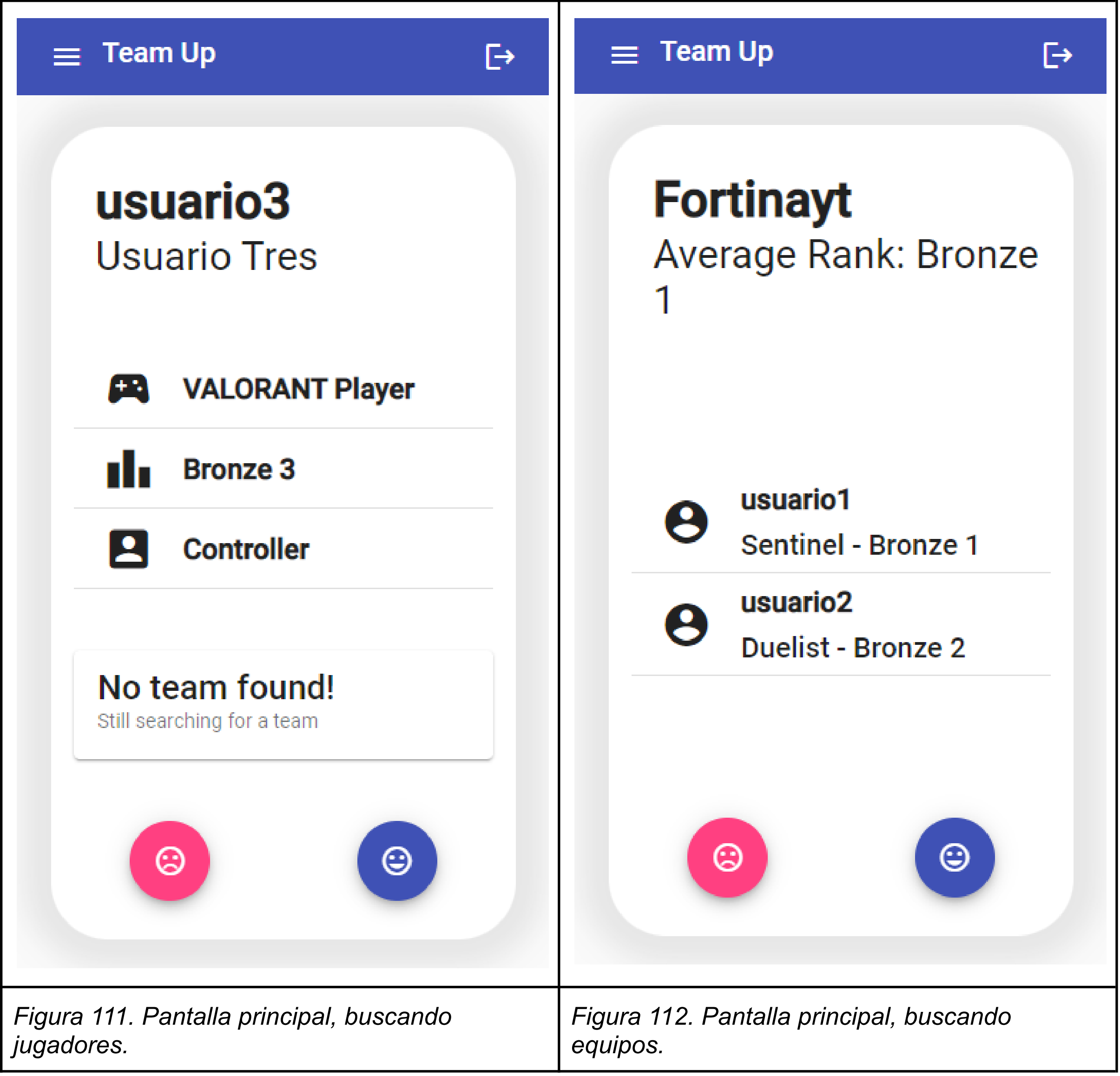

Figura 111. Pantalla principal, buscando jugadores.	*Figura 112. Pantalla principal, buscando equipos.*

En las Figuras 111 y 112 podemos observar la pantalla principal de la aplicación. En la Figura 111 podemos observar la tarjeta de un jugador, que contiene información básica sobre él, así como si tiene equipo o no. En la Figura 112 podemos observar un equipo, con sus miembros y sus respectivos roles y rangos. Además, también podemos ver el rango promedio del equipo así como su nombre.

En esta pantalla podemos deslizar las distintas tarjetas que nos aparecerán, dando *like* o no, para obtener futuros *matches* en caso de que dicho usuario te dé like de vuelta.

También podemos pulsar sobre los botones que aparecen en la parte inferior de la pantalla para realizar la misma acción que mencionamos anteriormente, dar like o no.

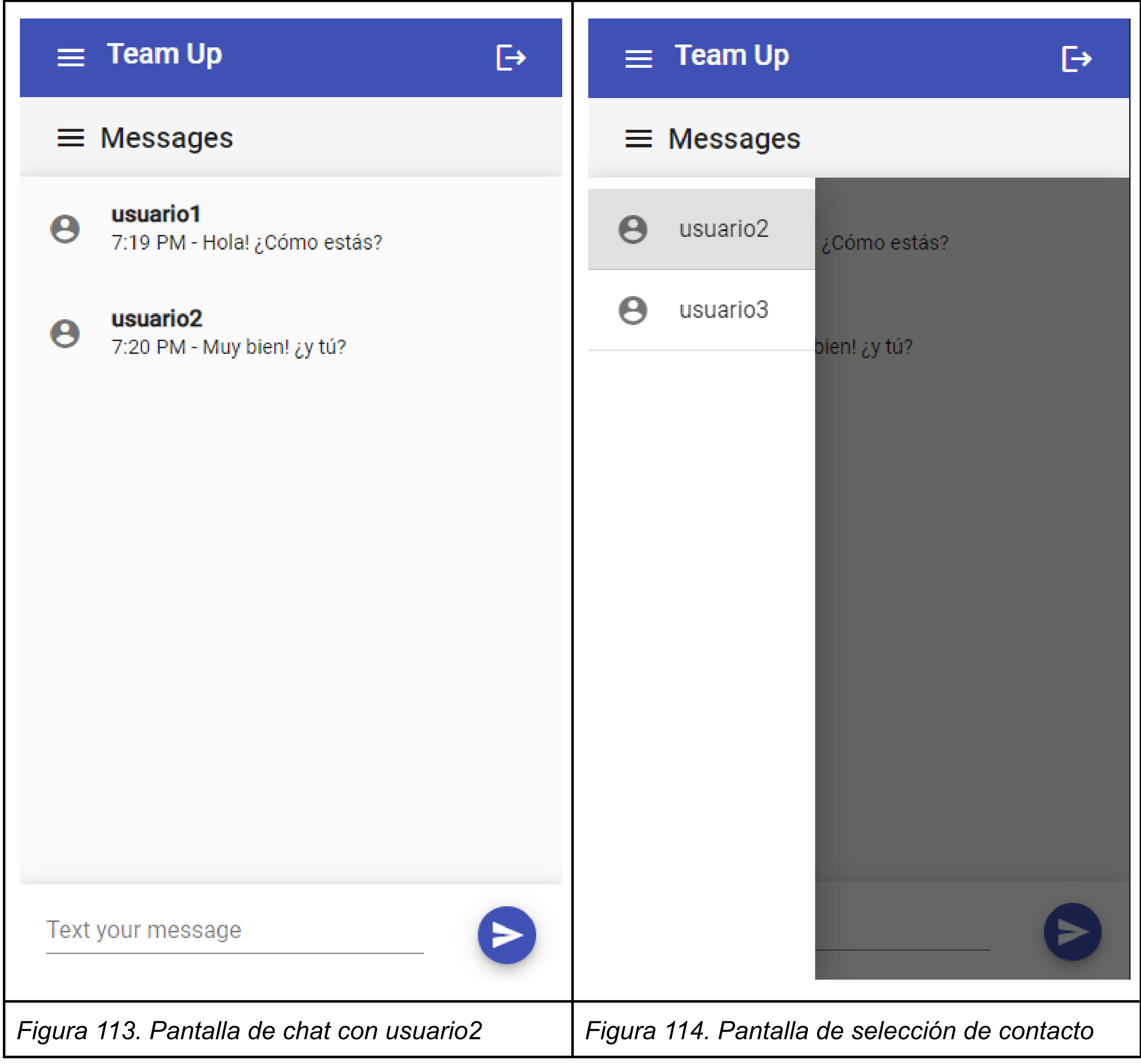

Figura 113. Pantalla de chat con usuario2

Figura 114. Pantalla de selección de contacto

En las Figuras 113 y 114 podemos observar el apartado de mensajería de la aplicación. En el botón arriba a la izquierda del título "Messages" podemos abrir y cerrar el menú de contactos.

En la Figura 113 podemos observar el historial de mensajes entre usuario1 (tú) y usuario2. Si envías un mensaje, este le llegará a la otra persona y podrá contestar.

En la Figura 114 puedes seleccionar el chat que quieres abrir con los diferentes matches que tienes, puesto que un contacto aparecerá en tu lista de contactos sólo si existe un match entre tú y el otro jugador.