Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN SISTEMAS DE INFORMACIÓN

Plataforma de alquiler de pistas de padel

Estudiante: Fernando Álvarez Vázquez

Dirección: Daniel Ismael Iglesia Iglesias

A Coruña, Septiembre de 2023.

A todos los que confiaron en mi.

Agradecimientos

Quiero agradecer este trabajo a todos los profesores que me han dado clase, en especial a mi tutor Daniel, que ha estado disponible en todo momento para cualquier duda que pudiera tener y por su confianza plena en mis capacidades para desarrollar este trabajo. Además agradecer también a todos los compañeros de la FIC que me han podido ayudar con alguna asignatura que se me pudiera enrevesar y a mi prima Patricia la cual me ha ayudado con la redacción de la memoria y el diseño del logo de la aplicación. A todos ellos: Gracias.

Resumen

El objetivo de este trabajo de Fin de Grado es el poder unificar las reservas de pistas de padel en un área metropolitana ya que, hoy en día, el intentar buscar un club donde poder reservar una pista es una odisea debido al auge que esta teniendo este deporte. Para ello, se ha decidido hacer una simulación de un sistema de gestión de reservas de pádel el cual cuenta por un lado con una aplicación móvil programada en flutter donde los clientes pueden reservar una pista de pádel para poder jugar un partido con amigos, además de poder comprar bonos de clases a los propios clubes a través de la aplicación y una aplicación web en la cual los clubes podran gestionar sus pistas y desde la cual se podrá también administrar el sistema. Los pagos se realizaran o bien a través de una pasarela de pago o también se dará la opción de pagar en el club para la reserva de pistas. Además hemos considerado el crear una aplicación web con la que los administradores de los clubes y los administradores de la aplicación puedan gestionar los clubes, pistas, las reservas y los usuarios. Para ello usaremos el framework Vue.js con tecnologías Java, Spring Boot e Hibernate. En cuanto al control del proyecto, utilizaremos una versión personalizada de SCRUM.

Abstract

The aim of this Final Degree project is to be able to unify paddle tennis court reservations in a metropolitan area since, nowadays, trying to find a club where you can book a court is an odyssey due to the boom that this sport is having. For this, it has been decided to make a simulation of a padel reservation management system which has on the one hand a mobile application programmed in flutter where customers can book a padel court to play a game with friends, in addition to being able to buy vouchers for classes to the clubs themselves through the application and a web application in which clubs can manage their courts and from which you can also manage the system. Payments will be made either through a payment gateway or there will also be the option to pay at the club for the reservation of courts. We have also considered creating a web application with which club administrators and application administrators can manage clubs, courts, reservations and users. For this we will use the Vue.js framework with Java, Spring Boot and Hibernate technologies. As for the control of the project, we will use a customised version of SCRUM.

Palabras clave:

- Padel
- Flutter
- Java
- Spring Boot
- Hibernate
- Vue.js
- SCRUM
- Postman

Keywords:

- Padel
- Flutter
- Java
- Spring Boot
- Hibernate
- Vue.js
- SCRUM
- Postman

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El objetivo principal de la aplicación en la que se basa este Trabajo de fin de Grado (TFG), Flash Booking, es la creación de un sistema de gestión de clubes de padel de un área metropolitana, así como promocionar este deporte y las escuelas de padel de los clubes, ofreciendo diferentes posibilidades para todos los niveles de juego. A continuación profundizaremos más en este proyecto, la motivación (sección 1.1) que nos ha llevado a desempeñarlo y los objetivos (sección 1.2) de este.

1.1 Motivación

En las últimas décadas el padel se ha convertido en un deporte mayoritario en España. Tal es la euforia que tiene la gente por practicar este deporte que el lograr reservar una pista de un día para otro se ha convertido en algo milagroso. Es por ello que los clubes de padel se han visto obligados a tener que cambiar sus métodos de reservas de padel. Generalmente se hacían a través de llamadas telefónicas a los clubes, y si en un club no tenían pista, pues habría que realizar otra y otra y otra llamada hasta poder encontrar una pista en la que jugar. Esto ha provocado que la mayoría de clubes cambiasen al método de reserva a través de su pagina web, lo cual hace un poco mas ágil el proceso, pero aun así, habría que ir consultando por las páginas de todos los clubes hasta encontrar uno donde haya una pista disponible.

Es por estos problemas por los que se ha decidido crear este sistema de gestión de reservas, para hacer el proceso de reserva mas eficiente y no perder clientes por lo costoso, refiriéndonos al tiempo que llevaría, que es la reserva de una pista.

1.2 Objetivos

El objetivo de este trabajo es el facilitar el proceso de reserva a los clientes que deseen hacer uso de esta opción, ya que los clubes podrán, a través de la aplicación, gestionar ellos

mismos sus pistas de padel aunque también se permitirá a los clubes poder realizar reservas a través de llamadas telefónicas. Además, los clubes podrán ofrecer bonos de clases a los clientes para fomentar el crecimiento de este deporte y gestionar desde la plataforma todo lo que tenga que ver con su club.

Por otro lado, para intentar hacer del padel algo mas adictivo, hemos añadido un nivel de juego a cada usuario, el cual irá variando dependiendo de los resultados que obtenga en sus partidos. De esta manera, se fomenta un poco el ”pique” entre amigos y se hace de la plataforma una competición continua por ver quién es el que mejor nivel tiene.

1.3 Estructura de la memoria

La estructura de este proyecto estará claramente diferenciada entre los capítulos que mostraremos a continuación, describiendo en cada uno de ellos las partes más importantes de manera resumida.

- **Capítulo 1**: Introducción. En este capitulo mostraremos la primera toma de contacto con el proyecto, explicando las razones por la cual se ha hecho y los objetivos principales.
- **Capítulo 2**: Tecnologías y software. En esta parte de la memoria hablaremos de las tecnologías empleadas en el proyecto, así como del software elegido para su realización.
- **Capítulo 3**: Metodología. Explicaremos la metodología empleada para este proyecto, así como la estimación de costes del trabajo.
- **Capítulo 4**: Análisis. En esta parte hablaremos del análisis de requisitos para implementar la aplicación y de las historias de usuario a realizar.
- **Capítulo 5**: Diseño. Mostraremos los mockups más importantes tanto de la aplicación web como de la app móvil.
- **Capítulo 6**: Implementación. Mostraremos partes del código que se han considerado de mayor relevancia y justificaremos el uso de patrones de diseño.
- **Capítulo 7**: Pruebas. Indicaremos cómo se han realizado las pruebas del proyecto adjuntando imágenes de las diferentes pruebas realizadas para cada parte del proyecto.
- **Capítulo 8**: Conclusiones y futuras mejoras. En este capítulo comentaremos las conclusiones finales del proyecto, posibles mejoras que se podrían implementar en el futuro o que no se han podido implementar por falta de tiempo y escalabilidad de la plataforma.

Capítulo 2

Tecnologías y software

2.1 Tecnologías

A continuación analizaremos las tecnologías usadas para este trabajo y el porqué de su elección.

2.1.1 Java

Para el desarrollo del *backend* hemos optado por el uso de este lenguaje ya que estoy familiarizado con él desde hace años y me siento cómodo programando con Java. Otra característica por la que he elegido Java ha sido por la programación orientada a objetos, ya que en nuestra aplicación hacemos uso de objetos (a través de los constructores) para representar a las clases. Además, desde Java se pueden importar una gran cantidad de paquetes y *framework*s que facilitan mucho el trabajo de programación.

2.1.2 PostgreSQL

Para la administración de la base de datos hemos decidido —el tutor del proyecto y yo— usar PostgreSQL ya que es un sistema de gestión de bases de datos de código abierto y, lo más importante, orientado a objetos. Además cuenta con una interfaz gráfica (PgAdmin) que ahorra mucho tiempo a la hora de crear una base de datos y gestionarla.

2.1.3 Spring

Como ya comentamos antes, Java permite el uso de *frameworks* para programar y Spring es una de las herramientas más populares y flexibles para realizar servicios API Rest de calidad.

2.1.4 Spring Boot

Java Spring Boot (Spring Boot) es una herramienta que hace que el desarrollo de aplicaciones web y micro servicios con Spring sea más rápido y fácil. Spring Boot permite construir el esqueleto de un proyecto en Java desde su pagina web [1], pudiendo añadir todas las dependencias necesarias que se vayan a poder utilizar en el proyecto aunque se puedan añadir más adelante también en el archivo *application.yml*.

2.1.5 Hibernate

Para la creación de las relaciones en la base de datos, así como las consultas que realizamos a PostgreSQL, se utilizarán anotaciones proporcionadas por Hibernate y usando el lenguaje JPQL. Hibernate se encarga de hacer la base de datos persistente, lo cual permite diseñar objetos persistentes y poder gestionarlos a través de las consultas realizadas en JPQL.

2.1.6 Paypal-Java

Para la pasarela de pago hemos decidido trabajar con Paypal, ya que creemos que es la que mejor se adapta a nuestra aplicación y futuro mercado [2]. Paypal nos ofrece ventajas en temas de micro pagos, tiene un paquete disponible en Java que facilita mucho su programación y, además, tiene cuentas en modo *sandbox* [1] para poder probar la pasarela con dinero ficticio. Para realizar la pasarela de pago en el *backend* se ha añadido al archivo *application.yml* un clientId, una clave secreta (clientSecret) y el modo en el que se desea probar la pasarela (sandbox/live). Para inicializar la pasarela es necesario llamar al paquete APIContext con estos datos [3].

```
@Bean
public APIContext apiContext() {
    APIContext context = new APIContext(clientId, clientSecret, mode);
    return context;
}
```

Figura 2.1: Llamada al API de Paypal

2.1.7 Vue.js

Este ha sido el *framework* utilizado para el *frontend* de los administradores web y de los administradores de cada club. Este *framework* es utilizado para crear interfaces de usuario, el cual nos permite combinar tres lenguajes de programación: HTML, JavaScript y CSS. Se ha escogido este *framework* frente a otros por su simplicidad, su clara documentación[4] y su

[1] Modo de pruebas de Paypal. No operacional.

versatilidad. Además ya he trabajado con este *framework* con anterioridad y la experiencia ha sido positiva.

2.1.8 Flutter

Para el desarrollo y la implementación de la aplicación móvil se ha usado Flutter, un *framework* de código abierto desarrollado por Google pensado para el desarrollo de aplicaciones móviles (Android, IOS) y también aplicaciones web. Se ha escogido este lenguaje (dart) ya que, a pesar de ser un lenguaje ”reciente”, ofrece una documentación bastante completa y actualizada y, en añadido, consta de ventajas como la opción de Hot Reload, que permite al programador ver los cambios sin tener que parar la aplicación y volver a ejecutarla, lo que favorece la productividad del creador. La aplicación móvil se ha programado con la versión 11 de Android.

2.1.9 Paypal-flutter

Para acceder a una pagina web en flutter usaremos la versión mas reciente del paquete WebView[5],la cual espera una respuesta de Paypal para verificar si la operación ha sido exitosa o no. La elección de este paquete para el acceso a la pasarela de pago ha sido la simplicidad frente a otros métodos de acceso y la redirección a la aplicación como UrlLauncher, uni-links o deep-links. Todos los paquete usados en flutter se pueden consultar aquí: [6].

2.2 Software

En esta sección listaremos tanto el software empleado como los diferentes IDEs para la programación y software adicional para la realización de pruebas.

2.2.1 Eclipse

Para el desarrollo de la API Rest en Java se ha optado por el uso de eclipse, uno de los IDEs más conocidos para la programación en Java. Además de Java, Eclipse es compatible con una gran variedad de lenguajes, multitud de plugins para ayudar al desarrollador y también con una opción de auto-completado para optimizar la programación.

2.2.2 Dia

Dia es una aplicación de escritorio destinada a la creación de todo tipo de diagramas. Para este proyecto hemos desarrollado un Diagrama Entidad-Relación, un Diagrama de Clases y un diagrama de Casos de Uso para poder analizar, de una manera visual, como interactúan nuestros objetos en la aplicación. (Figura 2.2)

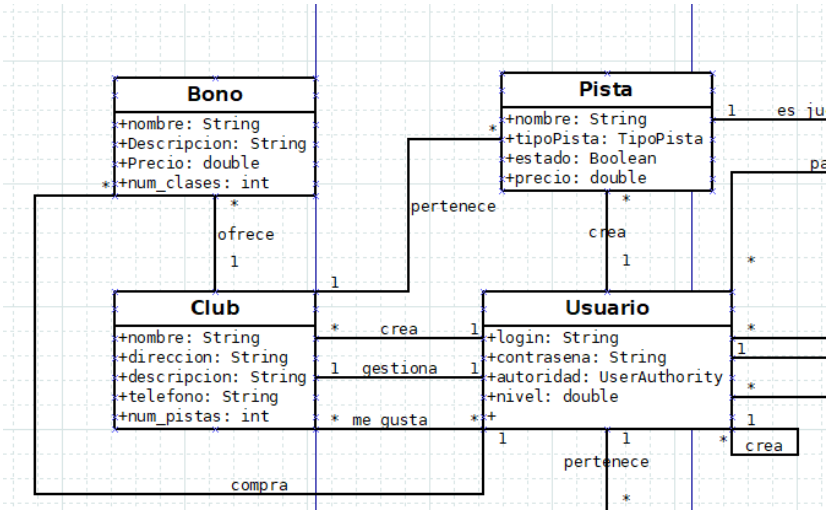


Figura 2.2: Muestra Diagrama ER

2.2.3 VisualStudio Code

En cuanto a la aplicación web, se ha optado por utilizar este IDE, el cual hoy en día es el más utilizado por los programadores de diferentes lenguajes al contar con una amplia lista de plugins y extensiones para diseñar cualquier tipo de aplicaciones. [7] (Figura 2.3).

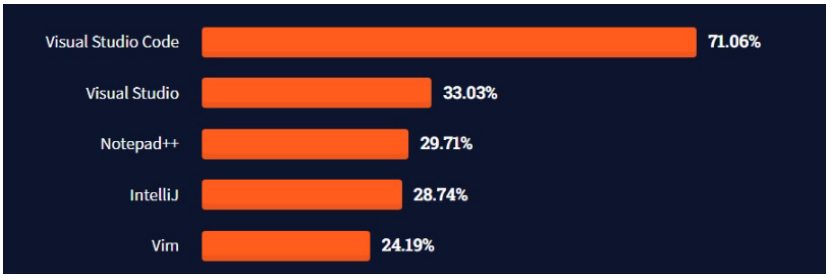


Figura 2.3: Uso de distintos editores por la comunidad

2.2.4 Android Studio

Hemos elegido este IDE para el desarrollo de la aplicación móvil, ya que es un editor de aplicaciones que me resulta basntante familiar. Además, cuenta ya con emuladores predefinidos y listos para su uso, lo cual ahorra bastante tiempo en comparación con otros editores, como VisualStudio Code, con el que hubiera tenido que descargar varios plugins y extensiones para poder ejecutar este proyecto.

2.2.5 Paypal- developer

Desde el navegador accedemos a Paypal como desarrollador para poder crear proyectos nuevos y allí obtener el ClientId y el ClientSecret mencionados anteriormente para poder conectarnos al API de Paypal[8]. (figura 2.4)

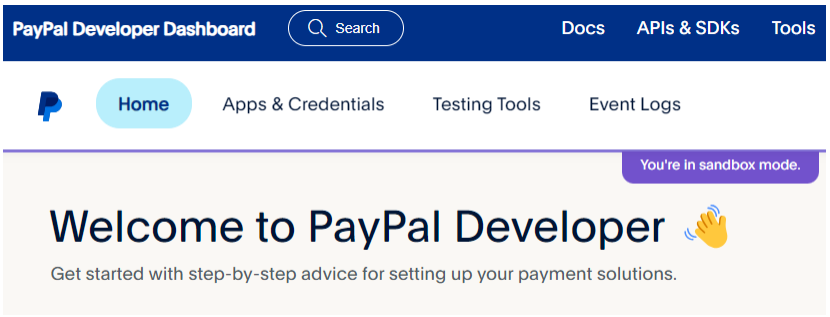


Figura 2.4: Página para desarrolladores Paypal

2.2.6 Postman

Ha sido el software elegido para realizar las pruebas en el API Rest, verificar que todas las peticiones al API se ejecutaban de la manera correcta, intentar hacer peticiones sin los permisos o autorizaciones necesarias y también para controlar los errores que puedan causar. [9]

2.2.7 PgAdmin

Interfaz de usuario para creación de bases de datos, consulta de datos y creación, modificación y eliminación de objetos. [10]

Capítulo 3
Metodología

En este capítulo hablaremos acerca de la metodología empleada para este proyecto desde sus inicios, con la primera reunión para definir la extensión del proyecto, así como el control de tareas y, por último, la estimación de costes.

3.1 Control de tareas

Para llevar un control de las tareas realizadas durante el proyecto hemos creado un documento Excel en el cual detallamos, día por día, los objetivos que se deberían cumplir. Así, en cada reunión quincenal con el tutor del proyecto, se iban ajustando de manera consciente la duración de las tareas futuras. Esto nos sirvió a modo de *backlog* [1] para poder estimar los costes finales del proyecto. Al finalizar este capítulo se muestra gráficamente el reparto de los sprints con sus tareas asociadas a lo largo de los meses de trabajo.

Las dos primeras semanas de trabajo las hemos dedicado a analizar y definir la aplicación, el mercado al cual va dirigido y la escalabilidad de esta. Adentrándonos un poco más hacia el valor de mercado, tenemos 3 diferentes ramas en las cuales creemos que este proyecto podría beneficiar: i) el jugador, para quien agilizamos el proceso de reserva; ii) el club ya consolidado, que vería reducida la carga de trabajo de su personal de atención al cliente y recepción; iii) el club de nueva creación, para no solo darse a conocer si no aumentar sus posibilidades de venta al estar, dentro de la aplicación, al lado de clubes más consagrados. Por ejemplo, cuando los clubes consagrados tenga sus pistas llenas, los jugadores probablemente acudirán a estos clubes menos conocidos para lograr reservar una pista. Durante esta semana también se ha comenzado con el análisis de requisitos, creando aquí un diagrama entidad-relación para darnos una mayor visión acerca del problema que vamos a abordar. Una vez realizado este diagrama decidimos crear también un diagrama de clases para tener una idea de cómo se estructuraría nuestra base de datos.

[1] Listado de todas las tareas que se pretenden hacer durante el desarrollo de un proyecto.

Las siguientes dos semanas estuvieron dedicas a la creación de las primeras entidades (Usuario, Club, Pista) en el API Rest. Por el momento se iban cumpliendo los plazos estipulados al inicio del proyecto ya que estas tareas no contenían ninguna dificultad especial, basándonos en un desarrollo incremental del proyecto. Al final de esta tarea confirmamos el funcionamiento de todas las operaciones de estas tres entidades a través de Postman.

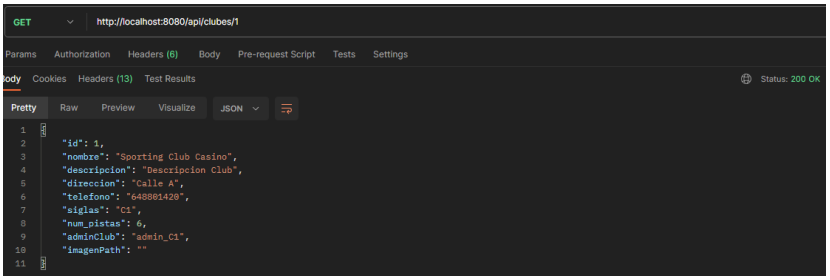


Figura 3.1: Prueba recuperación datos club con id = 1

Para la siguiente reunión con el tutor se esperaba tener completo el API Rest. Y así fue. Se completaron las tareas marcadas para la fecha y quedó la parte del servidor hecha a pesar de que en el futuro hubo que hacer varias modificaciones en esta parte.

Para el siguiente sprint estaba marcado el inicio del desarrollo de la Interfaz de Usuario (UI) de la parte del administrador web (AW). Durante estas dos semanas se realizó toda la parte de este actor y, en añadido, la pantalla de Inicio de la aplicación y un formulario para poder logearse en ella. Este formulario controla que el usuario que accede a la aplicación web sea o un AW o un administrador de club (AC). En este lado de la aplicación no se permiten registros de nuevos usuarios. Solo el administrador web puede crear nuevos usuarios con cualquier autoridad (Administrador Web, Administrador Club o Jugador) a través de un formulario. (3.2).

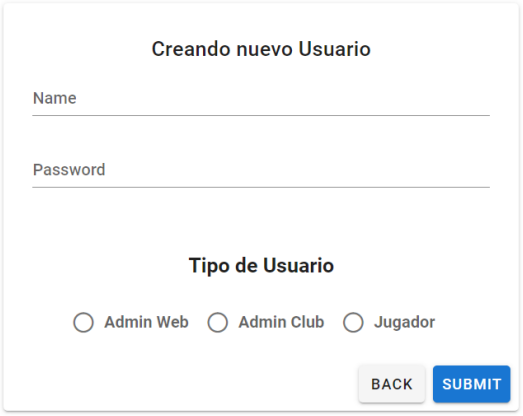


Figura 3.2: Formulario creación Usuario

El siguiente *sprint* consistía en comenzar con la parte del AC, en la cual se ha reciclado mucho código de la parte del AW, como las ventanas de visualización de un club por ejemplo. Encontramos bastante dificultad a la hora de crear las tablas de disponibilidad ya que en un principio no esperábamos que hubiese que controlar demasiados factores cambiantes, entre los que destacan: el de pistas ya reservadas, es decir, la limitación de que al pulsar el botón de reservar solo se ejecutase una reserva y que esta se extendiese en horas consecutivas y, además, la concurrencia a la hora de hacer la reserva, referida a que cuando dos usuarios distintos, independientemente de la autoridad, intenten reservar la misma pista a la misma hora, no sea posible. Esto lo controlamos a nivel de base de datos, añadiendo una *constraint* que controla que solo haya un objeto Partido con la misma pista y la misma hora de inicio.

El la figura 3.3 mostramos un ejemplo de lo que sería la tabla a la que tendrían acceso los AC. En rojo se muestran las franja horarias en las que la pista está reservada y en naranja se indican las franjas horarias donde el AC pretende reservar pista. Cada hueco ocupa una franja fija de 30 minutos.

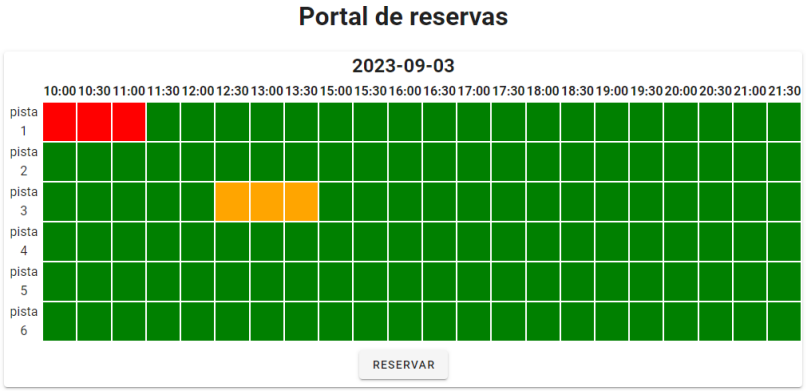


Figura 3.3: Ejemplo tabla de reservas de un día

Para finalizar con la aplicación web, creamos unos menús laterales para facilitar la navegación y hacer la app mas sencilla e intuitiva. Todo esto lo hicimos con apoyo la documentación de vue. [4]

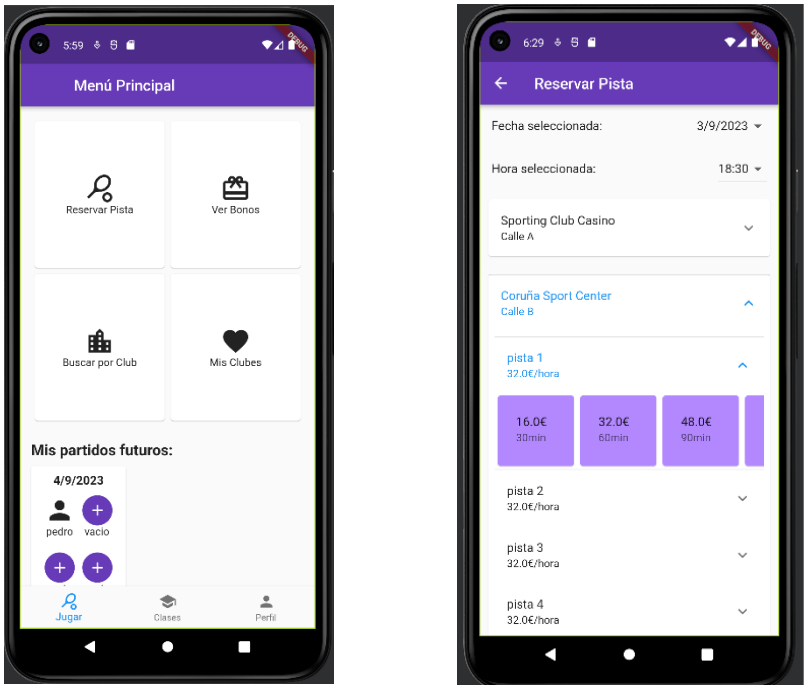
Hasta este punto llevábamos una demora de una semana laborable de cara a la entrega final del proyecto.

En este punto se decidió que los *sprint* pasasen de ser de dos semanas de duración a tan solo una. ya que de esta manera seríamos más precisos en la estimación del tiempo.

La semana del 17 de julio la dedicamos a documentarnos acerca de Flutter[11], preparar el entorno de desarrollo y a comenzar con las primeras implementaciones de la app móvil, más específicamente, con la pantalla de autenticación y con el menú principal. Este menú cuenta con 4 ventanas principales, además de una lista de partidos futuros del Jugador, si los tiene. En la parte inferior se cuenta con un menú inferior para poder acceder a distintas funcionalidades

de la aplicación. (Figura 3.4a).

En el siguiente *sprint* de trabajo se empezó con la lógica de reservas de la misma manera que la planteada para la realización de la aplicación web, a pesar de que la UI se construyó de manera diferente. En este caso optamos por permitir al Jugador introducir solo la fecha de inicio y, a partir de esa fecha, mostrarle una lista de pistas disponibles dentro de cada club; el Jugador puede entonces reservar pista seleccionando una duración predeterminada de 30, 60, 90 o 120 minutos en función de las reservas colindantes. (Figura 3.4b). Para finalizar con este *sprint* se realizó también la opción de buscar disponibilidad por club.



(a) Menú principal (b) Reservar pista en flutter

Figura 3.4: Primeras pantallas flutter

El siguiente *sprint* dio comienzo con el registro de nuevos Jugadores en la aplicación. Para ello creamos un formulario en el cual, a través de 4 preguntas genéricas, se le asigna al nuevo usuario un nivel de juego inicial que se irá ajustando a su nivel real a medida que vaya ganando/perdiendo partidos. Durante este sprint también agregamos los bonos de clases tanto en la parte del *backend* como el la del *frontend*. Creamos también la UI del perfil del Jugador, en el cual se pueden ver los partidos ya jugados, su nivel y su foto de perfil o avatar.

El siguiente *sprint* lo comenzamos creando la pantalla de los detalles de los partidos y dándole al usuario toda la información acerca de estos: la fecha y hora de inicio y fin, el precio total de la pista (dependiendo de su duración), la ubicación del partido (el club en el que disputará y en qué pista) y, en caso de que el partido se haya jugado ya o esté pendiente

de jugarse, diferentes alternativas para el Jugador:

- Si el partido aún no se jugo, se permitirá al usuario añadir o eliminar jugadores del partido.
- Si el partido ya se ha jugado, se podrá añadir un resultado al partido, que si ha estado formado por 4 jugadores, afectará al nivel de todos ellos en la app.

Durante este *sprint* hubo varias complicaciones en cuanto a la actualización de los datos que impidió cumplir con el tiempo establecido para realizar la totalidad de las tareas marcadas.

El siguiente *sprint* fue uno de los más importantes debido a las tareas asignadas. Tocaba hacer la pasarela de pago tanto para el *frontend*[3] como para el *backend*. Esta fue la única tarea a realizar durante este *sprint* por su complejidad, que implicaba una gran inversión de tiempo en la documentación acerca de qué pasarela de pago escoger, controlar su funcionamiento e implementarla a nuestro TFG. A pesar de haber escogido una pasarela de pago relativamente sencilla de implementar en la parte del *backend* (lo cual nos llevo 3 días laborables), para la implementación de la Interfaz de Usuario experimentamos varios problemas a la hora de programarla. El primer fallo fue no escoger bien el paquete con el que trabajar. En un primer momento nos decantamos por usar el paquete *URL Launcher*, en el que la programación de la redirección de la pasarela de pago a la aplicación era bastante confusa y no funcionaba. Se probó también con el paquete *Uri links*, el cual parecía ser el idóneo para trabajar con deep links, pero tampoco conseguíamos arreglar el problema anterior. Hasta que un amigo me aconsejó el uso de *WebView*[5], que abre una página web desde la aplicación y evita el problema de redirección. Todas estas complicaciones causaron una demora de 5 días laborables, con lo cual hubo que hacer un reajuste del cronograma del proyecto. (Figura 3.5).

Una vez acabada la pasarela de pago, empezamos a trabajar la Interfaz de Usuario para intentar hacer de ella una aplicación, además de intuitiva, atractiva a los ojos de los usuarios. Para ello nos centramos, en concreto, en la visualización de los clubes. En ellos decidimos añadir la posibilidad de tener una foto de portada que para mostrar tanto en la lista de clubes de los AW como en la información detallada del club (en la parte de los AW, en la de los AC de la aplicación web y también en la aplicación móvil). Además, también permitimos a los Jugadores poder añadir o editar su foto de perfil. Para ello requerimos hacer peticiones multipart[12]. (Figuras 3.6a y 3.6b).

La última semana previa a la entrega del TFG la dedicamos a finalizar la Interfaz de Usuario y a terminar de redactar la memoria.

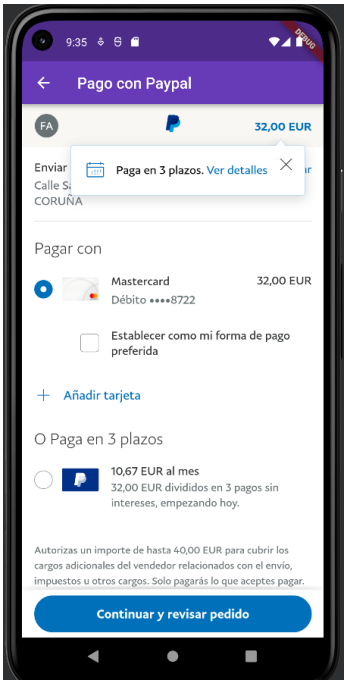
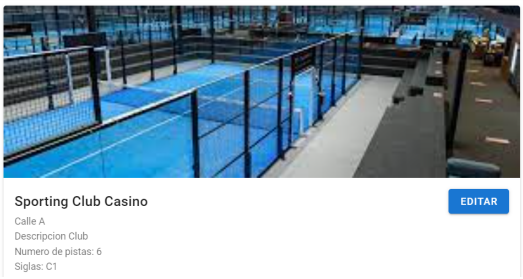


Figura 3.5: Pasarela de pago

(a) Perfil de Usuario



(b) Pagina principal Club

Figura 3.6: Imágenes en flutter

3.2 Estimación de costes vs. costes reales

En esta sección veremos los costes estimados para la realización de este proyecto en cuanto a inversión económica y de tiempo y su comparativa con los valores reales del proyecto.

Como ya se indicó anteriormente, durante varios *sprints* del proyecto experimentamos algunas dificultades, lo cual hizo que el proyecto se demorase o no se pudiesen completar todos los casos de uso estimados para la fecha de entrega.

3.2.1 Costes estimados del proyecto

Para el desarrollo de este proyecto se calculó un total de 320 horas de trabajo por parte del desarrollador junior, el cual, de media en España, cobra alrededor de 6.61 euros/hora. En total, el coste estimado relacionado con el desarrollador sería de 2 115,2 euros. [13]

Para el rol de director del proyecto se calcularon un total de 7 reuniones de entre 30 minutos y 1 hora. Este cargo en España cobra 30,37 euros la hora. En total, se estimó un coste de 106,32 euros para el director del proyecto. [14]

3.2.2 Costes reales del proyecto

En cuanto a los costes reales del proyecto, la duración de este fue de 380 horas trabajadas por parte del desarrollador, lo cual haría pasar su coste a 2 511.8 euros. También se vio afectada la parte del director, el cual pasó de 3.5 horas totales a 8 horas, lo cual significó que este coste ascendiese a 242,96 euros.

3.2.3 Software

En cuanto al software, se ha utilizado para la totalidad del proyecto software libre, por lo que su coste ha sido de 0 euros.

3.2.4 Hardware

Para el desarrollo de este proyecto se ha utilizado solamente un ordenador portátil que implicó un coste de 879 euros. [15]

3.2.5 Comparación de costes

Recurso	Coste estimado	Coste real
Desarrollador Junior	2 115,2€	2 511,8€
Director proyecto	106,32€	242,96€
Software	0€	0€
Hardware	879€	879€
Total	3 100,52€	3 663,76€

Tabla 3.1: Tabla de comparación de costes

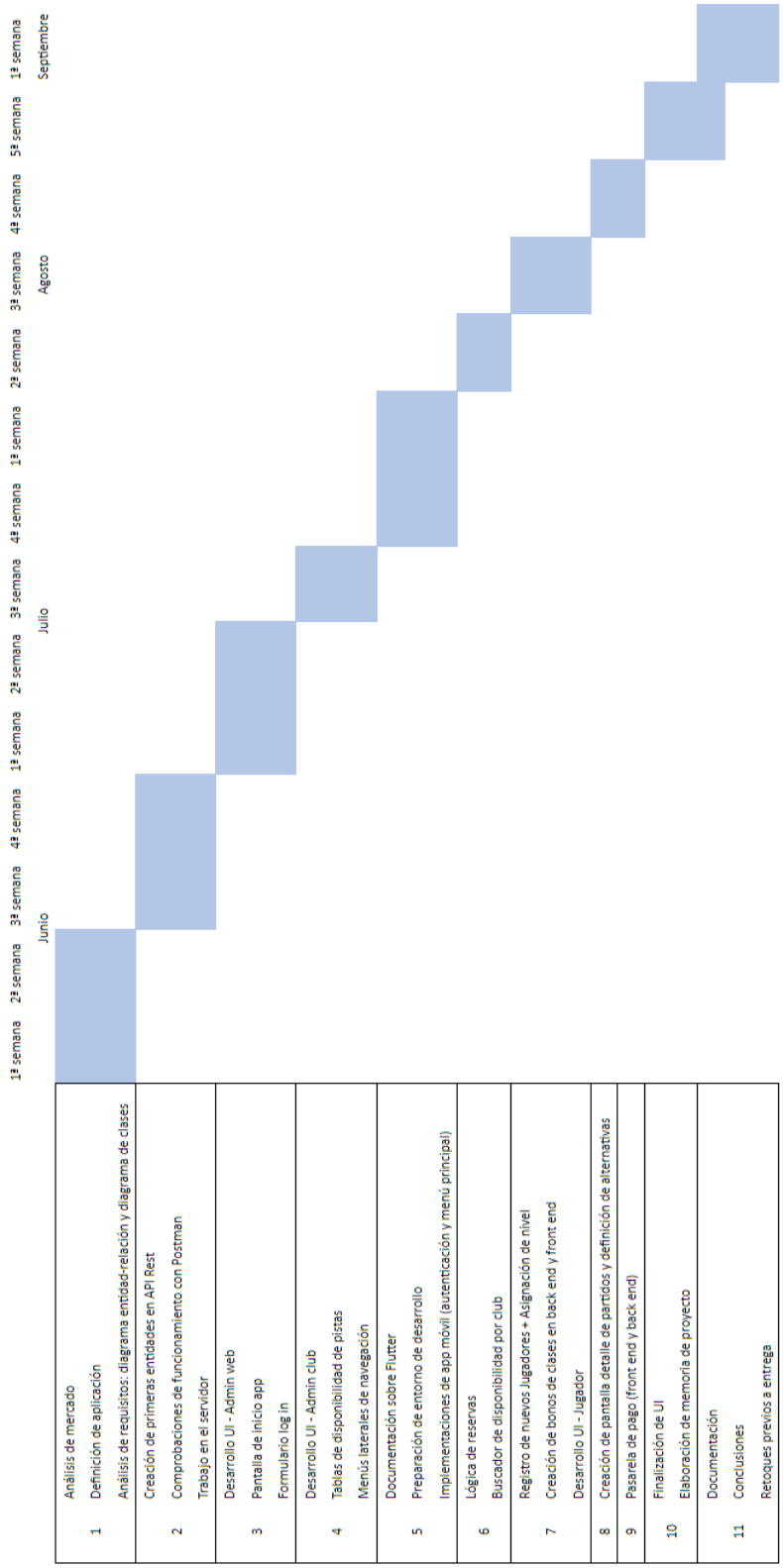


Figura 3.7: Cronograma de Sprints

Capítulo 4

Análisis

Es en este capítulo donde explicaremos la primera parte del proyecto, que consiste en estudiar el mercado al que va dirigido la aplicación y definir los actores que forman parte de la aplicación, así como los casos de uso de cada uno de ellos. En esta fase también se mostrarán el diagrama entidad-relación de la aplicación, el diagrama de clases y un diagrama de casos de uso del actor ”Jugador”.

4.1 Mercado al que va dirigido el proyecto

Previo al inicio del análisis e implementación de este Trabajo de fin de Grado, se ha realizado un estudio de mercado de la aplicación en el cual se sacaron varias propuestas de valor asociadas:

- En primer lugar, la aplicación puede potenciar el deporte en la ciudad y sus alrededores gracias al afán de superación y competitividad al que invita al usuario, que puede ir incrementando su nivel a medida que completa partidos. Esto generará un enganche del Jugador en paralelo al enganche que, de por sí, ya implica este deporte.
- Por otro lado, la aplicación supone un escaparate de incremento de ventas para los clubes, no solo por ser una cómoda vía de reserva de pistas sino también porque permite la venta de bonos de clases.
- Para terminar, los clubes de reciente creación también pueden ver en esta aplicación una oportunidad para darse a conocer y aprovecharse de la autoridad de los clubes ya consagrados que se muestran en la aplicación. Esto, referido a que, cuando un Jugador busque pista en un club popular y este se encuentre sin disponibilidad, lo más probable es que decida continuar con su decisión de reservar pista en otro club independientemente de su nombre de marca.

4.2 Actores y casos de uso

Para el diseño de la aplicación contamos con 3 actores claramente diferenciados: jugadores, administrador de club (AC) y administrador web (AW). A continuación se definen y muestran los casos de uso para cada uno de ellos.

4.2.1 Actores

Admin Web:

- **HU 01:** un usuario puede autenticarse en la aplicación web si tiene la autoridad "Admin-Web".
- **HU 02:** como AW, se podrá acceder a la lista de clubes, en la cual se pueden ver la totalidad de los clubes que gestiona la aplicación.
- **HU 03:** como AW, se podrá crear un club a través de el botón "+" en la parte superior derecha de la ventana de la lista de clubes.
- **HU 04:** desde la lista de clubes, como usuario AW se podrá clicar en cualquier club y se mostrará una ventana con los detalles de dicho club.
- **HU 05:** desde esta página, que muestra los detalles del club, podremos acceder a un formulario en el cual se podrán editar los datos del club. Todos menos la imagen de portada, ya que esta función solo se le permite a los administrador de club.
- **HU 06:** desde la página de detalles del club, el AW podrá eliminar el club de la base de datos. Al eliminar el club, también se eliminarán sus pistas y sus partidos.
- **HU 07:** desde la misma página de detalles de club se podrá añadir un AC en caso de que no se haya añadido uno previamente.
- **HU 08:** también en esta pantalla se podrán gestionar las pistas del club, pudiendo crear una nueva, editarla o eliminarla.
- **HU 09:** como usuario AW podremos crear cualquier tipo de usuario a través de un formulario de creación.
- **HU 10:** como usuario AW podremos ver una lista de todos los usuarios autenticados en la aplicación separándolos por sus roles. Desde esta lista también podremos borrar de la base de datos a cualquier usuario.
- **HU 11:** Como administrador web podremos desautenticarnos de la aplicación con el botón de la derecha de todo de la *app bar*.

Admin Club

- **HU 12:** un usuario puedo autenticarse en la aplicación web si tiene autoridad "Admin-Club".
- **HU 13:** desde la pantalla principal de los AC se podrá acceder también a una pantalla similar a la de los AW pero con menos funcionalidades. Siendo un administrador de club se podrán editar los detalles del club, y ahora sí, en este formulario se permitirá añadir una foto de portada del club.
- **HU 14:** desde los detalles del club también se podrá ver la lista de pistas que tiene ese club y se podrá editar y eliminar.
- **HU 15:** desde la pantalla principal de los AC se podrá ver la lista de partidos que hay ese día, con la posibilidad de editar cada partido o eliminarlo, además de navegar por los días para visualizar partidos futuros o pasados.
- **HU 16:** en la pantalla de reservas se visualizará, a través de tablas, la disponibilidad de las pistas del club: rojo, no disponible; verde, disponible y naranja, reservada.
- **HU 17:** en la pantalla de reservas un AC podrá reservar una pista en franjas horarias consecutivas. Solo se permite la reserva de pistas con una semana de margen.
- **HU 18:** como administrador de club se podrá ver una lista de bonos de clases que ofrece el club.
- **HU 19:** como administrador de club se podrán editar los bonos de clases ofertados por el club.
- **HU 20:** como administrador de club se podrán eliminar los bonos de clases ofertados por el club.
- **HU 21:** como administrador de club se podrán crear bonos de clases.
- **HU 22:** como administrador de club podremos desautenticarnos de la aplicación con el botón de la derecha de todo de la *app bar*.

Jugador

- **HU 23:** para acceder a la aplicación, el usuario deberá, o bien autenticarse, o bien registrarse como usuario nuevo rellenando un formulario sencillo el cual definirá su nivel de juego.

- **HU 24:** como Jugador autenticado, el usuario podrá reservar una pista indicando la fecha y hora de inicio, seleccionando el club y la pista donde desea jugar e indicando la duración del partido. Una vez se seleccionan todos estos datos, la pista quedará bloqueada hasta que el usuario confirme la reserva y elija el método de pago. Cuando la pista ha sido pagada, queda reservada, por lo que no se mostrará disponible en esa franja horaria para el resto de usuarios de la aplicación.
- **HU 25:** como jugador autenticado se puede ver la lista de partidos futuros en la página de inicio.
- **HU 26:** como jugador autenticado se puede añadir o eliminar jugadores de partidos futuros.
- **HU 27:** como jugador autenticado también es posible ver una lista de bonos de clases disponibles en los clubes, mostrando su precio y el club que los oferta.
- **HU 28:** como jugador autenticado, se puede comprar un bono de clases a través de la pasarela de pago.
- **HU 29:** el jugador autenticado tiene acceso a la lista de bonos comprados a través del menú lateral en la pestaña ”perfil”.
- **HU 30:** como jugador autenticado se puede buscar un club en concreto, y ahí ver la información del club y seleccionar una pista para realizar una reserva.
- **HU 31:** como jugador es posible ver una lista llamada ”Mis partidos jugados” a través del perfil.
- **HU 32:** también es posible añadir un resultado a ”Mis partidos jugados” si todavía no se ha añadido. Este resultado cambiará el nivel de juego de todos los jugadores de el partido estuvo formado por 4 jugadores.
- **HU 33:** como jugador autenticado se puede modificar la foto de perfil de su usuario.
- **HU 34:** como jugador autenticado es posible editar el login.
- **HU 35:** un usuario autenticado puede desautenticarse de la aplicación.

4.3 Diagramas

4.3.1 Diagrama de clases

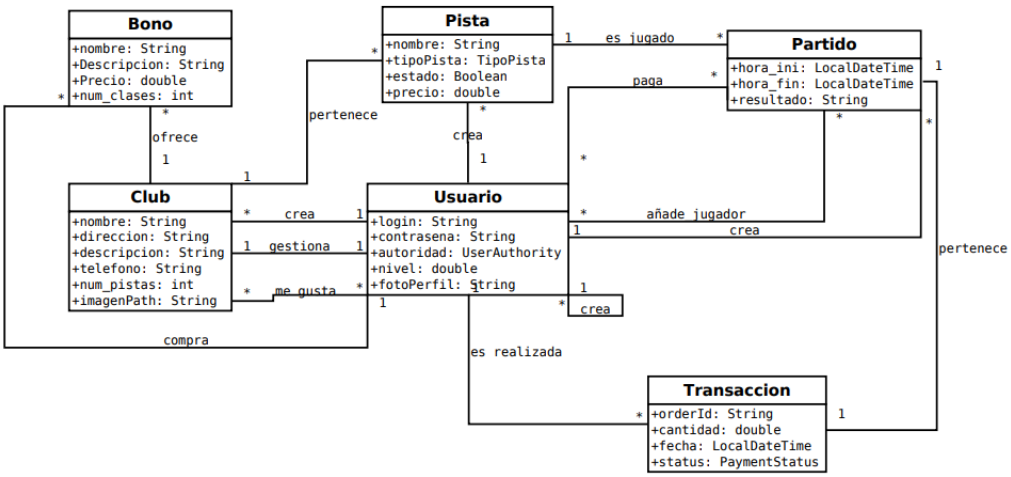


Figura 4.1: Diagrama de clases

4.3.2 Diagrama Entidad-Relación

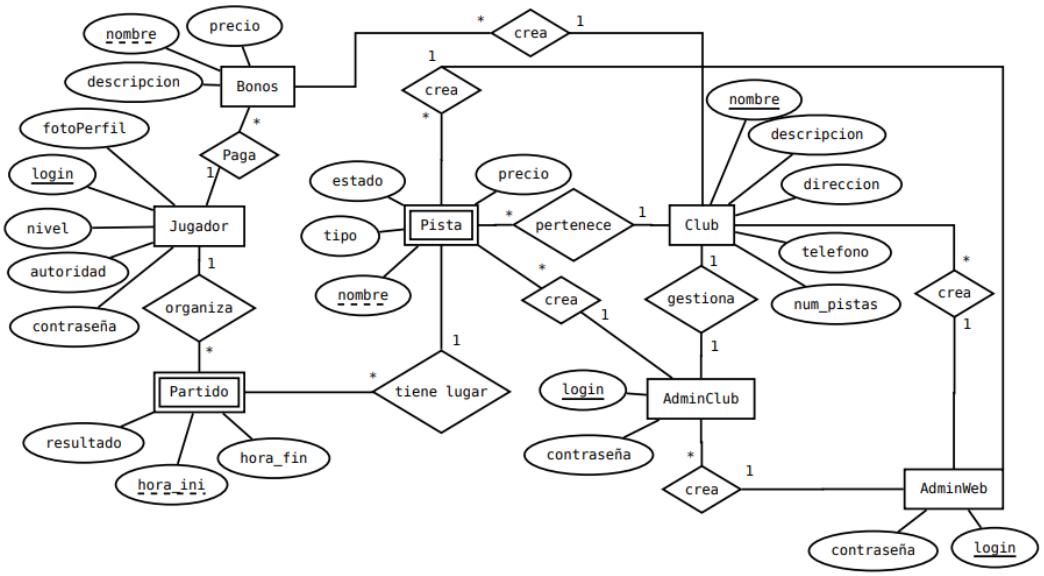


Figura 4.2: Diagrama Entidad-Relación

4.3.3 Diagramas Casos de Uso

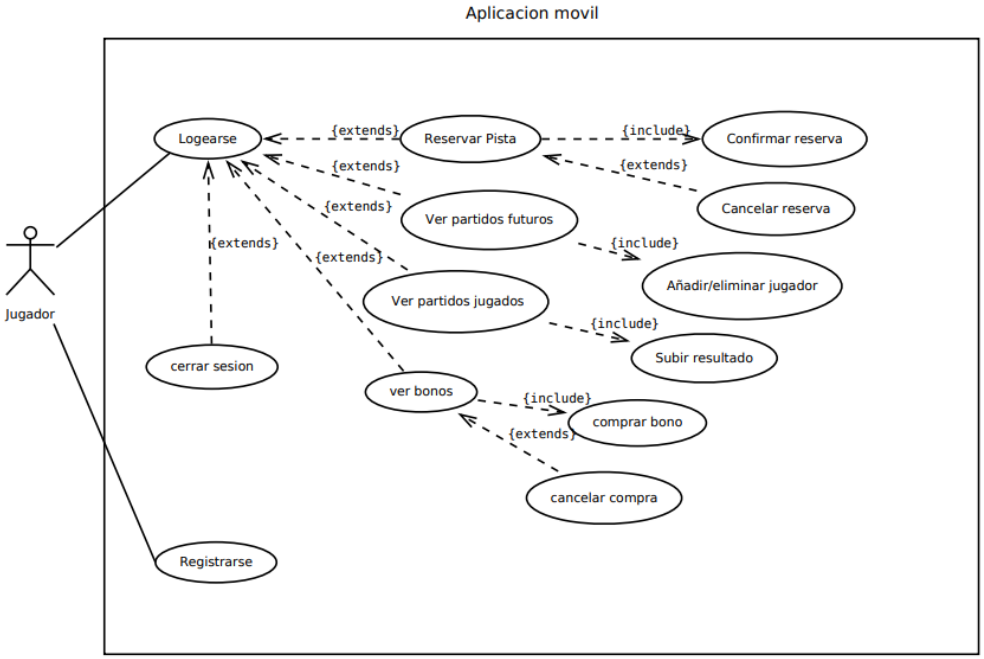


Figura 4.3: Diagrama Casos de Uso Jugador

Capítulo 5

Diseño

En este capitulo se mostrarán los *mock-ups* realizados para el proyecto. Hemos decidido solo mostrar los que consideramos más importantes.

5.1 Aplicación web

- Formulario de inicio de sesión:

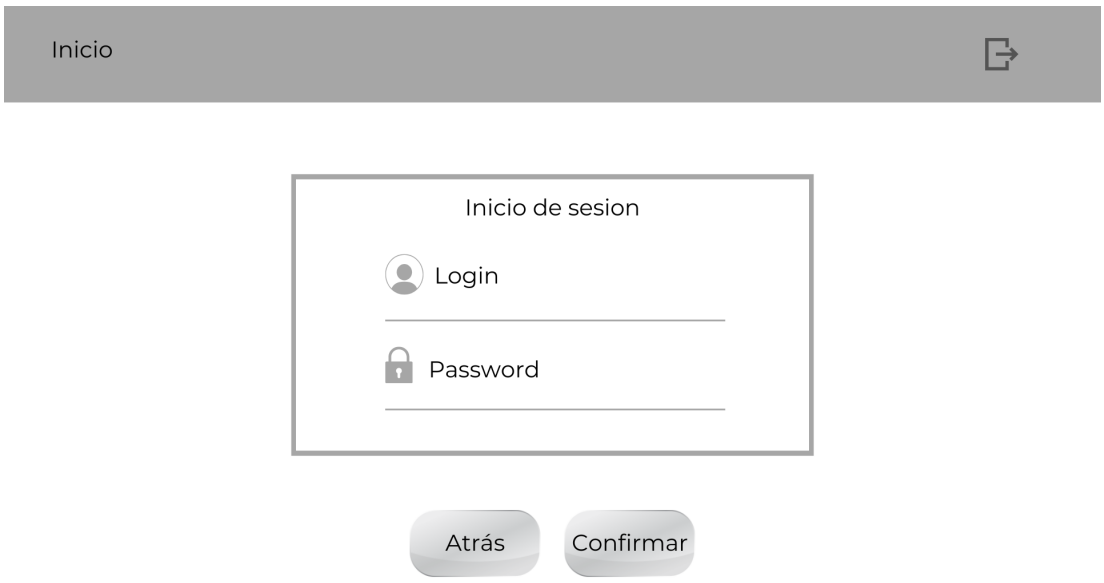


Figura 5.1: Login Web

- Menú principal (administrador web)

Figura 5.2: Menú principal Web

- Listado de clubes (administrador web)

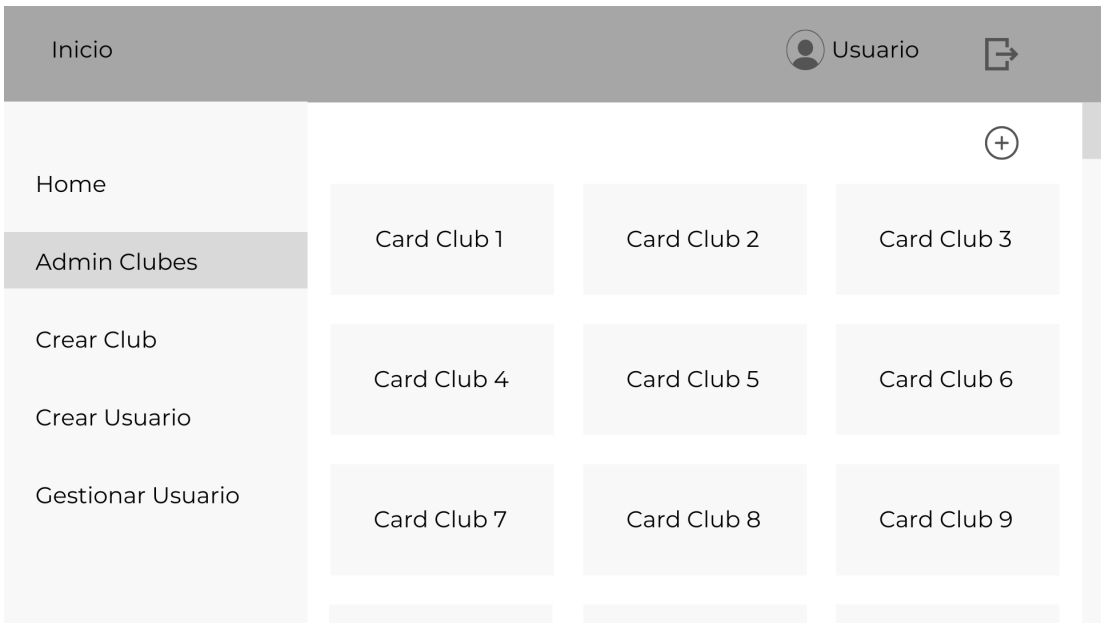


Figura 5.3: Listado de clubes Web

- Formulario creación nuevo club (administrador web)

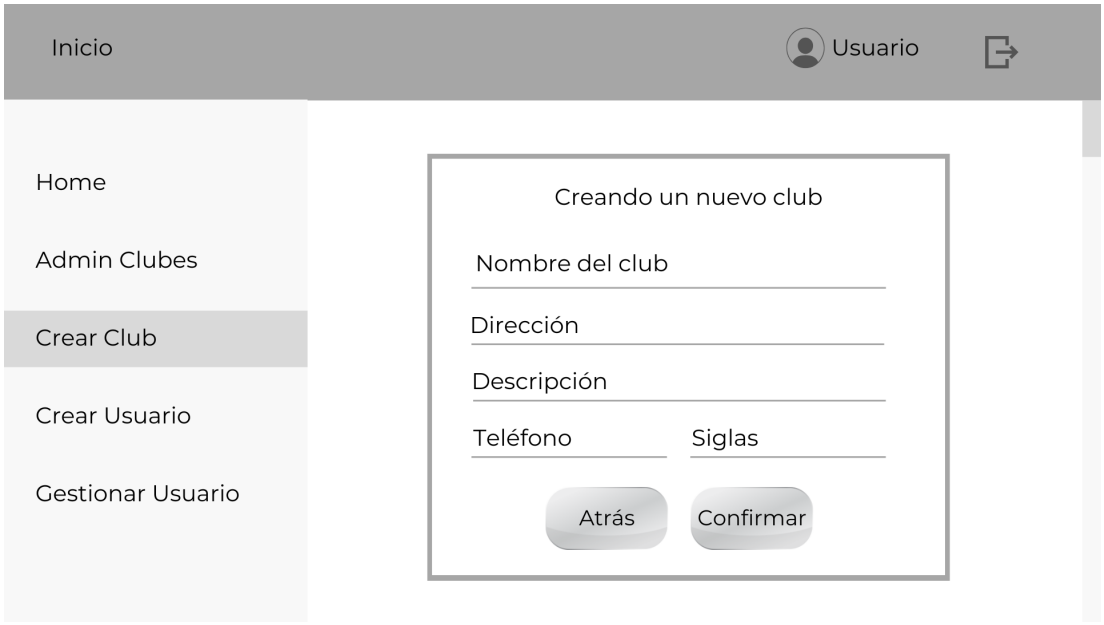


Figura 5.4: Formulario creación club Web

- Portal de reservas (administrador de club)

Figura 5.5: Portal de reservas

5.2 Aplicación móvil

- Menú principal

Figura 5.6: Menú principal móvil

- Preguntas registro

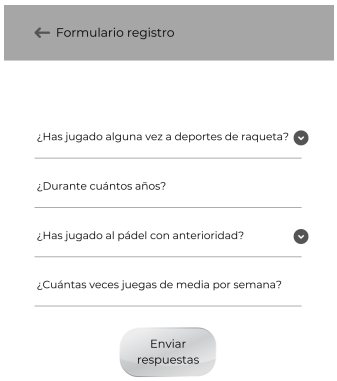


Figura 5.7: Preguntas Registro

- Reservas Aplicación móvil

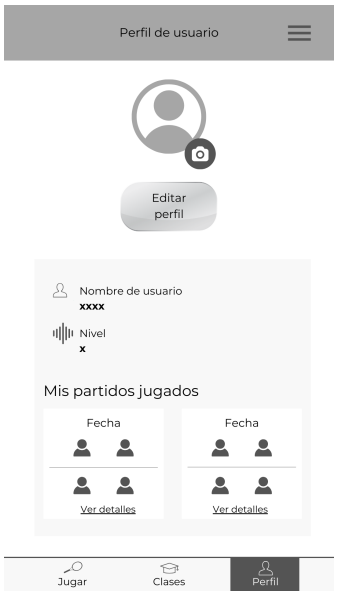


Figura 5.8: Pantalla perfil móvil

Capítulo 6

Implementación

En este capitulo hablaremos de los patrones de diseño escogidos para realizar la implementación de nuestro código. Mostraremos también partes del código que creemos que tienen una mayor relevancia.

6.1 Elección patrones de diseño

Comenzaremos explicando los patrones usados en el servidor Rest, a continuación los empleados en la parte web de la aplicación y, por último, en la app móvil.

6.1.1 Patrones de diseño en el API Rest

Dentro del API Rest podemos diferenciar distintos patrones de diseño:

- **Patron DAO:** el patrón DAO propone separar por completo la lógica de negocio de la lógica de acceso a los datos. En nuestro código usamos este patrón para realizar consultas a la base de datos, actualizarla o borrarla. [16]

```
1     @Override
2     public List<Club> findByName(String nombre) {
3         return entityManager.createQuery("from Club c where
      lower(c.nombre) like lower(:nombre)", Club.class)
4                 .setParameter("nombre",
      "%"+nombre+"%").getResultList();
5     }
6
7     @Override
8     public Club findByAdmin(Usuario user) {
9         return entityManager.createQuery("from Club c where
      c.adminClub = :user", Club.class)
10                .setParameter("user", user).getSingleResult();
11    }
```

```
12
```

- **Patron DTO:** el patrón DTO tiene como finalidad crear un objeto plano con una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación. [17]

```
1    @PreAuthorize("hasAuthority('ADMIN_WEB')")
2    @Transactional(readOnly = false)
3    public ClubDTO create(ClubDTO club) {
4        Club c = new Club(club.getNombre(), club.getDescripcion(),
     club.getDireccion(),
5                 club.getTelefono(), club.getSiglas(),
     usuarioDao.findByLogin(club.getAdminClub()),
     club.getImagenPath());
6        clubDAO.create(c);
7        return new ClubDTO(c);
8    }
9
```

- **Patron MVC:** las siglas MVC vienen dadas por Modelo(M), Vista(V) y Controlado(C). Este patrón es el encargado de separar, en una aplicación web, estas 3 funcionalidades. El Modelo se encarga de la administración de los datos; la Vista en el caso de un API Rest no se tiene en cuenta y, por último, el Controlador, que extrae, modifica y suministra datos al usuario. El Controlador es el enlace entre el Modelo y la Vista. [18]

  En nuestro caso:

```
1    @RestController
2    @RequestMapping("/api/clubes")
3    public class ClubResource {
4
5      @Autowired
6      private ClubService clubService;
7
8      @GetMapping
9      public List<ClubDTO> findAll(){
10       return clubService.findAll();
11     }
12
13     @GetMapping("/{id}")
14     public ClubDTO findById (@PathVariable Long id) throws
     NotFoundException{
15       return clubService.findById(id);
16     }
17   }
```

```
18
```

6.1.2 Patrones de diseño en la aplicación web

- **Patron Component-Based:** el patrón Component-Based (Arquitectura Basada en Componentes) ha ganado gran popularidad en el desarrollo frontend en la última década, especialmente con la aparición de frameworks y bibliotecas como React, Vue.js y Angular entre otros. [19]

  Se trata de un enfoque que descompone la interfaz de usuario en unidades independientes y reutilizables llamadas ”componentes”. Cada componente es una unidad encapsulada que contiene su propia lógica, estructura y estilos.

- **Patron MVC:** también tenemos este patrón presente en la interfaz web. En este caso sí que están bien marcadas todas las funciones del patrón. La Vista es la UI el Modelo son las entidades y el Controlador, las consultas a la API.

```
1   <template>
2       <div class="cList">
3           <Sidebar/>
4           <div class="content">
5               <v-col cols="auto">
6                   <div class="float-right">
7                       <v-btn :to="{ name: 'ClubCreate'}"
    color="green darken-1" rounded>
8                           <v-icon>mdi-plus</v-icon>
9                       </v-btn>
10                  </div>
11              </v-col>
12              <v-row>
13                  <v-col cols="auto" class="col" v-for="club in
    clubes" :key="club.id">
14                      <ClubCard :club="club"></ClubCard>
15                  </v-col>
16              </v-row>
17          </div>
18      </div>
19  </template>
20
```

6.1.3 Patrones presentes en la aplicación móvil

- **Patron MVC:** también tendremos presente este patrón arquitectónico en la aplicación web, donde separamos claramente las consultas a la API(controller), la creación de las Vistas y los Modelos, que serían las clases de datos. Se puede apreciar en la figura 6.1. Para hacer cumplir este patrón en nuestro proyecto de flutter usamos los componentes Stateful y Stateless, los cuales se basan el la separación de la lógica y la presentación. Mas adelante profundizaremos más en estos dos conceptos.

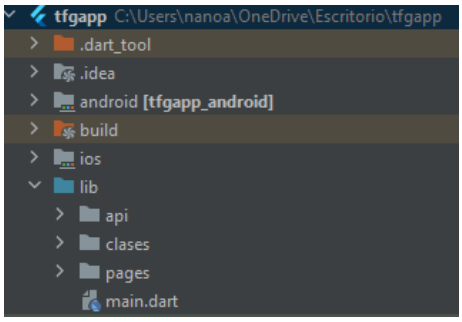


Figura 6.1: Estructura MVC en flutter

6.2 Implementación del backend

El primer paso antes de empezar la implementación del las entidades es crear la base de datos en PostgreSQL. Para ello, a través de PgAdmin, creamos una nueva base de datos llamada ”tfg” la cual conectaremos con nuestro lado servidor. Además de crear la base de datos, es necesario crear un usuario que gestione dicha base. En nuestro caso también se llamará ”tfg”.

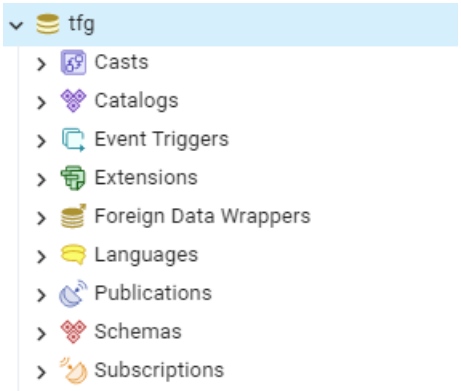


Figura 6.2: Base de datos ”tfg” en pgAdmin

Una vez configurada la base de datos será necesario conectarla con nuestro Rest API. Este proceso se realiza en nuestro archivo *application.yml* de nuestro proyecto de Java. (Figura 6.3).

```yaml
spring:
  datasource:
    host: localhost
    port: 5432
    database: tfg
    username: tfg
    password: tfg
    url: jdbc:postgresql://${spring.datasource.host}:${spring.datasource.port}/${spring.datasource.database}
    sql-script-encoding: UTF-8
    hikari:
      connectionTimeout: 20000
      maximumPoolSize: 5
    initialization-mode: always
    continue-on-error: false
  jpa:
    open-in-view: false
    show-sql: true
    hibernate:
      ddl-auto: create
```

Figura 6.3: Conectar la base de datos con el proyecto Java

Una vez conectada la base de datos con el proyecto, comenzaremos programando las entidades necesarias. Estas entidades deberán ser marcadas con "@Entity" antes de definir la clase para avisar a Spring que esta clase es una entidad. Tendremos 6 entidades en nuestro dominio: Club, Pista, Partido, Usuario, BonoClases y Transacción. Una vez creadas las entidades crearemos las consultas a la base de datos para creación, consulta, modificación y eliminación de objetos en la base de datos. Estas consultas serán en JPQL[20], lenguaje que se utiliza para crear consultas contra entidades para almacenar en una base de datos relacional. Este lenguaje, a diferencia de SQL, recupera objetos, no filas de las bases de datos.

```
1     @Override
2   public Partido findByPistaHora(Long idPista, LocalDateTime hora) {
3     return entityManager.createQuery("from Partido p where
      p.pista.id = :idPista AND p.hora_ini = :hora", Partido.class)
4         .setParameter("idPista", idPista).setParameter("hora",
      hora).getSingleResult();
5   }
6
7   @Override
8   public List<Partido> findByClub(Long idClub) {
9     return entityManager.createQuery("FROM Partido p JOIN FETCH
      p.pista WHERE p.pista.club.id = :idClub order by p.hora_ini",
      Partido.class)
10        .setParameter("idClub", idClub).getResultList();
11
12  }
13
```

En la porción de código anterior vemos un ejemplo de dos consultas JPA en las cuales estamos recuperando un partido a través de una pista y una hora en la primera consulta. En la segunda recuperamos la lista de partidos que se juegan en un club en concreto. Todas estas

clases deben estar marcadas con ”@Repository”.

Para conectar la Interfaz de Usuario (UI) con el *backend* haremos uso de *endpoints*, que son los que transmiten la información desde las páginas web a nuestra base de datos. El objeto que estos *endpoints* manejan se denominan DTOs. A continuación mostramos 3 ejemplos de *endpoints*:

```
    @GetMapping("/{id}")
  public ClubDTO findById (@PathVariable Long id) throws
    NotFoundException{
    return clubService.findById(id);
  }



  @GetMapping("/nombre")
  public List<ClubDTO> findByName(@RequestParam String nombre)
    throws NotFoundException{
    return clubService.findByName(nombre);
  }

  @GetMapping("/admin")
  public ClubDTO findByAdmin(@RequestParam Long idAdmin) throws
    NotFoundException{
    return clubService.findByAdmin(idAdmin);
  }

```

Los DTOs son clases especiales para transmitir los datos, de esta forma, podemos controlar los datos que enviamos, el nombre, el tipo de datos, etc. Además, si estos necesitan cambiar, no tiene impacto sobre la capa de servicios o datos, pues solo se utilizan para transmitir la respuesta.

```
    public class BonoClasesDTO {

      private Long id;
      private String nombre;
      private String descripcion;
      private Double precio;
      private int num_clases;
      private Long idClub;
      private Long idUser;

      public BonoClasesDTO() {}

      public BonoClasesDTO(BonoClases bono) {
        this.id = bono.getId();
```

```
15          this.nombre = bono.getNombre();
16          this.descripcion = bono.getDescripcion();
17          this.precio = bono.getPrecio();
18          this.num_clases = bono.getNum_clases();
19          this.idClub = bono.getClub().getId();
20          if(bono.getJugador() != null)
21            this.idUser = bono.getJugador().getId();
22        }
23
24          //getters and setters...
25      }
26
```

Para conectar la información de la base de datos y los *endpoints* es necesario convertir los DTOs en entidades. Para esto creamos todas la clases xxxService. Estas clases con el "@Service" al principio de la clase las utilizamos para convertir los DTOs en entidades y poder así almacenar o recuperar los datos que las aplicaciones web o móvil reclamen. Este sería un ejemplo de una clase Service:

```
1     @PreAuthorize("hasAuthority('ADMIN_WEB')")
2   @Transactional(readOnly = false)
3   public ClubDTO create(ClubDTO club) {
4     Club c = new Club(club.getNombre(), club.getDescripcion(),
      club.getDireccion(),
5         club.getTelefono(), club.getSiglas(),
      usuarioDao.findByLogin(club.getAdminClub()),
      club.getImagenPath());
6     clubDAO.create(c);
7     return new ClubDTO(c);
8   }
9
10  @PreAuthorize("hasAnyAuthority('ADMIN_WEB', 'ADMIN_CLUB')")
11  @Transactional(readOnly = false)
12  public ClubDTO update(ClubDTO club) {
13    Club bdClub = clubDAO.findById(club.getId());
14    Usuario adminClub = usuarioDao.findByLogin(club.getAdminClub());
15    bdClub.setNombre(club.getNombre());
16    bdClub.setDireccion(club.getDireccion());
17    bdClub.setDescripcion(club.getDescripcion());
18    bdClub.setTelefono(club.getTelefono());
19    bdClub.setSiglas(club.getSiglas());
20    bdClub.setAdminClub(adminClub);
21    bdClub.setImagenPath(club.getImagenPath());
22    clubDAO.update(bdClub);
23    return new ClubDTO(bdClub);
24  }
```

```
25
```

En estas dos funciones estamos creando un club a partir de la información proporcionada por la página web (a través de un DTO) y, en la segunda, estamos actualizando el objeto pasando como argumento el DTO del objeto actualizado. Como vemos en las cabeceras de las funciones, tenemos un *@PreAuthorize* y un *@Transactional*. El primero controla que los usuarios que acceden a este método estén autenticados y tengan como autoridad *ADMIN-WEB* o *ADMIN-CLUB*. El segundo @ nos muestra que estamos indicando que se realizarán cambios en la base de datos ya que estamos desactivando el modo readOnly, que está, de manera predeterminada, activado.

Para añadir imágenes y guardarlas elegimos la opción de crear una carpeta en nuestro proyecto y guardarlas allí. En nuestro proyecto la carpeta se llama *uploads*.

```
1  private String saveFile(MultipartFile file) throws IOException {
2      String fileName =
       StringUtils.cleanPath(file.getOriginalFilename());
3      File dir = new File("uploads");
4
5      if (!dir.exists()) {
6          Boolean success = dir.mkdir();
7          if(!success)
8              throw new IOException("no se pudo crear el dir");
9      }
10
11     File destinationFile = new File(dir, fileName);
12     try (InputStream fileInputStream = file.getInputStream()) {
13         Files.copy(fileInputStream, destinationFile.toPath(),
       StandardCopyOption.REPLACE_EXISTING);
14     }
15
16     return destinationFile.getAbsolutePath();
17 }
```

Una vez cargadas, solo habría que recuperarlas de la carpeta llamando a su ruta completa. Esta ruta la guardamos en un atributo del club, en este caso, en *imagenPath*.

```
1  @GetMapping("/{clubId}/imagen")
2  public ResponseEntity<UrlResource> getClubImage(@PathVariable Long
       clubId) throws NotFoundException {
3      ClubDTO club = clubService.findById(clubId);
4
5      if (club == null || club.getImagenPath() == null) {
6          return ResponseEntity.notFound().build();
7      }
8
```

```
9       Path path = Paths.get(club.getImagenPath());
10      UrlResource resource = null;
11
12      try {
13          resource = new UrlResource(path.toUri());
14          if (!resource.exists() || !resource.isReadable()) {
15              return ResponseEntity.notFound().build();
16          }
17      } catch (MalformedURLException e) {
18          return
        ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
19      }
20
21      String contentType = null;
22      try {
23          contentType = Files.probeContentType(path);
24      } catch (IOException e) {
25          contentType = "image/jpeg";
26      }
27
28      return ResponseEntity.ok()
29          .contentType(MediaType.parseMediaType(contentType))
30          .body(resource);
31  }
```

6.2.1 Implementación de la aplicación web

Antes de comenzar a programar la parte web de nuestro TFG, será necesario preparar el proyecto en Vue.js. Para ello descargamos node.js y creamos el proyecto a través de vue-create. [21]

Comenzando a programar, decidimos separar los archivos en diferentes carpetas para así tener más claro qué función se desempeñaría en cada archivo. Como vemos en la figura 6.4, en la carpeta *src* tenemos tres carpetas principales, las cuales definirán el patrón MVC: entities, components y repositories. A continuación vamos a mostrar partes del código que creemos que tienen especial importancia:

- **Parte Administrador WEB:** como en la aplicación web dos actores desempeñan sus funciones, comenzaremos explicando la parte del administrador web.

  Como se comentó anteriormente, primero diseñamos una página de inicio en la cual se da la bienvenida a la web y se aclara que esa página web es solo para administradores.

  Al iniciar sesión como AW tendremos un menú principal desde el que podremos navegar y realizar las funciones elegidas.

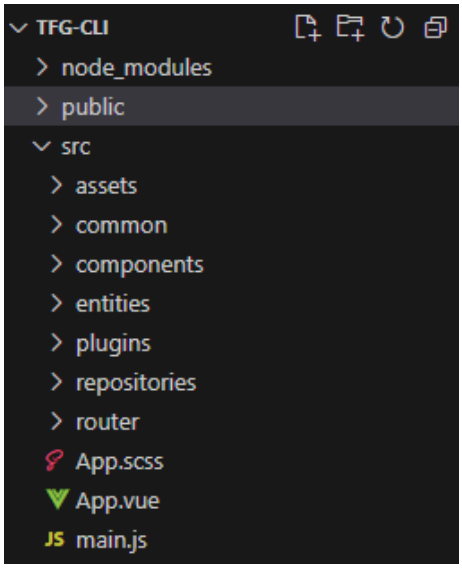


Figura 6.4: Cuerpo del proyecto en Vue.js

Figura 6.5: Pagina de inicio de sesión WEB

En la figura 6.6 creamos 4 botones que definen las funciones a las que un administrador web puede acceder. Entre ellas destacan la administración de clubes y la creación de nuevos clubes y usuarios. A continuación mostramos el código de esta pantalla:

```
1    <v-row v-if="user.authority == 'ADMIN_WEB'">
2        <v-col class="text-center">
3            <h1 style="text-align: center;">Bienvenido {{
     user.login }}</h1>
```

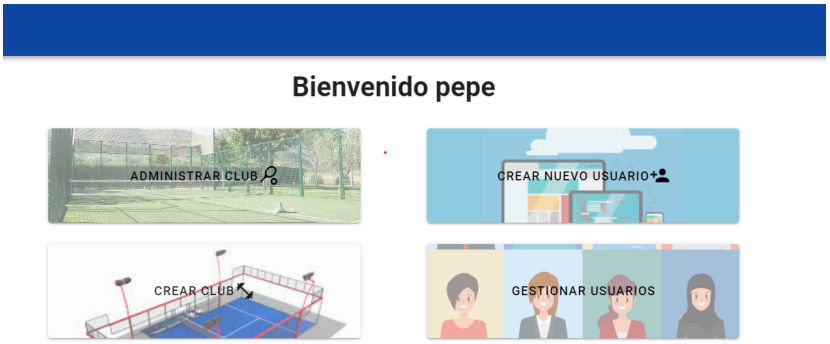


Figura 6.6: Menú principal de los admins WEB

```
        <div class="flex justify-center" style="text-align:
left">
            <v-btn class="btn-1" width="33%" min-width="220px"
height="110px" to="/clubes">
                Administrar Club
                <v-icon>mdi-tennis</v-icon>
            </v-btn>
            <v-btn class="btn-4" width="33%" min-width="220px"
height="110px" to="/register">
                Crear Nuevo Usuario
                <v-icon>
                    mdi-account-plus
                </v-icon>
            </v-btn>
        </div>
        <div class="flex justify-center" style="text-align:
left">
            <v-btn class="btn-3" width="33%" min-width="220px"
height="110px" to="/club/new">
                Crear club
                <v-icon>
                    mdi-dumbbell
                </v-icon>
            </v-btn>
            <v-btn class="btn-5" width="33%" min-width="220px"
height="110px" to="/usuarios">
                Gestionar Usuarios
            </v-btn>
        </div>
    </v-col>
</v-row>

```

Como se muestra en el código, creamos 4 botones centrados en la pantalla divididos en dos filas por un *div*. Estos botones acceden a diferentes rutas, las cuales se gestionan a través de la carpeta router.js perteneciente a nuestro proyecto. Ejemplo: en la línea 17 del código anterior se accede a la ruta */club/new* al pulsar en el botón "Crear Club".

Para ver la lista de clubes que gestiona la aplicación debemos acceder a nuestra base de datos a través de una petición definida en ClubRepository la cual recupera la totalidad de los clubes. Estos clubes los mostraremos a través del componente creado *ClubCard*, que muestra toda la información del club y genera un router-link hacia cada club. Adicionalmente, creamos un botón en la parte superior derecha que permite crear nuevos clubes.

```
1  <template>
2      <div class="cList">
3          <Sidebar/>
4          <div class="content">
5              <v-col cols="auto">
6                  <div class="float-right">
7                      <v-btn :to="{ name: 'ClubCreate'}" color="green
       darken-1" rounded>
8                          <v-icon>mdi-plus</v-icon>
9                      </v-btn>
10                 </div>
11             </v-col>
12             <v-row>
13                 <v-col cols="auto" class="col" v-for="club in
       clubes" :key="club.id">
14                     <ClubCard :club="club"></ClubCard>
15                 </v-col>
16             </v-row>
17         </div>
18     </div>
19 </template>
```

Como vemos en la línea 3 del código anterior, también hemos creado un componente *SideBar* para mejorar la visualización de nuestra app y simplificar la navegación entre pantallas. [22]

Ahora mostraremos un ejemplo de formulario de la parte de los administrador web, en concreto, el formulario de creación de usuarios:

```
1  <v-form ref="form" @submit.prevent="save">
2      <v-card class="card" width="500px" min-width="75px">
3          <v-card-title>
4              <v-col>
```

```
                Creando nuevo Usuario
                <v-text-field v-model="usuario.login" label="Name"
    :rules="requiredField"></v-text-field>
                <v-text-field v-model="usuario.contrasena"
    label="Password" :rules="requiredField"></v-text-field>
            </v-col>
        </v-card-title>

        <v-col class="user-type-title">
            Tipo de Usuario
            <v-row justify="center">
                <v-radio-group v-model="selectedRole" row>
                    <v-radio label="Admin Web"
    value="ADMIN_WEB"></v-radio>
                    <v-radio label="Admin Club"
    value="ADMIN_CLUB"></v-radio>
                    <v-radio label="Jugador"
    value="PLAYER"></v-radio>
                </v-radio-group>
            </v-row>
        </v-col>

        <v-card-actions>
            <v-spacer />
            <v-btn @click="back()">Atras</v-btn>
            <v-btn color="primary" type="submit">Confirmar</v-btn>
        </v-card-actions>
    </v-card>
</v-form>
```

En este formulario se permite crear usuarios nuevos introduciendo un login válido y una contraseña. Además, hay que seleccionar qué rol queremos que ejerza dicho usuario a través de un *v-radio-group*. [23]

- **Parte Administrador de Club:** en este punto hablaremos sobre cómo creamos las tablas de reservas, sobre todas sus restricciones y sobre la creación y vista de los bonos de clases.

  Comenzando con el menú principal de los administrador de club, es importante indicar que es ligeramente diferente al de los administrador web. En este menú solo se permitirá la edición de clubes (incluyendo la posibilidad de añadir imágenes de portada al club), la reserva de pistas, la creación y vista de bonos pertenecientes al club, y la lista de partidos que hay por día, ordenados por la hora de inicio de menor a mayor.

  La página de reservas muestra una tabla por día con franjas de 30 minutos que pueden

Figura 6.7: Menú principal de los admins CLUB

estar marcadas de varios colores: rojo indica que ya hay una reserva en esa pista durante esa franja horaria; verde indica que la pista está disponible y, naranja, se usa para mostrar las franjas seleccionadas al intentar realizar una reserva como se puede ver en la figura 3.3.

Esta sería una parte del código para la gestión de reservas:

```
<table class="table">
    <thead>
        <tr>
            <th></th>
            <th v-for="hora in horas" :key="hora">{{ hora }}</th>
        </tr>
    </thead>
    <tbody class="body">
        <tr v-for="pista in pistas" :key="pista.id">
            <td>{{ pista.nombre }}</td>
            <td
                v-for="hora in horas"
                :key="hora"
                @click="toggleDisponibilidad(pista, dia, hora)"
                :class="getEstado(pista, dia, hora)"
            >
            </td>
        </tr>
    </tbody>
</table>
```

Aquí la parte importante es el método *toggleDisponibilidad()*, la cual explicamos a continuación. La primera parte sería hacer que toda la tabla aparezca como disponible, es decir, en verde:

```
this.pistas.forEach((pista) => {
    this.$set(this.disponibilidad, pista.id, {});
    this.proximosDias.forEach((dia) => {
        this.$set(this.disponibilidad[pista.id], dia, {});
        this.horas.forEach((hora) => {
            const trimmedHora = hora.trim();
            this.$set(this.disponibilidad[pista.id][dia],
    trimmedHora, 'disponible');
        });
    });
});
```

El trozo de código anterior pertenece a la función *inicializarDisponibilidad()*. Continuando con esta función, ahora debemos incluir en nuestra tabla los partidos ya creados. Para ello añadimos el siguiente código:

```
this.partidos.forEach((partido) => {
    const horaInicioPartido =
    moment(partido.hora_ini).format("YYYY-MM-DD HH:mm");
    const horaFinPartido =
    moment(partido.hora_fin).format("YYYY-MM-DD HH:mm");
    this.proximosDias.forEach((dia) => {
        this.pistas.forEach((pista) => {
            if (pista.id === partido.pista) {
                this.horas.forEach((horaString, index) => {
                    const horaActual = moment(dia + " " +
    horaString);
                    const siguienteHora = moment(dia + " " +
    (this.horas[index + 1] || "21:30"));
                    if (horaActual.isSameOrAfter(horaInicioPartido)
    && siguienteHora.isSameOrBefore(horaFinPartido)) {

    this.$set(this.disponibilidad[pista.id][dia], horaString,
    'no-disponible');
                    }
                });
            }
        });
    });
});
```

Al añadir esto estamos gestionando los partidos ya reservados del día de hoy y de los siguientes 6 días, comparando las fechas de inicio y fin del partido. Con la fecha y hora de la tabla marcamos las pistas ya reservadas como *No-disponible* (en rojo). Este método será llamado al inicializar la pantalla a través del método created().

Ahora, para cambiar una pista de *Disponible* a *No-disponible* usaremos la función *toggleDisponibilidad*, de la cual ya hemos hablado con anterioridad.

```
1  toggleDisponibilidad(pista, dia, hora) {
2      if (this.getEstado(pista, dia, hora) === 'no-disponible') {
3          return;
4      }
5
6      const seleccionActual = { pista: pista.id, dia, hora };
7      const indiceSeleccionActual =
       this.reservasSeleccionadas.findIndex(reserva => reserva.pista
       === pista.id && reserva.dia === dia && reserva.hora === hora);
8
9      if (indiceSeleccionActual !== -1) {
10         this.$set(this.disponibilidad[pista.id][dia], hora,
       'disponible');
11         this.reservasSeleccionadas.splice(indiceSeleccionActual, 1);
12         return;
13     }
14
15     const ultimaSeleccion =
       this.reservasSeleccionadas[this.reservasSeleccionadas.length -
       1];
16     if (ultimaSeleccion && ultimaSeleccion.pista === pista.id &&
       ultimaSeleccion.dia === dia) {
17         const diferencia = this.horas.indexOf(hora) -
       this.horas.indexOf(ultimaSeleccion.hora);
18         if (Math.abs(diferencia) !== 1) {
19             alert('Por favor, seleccione horas consecutivas.');
20             return;
21         }
22     }
23
24     this.$set(this.disponibilidad[pista.id][dia], hora,
       'reservando');
25     this.reservasSeleccionadas.push(seleccionActual);
26 },
```

Con primer control, si una pista esta marcada como *No-disponible*, no permitiremos modificar su disponibilidad a través de esta pantalla.

Entre las lineas 9 y 12 se permite al usuario cancelar una reserva si ya ha seleccionado ese horario previamente. Si no existe una reserva previa para el mismo horario, entonces se procede a marcarlo como posible para la reserva.

El siguiente bloque de código (lineas 15-22) comprueban que la selección de la pista sea

consecutiva en lo que a horario se refiere y en la misma pista, en el mismo día. Si esto no es así, saltará una alerta al usuario. También saltaría el aviso si la pista es la misma pero las horas no son consecutivas.(Figura 6.8)

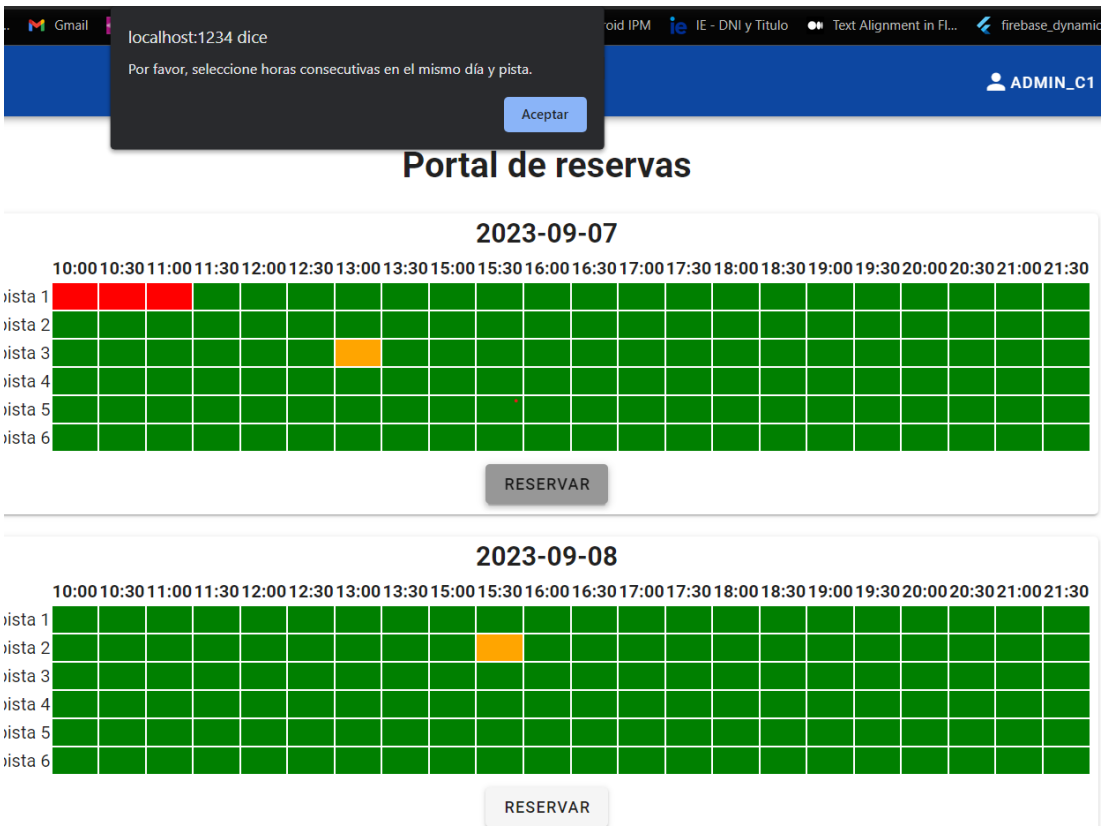


Figura 6.8: Alerta al reservar en horario no consecutivo

Es el administrador de club el encargado de añadir una foto de portada a su club. Esto se hace a través del formulario de edición del club.

```
<div>
    <label for="imagenArchivo">Imagen:</label>
    <input type="file" id="imagenArchivo" @change="onFileChange"
    accept="image/*" ref="imagenInput" required>
</div>
```

6.2.2 Implementación de la aplicación móvil

Antes de empezar con la aplicación móvil es necesario configurar el entorno de trabajo. Para ello necesitamos descargar Android Studio, copiar el flutter SDK y añadir flutter/bin a *Path* en las variables de entorno. Una vez aplicados estos cambios, en el terminal debemos ejecutar el comando *flutter doctor*. Este comando comprueba su entorno y muestra un informe

del estado de su instalación de Flutter. Comprobamos detenidamente el resultado para ver si necesitaremos instalar otro software o realizar otras tareas. [24]

Comenzando con la programación de la aplicación, como hicimos anteriormente con la aplicación web, creamos una pantalla de inicio en la cual se pedirá al usuario o bien iniciar sesión (si ya tiene una cuenta creada), o bien registrarse si es la primera vez que accede a la aplicación.

Para navegar entre pantallas en flutter usaremos rutas, la cuales definiremos en cada clase y las importaremos en nuestro *main.dart*.

En la autenticación controlamos que solo los jugadores puedan acceder a ella. Una vez autenticados, guardamos el token en el *secure-storage* de flutter [25]. En este punto, el menú principal estará formado por un grid con 4 opciones y, más abajo, una lista con los partidos futuros del jugador.

```
1    Container(
2        height: 425,
3        child: GridView.count(
4            crossAxisCount: 2,
5            padding: EdgeInsets.all(16.0),
6            childAspectRatio: 8.0 / 9.0,
7            children: [
8                _opcionMenu(context, 'Reservar Pista',
    Icons.sports_tennis, reservarPista.ROUTE),
9                _opcionMenu(context, 'Ver Bonos',
    Icons.card_giftcard, BonosList.ROUTE),
10               _opcionMenu(context, 'Buscar por Club',
    Icons.location_city, MostrarClubes.ROUTE),
11               _opcionMenu(context, 'Mis Clubes', Icons.favorite,
    MostrarClubes.ROUTE),
12           ],
13       ),
14   ),
```

En esta porción de código vemos cómo se ha creado el grid. El método *opcionMenu* nos devuelve un Widget Card al cual le pasamos un icono, un título y la ruta a la que se desea ir al pinchar en él.

A continuación mostramos también el código que genera los *PartidoCards*.

```
1    Widget _partidoCard(Partido partido) {
2    DateTime fecha = DateTime.parse(partido.hora_ini);
3
4    // Separar jugadores en dos listas
5    List<String> pareja1 = [];
6    List<String> pareja2 = [];
7
```

```
    for (int i = 0; i < partido.jugadores.length; i++) {...}
    //añadimos jugadores a pareja1 y pareja2

    if (pareja1.length < 2){...} //Si no hay dos jugadores en
    pareja 1 añadimos un hueco vacio.

    if(pareja2.length < 2){...}//Si no hay dos jugadores en pareja
    2 añadimos un hueco vacio.

    return Card(
      margin: EdgeInsets.symmetric(vertical: 10, horizontal: 15),
      child: Padding(
        padding: EdgeInsets.all(10),
        child: LimitedBox(
          maxHeight: 750,
          child: Column(
            mainAxisSize: MainAxisSize.min,
            children: <Widget>[
              Text(...), //Mostramos la fecha
              SizedBox(height: 10),
              if (pareja1.isNotEmpty)
                _generarFilaJugadores(pareja1),
              Divider(..),
              if (pareja2.isNotEmpty)
                _generarFilaJugadores(pareja2),
              TextButton(...)
            ],
          ),
        ),
      ),
    );
  }
```

El código anterior muestra cómo se han creado los PartidoCards en flutter. Este nos mostrará la fecha del partido y los jugadores que lo forman. Si se quieren más detalles del partido habrá un botón al final del Card. Para recuperar la lista de partidos tenemos que hacer una llamada al API Rest a través de la siguiente consulta:

```
// GET /api/usuarios/:id/partidos ------------------------------
  Future<List<Partido>> getPartidosFuturosUser(int id) async{
    List<Partido> partidos = [];
    var url =
    Uri.parse("http://$ip:8080/api/usuarios/$id/partidos");
    DateTime horaActual = DateTime.now();

    final response = await http.get(
```

```
        url
    );

    if (response.statusCode == 200){
      print('200 OK /api/usuarios/:id/partidos');
      String body = utf8.decode(response.bodyBytes);

      final jsonData = jsonDecode(body);

      for (var item in jsonData){
        if(DateTime.parse(item['hora_ini']).isAfter(horaActual) )
          if(item['resultado'] == null)
            partidos.add(Partido(item['id'], item['creador'],
    item['hora_ini'], item['hora_fin'], "", "",
                List<String>.from(item['jugadores'].map((x) =>
    x.toString())), item['pagado'], item['precioFinal'],
    item['pista']));
          else
            partidos.add(Partido(item['id'], item['creador'],
    item['hora_ini'], item['hora_fin'], item['resultado'],
    item['resultadoSets'],
                List<String>.from(item['jugadores'].map((x) =>
    x.toString())), item['pagado'], item['precioFinal'],
    item['pista']));
      }

      return partidos;

    } else{
      throw Exception('Error al recuperar los partidos futuros');
    }

  }
```

Con esta consulta se recuperan los partidos futuros del jugador con identificador=id.

A continuación se mostrará un tipo de consulta en la cual se necesita una autenticación mediante un *Bearer Token*:

```
Future<Usuario> getUser() async{
    Usuario user;
    final token = await getToken();
    var url = Uri.parse("http://$ip:8080/api/account");

    final response = await http.get(
        url,
        headers: {
```

```
9             'Authorization': 'Bearer $token',
10          },
11      );
12
13      print('Token(getUser()): ' + token!);
14
15      if (response.statusCode == 200){
16        String body = utf8.decode(response.bodyBytes);
17
18        final jsonData = jsonDecode(body);
19        user = Usuario.fromJson(jsonData);
20        return user;
21
22      } else{
23        throw Exception('Error al recuperar el usuario');
24      }
25  }
```

En esta consulta se recupera el token generado al autenticarse un usuario en la aplicación [26] haciendo uso del paquete *Shared-Preferences* de la documentación de Flutter[27]. Una vez recuperado el token, se añadirá a las cabeceras de la petición como se muestra en las lineas 8-10 del código anterior.

Otro ejemplo de consulta que se utiliza en nuestro código es la *Multipart Request*. Este tipo de consulta es utilizada para agregar o visualizar imágenes. A continuación se muestra un ejemplo de una consulta MultiPart Request en Flutter:

```
1  Future<void> updateFotoUser(Usuario user, String path) async{
2      var url =
       Uri.parse("http://$ip:8080/api/usuarios/cargar-imagen");
3      final token = await getToken();
4
5      var request = http.MultipartRequest('PUT', url);
6
7      request.headers.addAll({"Authorization": "Bearer $token"});
8
9      request.files.add(await
       http.MultipartFile.fromPath("imagenArchivo", path));
10
11     request.fields['id'] = user.id.toString();
12     request.fields['login'] = user.login;
13     request.fields['autoridad'] = user.authority;
14     request.fields['nivel'] = user.nivel.toString();
15
16     final response = await request.send();
17
```

```
    if (response.statusCode == 200){
      print("200 OK carga-imagen");
    } else{
      throw Exception('Error al actualizar la foto del usuario');
    }
}
```

La porción de código anterior muestra cómo se actualiza la imagen de perfil de un usuario pasando, como argumentos, al usuario en cuestión y la ruta donde la imagen está guardada en el API.

Otro punto de especial importancia de la aplicación móvil es la implementación de la pasarela de pago. A continuación se mostrará cómo se ha implementado en nuestra app.

```
//Aqui mostramos las acciones que suceden al hacer click en el
    boton "Pagar"
onPressed: () async {
    if (_selectedPaymentOption == PaymentOption.pagarAhora) {
        await reservar(usuario.login, widget.hora_ini,
    widget.hora_fin, widget.pista.id, true);
        String paymentUrl = await iniciarPago(widget.precio);

        Navigator.push(
            context,
            MaterialPageRoute(
                builder: (context) => WebviewPaypal(URLinicial:
    paymentUrl)
            ),
        ).then((result) {
        if(result == "completed"){
          snackbarPagoCompletado(context);
          Navigator.pushNamed(context, MenuPrincipal.ROUTE);
        }
        else if(result == "cancelled"){
          String hora =
    DateFormat("yyyy-MM-ddTHH:mm:ss").format(widget.hora_ini);
          borrarReserva(widget.pista.id, hora);
          snackbarPagoCancelado(context);
          Navigator.pushNamed(context, Perfil.ROUTE);
        }
      });
    } else {
      reservar(usuario.login, widget.hora_ini, widget.hora_fin,
    widget.pista.id, false);
      Navigator.pushNamed(context, MenuPrincipal.ROUTE);
    }
},
```

AL hacer clic en pagar online se llamará a la petición iniciarPago(). Esta petición devolverá una url de Paypal para ejecutar el pago de la pista. Una vez tenemos el *paymentURL*, la aplicación navegará al WebView, dentro de la página *WebViewPaypal*. Esta clase devuelve una url de redireccionamiento una vez se haya completado el pago o se haya cancelado.

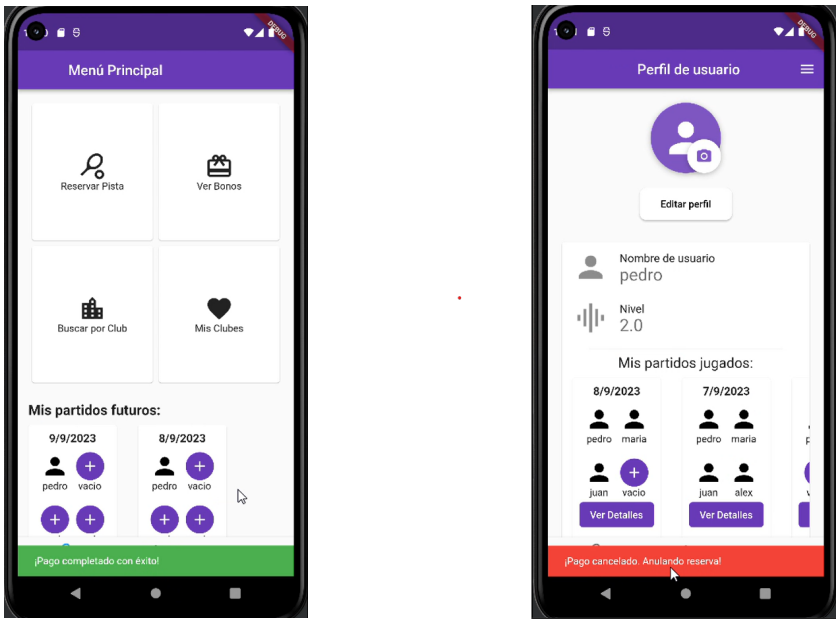
```
    onNavigationRequest: (NavigationRequest request){
        if (request.url.startsWith("http://tuapp.com/cancelled")) {
          Future.delayed(Duration.zero, () {
            Navigator.pop(context, 'cancelled');
          });
          return NavigationDecision.prevent;
        } else if
    (request.url.startsWith("http://tuapp.com/confirmed")){
          print('Resques Correcto -----------------');
          Future.delayed(Duration.zero, () {
            Navigator.pop(context, 'completed');
          });
          return NavigationDecision.prevent;
        }
        return NavigationDecision.navigate;
    },
```

Tal y como muestra el código anterior (perteneciente a la clase *WebViewPaypal*) si la url de redireccionamiento es *http://tuapp.com/cancelled*, quiere decir que el pago se ha cancelado. En este caso, se mostrará un mensaje de cancelación de pago y se eliminará la reserva. En caso contrario, si la url es *http://tuapp.com/comfirmed*, la operación habrá terminado con éxito. En ambos casos se mostrará un *ScanackBar* informando al usuario. (Figuras 6.9).

6.2.3 Repositorios

El código completo está subido a sus respectivos repositorios en GitLab:

- Repositorio API Rest: [28].
- Repositorio Aplicacion Web: [29].
- Repositorio Aplicacion Movil: [30].



(a) Confirmación pago (b) Cancelación pago

Figura 6.9: Mensajes pasarela de pago

Capítulo 7

Pruebas

En este capítulo se explicará cómo se han realizado las pruebas de todas las partes de este TFG.

7.1 API Rest

Empezaremos explicando cómo se ha probado el API Rest. A medida que se iba haciendo el lado servidor de la aplicación, se iba comprobando que los end-points funcionasen correctamente y que se cumplían los requerimientos establecidos por parte del servidor (por ejemplo, los tokens de autorización). En nuestro API se ejecutan varios tipos de consultas, en concreto, dos diferentes: consultas de tipo http normales y consultas multipart request. Se analizarán ambos tipos de consultas y las pruebas que se han realizado para probar el correcto funcionamiento de estas a través de Postman.

El primer caso que se analizará sera la creación de un partido. Como se puede observar en el archivo *PartidoService*, la función *create* tiene una restricción de autorización antes de ser ejecutada (@PreAuthorize(”hasAnyAuthority('ADMIN-CLUB', 'PLAYER')”)). Esto indica que solo los usuarios con autoridad de AC o de Jugador pueden ejecutar esta consulta (Figura 7.1).

La respuesta de error que se nos muestra al realizar esta consulta, al no estar autenticados como AC o Jugador, es **401 Unauthorized**. Si la autenticación es correcta, recibimos un **200 OK**. También se debe comprobar que el body que se pasa a la petición sea correcto y esté completo. Si, por ejemplo, se intenta recuperar un club por su identificador y este no existe, el error será **404 Not Found**. En otro caso, se recibirá un **500 Internal Server Error**.

También está la opción de no necesitar ningún tipo de autoridad para realizar una consulta. Es el caso de las consultas de recuperación de datos y no de modificación.

Para las actualizaciones o borrados de datos, primero ejecutaremos la operación escogida contra la base de datos y luego intentaremos recuperar el objeto modificado o eliminado con otra consulta.

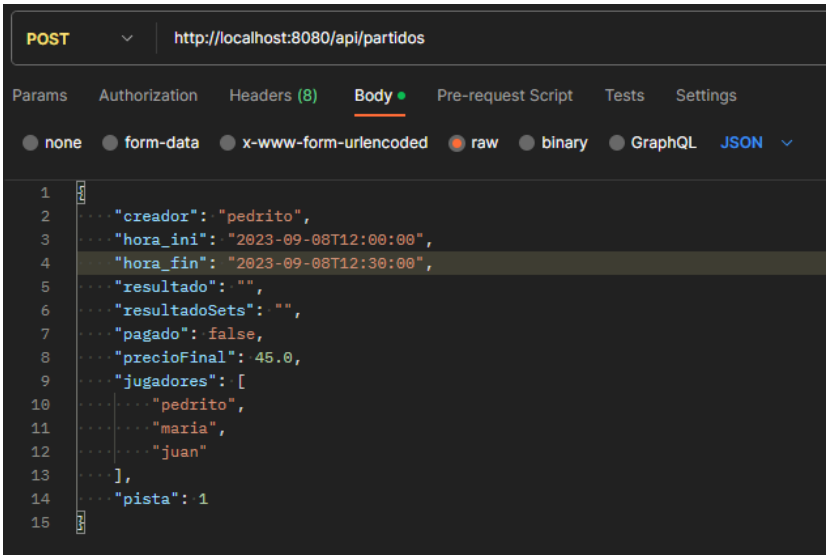


Figura 7.1: Prueba creación partido sin AUTH

Comprobamos también que el borrado de clubes sea en cascada, esto significa que, cuando borremos un club, también se borren las pistas asociadas a este y sus bonos. También se deben borrar los partidos asociados a este club, aunque no se verá modificado el nivel de los jugadores al desaparecer los partidos que jugaron en ese club. Adicionalmente, al hacer el borrado de un club, también se borrará a su administrador asociado.

Al hacer un borrado de un jugador o de un AC para representar al jugador en el partido o al administrador, se utilizará un usuario auxiliar (UserNoDisp).

7.2 Aplicación web

Para probar la aplicación web se ha usado un navegador web, Google Chrome. Las primeras pruebas realizadas tenían como objetivo comprobar que el usuario contaba con la autoridad correcta para acceder a la aplicación. En caso de que el usuario no cumpla con este requisito, se le avisa por pantalla mediante un mensaje de error. En esta pantalla también comprobamos que el login y la contraseña fuesen correctas, notificando al usuario en caso contrario.

En cuanto a las reservas de pistas, como ya se había comentado anteriormente, se controla que las franjas seleccionadas para realizar una reserva estén en la misma pista y sean consecutivas en lo que a horas se refiere. En caso de no cumplirse esto, se notifica al usuario mediante una alerta. Esta vez hemos elegido este método para el aviso ya que, para continuar, es necesario cerrar el *pop-up* y, de esta manera, nos aseguramos de que el usuario lee la notificación.

En el caso de los formularios, comprobamos, con reglas de entrada, que se cubran los campos obligatorios. Además, para la edición y creación de nuevos clubes, se controla también que los teléfonos sean una cadena de 9 números. (Figura 7.2)

```
1  <v-text-field v-model="club.nombre" label="Nombre del club"
   :rules="requiredField" maxlength="50"></v-text-field>
2  <v-text-field v-model="club.direccion" label="Direccion"
   :rules="requiredField" maxlength="100"></v-text-field>
3  <v-textarea v-model="club.descripcion" label="Descripcion"
   :rules="requiredField" rows="2" maxlength="150"></v-textarea>
4  <v-row>
5      <v-text-field v-model="club.telefono" label="Telefono"
   :rules="[...requiredField, exactLengthRule]"
   type="number"></v-text-field>
6      <v-text-field v-model="club.siglas" label="Siglas"
   :rules="requiredField" maxlength="10"></v-text-field>
7  </v-row>
```

Figura 7.2: Errores campos obligatorios

7.3 Aplicación móvil

Para probar la aplicación móvil se ha usado un emulador de Android Studio, en concreto, el emulador Pixel 5 con la versión de Android 12.0.

La primera prueba a realizar es la comprobación de que el login y la contraseña sean correctas para poder acceder a la aplicación. Aquí también se comprueba que el usuario que accede tiene la autoridad requerida (Jugador en el caso de esta app). En caso de que alguna de estas dos comprobaciones falle, se notificará al usuario mediante un *SnackBar*. También se comprueba si la longitud del login (que tiene que ser de 4 caracteres como mínimo) y, si no cumple esta condición, también se notificará al usuario.

```
if(existeLogin){
  ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(
        content: Text('El login ya existe. Por favor, elige otro.'),
        backgroundColor: Colors.red,
      )
  );
}
else if (login.isNotEmpty && password.isNotEmpty) {
  if (login.length >= 4){
    print("Registrando usuario con login: $login y contraseña: $password");
    await Peticiones().registerUser(login, password);
    await Peticiones().autenticar(login, password);
    Navigator.pushNamed(context, FormRegister.ROUTE);
  } else {
    ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('El login debe tener minimo 4 caracteres'), backgroundColor: Colors.red,)
    );
  }
} else {
  ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Por favor, rellena todos los campos.'), backgroundColor: Colors.red,)
  );
}
```

Empleamos el mismo método para el inicio de sesión, comprobando que el login y la contraseña sean correctos. En caso contrario, se le notificará al usuario el error.

En el archivo *Peticiones* se hacen controles al crear, actualizar o recuperar objetos para que ningún atributo de dicho objeto tenga valor nulo.

```
for (var item in jsonData){
    if(DateTime.parse(item['hora_ini']).isAfter(horaActual) )
        if(item['resultado'] == null)
                partidos.add(Partido(item['id'], item['creador'],
    item['hora_ini'], item['hora_fin'], "", "",
                List<String>.from(item['jugadores'].map((x) =>
    x.toString())), item['pagado'], item['precioFinal'],
    item['pista']));
    else
        partidos.add(Partido(item['id'], item['creador'],
    item['hora_ini'], item['hora_fin'], item['resultado'],
    item['resultadoSets'],
            List<String>.from(item['jugadores'].map((x) =>
    x.toString())), item['pagado'], item['precioFinal'],
    item['pista']));
  }
```

En el código anterior mostramos cómo recuperamos la totalidad de partidos de un club. Los partidos que aún no han tenido lugar no tienen resultado y en la base de datos aparece como null, pero Flutter no acepta nulos, por lo que hay que gestionar estos valores convirtiéndolos en cadenas vacías.

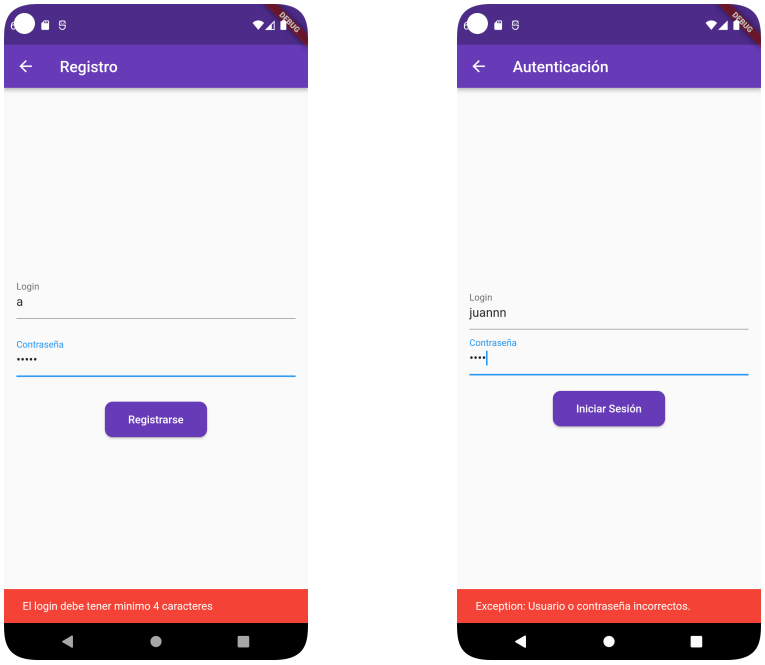
En la pasarela de pago se crean dos Snackbars para notificar al usuario el pago o la cancelación del mismo.

```
void snackbarPagoCompletado(BuildContext context) {
    final snackBar = SnackBar(
      content: Text('¡Pago completado con éxito!'),
      backgroundColor: Colors.green,
      duration: Duration(seconds: 3),
    );

    ScaffoldMessenger.of(context).showSnackBar(snackBar);
}

void snackbarPagoCancelado(BuildContext context) {
    final snackBar = SnackBar(
      content: Text('¡Pago cancelado. Anulando reserva!'),
      backgroundColor: Colors.red,
      duration: Duration(seconds: 3),
    );

    ScaffoldMessenger.of(context).showSnackBar(snackBar);
}
```



(a) Fallo en registro (b) Fallo auntenticación

Figura 7.3: Mensajes de fallo en registro y autenticación

Capítulo 8
Conclusiones

En este último capitulo de la memoria trataré aquellos aspectos de proyecto que me han permitido seguir mejorando mis habilidades en este área de desarrollo de aplicaciones y, en añadido, hablaré sobre otras posibles funcionalidades que se podrían añadir a futuro en la app y sobre la escalabilidad del negocio a otros sectores.

8.1 Lecciones aprendidas

Tras haber concebido, diseñado y desarrollado la aplicación Flash Booking, considero que he mejorado tanto mi capacitación de gestión de proyectos (referida, por ejemplo, al manejo de metodologías ágiles) como de desarrollo de plataformas y sus lenguajes de programación.

En concreto, he podido conocer más a fondo el proceso de implementación de pasarelas de pago para aplicaciones de venta.

Otro aspecto importante a destacar sobre lo aprendido en esta etapa de mi carrera universitaria consiste en que he comprobado cómo una buena organización y cronograma en la fase de concepción del proyecto son claves tanto para el cumplimiento de objetivos como para una valoración real del coste del proyecto.

8.2 Mejoras futuras

Las posibilidades de este proyecto se extienden tanto en la esencia actual de la aplicación como hacia su entrada en otros sectores.

En lo que a mejoras de la aplicacion se refiere, a continuación se indican algunas de ellas:

- Proporcionar al Jugador la posibilidad de configurar el partido a crear como público o privado de manera que, en el primer caso, se puedan unir al partido otros usuarios de la aplicación.

- Crear de un chat que fomente la interacción entre usuarios que comparten partido.
- Ofrecer la posibilidad de contactar, a través de la aplicación, con los clubes para realizar cualquier gestión relacionada con la actividad.

En cuanto a la escalabilidad del negocio, este modelo de aplicación se puede llevar a otras practicas deportivas en las que sea necesario disponer de las instalaciones de los centros para practicar el deporte (tenis, piscina, escalada, etc.).

Otro posible potencial del modelo de la aplicación se basa en su uso aplicado a organismos públicos, como ayuntamientos, para gestionar los préstamos de servicios y lugares públicos (instalaciones deportivas como canchas de tenis o campos de fútbol, casas para eventos, actividades organizadas por los ayuntamientos, etc.).

Flash Booking ha sido un gran colofón a mi carrera universitaria. En el proyecto, además de poder aplicar y afianzar una gran suma de conocimientos adquiridos durante estos años, he podido aprender nuevos aspectos relacionados con mi materia y ver materializado todo ello en un proyecto que tiene potencial para hacerse real.

Lista de acrónimos

AC administrador de club. 9, 10, 12, 18, 19, 26, 41, 45, 53, 54

AW administrador web. 9, 10, 12, 18, 19, 25, 26, 37, 38, 40, 41

TFG Trabajo de fin de Grado. 1, 12, 17, 37, 53

UI Interfaz de Usuario. 9, 11, 12, 31, 34

Glosario

API Conjunto de reglas que definen cómo las aplicaciones o los dispositivos pueden conectarse y comunicarse entre sí.. 50, 53

API Rest Una API que cumple los principios de diseño del estilo de arquitectura REST.. 53

backend Es la parte del desarrollo web que se encarga de que toda la lógica de una página web funcione. También conocido como lado servidor.. 3, 4

framework Un esquema o marco de trabajo que ofrece una estructura base para elaborar un proyecto con objetivos específicos. 3–5

frontend Se encarga de la realización de la interfaz de un sitio web, desde su estructura hasta los estilos, como pueden ser la definición de los colores, texturas, tipografías, secciones, entre otros. Comúnmente conocido como lado cliente.. 4

Bibliografía

[1] Spring Initializr. (2023) Spring initializr. [En línea]. Disponible en: https://start.spring.io/

[2] IONOS Digital Guide. (2023) Stripe vs. paypal: ¿cuál es la mejor solución de pago en línea? [En línea]. Disponible en: https://www.ionos.es/digitalguide/online-marketing/vender-en-internet/stripe-vs-paypal/

[3] K. Khezami. (2023) Integrating paypal payment in spring-boot backend: Full guide with all the steps (charge and payout). [En línea]. Disponible en: https://khouloudkhezami.medium.com/integrating-paypal-payment-in-springboot-backend-full-guide-with-all-the-steps-charge-and-payout-

[4] J. Leider and contributors, "Vuetify: A material design framework for vue.js," 2021, biblioteca de componentes Vue.js que implementa el Material Design de Google. [En línea]. Disponible en: https://vuetifyjs.com/

[5] pub.dev. (2023) webview_flutter 3.0.4. [En línea]. Disponible en: https://pub.dev/packages/webview_flutter/versions/3.0.4

[6] Google, "pub.dev," 2021, repositorio oficial de paquetes para Dart y Flutter. [En línea]. Disponible en: https://pub.dev/

[7] OpenWebinars. (2023) ¿qué es visual studio code y qué ventajas ofrece? [En línea]. Disponible en: https://openwebinars.net/blog/que-es-visual-studio-code-y-que-ventajas-ofrece/

[8] PayPal Developer. (2023) Paypal developer central. [En línea]. Disponible en: https://developer.paypal.com/home

[9] Postman. (2023) Postman. [En línea]. Disponible en: https://www.postman.com/

[10] pgAdmin. (2023) Descargar pgadmin. [En línea]. Disponible en: https://www.pgadmin.org/download/

[11] Flutter Dev. (2023) Publicación para android. [En línea]. Disponible en: https://esflutter.dev/docs/deployment/android

[12] pub.dev. (2023) Multipartrequest class - http package. [En línea]. Disponible en: https://pub.dev/documentation/http/latest/http/MultipartRequest-class.html

[13] Bankinter Comité. (2023) NÓmina convenio tic: Tablas salariales 2023. [En línea]. Disponible en: https://bankintercomite.es/pag/bk/general/nomina/tic/index.php?conc=tablas

[14] Glassdoor. (2023) Sueldo de director de proyecto en españa. [En línea]. Disponible en: https://www.glassdoor.es/Sueldos/director-de-proyecto-sueldo-SRCH_KO0,20.htm

[15] PCComponentes. (2023) Msi katana gf66 12ud-081xes intel core i7-12700h/16gb/512gb ssd/rtx3050ti/15.6". [En línea]. Disponible en: https://www.pccomponentes.com/msi-katana-gf66-12ud-081xes-intel-core-i7-12700h-16gb-512gb-ssd-rtx3050ti-156

[16] O. Blancarte. (2018) Data access object (dao) pattern. [En línea]. Disponible en: https://www.oscarblancarteblog.com/2018/12/10/data-access-object-dao-pattern/

[17] ReactiveProgramming.io. (2023) Patrones arquitectónicos: Dto. [En línea]. Disponible en: https://reactiveprogramming.io/blog/es/patrones-arquitectonicos/dto

[18] CampusMVP. (2023) ¿qué es el patrón mvc en programación y por qué es útil? [En línea]. Disponible en: https://www.campusmvp.es/recursos/post/que-es-el-patron-mvc-en-programacion-y-por-que-es-util.aspx

[19] O. El-Gabry. (2023) Component-based architecture. [En línea]. Disponible en: https://medium.com/omarelgabrys-blog/component-based-architecture-3c3c23c7e348

[20] TutorialsPoint. (2023) Jpa - lenguaje de consulta java persistence (jpql). [En línea]. Disponible en: https://www.tutorialspoint.com/es/jpa/jpa_jpql.htm

[21] LenguajeJS. (2023) Primeros pasos con vue.js. [En línea]. Disponible en: https://lenguajejs.com/vuejs/introduccion/primeros-pasos/

[22] T. Potts. (2023) Build an animated responsive sidebar menu with vue js, vue router, scss and vite in 2022. [En línea]. Disponible en: https://www.youtube.com/watch?v=Ul4KOXNQJSk&ab_channel=TylerPotts

[23] Vuetify. (2023) Radio buttons - vuetify documentation. [En línea]. Disponible en: https://vuetifyjs.com/en/components/radio-buttons/

[24] Flutter. (2023) Install flutter on windows. [En línea]. Disponible en: https://docs.flutter.dev/get-started/install/windows

[25] Flutter Community. (2023) flutter-secure-storage. [En línea]. Disponible en: https://pub.dev/packages/flutter_secure_storage

[26] U. de Stack Overflow. (2023) Bearer token request en http - flutter. [En línea]. Disponible en: https://stackoverflow.com/questions/58079131/bearer-token-request-http-flutter

[27] Flutter Community. (2023) shared-preferences. [En línea]. Disponible en: https://pub.dev/packages/shared_preferences

[28] F. A. Vazquez. (2023) Repositorio del servicio rest - tfg. [En línea]. Disponible en: https://gitlab.com/fernandoalvarezvazquez/rest-service-tfg

[29] ——. (2023) Repositorio del tfg - cli. [En línea]. Disponible en: https://gitlab.com/fernandoalvarezvazquez/tfg-cli

[30] ——. (2023) Repositorio de tfg (aplicacion movil). [En línea]. Disponible en: https://gitlab.com/fernandoalvarezvazquez/tfgapp