Universidad de Valladolid

Escuela Técnica Superior de Ingenieros de Telecomunicación

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías Específicas de Telecomunicación, Mención en Telemática

Diseño e implementación de una aplicación web para la gestión de clubes de pádel orientado a mejorar de la experiencia del usuario

Autor
Guillermo Sanz López

Tutora
María Jesús Verdú Peréz

Valladolid, 24 de junio de 2025

Título:	**Diseño e implementación de una aplicación web para la gestión de clubes de pádel orientado a mejorar de la experiencia del usuario**
Autor:	**Guillermo Sanz López**
Tutora:	**María Jesús Verdú Peréz**
Departamento:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

Tribunal

Presidente:	**Juan Pablo de Castro Fernández**
Secretario:	**Luisa M. Regueras Santos**
Vocal:	**María Jesús Verdú Pérez**

Fecha:	**24 de junio de 2025**
Calificación:	

Resumen

El pádel es un deporte en crecimiento que requiere una gestión eficiente de las instalaciones y servicios asociados. La administración de pistas, reservas y usuarios suele realizarse mediante herramientas poco intuitivas o sistemas manuales, lo que puede dificultar la experiencia tanto de gestores como de clientes.

Con el objetivo de facilitar esta gestión, en este Trabajo de Fin de Grado se desarrolla una aplicación web que permite a los clubes de pádel gestionar sus instalaciones de forma eficiente. La aplicación consta de un sistema de administración desde el que los responsables del club pueden gestionar pistas, horarios, usuarios y servicios adicionales. Por otro lado, los usuarios registrados pueden acceder al sistema, consultar la disponibilidad y realizar reservas de pistas de forma sencilla e intuitiva.

La aplicación se basa en una arquitectura cliente-servidor, con un backend desarrollado en Python utilizando el framework FastAPI, encargado de gestionar las solicitudes, almacenar los datos de usuarios, reservas y pistas, así como garantizar la seguridad y eficiencia de las operaciones. El frontend, por su parte, ha sido desarrollado en React, proporcionando una interfaz dinámica, adaptable a diferentes dispositivos y fácil de usar para los clientes del club.

El sistema de reservas permite a los usuarios seleccionar la pista, fecha y hora deseada, recibir confirmaciones de reserva y visualizar su historial de reservas. Además, el administrador puede acceder a estadísticas sobre la ocupación de las pistas, gestionar usuarios y realizar modificaciones en la oferta de servicios del club.

El sistema implementado incorpora funcionalidades adicionales que mejoran significativamente la experiencia del usuario. Entre ellas, destaca el envío automático de recordatorios por correo electrónico 24 horas antes de cada reserva, incluyendo información meteorológica actualizada para alertar al usuario en caso de cambios sustanciales respecto al pronóstico inicial. Además, se ha integrado un sistema de valoraciones posterior a cada uso, que permite recoger opiniones de los usuarios sobre la calidad de las pistas. Esta información no solo sirve como herramienta de mejora continua para los administradores, sino que también facilita a futuros usuarios una toma de decisiones más informada al momento de realizar nuevas reservas.

La aplicación ha sido probada en distintos navegadores y dispositivos, asegurando su correcto funcionamiento y facilidad de uso. Los resultados obtenidos muestran que el sistema facilita significativamente el proceso de reserva para los usuarios y optimiza la gestión diaria de los administradores, contribuyendo a una administración más eficiente y a una mejor experiencia para los clientes del club de pádel.

Palabras clave

Pádel, Aplicación web, React, Python, Desarrollo software

Abstract

Padel is a growing sport that requires efficient management of its facilities and associated services. The management of courts, reservations, and users is often carried out using outdated or manual systems, which can complicate the experience for both managers and customers.

To address this challenge, this Final Degree Project presents a web application designed to streamline the management of padel clubs. The application features an administration panel that allows club managers to manage courts, schedules, users, and additional services. Additionally, registered users can log in to the system, check court availability, and make reservations easily and intuitively.

The application is based on a client-server architecture, with a backend developed in Python using the FastAPI framework. The backend handles service requests, stores user, reservation, and court data, and ensures secure and efficient operations. The frontend, developed with React, provides a dynamic, responsive interface that adapts to different devices, offering a smooth user experience.

The reservation system enables users to select a court, date, and time, receive booking confirmations, and view their reservation history. On the administrative side, managers can access court occupancy statistics, manage users, and update available services.

The implemented system incorporates additional features that significantly enhance the user experience. Among them, the automatic sending of email reminders 24 hours before each reservation stands out, including updated weather information to notify the user in case of significant changes compared to the initial forecast. Furthermore, a post-reservation review system has been integrated, allowing users to provide feedback on the quality of the courts. This information not only serves as a continuous improvement tool for administrators but also helps future users make more informed decisions when booking a court.

The application has been tested across different browsers and devices to ensure its reliability and ease of use. The results demonstrate that the system significantly simplifies the reservation process for users and optimizes daily administrative tasks, contributing to more efficient club management and a better experience for padel club customers.

Keywords

Padel, Web application, React, Python, Software development

Agradecimientos

- Gracias a María Jesús por la paciencia que ha tenido conmigo y por su ayuda en todo momento.
- Gracias a mi familia por el apoyo cercano siempre que lo he necesitado.
- Gracias a mis amigos que me han acompañado en este camino siempre dispuestos a escuchar y apoyar.

Índice

Índice de figuras

Índice de códigos

Capítulo 1

Introducción

El pádel lleva unos años incrementando exponencialmente su popularidad, lo que ha supuesto que todo lo que se genera a su alrededor necesite profesionalizarse al máximo nivel. En este punto, las aplicaciones tecnológicas asociadas a la gestión de los clubes de pádel han ido mejorando, pero en muchos casos se siguen utilizando métodos manuales o que desaprovechan las muchas posibilidades de las herramientas digitales.

Desde este Trabajo de Fin de grado se va a intentar dar respuesta a la necesidad de una aplicación web desde la que gestionar las reservas, pistas y usuarios en clubes de pádel, facilitando el proceso para los administradores y ofreciendo soluciones accesibles e intuitivas para los clientes.

A lo largo del documento se contextualiza el problema, se describen los objetivos planteados y se detallan las decisiones de diseño y desarrollo adoptadas, con el propósito de ofrecer una solución escalable y alineada con las demandas reales del sector.

1.1. Contexto y motivación

La popularidad del pádel a nivel mundial ha provocado la proliferación de numerosas instalaciones deportivas, y sus servicios asociados, que dan respuesta a la gran demanda reciente por este deporte. Uno de los hitos detectados es la gestión eficiente de estas instalaciones, como lo es el caso de cuando un mismo club tiene instalaciones en distintas localizaciones, y resulta complicado unificarlas bajo el mismo sistema. Esta aplicación propone una única herramienta que de servicio a múltiples localizaciones, centralizando su gestión, optimizando así los procesos administrativos y mejorando la experiencia de usuario.

1.2. Objetivos

El objetivo principal de este proyecto es el desarrollo de una aplicación web que facilite la gestión centralizada de clubes de pádel distribuidos geográficamente, mejorando al mismo tiempo la experiencia del usuario. Para lograrlo, se han definido los siguientes objetivos específicos:

- Centralizar la gestión de clubes, permitiendo una administración unificada de pistas, reservas, usuarios y servicios.

- Optimizar la experiencia del usuario mediante una interfaz web moderna y accesible desde cualquier dispositivo.
- Utilizar tecnologías actuales y eficientes, como FastAPI para el backend, React para el frontend, PostgreSQL para el almacenamiento de datos y Docker para su despliegue y escalabilidad.
- Implementar un sistema de reservas sencillo e intuitivo para los usuarios, que permita seleccionar pista, fecha y hora, así como gestionar sus reservas.
- Ofrecer al usuario servicios adicionales como la climatología o la consulta de las últimas noticias en el mundo del padel.
- Ofrecer al gestor del club un panel de administración con funcionalidades avanzadas, desde el que se permita consultar rápidamente el estado de las pistas, creación y modificación de servicios y la gestion de usuarios.

1.3. Metodología

Este proyecto se ha desarrollado de forma individual, siguiendo una metodología acorde a lo utilizado en el sector actualmente y basada en la progresiva mejora de funcionalidades. Aunque tratándose de un proyecto individual es sencillo desviarse de las buenas prácticas utilizadas en entornos colaborativos, se ha tratado siempre de respetarlas utilizando un control de versiones como Git, organizando las principales funcionalidades añadidas en ramas y creando *pull requests* para integrarlas en la rama principal en GitHub.

Esto nos ha permitido mantener un historial limpio de todos los avances, consiguiendo de esta manera estructurar bien los pasos que se iban acometiendo en el desarrollo y asegurando que cada característica añadida no suponía ningún riesgo para la versión estable. Cada funcionalidad del backend, frontend, o del despliegue final ha tratado de ser desarrollada en ramas independientes, testada de forma local e integrada finalmente en la rama principal de nuestro sistema de control de versiones. Gracias a esto, hemos podido tener siempre un sistema estable, al mismo tiempo que hacía más fácil la documentación de cara a redactar la memoria final.

La metodología utilizada ha seguido un orden lógico, con una división en las principales tareas acometidas. Se comenzó con una fase de análisis de las necesidades, se continuó con un diseño de la arquitectura y del modelo de datos, para finalmente hacer el desarrollo funcional del backend en FastAPI y del frontend en React. Una vez finalizado el desarrollo, se procedió a la fase de despliegue utilizando Docker y Docker Compose, lo que facilita la portabilidad y escalabilidad del proyecto. Para finalizar, se realizaron pruebas funcionales en diferentes entornos, se documentó el proyecto y se preparó esta memoria.

1.4. Estructura del documento

Esta memoria se ha organizado en capítulos que permiten entender la motivación y las decisiones técnicas que se han ido tomando durante su desarrollo.

En el **Capítulo 2 - Estado del Arte**, se analizan las soluciones existentes en el mercado, analizando las limitaciones que tienen y las posibles mejoras.

En el **Capítulo 3 - Análisis**, se recogen los requisitos funcionales y no funcionales del sistema, y un ejemplo del uso típico de la aplicación.

En el **Capítulo 4 - Diseño**, se explica la arquitectura a desarrollar en la aplicación web, como el diseño de la base de datos, la estructura del backend y los componentes de la interfaz de usuario.

En el **Capítulo 5 - Implementación**, se detallan las tecnologías empleadas, la organización del código, los mecanismos de despliegue y las pruebas funcionales realizadas sobre el sistema. Se incluye también una explicación de cómo se automatiza el envío de correos electrónicos de confirmación, recordatorio de pista junto con la confirmación del estado climatológico para la reserva y solicitud de valoración final.

En el **Capítulo 6 - Conclusiones**, se resume el estado final del proyecto, destacando las funcionalidades que se han considerado claves y diferenciales como la valoración de las pistas o los recordatorios de avisos meteorológicos. También se incluyen posibles mejoras futuras en una ampliación de la aplicación.

Esta división en capítulos permite entender los diferentes pasos que se han seguido en la realización de este Trabajo de Fin de Grado y el despliegue tecnológico utilizado en el mismo.

Capítulo 2

Estado del arte

2.1. Introducción

El Estado del Arte tiene como objetivo analizar las soluciones existentes y las tecnologías que pueden influir en el desarrollo del proyecto. En este capítulo, se van a revisar las herramientas similares existentes en la gestión de clubes deportivos y también se explicarán las tecnologías clave elegidas para la implementación de la aplicación, explicando la razón por la que han sido seleccionadas.

El análisis de la competencia es un paso fundamental en cualquier desarrollo tecnológico, ya que nos va a permitir identificar las fortalezas y debilidades en las soluciones previas, evitando la reinvención de la rueda y mejorando las características del sistema en comparación con otros sistemas. En el caso de la gestión de clubes de pádel, existen diversas plataformas diseñadas para administrar reservas de pistas, controlar la disponibilidad de servicios y gestionar a los usuarios. Sin embargo, muchas de estas soluciones presentan limitaciones, como la falta de una infraestructura centralizada para gestionar múltiples clubes desde una misma plataforma, funcionalidades extras como el aviso de un cambio meteorológico o las valoraciones de pistas.

La selección de las tecnologías utilizadas en este proyecto se ha basado en la evaluación de los frameworks y herramientas más utilizadas en la industria, teniendo en cuenta parámetros como escalabilidad, facilidad de desarrollo y compatibilidad con estándares modernos. De este modo, las tecnologías seleccionadas han sido FastAPI para el backend, React para el frontend, PostgreSQL como sistema de base de datos y Docker junto con Nginx para el despliegue de la aplicación.

Durante este capítulo se comentan las aplicaciones existentes para la gestión de clubes deportivos destacando sus ventajas e inconvenientes, justificando de esta manera la elección de las tecnologías utilizadas en el desarrollo de esta aplicación. Finalmente, se presentan unas conclusiones obtenidas tras este análisis y cómo han influido en el diseño del sistema.

2.2. Aplicaciones similares

Junto a la creciente popularidad que ha sufrido el pádel, lógicamente las herramientas para la gestión de sus clubes deportivos han ido mejorando desde la utilización de métodos más tradicionales como una sencilla llamada telefónica, hasta la aparición de plataformas y aplicaciones

diseñadas para optimizar la administración de las instalaciones, reservas y usuarios. Existen diferentes soluciones en función de si responden a un pequeño negocio o si están más orientadas a grandes empresas con múltiples sedes.

A continuación, se presentan algunas de las soluciones más relevantes en este ámbito, analizando sus funcionalidades principales, ventajas y limitaciones en comparación con la aplicación propuesta en este trabajo.

2.2.1. Playtomic

Playtomic es una de las plataformas más populares para la gestión de reservas de pistas de pádel y otros deportes de raqueta. Esta aplicación permite la reserva de pista, la organización de partidos públicos en la que pueden apuntarse jugadores desconocidos y el pago a través de ella, dando servicio de esta manera a clubes y jugadores [1].

Ventajas:

- Gran cantidad de usuarios absorbiendo gran parte del mercado.
- Interfaz moderna e intuitiva, accesible desde web y aplicaciones móviles.
- Integración con sistemas de pago online.
- Funcionalidades sociales: creación de partidos abiertos, perfiles con niveles de juego, chat entre jugadores.
- Estadísticas de partidos y ranking personal.

Limitaciones:

- Dependencia de la plataforma y sus comisiones.
- Falta de personalización para clubes individuales.
- Limitada capacidad de personalización a nivel técnico o de funcionalidades.
- Falta de gestión propia del club desde la aplicación.

Fig. 2.1: Logotipo oficial de Playtomic

2.2.2. Reservaplay

Reservaplay es una plataforma para la gestión de reservas en clubes deportivos. Su principal función es la gestión de reservas, los usuarios que acceden y los pagos a través de ella. Esta aplicación permite reservas no únicamente de pistas de pádel, sino de pistas de tenis y campos de fútbol también [2].

Ventajas:

- Plataforma dedicada a la gestión de reservas y pagos online.
- Integración con sistemas de control de accesos.
- Gestión de torneos y rankings internos.

Limitaciones:

- No ofrece gestión completa de múltiples clubes.
- Interfaz menos intuitiva en comparación con otras soluciones.
- No presenta funcionalidades modernas como notificaciones o recomendaciones.
- Escasa flexibilidad para integrar nuevas funcionalidades personalizadas.

Fig. 2.2: Logotipo oficial de Reservaplay

2.2.3. Clupik

Clupik es una plataforma más generalista, enfocada en la gestión integral de clubes deportivos. Su enfoque no es exclusivo del pádel, sino que abarca distintos deportes y ofrece un enfoque más completo que va desde la gestión de jugadores, equipos y entrenamientos hasta comunicación interna y venta de merchandising [3].

Ventajas:

- Gestión integral de clubes, incluyendo equipos, socios y entrenadores.
- Sistema de comunicación interna con notificaciones y publicaciones.
- Tienda online integrada para venta de productos del club.

Limitaciones:

- No está especializada en clubes de pádel.
- Menos eficiente para la gestión de pistas y reservas.
- Tiene barrera de entrada en su utilización pues es complicada para usuarios sin experiencia previa en este tipo de aplicaciones.

Fig. 2.3: Logotipo oficial de Clupik

2.3. Tecnologías utilizadas

El desarrollo de esta aplicación web se ha basado en tecnologías modernas y eficientes que permiten una gestión robusta, escalable y fácil de mantener. La elección de cada herramienta ha sido fundamentada en su rendimiento, facilidad de desarrollo y compatibilidad con los estándares actuales. En esta sección se describen las principales tecnologías utilizadas, destacando sus ventajas y el motivo de su elección.

2.3.1. FastAPI: Framework para el backend

FastAPI es un framework moderno para el desarrollo de APIs en Python, diseñado para ofrecer alto rendimiento y facilidad de uso. Se basa en Python 3.7 y utiliza tipado estático para mejorar la validación de datos y la documentación automática.

Ventajas de FastAPI:

- Alto rendimiento: Comparable con frameworks como Node.js gracias a su uso de *async/await*.
- Validación automática de datos: Utiliza Pydantic para garantizar la integridad de los datos de entrada y salida.
- Gracias a la utilización de la biblioteca Pydantic podemos hacer la validación automática de los datos y conversión de tipos, garantizando de esta manera la integridad a la entrada y salida del sistema.
- Documentación automática: Genera automáticamente documentación interactiva con OpenAPI (*Swagger UI* y *ReDoc*).
- Framework sencillo que favorece el desarrollo ordenado y el futuro mantenimiento del código.

- Ejecución de tareas en segundo plano: Permite definir tareas asincrónicas que se ejecutan periódicamente sin bloquear el servidor, como el envío automatizado de recordatorios y solicitudes de reseñas al usuario tras realizar una reserva.

En este proyecto se ha elegido FastAPI teniendo en cuenta la facilidad que da para manejar solicitudes concurrentes de manera eficiente, lo cual es muy útil al desarrollar una aplicación web con múltiples usuarios gestionando reservas en tiempo real [4].

Fig. 2.4: Logo oficial de FastAPI

FastAPI ha experimentado un notable crecimiento en los últimos años, consolidándose como uno de los frameworks más populares para el desarrollo de APIs en Python. Según el informe *The State of Python 2024* elaborado por Michael Kennedy [5], el 46 % de los desarrolladores lo emplean para desarrollo web, y el 31 % para ciencia de datos, superando en adopción para APIs a frameworks veteranos como Flask y Django.

En la Figura 2.5 podemos observar una gráfica de barras horizontal comparando el uso principal que da cada framework de Python. Esta gráfica elaborada a partir del informe JetBrains (2024) [5], indica que, aunque Django sigue dominando el desarrollo web, FastAPI le sigue muy de cerca a pesar de ser el más reciente de los tres.

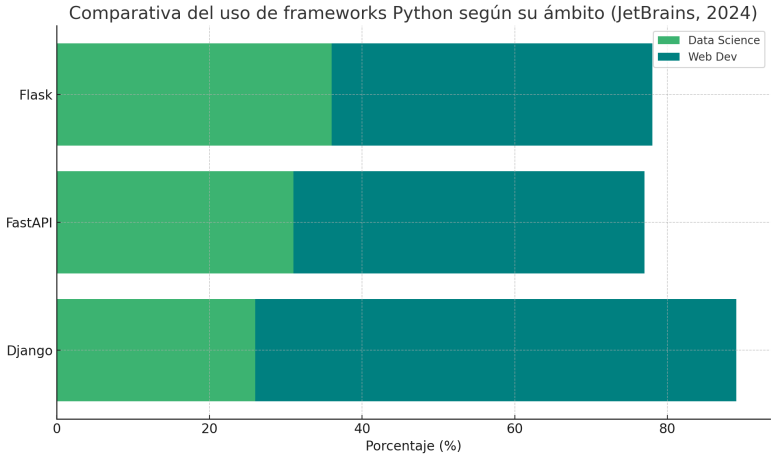

Fig. 2.5: Comparativa del uso de frameworks Python según su ámbito, elaborada a partir del informe JetBrains (2024) [5]

Este auge en su adopción se debe a su rendimiento, tipado estático, integración nativa con OpenAPI y su enfoque asincrónico, lo que lo convierte en una opción ideal para aplicaciones modernas que requieren escalabilidad, validación robusta y documentación automática.

Además, medios especializados destacan su creciente uso en sectores como inteligencia artificial, microservicios y plataformas de alto rendimiento, siendo adoptado incluso por grandes empresas tecnológicas [6].

FastAPI ha tenido un crecimiento enorme desde su lanzamiento en 2018, siendo rápidamente adoptado en la industria y consolidándose como uno de los frameworks más populares para la construcción de APIs en Python. Esto podemos verlo de manera visual en la Figura 2.6 que muestra la evolución estimada del uso de FastAPI entre 2019 y 2024 según el informe *The State of Python 2024* de JetBrains [5]. Según este estudio, FastAPI ha crecido desde 2019 a 2024 desde un 2 % de utilización por los desarrolladores hasta un 31 % en 2024.

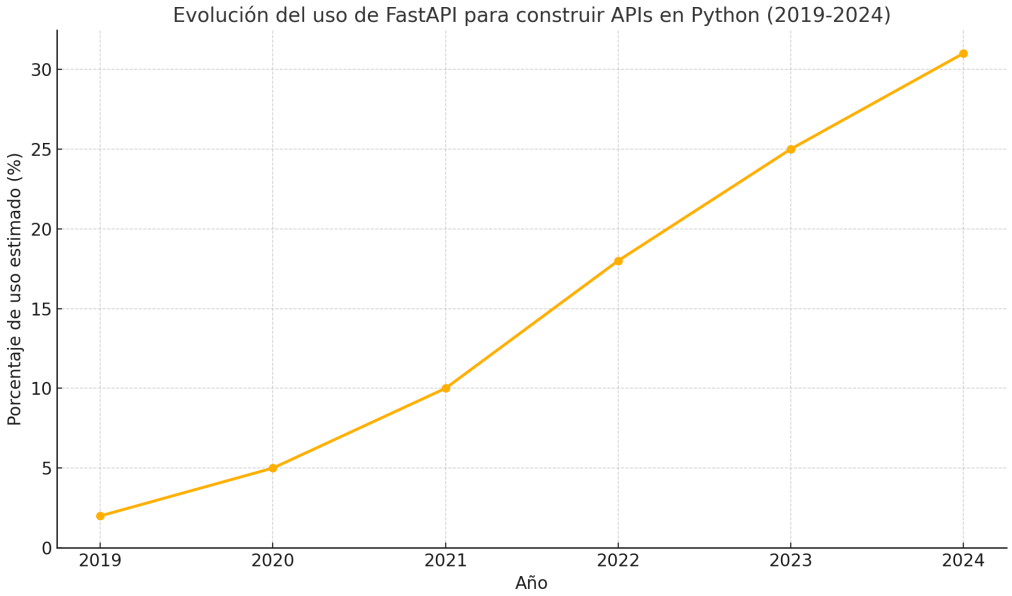

Fig. 2.6: Evolución del uso de FastAPI como framework para APIs en Python (2019–2024). Fuente: elaboración propia a partir de JetBrains [5].

2.3.2. React: Librería para el frontend

React es una librería de JavaScript desarrollada por Facebook, diseñada para la construcción de interfaces de usuario dinámicas y reutilizables. Su popularidad se debe a su enfoque basado en componentes, que facilita la escalabilidad y el mantenimiento de la aplicación.

Ventajas de React:

- Arquitectura basada en componentes: Permite el desarrollo modular y la reutilización de código.
- Virtual DOM: Optimiza la actualización y renderización de la interfaz de usuario.
- Ecosistema amplio: Compatible con múltiples librerías como React Router, Redux y Material-UI.

- Integración con RTK Query: Facilita la gestión de peticiones HTTP, el estado de carga y la caché mediante hooks automáticos generados a partir de los endpoints definidos [7].

React ha sido elegido para el frontend de la aplicación debido a su capacidad de construir interfaces interactivas y su compatibilidad con arquitecturas modernas basadas en APIs REST [8].

Fig. 2.7: Logotipo oficial de React

React se ha consolidado como una de las tecnologías más adoptadas en el desarrollo frontend moderno. Según la encuesta anual *State of JS* [9], React ha mantenido una posición de liderazgo durante más de un lustro, siendo utilizada por más del 82 % de los desarrolladores encuestados en 2023.

Esta popularidad sostenida se debe a su ecosistema robusto, la gran comunidad que lo respalda y su continua evolución, lo cual lo convierte en una opción de confianza tanto para pequeñas startups como para grandes empresas tecnológicas.

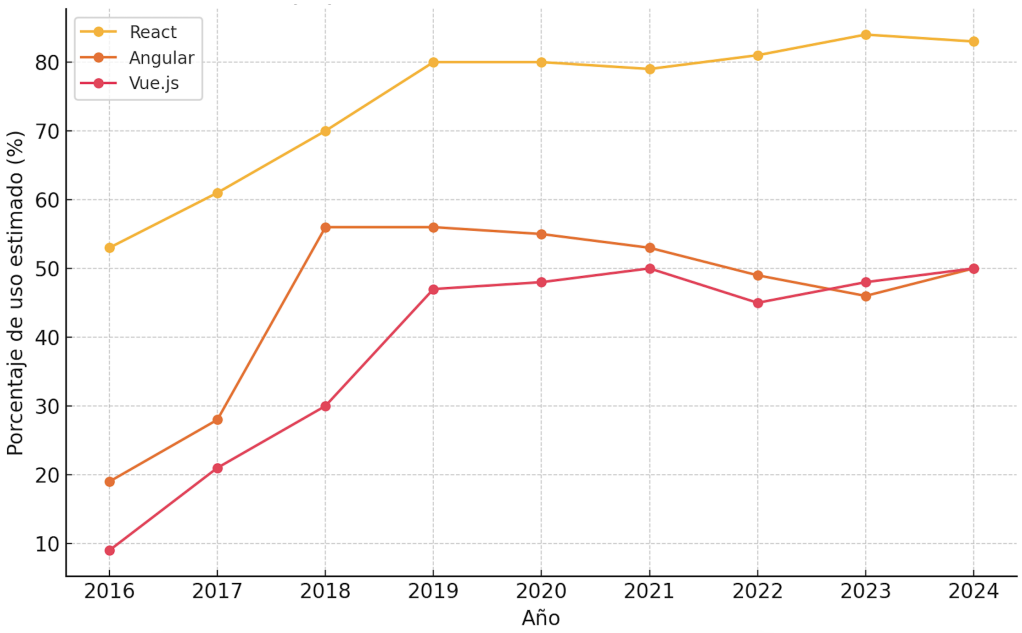

Fig. 2.8: Evolución de popularidad de frameworks frontend (2016–2024). Fuente: elaboración propia a partir de datos de State of JS [10].

La Figura 2.8 muestra la evolución del uso de bibliotecas frontend desde 2016 hasta 2024. En ella se observa cómo React se ha mantenido como la tecnología dominante durante todo el periodo, alcanzando picos de uso por encima del 80 % y consolidando su posición como la

herramienta preferida por los desarrolladores. Aunque otras opciones como Vue.js y Angular también han tenido una adopción significativa, ninguna ha logrado superar a React en cuanto a popularidad sostenida. Esta tendencia evidencia la madurez y aceptación generalizada de React en la industria del desarrollo frontend.

A parte de ser el más utilizado, React tiene una alta tasa de retención, indicando de esta manera que cuando un usuario lo prueba su utilización favorece que continué con el mismo entorno, favoreciendo de esta manera su posición dominante del mercado.

2.3.3. PostgreSQL: Sistema de base de datos

PostgreSQL es un sistema de gestión de bases de datos relacional de código abierto, reconocido por su fiabilidad, extensibilidad y conformidad con estándares SQL.

Ventajas de PostgreSQL:

- Escalabilidad y rendimiento: Soporta grandes volúmenes de datos y optimización de consultas avanzadas.
- Soporte para transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad): Permite que las operaciones sobre los datos se realicen de forma segura y sin errores.
- Extensibilidad: Permite el uso de funciones personalizadas y almacenamiento de datos JSON.
- Seguridad avanzada: Soporta autenticación robusta y control de acceso mediante roles y permisos.

PostgreSQL ha sido seleccionado para este proyecto debido a su capacidad de manejar múltiples conexiones simultáneas y su compatibilidad con aplicaciones escalables en producción [11].

Fig. 2.9: Logotipo oficial de PostgreSQL.

Postgres es uno de los gestores de bases de datos más populares entre los profesionales, con un 49 % de uso, superando a soluciones como MySQL y SQLite según la encuesta de desarrolladores de Stack Overflow 2024 [12].

Además, el informe *State of PostgreSQL 2024* publicado por Timescale [13] señala que más del 60 % de los encuestados utilizan PostgreSQL tanto en proyectos personales como profesionales, lo que evidencia su versatilidad en distintos entornos.

Por otro lado, el ranking de DB-Engines [14] posiciona a PostgreSQL como la cuarta base de datos más popular a nivel global en abril de 2025, con una tendencia sostenida al alza en los últimos años.

2.3.4. Docker y Nginx: Despliegue y gestión de contenedores

Para garantizar un despliegue eficiente y una infraestructura escalable, se ha optado por el uso de Docker y Nginx.

Uno de los problemas más comunes a la hora de desarrollar código es tener siempre instaladas las mismas dependencias y configuraciones en todos los entornos. Docker es una herramienta soluciona este problema creando contenedores aislados que encapsulan la aplicación siempre con las mismas características, haciendo que sea fácilmente portable y reproducible.

Docker permite empaquetar y desplegar la aplicación en contenedores, asegurando que el entorno de ejecución sea idéntico en desarrollo y producción [15].

Ventajas de Docker:

- Portabilidad: La aplicación se ejecuta de la misma manera en cualquier sistema sin depender de configuraciones específicas.
- Aislamiento de servicios: Cada componente (backend, frontend, base de datos) se ejecuta en su propio contenedor, evitando conflictos entre dependencias.
- Facilidad de escalado: Permite gestionar múltiples instancias de la aplicación de manera eficiente.

Fig. 2.10: Logotipo oficial de Docker

Nginx es un servidor web ligero y de alto rendimiento utilizado para servir los archivos estáticos del frontend y gestionar el tráfico HTTP de la aplicación [16].

Ventajas de Nginx:

- Optimización del rendimiento: Maneja múltiples conexiones concurrentes de manera eficiente.
- Balanceo de carga: permite distribuir el tráfico entre diferentes instancias favoreciendo la disponibilidad en todo momento.
- Cacheo y compresión de contenido: Reduce los tiempos de carga mejorando la experiencia del usuario.

El uso combinado de Docker y Nginx en este proyecto permite un despliegue seguro, eficiente y escalable, garantizando la estabilidad del sistema en entornos de producción [15, 16].

Fig. 2.11: Logotipo oficial de Nginx

Docker ha experimentado una adopción masiva en los últimos años, convirtiéndose en una herramienta clave en el desarrollo y despliegue de aplicaciones modernas. Según la encuesta anual *Stack Overflow Developer Survey 2023* [17], Docker es utilizado por más del 51 % de los desarrolladores encuestados, superando a otras herramientas como npm, Pip o Kubernetes. La Figura 2.12 muestra la popularidad comparativa de Docker frente a otras herramientas de desarrollo, destacando su posición de liderazgo.

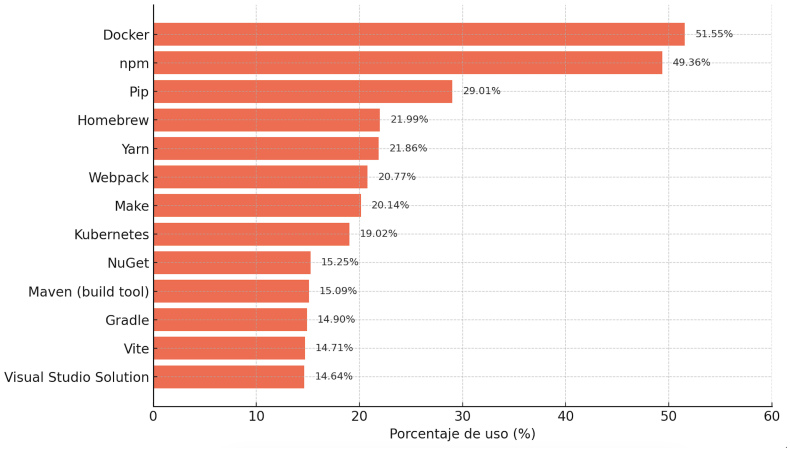

Fig. 2.12: Popularidad de Docker frente a otras herramientas de desarrollo en 2023. Docker lidera el ranking con más del 51 % de uso entre los desarrolladores encuestados. (Fuente: Stack Overflow Developer Survey 2023)

Por su parte, Nginx también goza de una alta adopción, siendo el servidor web más utilizado en plataformas modernas. Según W3Techs [18], Nginx es empleado por más del 33 % de todos los sitios web del mundo, incluyendo plataformas de alto tráfico como Netflix, Dropbox y GitHub, lo que evidencia su robustez y rendimiento.

Esta combinación de tecnologías no solo es ampliamente adoptada en la industria, sino que además representa una solución probada y escalable para entornos de microservicios y arquitectura basada en contenedores.

—

Capítulo 3

Análisis

3.1. Introducción

En este capítulo se detallan los requisitos que debe cumplir la aplicación desarrollada, tanto desde el punto de vista funcional como técnico. El análisis de requisitos constituye una fase fundamental en cualquier proceso de desarrollo de software, ya que permite comprender qué necesita el usuario, cuáles son los objetivos del sistema y qué restricciones técnicas o de negocio deben considerarse desde las primeras etapas del proyecto.

El objetivo principal es definir una solución que permita a los clubes de pádel gestionar de forma eficiente sus instalaciones, usuarios y reservas, así como facilitar la experiencia del cliente final a través de una plataforma accesible, moderna e intuitiva. Para ello, se toma como referencia el análisis del estado del arte descrito en el capítulo anterior, identificando las carencias de las soluciones existentes y las oportunidades de mejora.

A lo largo de este capítulo se presentará un caso de uso representativo de la aplicación, así como una clasificación de los requisitos funcionales y no funcionales que guían el diseño e implementación del sistema. Esta información servirá como base para el posterior desarrollo técnico que se expone en los siguientes capítulos del documento.

3.2. Ejemplo de uso de la aplicación

A continuación, se presenta un ejemplo práctico que describe el flujo de uso habitual de la aplicación desde la perspectiva de un cliente que desea reservar una pista, y de un administrador encargado de la gestión del club. Este escenario permite ilustrar cómo interactúan los distintos perfiles de usuario con el sistema, así como cómo las funcionalidades implementadas responden a las necesidades reales de un entorno deportivo digitalizado.

Cliente: búsqueda y reserva de pista

Un usuario accede a la plataforma desde su navegador o dispositivo móvil. Tras iniciar sesión, puede seguir dos rutas principales. En un primer vistazo en la página de *Home* puede descubrir las distintas localizaciones que tiene el club y, si se selecciona una de las mismas, verá la visualización de esta localización, su tiempo actual y la posibilidad de reservar una pista. Por

otra parte, se puede ir al área de miembros, y allí, al ver las reservas actuales, se puede crear una nueva reserva. Por ambos caminos se llega al mismo lugar para la reserva, allí se selecciona el club de pádel al que desea asistir, y se consulta la disponibilidad de pistas para una fecha concreta.

Una vez seleccionada la fecha, el sistema le muestra las franjas horarias que hay disponibles, dado que solo se permiten reservas de 1 hora y media de duración entre las nueve de la mañana y las doce de la noche. Al seleccionar la franja horaria deseada, el sistema le muestra las pistas disponibles, junto con información adicional como la puntuación media de cada pista y la última valoración recibida. Adicionalmente, puede expandir esto y ver las 5 últimas valoraciones de esta pista. El cliente elige una pista en función de la puntuación y los comentarios recibidos, y completa el proceso de reserva confirmando fecha, hora y pista de su reserva.

Una vez hecha la reserva, recibe un correo electrónico de confirmación. Además, 24 horas antes de la hora reservada, el sistema le envía un nuevo correo con la previsión meteorológica actualizada, indicando si ha habido algún cambio respecto a la previsión inicial. Si, por ejemplo, se preveía sol y ahora se espera lluvia, el usuario puede valorar modificar o cancelar su reserva con antelación.

Al finalizar la sesión de juego, el sistema envía automáticamente un nuevo correo solicitando una reseña de la pista utilizada. Esta valoración se almacena y se muestra a otros usuarios que busquen reservar esa pista en el futuro.

Administrador: gestión del club y reservas

Los usuarios con perfil de administrador acceden a un área de gestión diseñada específicamente para facilitar la administración de uno o varios clubes deportivos. Al iniciar sesión, se presenta un panel inicial en el que pueden visualizar los distintos clubes que tienen asignados, junto con un resumen general del estado de cada uno, incluyendo el número total de reservas registradas.

Una vez seleccionado un club, el sistema permite al administrador *impersonarse* en la gestión del mismo. Desde el *dashboard principal* se ofrece una vista en tiempo real de las pistas que están actualmente ocupadas, así como un listado de las reservas programadas para las próximas horas. Esta funcionalidad permite tener una visión rápida y clara de la operativa diaria del club.

Para gestionar las reservas, el administrador puede ir a la sección correspondiente donde puede obtener un listado de todas las reservas, tanto las futuras que aún están por llegar, como el histórico de todas las reservas, con la posibilidad de ordenarlas y filtrarlas por diferentes criterios (fecha, pista, usuario, etc.). Desde aquí, también tiene la posibilidad de cancelar manualmente cualquier reserva en caso de que surja una incidencia.

Esta funcionalidad integral permite al administrador mantener el control operativo del club de forma ágil y eficiente, contando con información actualizada y herramientas prácticas para la toma de decisiones y la atención a los usuarios.

3.3. Requisitos funcionales

Los requisitos funcionales definen el conjunto de funcionalidades que la aplicación debe ofrecer para cubrir las necesidades de los distintos tipos de usuarios (clientes y administradores). A

continuación, se detallan los principales:

- **Gestión de clubes:** la plataforma permite la administración de múltiples clubes, cada uno con sus propias ubicaciones, horarios de apertura y pistas disponibles. Además, es posible gestionar los servicios ofrecidos por cada club de forma individualizada.
- **Gestión de usuarios:** en el sistema tenemos dos roles de usuario, uno para los clientes y otro para los administradores. Los administradores pueden gestionar clubes, administrar usuarios, crear, eliminar y supervisar las reservas, mientras que los clientes pueden realizar reservas y consultar información relevante.
- **Reservas de pistas:** los clientes pueden consultar la disponibilidad de pistas para una fecha y hora determinada, visualizar la puntuación de cada pista y completar el proceso de reserva a través de una interfaz sencilla.
- **Otros servicios:** además de pistas, el sistema está diseñado para permitir la reserva de otros servicios complementarios que pueda ofrecer el club (como clases) o servicios como leer las últimas noticas del mundo del padel.
- **Notificaciones:** el sistema envía correos electrónicos en diferentes momentos del proceso: confirmación de la reserva, recordatorio con la previsión meteorológica actualizada 24 horas antes del evento, y solicitud de valoración una vez finalizada la reserva.

3.4. Requisitos no funcionales

Además de cumplir con los requisitos funcionales, el sistema debe ofrecer un rendimiento adecuado, seguridad y facilidad de mantenimiento. A continuación, se presentan los requisitos no funcionales clave:

- **Escalabilidad:** la arquitectura de la aplicación debe permitir la gestión de un número creciente de clubes, usuarios y reservas sin pérdida de rendimiento. Para ello, se emplea una arquitectura cliente-servidor desacoplada basada en API REST.
- **Seguridad:** es importante proteger los datos personales de los usuarios. Para ello, el sistema usa autenticación, control de acceso según el tipo de usuario y conexión segura entre el frontend y el backend.
- **Despliegue eficiente:** mediante el uso de *Docker* se facilita la contenedorización de cada componente del sistema (frontend, backend y base de datos), garantizando entornos replicables. *Nginx* actúa como servidor de aplicaciones y proxy inverso, permitiendo escalar y distribuir la carga del sistema de forma eficiente.
- **Mantenibilidad y modularidad:** la aplicación se desarrolla siguiendo principios de separación de responsabilidades, utilizando frameworks modernos como FastAPI y React, que facilitan futuras extensiones o modificaciones del sistema.

Capítulo 4

Diseño

4.1. Introducción

El diseño del sistema representa una de las fases clave del desarrollo de esta aplicación web, ya que define la estructura interna, la organización de los distintos componentes y la forma en que interactúan entre sí. Un diseño bien estructurado garantiza que la aplicación sea escalable, mantenible y eficiente, facilitando su evolución a lo largo del tiempo.

Para abordar esta etapa, se ha seguido un enfoque modular y orientado a componentes, tanto en el backend como en el frontend. La arquitectura está basada en el patrón cliente-servidor, en el que el frontend (desarrollado con React) se comunica con el backend (implementado en FastAPI) a través de una API RESTful. La persistencia de los datos se gestiona mediante PostgreSQL, un sistema de bases de datos relacional robusto y altamente escalable.

Este sistema se ha construido dividido en bloques independientes, separando la lógica del backend, la interfaz de usuario y la base de datos. El backend (implementado en FastAPI) y el frontend (desarrollado con React) se comunican a través de un API REST, permitiendo su conexión en todo momento pero trabajando de forma independiente. PostgreSQL es el sistema de base datos relacional que gestiona la persistencia de los datos, almacenándolos de forma segura y eficiente.

Además, con el objetivo de simplificar el despliegue, se ha optado por una solución basada en contenedores mediante Docker. El servidor Nginx se utiliza exclusivamente para servir los archivos estáticos generados por el frontend, permitiendo una entrega eficiente de los recursos al navegador del usuario.

En los siguientes apartados se describen los componentes principales de la arquitectura, el diseño del modelo de datos, la estructura del backend y el diseño de la interfaz en el frontend.

4.2. Arquitectura del sistema

La arquitectura de la aplicación desarrollada se basa en un modelo cliente-servidor, donde el frontend y el backend se encuentran desacoplados y se comunican a través de una API REST. Este enfoque permite una mayor flexibilidad, escalabilidad y facilidad de mantenimiento.

A continuación, se describen los principales componentes del sistema y su interacción:

- **Frontend (React):** El cliente es una aplicación web desarrollada con React. Se encarga de mostrar la interfaz gráfica al usuario y gestionar la navegación y las interacciones. Realiza peticiones HTTP al backend para enviar o solicitar datos (por ejemplo, iniciar sesión, consultar disponibilidad o crear reservas). Gracias al uso de la librería RTK Query, se simplifica la gestión de estado y el consumo de la API.
- **Backend (FastAPI):** El servidor está desarrollado con FastAPI, un framework moderno y eficiente para construir APIs en Python. Se encarga de la lógica de negocio, el control de acceso, la validación de datos y la conexión con la base de datos. Define endpoints organizados por rutas (reservas, usuarios, clubs, reseñas, etc.) y devuelve respuestas en formato JSON al cliente. También incluye funcionalidades en segundo plano, como envío de correos recordatorio o actualización de datos meteorológicos mediante tareas asíncronas.
- **Base de datos (PostgreSQL):** El sistema utiliza PostgreSQL como motor de base de datos relacional para almacenar toda la información relevante: usuarios, reservas, pistas, valoraciones, etc. Se accede a ella desde el backend utilizando SQLAlchemy como ORM (Object-Relational Mapping) para facilitar la interacción con las tablas y mantener la coherencia de los datos.
- **Docker y Nginx:** La aplicación se despliega en contenedores Docker, lo que garantiza un entorno homogéneo entre desarrollo y producción. Cada componente (frontend, backend, base de datos) se ejecuta en su propio contenedor, facilitando el despliegue, el mantenimiento y la escalabilidad del sistema. Nginx actúa como servidor web ligero para servir los archivos estáticos generados por el frontend, mejorando el rendimiento en producción.

Esta arquitectura modular permite que cada parte de la aplicación se desarrolle y escale de forma independiente, lo cual resulta clave en proyectos con expectativas de crecimiento. Además, el uso de tecnologías ampliamente adoptadas y modernas garantiza la sostenibilidad del proyecto a largo plazo.

4.3. Modelo de datos

El modelo de datos define la estructura de la información gestionada por la aplicación y establece las relaciones entre las distintas entidades del sistema. En este proyecto, se ha seguido un enfoque relacional empleando **PostgreSQL** como sistema de gestión de bases de datos y **SQLAlchemy ORM** para definir los modelos en Python y mapearlos directamente a las tablas correspondientes [11, 19].

Con **SQLAlchemy** podemos comunicarnos con la base de datos utilizando clases de Python, simplificando mucho el el desarrollo del sistema. Es un enfoque orientado a objetos, en el que cada elemento del sistema (usuarios, pistas, reservas, etc) se modela como una clase con sus respectivos atributos y vínculos.

Además, se ha empleado **Pydantic** para la validación de datos y la serialización entre objetos Python y estructuras JSON. Esta librería, ampliamente utilizada junto a FastAPI, permite definir esquemas de entrada y salida basados en clases que heredan de la clase `BaseModel` de `Pydantic`, lo cual garantiza que los datos intercambiados en las peticiones cumplan con los formatos y tipos esperados [20].

La base de datos incluye varias entidades principales: *usuarios*, *reservas*, *pistas*, *clubes*, *valoraciones*, *preguntas frecuentes*. A continuación, se describen todas las tablas del sistema.

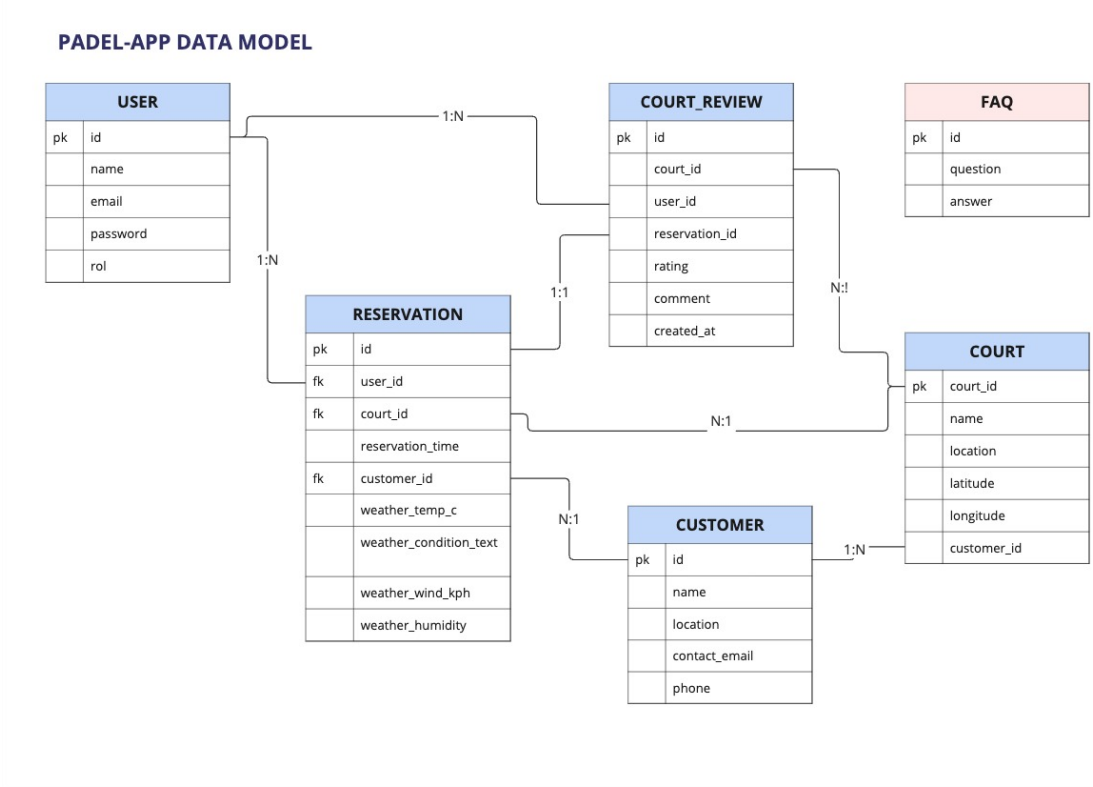

Fig. 4.1: Modelo entidad-relación del sistema.

Usuarios (`users`)

La tabla de usuarios almacena la información de las personas que acceden a la plataforma. Un usuario puede tener distintos roles como `operator` (administrador) o `customer` (cliente).

- `id`: Identificador único del usuario.
- `name`: Nombre de usuario (único).
- `email`: Correo electrónico del usuario.
- `password`: Contraseña cifrada.
- `role`: Rol del usuario (por ejemplo, `customer` o `operator`).

Clubes (`customers`)

Representa los clubes de pádel que utilizan la plataforma. Cada club puede tener múltiples pistas asociadas.

- `id`: Identificador único del club.
- `name`: Nombre del club.
- `location`: Ciudad o ubicación del club.

- `contact_email`: Correo de contacto.
- `phone`: Teléfono del club.

Pistas (`courts`)

Tabla que representa las pistas de cada club. Cada pista pertenece a un único club.

- `court_id`: Identificador único de la pista.
- `name`: Nombre de la pista (normalmente un número).
- `location`: Zona dentro del club.
- `latitude` y `longitude`: Coordenadas geográficas de la pista.
- `customer_id`: Identificador del club al que pertenece.

Reservas (`reservations`)

Tabla que gestiona la información de las reservas realizadas por los clientes.

- `id`: Identificador único de la reserva.
- `user_id`: Usuario que realizó la reserva.
- `court_id`: Pista reservada.
- `customer_id`: Club al que pertenece la pista.
- `reservation_time`: Fecha y hora de la reserva.
- `weather_temp_c`: Temperatura prevista.
- `weather_condition_text`: Condición climática (por ejemplo, `Soleado`).
- `weather_wind_kph`: Velocidad del viento.
- `weather_humidity`: Humedad relativa.

Valoraciones (`court_reviews`)

Los usuarios pueden valorar una pista tras haber realizado una reserva. Estas valoraciones se muestran para ayudar a otros usuarios en futuras decisiones.

- `id`: Identificador de la valoración.
- `court_id`: Pista valorada.
- `user_id`: Usuario que realizó la valoración.
- `reservation_id`: Reserva asociada a la valoración.

- `rating`: Puntuación del 1 al 5.
- `comment`: Comentario opcional.
- `created_at`: Fecha de la valoración.

Preguntas frecuentes (`faqs`)

Contiene una colección de preguntas y respuestas para mejorar la experiencia de los usuarios en la plataforma.

- `id`: Identificador único de la entrada.
- `question`: Pregunta frecuente.
- `answer`: Respuesta asociada.

Este modelo de datos ha sido diseñado para garantizar la integridad referencial, facilitar las consultas necesarias para la lógica del negocio y asegurar una buena escalabilidad de cara al futuro.

El diagrama entidad-relación ha sido elaborado utilizando la herramienta *Miro* [21], lo que ha permitido representar de forma visual las relaciones entre entidades del sistema.

4.4. Clases y módulos de diseño (backend)

El backend de esta aplicación ha sido desarrollado utilizando **FastAPI**, un framework moderno y eficiente basado en Python para la construcción de APIs. Una de las principales ventajas de FastAPI es su integración nativa con `Pydantic`, lo que permite validar y documentar automáticamente los datos de entrada y salida mediante anotaciones de tipos.

Estructura del backend

La arquitectura del backend sigue una estructura modular, organizada en directorios que separan claramente las responsabilidades. Esta separación facilita la escalabilidad del sistema y mejora el mantenimiento del código:

- `config/`: Contiene la configuración global de la aplicación, como claves secretas, parámetros de JWT, y carga de variables de entorno mediante `Pydantic.BaseSettings`.
- `db/`: Define la configuración de la base de datos mediante SQLAlchemy, incluyendo la creación del motor de conexión, la clase base para los modelos (`Base`), y la creación de sesiones.
- `models/`: Incluye los modelos ORM, donde cada clase representa una tabla de la base de datos. Las relaciones entre entidades se definen con `relationship` y claves foráneas (`ForeignKey`).

- `schemas/`: Define los modelos de datos para entrada y salida mediante clases `Pydantic`, proporcionando validación automática, conversión a JSON y generación de documentación para la API.
- `routers/`: Contiene los distintos módulos de rutas, organizados por dominio funcional (usuarios, reservas, preguntas frecuentes, etc.). Cada módulo agrupa endpoints relacionados y facilita el desarrollo colaborativo.
- `dependencies/`: Almacena funciones reutilizables para inyectar dependencias como la sesión de base de datos o la autenticación JWT.
- `services/`: Implementa lógica de negocio auxiliar, como el envío de correos electrónicos, acceso a APIs externas (clima), o comprobaciones automáticas con tareas programadas.
- `main.py`: Es el punto de entrada de la aplicación. Configura CORS, registra los routers y lanza tareas asincrónicas de fondo (como recordatorios por correo).

Herramientas utilizadas

- **FastAPI** [4]: Framework principal para la construcción de endpoints HTTP.
- **SQLAlchemy ORM** [19]: Mapeo objeto-relacional que permite definir clases Python como tablas relacionales.
- **Pydantic** [20]: Librería para validación de datos y definición de esquemas de forma declarativa.
- **PostgreSQL** [11]: Sistema de gestión de bases de datos relacional utilizado como almacenamiento principal.
- **JWT (JSON Web Tokens)**: Método de autenticación que utiliza tokens desde los que se puede identificar a los usuarios y parte de su información.
- **Docker**: Contenedores para el backend y base de datos, simplificando la ejecución y el despliegue.
- **PgAdmin**: Herramienta web para administración visual de la base de datos PostgreSQL.

Sistema de autenticación con JWT

El sistema de autenticación implementado en esta aplicación se basa en el uso de **JWT (JSON Web Tokens)**, una tecnología ampliamente utilizada para autenticar usuarios en aplicaciones modernas, especialmente en arquitecturas de tipo *frontend-backend* desacopladas, como la que se emplea en este proyecto con React y FastAPI.

JWT es un formato estándar definido en el RFC 7519 [22] para transmitir información segura en una comunicación. Un token JWT consta de tres partes separadas por puntos:

- **Header (cabecera)**: contiene el tipo de token ("JWT") y el algoritmo de firma, como HMAC SHA256 o RSA.
- **Payload (carga útil)**: contiene las *claims*, es decir, los datos que se quieren transmitir. En este caso, incluye el ID del usuario, su rol (por ejemplo, `operator` o `customer`), y otra información adicional si se desea.

- **Signature (firma)**: se genera aplicando el algoritmo especificado al *header* y *payload*, junto con una clave secreta. Sirve para verificar que el contenido no ha sido modificado.

¿Cómo se genera un token?

Durante el proceso de `/login`, el usuario envía su nombre y contraseña. Si las credenciales son correctas, se genera un token JWT con los siguientes pasos:

1. Se crea una carga útil (`payload`) con el ID del usuario (`sub: user.id`) y su rol.
2. Se firma el token con una clave secreta definida en el archivo de configuración (`config.py`).
3. Se devuelve el token como respuesta al cliente, junto con un token de refresco.

Este proceso se implementa mediante la librería `fastapi-jwt-auth`, que proporciona herramientas para la creación, validación y renovación de tokens.

¿Cómo se usa el token en las peticiones?

Una vez autenticado, el cliente (por ejemplo, el frontend en React) incluye el token en cada petición protegida dentro de la cabecera `Authorization`, con el formato:

```
Authorization: Bearer <token>
```

En el backend, FastAPI utiliza una dependencia personalizada (`jwt_required`) que se encarga de verificar la validez del token y extraer la identidad del usuario. Si el token es válido, se permite el acceso al endpoint; en caso contrario, se devuelve un error 401 (Unauthorized).

Control de acceso por roles

Además de verificar la autenticación, algunos endpoints aplican un control de acceso adicional en función del **rol del usuario**. Esto se logra mediante una función llamada `require_role()`, que extrae el campo `role` del token y compara su valor. Por ejemplo, sólo los usuarios con el rol `operator` pueden eliminar usuarios o acceder a ciertos recursos administrativos.

Este enfoque aporta un modelo de autorización flexible y escalable, permitiendo implementar políticas de acceso más avanzadas en el futuro (como múltiples niveles de permisos o integración con OAuth2).

Ventajas de JWT

- **Independencia del servidor**: el backend no necesita mantener el estado de la sesión. Toda la información está contenida en el token.
- **Escalabilidad**: ideal para sistemas distribuidos y microservicios, donde se requiere autenticación sin almacenamiento centralizado de sesiones.

- **Seguridad**: los tokens tienes una duración limitada, que obliga a volver a iniciar sesión. Como están firmados, no pueden ser modificados sin invalidar la firma.

Gracias a este sistema, se garantiza que cada solicitud al backend está autenticada y autorizada de manera segura, manteniendo una arquitectura sin estado (*stateless*) y preparada para escalar horizontalmente.

Tareas en segundo plano con `asyncio` y eventos de arranque en FastAPI

FastAPI proporciona una funcionalidad muy útil para lanzar procesos en segundo plano de manera asíncrona durante el arranque del servidor mediante el decorador `@app.on_event("startup")`. Este mecanismo permite ejecutar lógica personalizada justo cuando la aplicación se inicia, antes de comenzar a atender peticiones HTTP.

Inicialización del sistema

En este proyecto, se hace uso de este evento de arranque para realizar dos tareas clave al iniciar la aplicación:

1. Crear las tablas en la base de datos (si no existen), a partir de los modelos ORM definidos mediante SQLAlchemy.
2. Inicializar los datos del sistema: clientes, pistas, usuarios, FAQs, etc., mediante la función `create_initial_data()`.

Estas acciones se llevan a cabo de forma síncrona, garantizando que la base de datos está correctamente estructurada antes de aceptar peticiones.

```
@app.on_event("startup")
async def create_tables():
    Base.metadata.create_all(bind=engine)
    create_initial_data()
    asyncio.create_task(reminder_scheduler())
```

Ejecución de tareas en segundo plano

La funcionalidad clave de este bloque es el uso de `asyncio.create_task()`, que permite lanzar una tarea asíncrona que se ejecuta en segundo plano de manera independiente al hilo principal del servidor.

En este caso, la tarea que se lanza es `reminder_scheduler()`, un *loop* infinito que cada cierto tiempo realiza las siguientes acciones:

- Verifica si hay reservas programadas para las próximas 24 horas y envía correos electrónicos recordatorios a los usuarios.

- Comprueba si la previsión meteorológica ha cambiado desde que se hizo la reserva e informa al usuario por si desea modificarla.
- Comprueba si hay reservas que han finalizado recientemente (en los últimos 15 minutos) y, en tal caso, envía un correo automático invitando al usuario a dejar una valoración sobre la pista utilizada.

Esto se gestiona como una tarea asíncrona en segundo plano, por lo que no afecta al funcionamiento para el procesamiento de las peticiones HTTP normales. Esta tarea asíncrona lanza el event loop de `asyncio` que coteja cada parámetro periódicamente.

Ventajas del enfoque asíncrono

- **No bloqueante**: Al ejecutarse en segundo plano de forma asíncrona, no se interrumpe el procesamiento de peticiones ni se ralentiza la respuesta del servidor.
- **Flexibilidad**: Permite la ejecución de distintas tareas al mismo tiempo sin necesidad de utilizar servicios adicionales como pueden ser jobs externos.
- **Integración nativa**: Todo se gestiona dentro del propio servidor FastAPI, lo que facilita el mantenimiento, la depuración y la coherencia del sistema.

Este enfoque demuestra una clara ventaja en términos de escalabilidad y simplicidad arquitectónica, ya que permite incorporar funcionalidades complejas (como alertas inteligentes o automatización de procesos) sin recurrir a herramientas externas como Celery y Redis, o la utilización de procesos más clásicos como cron.

Integración con servicios externos: API meteorológica

Una de las ventajas clave de las arquitecturas modernas basadas en microservicios y APIs es la posibilidad de incorporar datos de terceros para enriquecer la experiencia del usuario. En este proyecto se ha optado por integrar información meteorológica contextual a cada reserva deportiva, con el objetivo de ofrecer al usuario una visión más completa de las condiciones en las que tendrá lugar su actividad.

WeatherAPI: un servicio de predicción meteorológica

Para ello se ha utilizado **WeatherAPI** [23], un servicio web RESTful que permite acceder a información meteorológica actual, histórica y futura. Desarrollada por Apixu Ltd., WeatherAPI proporciona una gran variedad de datos como temperatura, velocidad del viento, humedad, condiciones climáticas, visibilidad, presión atmosférica, entre otros. Además, permite realizar consultas por localización geográfica (latitud y longitud) y por fecha, lo que la convierte en una solución ideal para aplicaciones que requieren predicción horaria o seguimiento climático personalizado.

La API está organizada en múltiples endpoints especializados. Entre ellos, destacan:

- `/current.json`: Devuelve el estado actual del clima en una localización determinada.

- `/forecast.json`: Proporciona predicciones meteorológicas por horas hasta un máximo de 14 días.
- `/history.json`: Permite consultar datos climáticos históricos para análisis o comparativas.

Aplicación en el sistema desarrollado

En el contexto de esta aplicación, la integración con WeatherAPI permite asociar una reserva deportiva a una previsión meteorológica específica. Esto se logra realizando una consulta en tiempo real a la API, en función de la ubicación de la pista (coordenadas geográficas) y la fecha y hora de la reserva.

Esta funcionalidad tiene varios usos prácticos:

- **Prevención**: el usuario puede decidir si reservar en función de condiciones adversas (viento fuerte, lluvia, etc.).
- **Notificación inteligente**: el sistema es capaz de enviar recordatorios previos con información actualizada del clima.
- **Contextualización**: se añade valor a la reserva mostrando la temperatura y estado del cielo previstos.

Además, esta integración es un claro ejemplo del aprovechamiento de servicios externos mediante API REST, algo especialmente útil en aplicaciones como la nuestra, donde la información contextual —como el clima en una ubicación y momento concretos— aporta valor añadido a la experiencia del usuario, permite personalizar las comunicaciones y contribuye a una toma de decisiones más informada por parte del cliente.

4.5. Interfaz de usuario (frontend)

La interfaz de usuario de la aplicación ha sido desarrollada empleando React [8], una biblioteca declarativa y basada en componentes para construir interfaces web interactivas. Esta tecnología permite crear vistas dinámicas y escalables que reaccionan al estado de la aplicación sin necesidad de recargar la página, lo que mejora notablemente la experiencia de usuario.

Arquitectura y estructura del proyecto

El proyecto sigue una estructura modular basada en la separación de responsabilidades. El código fuente está organizado de forma funcional, agrupando páginas, componentes, lógica de negocio y servicios en carpetas independientes:

- `components/`: Contiene componentes reutilizables clasificados según el patrón atómico (atoms, molecules, organisms).
- `pages/`: Vistas principales como el panel de reservas, sección de noticias, FAQs, login, etc.

- `modules/`: Contiene lógica específica como validaciones, gestión de errores, navegación protegida, entre otros.
- `domain/`: Centraliza la lógica del negocio, incluyendo *store.js*, slices de *Redux*, servicios de *RTK Query* y definiciones de endpoints.
- `custom/`: Contiene temas personalizados, fuentes tipográficas y configuraciones visuales.
- `translations/`: Gestión de la internacionalización mediante `i18next`, con soporte multilingüe.

Gestión de rutas y acceso

La navegación se implementa con *React Router*, permitiendo rutas protegidas basadas en roles mediante el componente *RequireAuth*, que restringe el acceso a ciertas páginas dependiendo del tipo de usuario (por ejemplo, *customer* u *operator*). De esta forma, si el usuario autenticado es un *customer* verá las páginas de cierta manera, el perfil de un cliente con limitadas funciones, y si el usuario autenticado es un *operator* tendrá acceso a páginas exclusivas de administradores con un estilo personalizado para ello.

En el `App.js` se define cada ruta utilizando la carga diferida (*lazy*) y el componente *Suspense*. Gracias a esto, se mejora el rendimiento al dividir el código y cargar las vistas en el momento en el que son necesarias.

Gestión del estado y llamadas a la API

Para el manejo del estado global se emplea **Redux Toolkit**, junto con **RTK Query** [7], una herramienta que automatiza la gestión de peticiones asíncronas y la caché de datos. Esta combinación permite:

- Consultas declarativas con lógica desacoplada del componente.
- Actualización automática de los datos al mutar entidades (por ejemplo, crear, editar o eliminar reservas).
- Gestión del estado de carga, errores y caché de forma centralizada.
- Resiliencia en caso de error, pues al detectar un código 401, se reintenta automáticamente. También incluye renovación de tokens mediante *refresh token*.
-

Diseño visual e interfaz

El diseño visual de la plataforma se ha construido utilizando **Material UI (MUI)** [24], una de las bibliotecas de componentes más populares y maduras del ecosistema React. MUI proporciona un conjunto completo de componentes listos para usar, como botones, formularios, diálogos, tarjetas, menús desplegables, iconos, tablas y muchos más, todos diseñados siguiendo las directrices de diseño de Google Material Design.

Esta biblioteca permite desarrollar interfaces modernas, accesibles y completamente responsivas sin necesidad de escribir gran cantidad de CSS personalizado. Además, incluye soporte para temas, lo que permite personalizar los colores, tipografías y espaciados globales de la aplicación para adaptarlos a una identidad visual concreta, como la utilizada en este proyecto.

En esta aplicación, MUI se ha integrado estrechamente con el sistema de rutas y la lógica de negocio para garantizar una experiencia de usuario fluida y coherente. Entre sus ventajas más destacadas, cabe señalar:

MUI se ha integrado en todo el sistema, lo que garantiza una experiencia de usuario fluida y dinámica. Tiene numerosas ventajas entre las que cabe destacar:

- **Componentes accesibles y adaptativos**: MUI garantiza compatibilidad con dispositivos móviles y estándares de accesibilidad.
- **Integración con iconografía**: Se han utilizado iconos de la colección oficial de Material Icons, fácilmente importables como componentes.
- **Sistema de estilos basado en *sx* y *styled components***: Facilita la personalización puntual sin comprometer la coherencia visual global.
- **Documentación detallada y comunidad activa**: Contribuye a que sea sencillo solventar dudas durante el desarrollo.

La estructura de estilos del frontend se refuerza con hojas de estilo adicionales definidas en `index.css`, donde se cargan tipografías personalizadas mediante la directiva *@font-face*, proporcionando una identidad visual propia a la aplicación.

El uso de Material UI ha sido clave para el desarrollo ágil y profesional de la interfaz de usuario, permitiendo centrarse en la lógica funcional del sistema sin necesidad de reinventar elementos visuales comunes. Gracias a su sistema de diseño modular, se ha conseguido mantener la coherencia visual entre páginas y facilitar la escalabilidad futura del frontend.

Componentes clave de la plataforma

La interfaz ofrece una experiencia completa al usuario final, que puede navegar por las siguientes secciones destacadas:

- **Panel de reservas**: Permite seleccionar fecha, pista y visualizar disponibilidad, todo sincronizado con el backend.
- **Área privada del usuario**: Muestra reservas futuras y acceso a edición de perfil.
- **Valoraciones**: Sistema para puntuar y comentar sobre las pistas, integrado con recordatorios automáticos por correo electrónico.
- **Sección de FAQs**: Información dinámica obtenida del backend sobre preguntas frecuentes.
- **Noticias**: Integración con *NewsAPI* para mostrar noticias actualizadas sobre el mundo del pádel.
- **Panel de operador**: Sección administrativa para operadores, con gestión de usuarios, panel de estado de los clubes y gestion de reservas de las pista.

Internacionalización y accesibilidad

La aplicación es multilingüe, con soporte para los idiomas `es` (español), `en` (inglés), `de` (alemán) y `pt` (portugués). Para implementar esta funcionalidad se ha utilizado la librería **i18next** [25], una solución ampliamente adoptada en el ecosistema JavaScript para la internacionalización de interfaces web y móviles.

i18next proporciona un sistema de traducción flexible y escalable basado en ficheros externos JSON y el uso de namespaces. En este proyecto, cada módulo o página del frontend cuenta con su propio espacio de nombres (*namespace*), lo cual facilita la organización modular de las cadenas traducibles, así como su mantenimiento a largo plazo. Esto resulta especialmente útil para colaborar con traductores externos, ya que cada conjunto de traducciones está claramente delimitado.

El sistema está integrado a través del proveedor *TranslationProvider*, el cual encapsula la lógica de carga dinámica de recursos según el idioma actual. Las traducciones se obtienen bajo demanda usando el hook *useTranslation()*, y gracias a esta arquitectura, el cambio de idioma es inmediato y reactivo en toda la aplicación.

Cuenta con numerosas ventajas entre las que destacan:

- **Compatibilidad con React**: i18next cuenta con soporte oficial para React a través de *react-i18next*.
- **Carga asincrónica de recursos**: Las traducciones se pueden cargar de forma remota o local, lo que permite optimizar el rendimiento.
- **Interpolación de variables y pluralización**: Permite construir mensajes complejos adaptados al contexto del idioma.
- **Detección de idioma automática**: Puede configurarse para detectar el idioma del navegador del usuario.

Además de su valor funcional, la internacionalización abre las puertas a una expansión global del producto, haciendo posible que usuarios de distintas regiones accedan a la aplicación en su idioma nativo, mejorando la accesibilidad, la usabilidad y la satisfacción general del usuario.

4.6. Despliegue y ejecución

La aplicación frontend está construida utilizando **React**, lo que permite compilar el código fuente (JSX, CSS, assets, etc.) en una colección de *ficheros estáticos* optimizados para producción. Estos archivos, normalmente alojados en la carpeta `build/`, incluyen HTML, JavaScript minificado, hojas de estilo y recursos multimedia, y son ideales para ser servidos rápidamente al cliente sin necesidad de lógica de backend.

Para hacer el despliegue en producción, se cogen los ficheros estáticos del frontend y se sirve en un contenedor Docker que utiliza `nginx` como servidor web ligero. Nginx distribuye el frontend, encargándose de gestionar la caché, lo que mejora el rendimiento, especialmente cuando hay entorno de baja latencia o rendimiento.

El sistema completo se orquesta utilizando **Docker Compose**, organizando los servicios en cuatro contenedores principales:

- **Frontend**: Sirve los archivos estáticos generados por React mediante Nginx.
- **Backend**: Ejecuta la API desarrollada con FastAPI, gestionando la lógica de negocio y las comunicaciones con la base de datos.
- **Base de datos (PostgreSQL)**: Sistema relacional donde se almacenan todas las entidades del sistema (usuarios, reservas, pistas, valoraciones...).
- **pgAdmin**: Herramienta visual para inspeccionar y gestionar la base de datos de forma gráfica.

El uso de **Docker** permite encapsular cada servicio en un contenedor independiente, creando una arquitectura de tipo **microservicios**, donde cada componente es autónomo, escalable y fácil de mantener. Algunas de sus principales ventajas son:

- **Reproducibilidad del entorno**: El sistema se puede ejecutar de la misma forma en diferentes entornos (desarrollo, testing, producción) sin conflictos por diferencias en librerías o versiones.
- **Aislamiento de servicios**: Cada aplicación corre en un contenedor diferente evitando interferencias y mejorando la seguridad y estabilidad.
- **Escalabilidad**: Permite escalar horizontalmente servicios concretos, como múltiples instancias del backend, sin necesidad de replicar todo el sistema.
- **Facilidad de despliegue**: Con un solo comando (`docker-compose up`), se puede levantar todo el ecosistema completo.
- **Monitorización y mantenimiento**: Integración de herramientas (como pgAdmin) en contenedores adicionales para administrar otras tareas.

Este sistema resulta especialmente adecuado para entornos modernos gracias a su diseño modular y desacoplado. Al separar cada componente funcional en un contenedor independiente, se logra una arquitectura robusta, escalable y fácilmente mantenible.

Por ejemplo, si en el futuro se quisiera incorporar un nuevo servicio -como un sistema de analítica, un panel administrativo avanzado, un módulo de pagos o incluso un microservicio externo especializado— no sería necesario reestructurar toda la aplicación. Bastaría con definir un nuevo contenedor en el archivo `docker-compose.yml` e integrarlo mediante la red interna entre servicios.

Este enfoque permite evolucionar el sistema de forma controlada y segura, minimizando los riesgos al modificar componentes existentes y facilitando tanto el desarrollo en paralelo como el despliegue continuo. Además, este modelo favorece la escalabilidad horizontal: si uno de los servicios necesita mayor rendimiento, se puede escalar de manera aislada sin afectar al resto de la infraestructura.

—

Capítulo 5

Implementación

5.1. Introducción

Tras definir la arquitectura y el diseño del sistema, se procedió a la fase de implementación, en la que se materializaron los distintos componentes del proyecto. Esta etapa comprendió el desarrollo del backend, la creación de la interfaz de usuario en el frontend, la configuración de la base de datos, la integración de servicios externos y el despliegue del sistema utilizando contenedores Docker.

La implementación se realizó de manera gradual, partiendo de las funcionalidades más esenciales para después ir incorporando mejoras y características más complejas. Se ha utilizado una metodología ágil, que ha permitido un avance rápido y preciso, con pruebas constantes y los ajustes oportunos cuando ha sido necesario.

El backend, desarrollado con FastAPI, se centró en la creación de una API REST robusta, segura y eficiente, incluyendo la gestión de usuarios, reservas, pistas y valoraciones, además de integrar servicios externos como la API meteorológica.

El frontend, basado en React, se estructuró en componentes reutilizables, gestionando el estado de la aplicación mediante Redux Toolkit y automatizando las llamadas a la API mediante RTK Query. Se prestó especial atención a la experiencia del usuario, garantizando una navegación fluida y un diseño responsivo.

En paralelo, se configuró PostgreSQL como sistema de base de datos relacional, asegurando la persistencia de los datos y la integridad referencial. Para facilitar el acceso y la administración de la base de datos, se incorporó la herramienta pgAdmin.

Finalmente, todos los servicios se encapsularon en contenedores utilizando Docker y se orquestaron mediante Docker Compose, garantizando un entorno de despliegue homogéneo, portátil y escalable.

En los siguientes apartados se describen en detalle las fases de implementación de cada uno de los componentes principales del sistema, así como las pruebas realizadas para validar su correcto funcionamiento.

5.2. Backend en Python (FastAPI)

El backend de la aplicación ha sido desarrollado empleando **FastAPI**, un framework moderno y de alto rendimiento para la creación de APIs en Python. Esta sección describe la estructura general del servidor, su comunicación con la base de datos y algunos aspectos destacados de su implementación.

5.2.1. Estructura del backend en Python (FastAPI)

Antes de explicar la implementación detallada del backend, se presenta la organización de ficheros y directorios del proyecto. En la Fig. 5.1, se muestra la estructura de carpetas que conforman la aplicación.

Todo el backend se estructura dentro de un directorio principal llamado `app/`, adaptando un diseño modular para facilitar la organización y el mantenimiento del sistema, como vemos en las siguientes carpetas:

- `config/`: Contiene los ficheros de configuración general, como claves secretas y parámetros relacionados con la autenticación.
- `db/`: Gestiona la conexión a la base de datos y define el motor SQLAlchemy.
- `dependencies/`: Incluye dependencias reutilizables como autenticación y gestión de sesiones.
- `models/`: Define las clases ORM que representan las tablas de la base de datos.
- `routers/`: Contiene los módulos de rutas agrupados por funcionalidad.
- `schemas/`: Define los esquemas Pydantic para validación y serialización.
- `services/`: Contiene servicios auxiliares como envío de correos, planificación de tareas y consultas externas.
- `initialize_db.py`: Script para inicializar la base de datos con datos precargados.
- `main.py`: Archivo principal y punto de entrada de la aplicación.

Esta estructura sigue principios de separación de responsabilidades, facilitando el mantenimiento y las futuras extensiones.

5.2.2. Archivo principal (`main.py`)

El archivo `main.py` es el núcleo del backend de la aplicación, donde se define y configura la instancia principal del servidor. Este archivo contiene todas las piezas clave para levantar el backend, gestionar las rutas y lanzar tareas de fondo. A continuación, se explica cada sección referenciada por línea de código.

Importaciones necesarias: En las líneas 1–8 del Código 5.1 se importan los módulos esenciales: `FastAPI` para crear el servidor, `engine` y `Base` para gestionar la base de datos con SQLAlchemy, todos los routers funcionales desde el directorio `routers/`, la configuración de middleware CORS, los modelos de usuario, la función para inicializar datos (`create_initial_data`) y el programador asíncrono de recordatorios (`reminder_scheduler`).

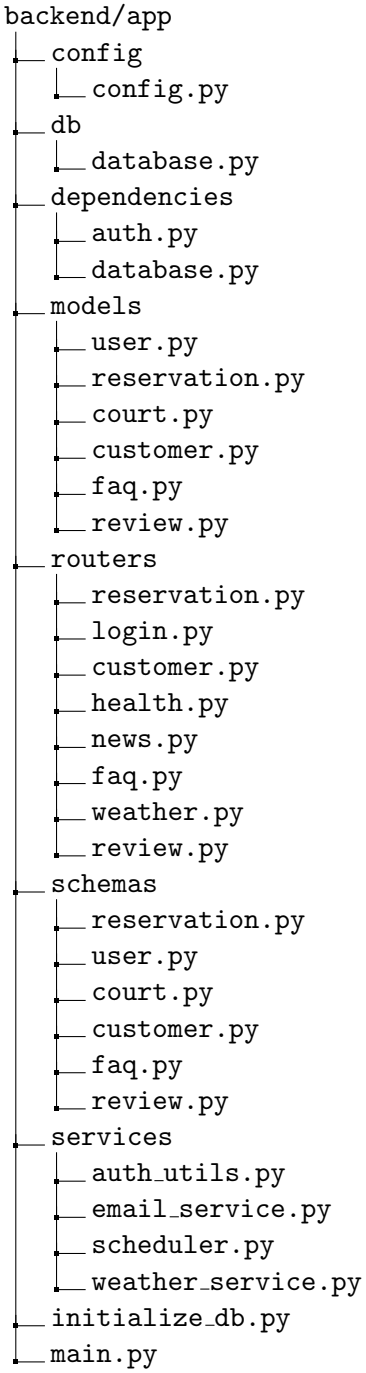

Fig. 5.1: Estructura de directorios del backend desarrollado en Python con FastAPI.

Creación de la instancia FastAPI: En la línea 10 del Código 5.1 se instancia la aplicación principal mediante `app = FastAPI()`, creando el objeto sobre el que se montarán todas las rutas, middlewares y eventos.

Configuración de CORS (Cross-Origin Resource Sharing): Entre las líneas 13–24 del Código 5.1 se configuran los orígenes permitidos (`origins`), permitiendo al frontend (que puede estar alojado en otro puerto o dominio) comunicarse con el backend sin problemas de seguridad. El middleware `CORSMiddleware` autoriza el uso de credenciales, todos los métodos (`GET`, `POST`, etc.) y todas las cabeceras.

Evento `startup`: En las líneas 26–32 del Código 5.1, el decorador `@app.on_event("startup")` define una función que se ejecuta automáticamente al iniciar el servidor. Esta función:

- Crea todas las tablas necesarias en la base de datos con `Base.metadata.create_all(bind=engine)`. Aquí, `Base.metadata` recopila toda la información de los modelos ORM definidos en el directorio `models/` y genera automáticamente las tablas correspondientes en la base de datos especificada en el `engine`. Esto asegura que la estructura de la base de datos esté sincronizada con el modelo de datos de la aplicación sin necesidad de crear las tablas manualmente.
- Inicializa los datos base ejecutando `create_initial_data()`;
- Lanza en segundo plano (`asyncio.create_task()`) el `reminder_scheduler`. Esta función se ejecuta en un hilo independiente como una tarea asíncrona no bloqueante, lo que significa que corre en paralelo al procesamiento normal de peticiones HTTP.

Registro de routers: En las líneas 34–43 del Código 5.1 se agregan los distintos módulos de rutas (`login`, `reservation`, `customer`, etc.) usando `app.include_router()`. Esto permite organizar las rutas por responsabilidad, facilitando el mantenimiento y la escalabilidad del proyecto.

```
1  from fastapi import FastAPI
2  from db.database import engine, Base  # utilizado para interactuar con la base
       de datos
3  from routers import reservation, login, customer, health, news, faq, weather,
       review
4  from fastapi.middleware.cors import CORSMiddleware
5  from models.user import User  # Asegúrate de importar los modelos
6  from initialize_db import create_initial_data
7  from services.scheduler import reminder_scheduler  # Importamos el scheduler
8  import asyncio
9
10 app = FastAPI()
11
12 # Configurar los orígenes permitidos
13 origins = [
14     "http://localhost:8081",
15     "http://127.0.0.1:8081",
16     "http://localhost:3000",
17     "http://127.0.0.1:3000",
18 ]
19
20 # Habilita CORS para todas las rutas y métodos
21 app.add_middleware(
22     CORSMiddleware,
23     allow_origins=origins,  # Orígenes permitidos
24     allow_credentials=True,  # Si usas cookies o autenticación
25     allow_methods=["*"],  # Métodos permitidos (GET, POST, etc.)
26     allow_headers=["*"],  # Cabeceras permitidas
27 )
28
29 @app.on_event("startup")
30 async def create_tables():
31     Base.metadata.create_all(bind=engine)  # Crea las tablas definidas en los
           modelos
32     create_initial_data()
33     print("Servidor iniciado. Lanzando el recordatorio en segundo plano...")
```

```
34      asyncio.create_task(reminder_scheduler())
35
36  app.include_router(login.router, tags=["login"])
37  app.include_router(reservation.router, tags=["reservation"])
38  app.include_router(customer.router, tags=["customer"])
39  app.include_router(health.router, tags=["health"])
40  app.include_router(news.router, tags=["news"])
41  app.include_router(faq.router, tags=["FAQs"])
42  app.include_router(weather.router, tags=["weather"])
43  app.include_router(review.router, tags=["review"])
```

Código 5.1: Archivo principal del backend: main.py

5.2.3. Routers y organización modular

El backend sigue una arquitectura modular que divide las responsabilidades en diferentes routers, agrupando los endpoints según su funcionalidad. Esto permite mantener el código organizado, escalable y fácil de mantener.

Cada router está definido en el directorio `routers/`, donde se encuentran módulos como `login.py`, `reservation.py`, `customer.py`, entre otros. Dentro de cada archivo, se define un objeto `APIRouter` de FastAPI como podemos ver en la línea 4 del Código 5.2, al que se le añaden las rutas específicas con sus respectivos métodos HTTP como vemos en la línea 6 del Código 5.2 y lógica de negocio.

```
1  from fastapi import APIRouter, Depends, HTTPException, Query, BackgroundTasks
2  from schemas.reservation import ReservationResponse
3
4  router = APIRouter()
5
6  @router.post("/reservations/", response_model=ReservationResponse)
7  def create_reservation(background_tasks: BackgroundTasks, reservation:
       ReservationCreate, db: Session = Depends(get_db)):
8  ...
```

Código 5.2: Archivo dentro del /routers del backend: reservation.py

En el archivo `main.py`, todos estos routers son registrados en la aplicación principal mediante la función `app.include_router()`, lo que permite que la aplicación detecte y active automáticamente todas las rutas al iniciarse. Esta técnica permite mantener el código desacoplado y favorece la reutilización y la expansión futura del backend.

A continuación, se resumen los routers principales utilizados en este proyecto:

- `login`: Maneja la autenticación de usuarios y la generación de tokens JWT.
- `reservation`: Gestiona la creación, consulta y eliminación de reservas.
- `customer`: Administra la información de clubes y pistas.
- `health`: Proporciona un endpoint de comprobación de estado del servidor.
- `news`: Permite obtener noticias relacionadas con el pádel.
- `faq`: Gestiona las preguntas frecuentes (FAQs) mostradas al usuario.

- `weather`: Integra las consultas a la API meteorológica.
- `review`: Permite la gestión y consulta de valoraciones de pistas.

En el Código 5.1 (líneas 36–43) se puede observar cómo se utiliza `app.include_router()` para incluir todos los routers en la aplicación principal. De esta manera podemos garantizar que todas estas rutas estén activas y listas para recibir peticiones cuando se despliega el servidor.

5.2.4. Estructura de las rutas y lógica del backend

La lógica del backend se organiza siguiendo el patrón REST, lo que significa que cada recurso del sistema (usuarios, reservas, pistas, clientes, valoraciones, etc.) tiene un conjunto de rutas o *endpoints* bien definidos para realizar operaciones CRUD (crear, leer, actualizar, eliminar).

Cada módulo dentro del directorio `routers/` define estas rutas usando decoradores de FastAPI como `@router.get()`, `@router.post()`, `@router.put()` y `@router.delete()`. Estas funciones manejan las peticiones entrantes, realizan validaciones básicas y delegan la lógica de negocio más compleja a los servicios ubicados en el directorio `services/`.

En el Código 5.2 podemos ver cómo la ruta `/reservations/` definida con `@router.post()`, que permite crear una nueva reserva, verifica la existencia del usuario, cliente y pista antes de realizar ninguna operación de reserva. Tras esto, comprueba la disponibilidad de la pista y si está libre, hará la reserva y enviará un correo de confirmación al usuario.

La validación de los datos de entrada se realiza utilizando los esquemas de Pydantic definidos en el directorio `schemas/`, que aseguran que las peticiones cumplen con el formato y los tipos de datos esperados.

En resumen, la lógica de cada endpoint se puede entender como una cadena de pasos:

- Recepción y validación de datos usando `schemas/`.
- Acceso a la base de datos usando SQLAlchemy a través de `models/`.
- Ejecución de lógica de negocio (por ejemplo, cálculos, validaciones complejas) usando `services/`.
- Preparación y envío de la respuesta al cliente.

Esta organización clara permite mantener el código limpio, desacoplado y fácil de extender en el futuro.

5.2.5. Sistema de autenticación y protección con tokens

El backend implementa un sistema de autenticación basado en **JSON Web Tokens (JWT)** mediante la librería `fastapi-jwt-auth`. Esto garantiza que solo los usuarios autenticados puedan acceder a los recursos protegidos de la API, y que ciertas operaciones estén restringidas según el rol del usuario.

Proceso de autenticación y generación de tokens

La autenticación se gestiona en el endpoint `/login`, definido en `routers/login.py`. Cuando un usuario envía sus credenciales (nombre de usuario y contraseña), ocurre lo siguiente:

- Primero, el backend busca al usuario en la base de datos usando SQLAlchemy.
- Si las credenciales son válidas, se genera un par de tokens:
  - Un `access_token`, con una validez de 8 horas, que permite acceder a los endpoints protegidos.
  - Un `refresh_token`, que permite obtener un nuevo access token cuando expire.
- El access token se crea usando `Authorize.create_access_token()`, incluyendo:
  - El identificador del usuario como `subject`.
  - El tiempo de expiración del token.
  - Un claim adicional `role`, que almacena el rol del usuario (por ejemplo, `operator` o `customer`).

La clase `Settings` configura el sistema y es donde se define la clave secreta utilizada para firmar y verificar los tokens. Esta clase extiende de `BaseSettings`, lo que habilita a FastAPI para cargar automáticamente los parámetros desde variables de entorno, el archivo `.env` o valores por defecto definidos en el código.

La integración con la librería `fastapi-jwt-auth` se realiza mediante el decorador `@AuthJWT.load_config`, que indica que la función `get_config()` proporcionará la configuración necesaria para inicializar y operar el sistema de autenticación. En concreto, al devolver la clase `Settings`, FastAPI y `fastapi-jwt-auth` obtienen el valor de `authjwt_secret_key`, que es el valor usado internamente para:

- **Firmar** los tokens generados en el servidor, garantizando que no puedan ser manipulados por el cliente.
- **Verificar** la validez y la firma de los tokens recibidos en cada petición protegida.

La clave secreta actúa como una firma digital. Cuando se genera un token, la biblioteca lo codifica (incluyendo los datos del usuario y los claims como el rol) y lo firma criptográficamente usando la clave. Cuando el servidor recibe de vuelta ese token en futuras peticiones, verifica la firma usando la misma clave secreta para asegurarse de que el token no ha sido alterado.

El siguiente bloque de código muestra cómo se define y carga esta configuración:

```
1 class Settings(BaseSettings):
2     authjwt_secret_key: str = "TU_CLAVE_SECRETA_AQUI"
3
4 @AuthJWT.load_config
5 def get_config():
6     return Settings()
```

Código 5.3: Configuración de clave secreta para JWT

De esta manera, la clase `Settings` se convierte en el punto central de configuración para toda la autenticación basada en JWT, y puede ampliarse fácilmente con otros parámetros, como tiempo de expiración, algoritmos de firma o cabeceras personalizadas.

Cuando se genera un token, la librería JWT lo codifica y firma usando esta clave secreta. El token resultante contiene información encriptada que solo puede ser verificada por el backend.

Protección de endpoints mediante dependencias

FastAPI permite proteger endpoints utilizando **dependencias**, un mecanismo que permite declarar funciones que se ejecutan automáticamente antes de que se procese la petición principal.

En este proyecto, se usa la dependencia `jwt_required`, definida en `dependencies/auth.py`, para validar los tokens. Ejemplo de uso en un endpoint:

```
@router.get('/protected-endpoint')
def protected_endpoint(
    _: None = Depends(jwt_required),  # Verifica token
    Authorize: AuthJWT = Depends()
):
```

Código 5.4: Uso de la dependencia jwt_required para proteger un endpoint

La función `jwt_required` que vemos en Código 5.5 verifica que:

- El token esté presente en la cabecera `Authorization`.
- El token no esté expirado, ni revocado.
- La firma del token sea válida usando la clave secreta configurada.

El código de validación es el siguiente:

```
def jwt_required(Authorize: AuthJWT = Depends()):
    try:
        Authorize.jwt_required()  # Verifica el token
    except (MissingTokenError, InvalidHeaderError, RevokedTokenError,
        AccessTokenRequired) as e:
        raise HTTPException(status_code=401, detail="Not authenticated
            or token missing")
    except Exception as e:
        raise HTTPException(status_code=401, detail=f"Token error: {str(
            e)}")
```

Código 5.5: Función jwt_required para verificar tokens

Este proceso garantiza que el backend:

- Valida la firma criptográfica del token con la clave secreta.
- Verifica que el token no haya sido manipulado.
- Extrae los claims del token para usarlos en la lógica de negocio.

Si el método `Authorize.jwt_required()` detecta algún problema con el token (como ausencia del token, expiración, firma inválida o manipulación), lanza automáticamente una excepción

que es capturada por el bloque `try-except`. El backend responde entonces con un error HTTP 401 (`Unauthorized`) y un mensaje descriptivo en el campo `detail`, informando al cliente de que la autenticación ha fallado. Esto garantiza que los endpoints protegidos no se ejecuten si no se cumplen las condiciones mínimas de seguridad.

Control de acceso por roles

Además de verificar que un usuario esté autenticado, el sistema permite restringir operaciones según el rol usando la función `require_role`, definida en `services/auth_utils.py`. Esta función:

- Valida el token JWT.
- Extrae el claim `role` del token.
- Verifica si el rol coincide con el requerido (por ejemplo, `operator`).

Si el rol no es correcto, lanza un error 403 (`Access forbidden`). Existen dos roles principales:

- `operator`: para usuarios administradores.
- `customer`: para usuarios clientes (clubes).

Este sistema asegura un control granular y seguro de las acciones permitidas en la plataforma.

5.2.6. Envío de correos electrónicos y recordatorios automáticos

Uno de los elementos más destacables del sistema es la automatización en la comunicación con los usuarios mediante el envío de correos electrónicos en distintos momentos del ciclo de vida de una reserva. Esto permite mejorar la experiencia del usuario, reducir el número de inasistencias y fomentar la retroalimentación mediante valoraciones.

Tecnología utilizada: FastAPI-Mail

Para la gestión de correos electrónicos se ha utilizado la librería `FastAPI-Mail`, una extensión compatible con FastAPI que permite definir plantillas, configurar servidores SMTP y enviar mensajes de forma sencilla y asíncrona. En este caso, se emplea una cuenta de Gmail configurada específicamente para este propósito. La configuración se encuentra en el archivo `services/email_service.py`, y utiliza el servidor SMTP de Gmail con autenticación mediante contraseña de aplicación.

5.2.6.1. Correo de confirmación al crear una reserva

Cuando un usuario realiza una reserva mediante el endpoint `POST /reservations/`, se verifica que el usuario, la pista y el club existen, y que la pista está disponible en la franja horaria solicitada. Si todo es correcto, se consulta el API meteorológico para almacenar la previsión en ese momento (temperatura, condición, viento y humedad). Esta información queda registrada en la base de datos junto con la reserva.

Acto seguido, se agenda una tarea en segundo plano usando `background_tasks.add_task(...)` que ejecuta la función `send_email()` como vemos en Código 5.6 para enviar un correo de confirmación. El contenido incluye los datos de la reserva y un mensaje de bienvenida personalizado.

```
async def send_email(to_email: str, court_name: str, club_name: str,
    reservation_time: str):
    email_body = f"""
    Hola,

    Tu reserva ha sido confirmada con éxito.

            **Club:** {club_name}
            **Pista:** {court_name}
        **Fecha y hora:** {reservation_time}

     Nos  vemos en la pista!

    Equipo del Club de Pádel
    """

    message = MessageSchema(
        subject="   ␣Confirmación␣de␣tu␣reserva",
        recipients=[to_email],
        body=email_body,
        subtype="plain"
    )

    fm = FastMail(conf)
    await fm.send_message(message)
    print(f"Email␣enviado␣correctamente␣a␣{to_email}")
```

Código 5.6: Función send_email() que envía el correo de confirmación.

A continuación se muestra en la figura 5.2 un ejemplo visual del correo que recibe el usuario tras completar la reserva.

5.2.6.2. Bucle de comprobación de recordatorios y correos post-reserva

Además del correo de confirmación enviado en el momento de la reserva, el sistema incluye un mecanismo automatizado que se ejecuta en segundo plano para realizar tareas periódicas relacionadas con la gestión de recordatorios y valoraciones.

Este mecanismo es un **bucle asíncrono** que empieza a correr automáticamente cuando arranca el servidor FastAPI. Esto se encuentra definido en la función `reminder_scheduler()`.

Inicialización del bucle en FastAPI

Para que el bucle comience a ejecutarse automáticamente al iniciar la aplicación, se utiliza el evento especial `@app.on_event("startup")`, proporcionado por FastAPI. Este evento permite definir lógica que debe ejecutarse una vez que el servidor ha arrancado, como por ejemplo, lanzamiento de procesos en segundo plano. En este caso, dentro del evento `startup` se lanza el bucle de recordatorios como una tarea asíncrona mediante `asyncio.create_task()`, tal y como se muestra en el código siguiente, definido en el archivo `main.py`:

Fig. 5.2: Correo de confirmación recibido tras realizar una reserva.

```
@app.on_event("startup")
async def create_tables():
    ...
    print("Servidor iniciado. Lanzando el recordatorio en segundo plano
        ...")
    asyncio.create_task(reminder_scheduler())
```

Código 5.7: Inicialización del bucle de recordatorios en FastAPI

Lógica del bucle reminder_scheduler()

El bucle ejecuta dos funciones clave cada 15 minutos: `check_and_send_reminders()` y `check_and_send_review_requests()`. El primero busca reservas cuya hora de inicio sea dentro de 24 horas, mientras que el segundo detecta reservas que han finalizado recientemente para solicitar una valoración al usuario.

```
async def reminder_scheduler():
    print("Hora actual:", datetime.now(ZoneInfo("Europe/Madrid")))
    while True:
        await check_and_send_reminders()
        await check_and_send_review_requests()
        await asyncio.sleep(900)  # 900 segundos = 15 minutos
```

Código 5.8: Bucle asíncrono para gestionar recordatorios y reviews

Recordatorio 24 horas antes de la reserva

Cada vez que se ejecuta el bucle de comprobación, se invoca la función `check_and_send_remin-`

ders() que podemos ver en el Código 5.9, cuya responsabilidad es identificar todas aquellas reservas que están programadas para comenzar en un margen muy concreto: entre 24 horas menos 15 minutos y 24 horas exactas desde el momento actual. Esta ventana temporal de 15 minutos garantiza que las reservas próximas a ese punto no se pasen por alto ni se notifiquen repetidamente si el bucle se ejecuta varias veces durante el día.

Una vez detectadas las reservas dentro de ese intervalo, la función accede a la base de datos para recuperar los datos del usuario que realizó la reserva, así como de la pista y el club asociados. Posteriormente, se realiza una nueva consulta meteorológica utilizando la función `get_weather_for_reservation()`, que devuelve la previsión más reciente para la ubicación y hora de la reserva.

El objetivo principal es comparar esta nueva previsión con los datos que fueron almacenados en el momento en que se creó la reserva. Si se detectan cambios significativos en la temperatura (más de 1°C), la velocidad del viento (más de 5 km/h), la humedad (más de un 10 %) o en el estado general del cielo (por ejemplo, de "Despejado" a "Lluvia ligera"), se genera un mensaje informativo que resalta estas diferencias. En caso contrario, se notifica al usuario que las condiciones se mantienen estables, incluyendo los valores originales como recordatorio.

Finalmente, toda esta información se envía al usuario mediante un correo electrónico utilizando la función `send_reminder_email()`, que presenta los datos de forma clara y estructurada, ayudando al usuario a tomar decisiones informadas, como reprogramar o cancelar la reserva si el tiempo ha empeorado.

```
async def check_and_send_reminders():
    db: Session = SessionLocal()
    now = datetime.now(ZoneInfo("Europe/Madrid"))
    reminder_window_start = now + timedelta(hours=24) - timedelta(
        minutes=15)
    reminder_window_end = now + timedelta(hours=24)

    reservations = db.query(Reservation).filter(
        Reservation.reservation_time >= reminder_window_start,
        Reservation.reservation_time <= reminder_window_end
    ).all()

    for reservation in reservations:
        user = db.query(User).filter(User.id == reservation.user_id).
            first()
        court = db.query(Court).filter(Court.court_id == reservation.
            court_id).first()
        customer = db.query(Customer).filter(Customer.id == reservation.
            customer_id).first()

        if user and court and customer:
            new_weather = get_weather_for_reservation(
                court.latitude, court.longitude, reservation.
                    reservation_time
            )

            if new_weather:
                changes = []

                if reservation.weather_temp_c is not None and abs(
                    reservation.weather_temp_c - new_weather.get("temp_c"
```

```
                    , 0)) > 1:
26                      changes.append(f"- Temperatura: antes {reservation.
                            weather_temp_c} C , ahora {new_weather['temp_c']}
                             C ")
27
28                  if reservation.weather_condition_text and reservation.
                        weather_condition_text != new_weather.get("condition"
                        , {}).get("text"):
29                      changes.append(f"- Condición: antes '{reservation.
                            weather_condition_text}', ahora '{new_weather['
                            condition']['text']}'")
30
31                  if reservation.weather_wind_kph is not None and abs(
                        reservation.weather_wind_kph - new_weather.get("
                        wind_kph", 0)) > 5:
32                      changes.append(f"- Viento: antes {reservation.
                            weather_wind_kph} km/h, ahora {new_weather['
                            wind_kph']} km/h")
33
34                  if reservation.weather_humidity is not None and abs(
                        reservation.weather_humidity - new_weather.get("
                        humidity", 0)) > 10:
35                      changes.append(f"- Humedad: antes {reservation.
                            weather_humidity}%, ahora {new_weather['humidity
                            ']}%")
36
37                  if changes:
38                      weather_message = "      La previsión meteorológica
                            ha cambiado:\n" + "\n".join(changes)
39                  else:
40                      weather_message = (
41                          "    La previsión se mantiene similar:\n"
42                          f"- Condición: {reservation.
                                weather_condition_text}\n"
43                          f"- Temperatura: {reservation.weather_temp_c} C
                                \n"
44                          f"- Viento: {reservation.weather_wind_kph} km/h\
                                n"
45                          f"- Humedad: {reservation.weather_humidity}%"
46                      )
47              else:
48                  weather_message = "      No se pudo obtener la previsió
                        n del tiempo."
49
50              await send_reminder_email(
51                  user.email,
52                  court.name,
53                  customer.name,
54                  reservation.reservation_time.strftime("%d/%m/%Y %H:%M"),
55                  weather_message
56              )
57
58      db.close()
```

Código 5.9: Función check_and_send_reminders() que detecta reservas a 24h y envía recordatorios.

Una vez construido el contenido del mensaje con la información meteorológica correspondiente, el sistema utiliza la función `send_reminder_email()` para componer y enviar el correo al usuario. Esta función se encuentra definida en `services/email_service.py` y está diseñada para ser completamente asíncrona, haciendo uso de la librería `FastAPI-Mail`.

El cuerpo del mensaje contiene de forma clara los datos esenciales de la reserva: el nombre del club, la pista reservada y la fecha y hora. A continuación, se incorpora el mensaje meteorológico generado anteriormente, que puede ser una advertencia sobre cambios relevantes en la previsión o bien una confirmación de que las condiciones se mantienen similares a las inicialmente previstas. Este mensaje se incluye dinámicamente en el cuerpo del correo gracias al parámetro `weather_message`.

A nivel técnico, la función construye un objeto `MessageSchema` con el asunto, el destinatario y el contenido del mensaje en texto plano. Después, mediante una instancia de `FastMail` configurada con las credenciales SMTP, se envía el correo de forma asíncrona. En el Código 5.10 se muestra la implementación de la función encargada de este proceso.

```
async def send_reminder_email(to_email: str, court_name: str, club_name:
    str, reservation_time: str, weather_message: str):
    email_body = f"""
        **Recordatorio de tu reserva**

           **Club:** {club_name}
           **Pista:** {court_name}
        **Fecha y hora:** {reservation_time}

    {weather_message.strip()}

    No olvides tu cita.  Te  esperamos en la pista!

    Equipo del Club de Pádel
    """

    message = MessageSchema(
        subject="      Recordatorio: Tu reserva de pádel",
        recipients=[to_email],
        body=email_body,
        subtype="plain"
    )

    fm = FastMail(conf)
    await fm.send_message(message)
    print(f"      Recordatorio enviado a {to_email}")
```

Código 5.10: Función send_reminder_email() que envía el recordatorio con previsión meteorológica actualizada.

A continuación se muestra en la figura 5.3 un ejemplo visual del correo que recibe el usuario 24 horas antes de su reserva, incluyendo la previsión meteorológica actualizada.

Este sistema permite al usuario mantenerse actualizado sobre las condiciones meteorológicas esperadas en el momento de su reserva y mejora significativamente la experiencia general, aportando un valor añadido respecto a otras plataformas de reservas deportivas.

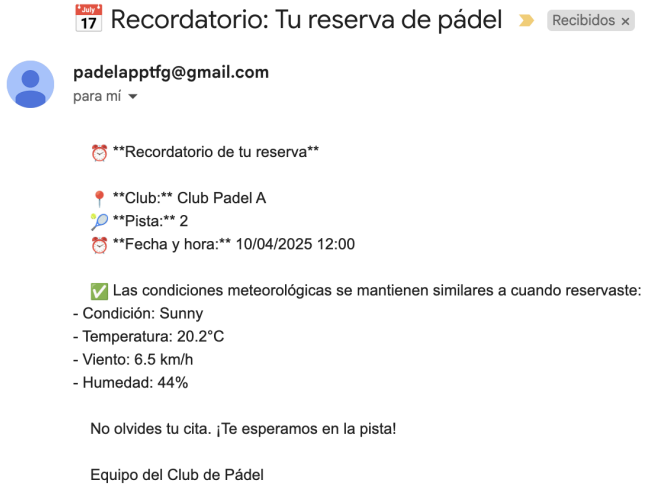

Fig. 5.3: Correo de recordatorio enviado 24 horas antes de la reserva con información meteorológica actualizada.

Correo de agradecimiento y solicitud de review

Una vez que el sistema detecta que una reserva ha finalizado recientemente, se activa un mecanismo automático de agradecimiento y solicitud de valoración. Esta funcionalidad está implementada en la función `check_and_send_review_requests()`, cuyo objetivo es identificar aquellas reservas que hayan terminado hace menos de 15 minutos. Para ello, se calcula el momento final de cada reserva como 90 minutos después de su hora de inicio, y se compara con el instante actual para determinar si está dentro de esa ventana temporal. Si la condición se cumple, se procede a recuperar los datos del usuario, la pista y el club asociados a la reserva.

Una vez obtenida esta información, se genera un enlace personalizado que redirige al usuario a la interfaz de valoraciones del sistema (`members-area/review`) e incluye en la URL los identificadores de la reserva y de la pista. Esto permite que el frontend pueda renderizar automáticamente la vista adecuada sin necesidad de autenticación adicional ni pasos intermedios.

A continuación, el sistema invoca la función `send_thank_you_email()`, encargada de construir y enviar el mensaje de correo. En el Código 5.11 se muestra la lógica encargada de identificar las reservas finalizadas y disparar la notificación correspondiente.

```
1  async def check_and_send_review_requests():
2      db: Session = SessionLocal()
3      now = datetime.now(ZoneInfo("Europe/Madrid"))
4      review_window_start = now - timedelta(minutes=15)
5
6      all_reservations = db.query(Reservation).all()
7
8      for reservation in all_reservations:
9          start_time = reservation.reservation_time.replace(tzinfo=
               ZoneInfo("Europe/Madrid"))
10         end_time = start_time + timedelta(minutes=90)
11
12         if review_window_start <= end_time <= now:
13             user = db.query(User).filter(User.id == reservation.user_id)
                   .first()
```

```
            court = db.query(Court).filter(Court.court_id == reservation
                .court_id).first()
            customer = db.query(Customer).filter(Customer.id ==
                reservation.customer_id).first()

            if user and court and customer:
                review_link = f"http://localhost:3000/members-area/
                    review?reservation_id={reservation.id}&court_id={
                    court.court_id}"

                await send_thank_you_email(
                    user.email,
                    user.name,
                    court.name,
                    customer.name,
                    reservation.reservation_time.strftime("%d/%m/%Y␣%H:%
                        M"),
                    review_link
                )

    db.close()
```

Código 5.11: Función check_and_send_review_requests() que detecta reservas finalizadas y envía correos de review.

La función `send_thank_you_email()` envía un correo personalizado con los datos de la reserva que acaba de terminar deseando que los jugadores hayan disfrutado y agradeciendo por la visita. En el correo se pide una valoración de la pista y se adjunta un acceso directo que llevará directamente a la página de valoraciones con los datos de la reserva ya completados para que el usuario deje su opinión y sirva para mejorar el servicio.

Este correo también se construye utilizando la librería `FastAPI-Mail` y se envía de manera asíncrona para no bloquear el flujo del sistema. Su implementación se muestra en el Código 5.12.

```
async def send_thank_you_email(to_email: str, user_name: str, court_name
    : str, club_name: str, reservation_time: str, review_link: str):
    email_body = f"""
     Hola  {user_name}!

    Esperamos que hayas disfrutado tu reserva en el club **{club_name
        }**.

            **Pista:** {court_name}
            **Fecha y hora:** {reservation_time}

    Nos encantaría conocer tu opinión sobre la pista.
            Puedes dejar tu valoración aquí: {review_link}

      Gracias  por ayudarnos a mejorar!

    Equipo del Club de Pádel
    """

    message = MessageSchema(
        subject="      ␣ Qu é␣te␣pareció␣tu␣reserva?",
```

```
        recipients=[to_email],
        body=email_body,
        subtype="plain"
    )

    fm = FastMail(conf)
    await fm.send_message(message)
    print(f"  Email de agradecimiento y review enviado a {to_email}")
```

Código 5.12: Función send_thank_you_email() que envía el correo de agradecimiento y valoración.

La figura 5.4 ilustra el correo que se envía automáticamente al usuario poco después de haber finalizado su reserva. Este mensaje incluye un agradecimiento personalizado y un enlace directo para dejar una valoración.

Fig. 5.4: Correo de agradecimiento y enlace para dejar una valoración de la pista tras finalizar la reserva.

Gracias a esta funcionalidad, el sistema no solo mejora la experiencia del usuario al cerrar el ciclo de la reserva con un mensaje de cortesía, sino que también promueve la participación activa de los clientes en la mejora continua del servicio, permitiendo recopilar feedback de forma estructurada y automatizada.

Este mecanismo, al estar totalmente automatizado y ejecutarse en segundo plano, mejora la experiencia del usuario y fomenta la fidelización sin necesidad de intervención humana adicional. Además, la comparación meteorológica en tiempo real proporciona una ventaja competitiva frente a plataformas que no tienen en cuenta posibles cambios en la previsión.

5.2.7. Comunicación con la base de datos mediante ORM (SQLAlchemy)

La comunicación entre el backend y la base de datos se gestiona mediante **SQLAlchemy**, una librería ampliamente utilizada en Python que proporciona un ORM (Object Relational Mapper). Este enfoque permite trabajar con los datos de la base de datos como si fueran objetos y clases de Python, evitando la necesidad de escribir consultas SQL directas. Además, SQLAlchemy ofrece control total sobre las transacciones y un sistema flexible para definir relaciones entre tablas, simplificando la implementación y el mantenimiento del backend.

La configuración principal se encuentra en `db/database.py` como vemos en Código 5.13, donde:

- Se define la URL de conexión a PostgreSQL mediante la variable `SQLALCHEMY_DATABASE_URL`.
- Se crea el motor de conexión con `create_engine()`.
- Se configura la fábrica de sesiones `SessionLocal` usando `sessionmaker`, que permite crear sesiones para comunicarse con la base de datos.
- Se define `Base`, la clase base declarativa de SQLAlchemy, de la cual heredarán todos los modelos ORM del proyecto.

```
1 engine = create_engine(SQLALCHEMY_DATABASE_URL)
2 SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=
      engine)
3 Base = declarative_base()
```

Código 5.13: Configuración de SQLAlchemy

Para gestionar la apertura y cierre de sesiones de manera automática, se define la dependencia `get_db()` en `dependencies/database.py` como podemos ver en Código 5.14, que proporciona una sesión de base de datos por cada petición:

```
1 def get_db() -> Session:
2     db = SessionLocal()
3     try:
4         yield db
5     finally:
6         db.close()
```

Código 5.14: Dependencia para obtener la sesión de base de datos

La función `get_db()` actúa como una dependencia que garantiza que cada petición HTTP disponga de su propia sesión de base de datos, evitando conflictos entre peticiones concurrentes. Cuando se llama al endpoint, FastAPI ejecuta la función `get_db()` hasta llegar a la instrucción `yield`, momento en el que entrega el objeto `db` al endpoint que lo ha solicitado como dependencia. Cuando el endpoint termina de ejecutarse (independientemente de si tuvo éxito o falló), FastAPI retoma la función después del `yield` y ejecuta el bloque `finally`, asegurándose de llamar a `db.close()` para cerrar la conexión. Este comportamiento es posible gracias al uso de generadores en Python, que permiten pausar y reanudar funciones en distintos puntos de su ejecución.

Este patrón asegura que:

- Se abre una sesión nueva por cada petición al backend.
- La sesión se cierra automáticamente al finalizar la petición, liberando recursos.
- No se producen fugas de conexiones ni bloqueos en la base de datos.

Además, al integrarse con el sistema de dependencias de FastAPI, este manejo es completamente transparente para el desarrollador, lo que simplifica la implementación y aumenta la robustez del código.

Este patrón se integra en los endpoints usando el sistema de dependencias de FastAPI (`Depends`). Por ejemplo, en la operación de eliminar reservas de tenemos en Código 5.15, se incluye el parámetro `db: Session = Depends(get_db)`, lo que garantiza que la sesión esté disponible durante la ejecución del endpoint:

```python
@router.delete("/reservations/{reservation_id}/", response_model=
    ReservationResponse)
def delete_reservation(reservation_id: int, db: Session = Depends(get_db
    )):
    reservation = db.query(Reservation).filter(Reservation.id ==
        reservation_id).first()
    if not reservation:
        raise HTTPException(status_code=404, detail="Reservation not
            found")

    db.delete(reservation)
    db.commit()
    return reservation
```

Código 5.15: Uso de la sesión en un endpoint

Gracias al uso de SQLAlchemy, no es necesario escribir consultas SQL manualmente. Por ejemplo, la instrucción `db.query(Reservation).filter(Reservation.id == reservation_id).first()` permite recuperar una reserva específica directamente como un objeto Python, sin tener que escribir una sentencia `SELECT`.

De manera similar, operaciones como `db.delete(reservation)` y `db.commit()` permiten eliminar registros y confirmar los cambios en la base de datos, respectivamente, delegando en SQLAlchemy la generación y ejecución de las sentencias SQL necesarias.

Gracias a esto se mejora la legibilidad del código, reduciendo de esta manera la probabilidad de errores y permitiendo trabajar con objetos y métodos de alto nivel en lugar de utilizar directamente cadenas SQL sin procesar.

Los modelos ORM, como `Reservation`, se definen en el directorio `models/`. Cada modelo hereda de `Base`, establece un nombre de tabla (`__tablename__`) y define columnas y relaciones:

```python
class Reservation(Base):
    __tablename__ = "reservations"

    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(Integer, ForeignKey("users.id"))
    ...
    customer = relationship("Customer")
```

Código 5.16: Modelo ORM de reservas

Gracias a esta arquitectura, el backend puede realizar operaciones complejas sobre la base de datos de forma eficiente, segura y desacoplada del lenguaje SQL.

5.2.8. Inicialización de la base de datos (`initialize_db.py`)

El archivo `initialize_db.py` permite precargar datos esenciales en la base de datos al iniciar la aplicación, garantizando que el sistema disponga de usuarios, clubes, pistas, reservas, preguntas frecuentes y valoraciones mínimas para funcionar correctamente.

Este script realiza las siguientes tareas principales:

- Verifica que las tablas estén creadas en la base de datos usando `Base.metadata.create_all`

(bind=engine).

- Comprueba si ya existen datos previos para evitar duplicados.
- Inserta registros por defecto para entidades clave como `Customer`, `User`, `Court`, `Reservation`, `Faq` y `CourtReview`.
- Realiza commits intermedios tras cada bloque de datos para garantizar la persistencia.

El proceso sigue un patrón general como podemos ver en Código 5.17 que repetirá para el resto de entidades:

1. Abre una sesión de base de datos (`db = SessionLocal()`).
2. Itera sobre una lista de datos predefinidos.
3. Consulta si el registro ya existe con `db.query(...).filter(...).first()`.
4. Si no existe, lo crea usando `db.add(...)`.
5. Finalmente, ejecuta `db.commit()` para guardar los cambios.
6. Cierra la sesión con `db.close()`.

```
1
2  def create_initial_data():
3      db = SessionLocal()
4      try:
5          Base.metadata.create_all(bind=engine)  # Asegúrate de que todas
               las tablas estén creadas
6
7          # Crear Customers (Clientes/Clubs)
8          customers_data = [
9              {"name": "Club␣Padel␣A", "location": "Valladolid", "
                   contact_email": "clubpadelA@padel.com", "phone": "
                   123456789"},
10             {"name": "Club␣Padel␣B", "location": "Palencia", "
                   contact_email": "clubpadelB@padel.com", "phone": "
                   987654321"},
11             {"name": "Club␣Padel␣C", "location": "Madrid", "
                   contact_email": "clubpadelC@padel.com", "phone": "
                   456789123"}
12         ]
13
14         customers = []
15         for customer_data in customers_data:
16             customer = db.query(Customer).filter(Customer.name ==
                   customer_data["name"]).first()
17             if not customer:
18                 customer = Customer(**customer_data)
19                 db.add(customer)
20                 customers.append(customer)
21                 print(f"Customer␣{customer_data['name']}␣created")
22             else:
23                 print(f"Customer␣{customer_data['name']}␣already␣exists"
                       )
```

```
24
25          db.commit()
```

Código 5.17: Introducción de Clubs en la base de datos

5.3. Base de datos PostgreSQL

La base de datos del proyecto se implementa usando **PostgreSQL**, un sistema de gestión de bases de datos relacional ampliamente utilizado en entornos de producción por su robustez, rendimiento y fiabilidad. En este proyecto, la base de datos se levanta directamente en un contenedor Docker usando la imagen oficial de PostgreSQL, lo que permite tener un entorno totalmente aislado, reproducible y fácil de desplegar. Además, carga una serie de datos al iniciarse, garantizando una configuración lista para usar desde el primer arranque.

Esta sección describe cómo se despliega el entorno con Docker, qué tablas y relaciones componen el modelo de datos y cómo se ha integrado `pgAdmin` para facilitar el desarrollo.

5.3.1. Despliegue automático con Docker Compose

En el archivo `docker-compose.yml`, la base de datos se define bajo el servicio `db`:

```
1  db:
2    build:
3      context: ./backend/postgres
4    container_name: postgres_db
5    volumes:
6      - postgres_data:/var/lib/postgresql/data
7    environment:
8      POSTGRES_DB: dbname
9      POSTGRES_USER: user
10     POSTGRES_PASSWORD: password
11   ports:
12     - "5432:5432"
```

Código 5.18: Definición del servicio de base de datos en docker-compose.yml

Aquí se configuran los siguientes parámetros:

- `POSTGRES_DB`: nombre de la base de datos que se creará al iniciar el contenedor.
- `POSTGRES_USER`: nombre del usuario administrador.
- `POSTGRES_PASSWORD`: contraseña del usuario administrador.
- `ports`: expone el puerto 5432 del contenedor al host, permitiendo conexiones externas.
- `volumes`: asocia un volumen persistente para guardar los datos aunque se reinicie el contenedor.

Utilizamos el Dockerfile, que vemos en el Código 5.19, y que está ubicado en `backend/postgres/Dockerfile` para el servicio `db`. Este fichero se basa en la imagen oficial de PostgreSQL 13 e

instala algunas herramientas adicionales para facilitar el trabajo si hay que depurar dentro del contenedor.

```
FROM postgres:13

RUN apt-get update && apt-get install -y nano

COPY pg_hba.conf /etc/postgresql/pg_hba.conf

CMD ["postgres"]
```

Código 5.19: Dockerfile para el servicio de base de datos PostgreSQL

Además, se incluye un archivo de configuración personalizado, `pg_hba.conf`, mostrado en el Código 5.20, que define las reglas de autenticación y acceso al servidor PostgreSQL:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     md5
host    all             all             127.0.0.1/32            md5
host    all             all             ::1/128                 md5
local   replication     all                                     md5
host    replication     all             127.0.0.1/32            md5
host    replication     all             ::1/128                 md5
host    all             all             all                     md5
```

Código 5.20: Archivo pg_hba.conf para configurar autenticación

Este archivo `pg_hba.conf` controla qué usuarios pueden conectarse a qué bases de datos, desde qué direcciones IP y utilizando qué método de autenticación. En este caso, se permite el acceso a todas las bases de datos y usuarios usando el método `md5`, que requiere contraseña, tanto para conexiones locales como remotas. Esto asegura un nivel básico de seguridad durante el desarrollo.

5.3.2. Tablas y relaciones principales

El modelo de datos, representado en la Fig. 4.1, define las entidades principales del sistema y las relaciones entre ellas. Gracias al diseño de los modelos en Python mediante SQLAlchemy, es posible generar automáticamente todas las tablas en la base de datos ejecutando el comando `Base.metadata.create_all(bind=engine)`. Este comando recorre la metadata recopilada por SQLAlchemy, que incluye las definiciones de tablas, columnas, claves primarias y foráneas, y crea las estructuras necesarias en la base de datos sin necesidad de escribir consultas SQL manualmente.

Las relaciones más destacadas del modelo son:

- **User** → **Reservation**: Relación 1:N, un usuario puede tener múltiples reservas.
- **Customer** → **Court**: Relación 1:N, un club tiene varias pistas asociadas.
- **Court** → **Reservation**: Relación 1:N, una pista puede tener múltiples reservas.
- **Reservation** → **CourtReview**: Relación 1:1, cada valoración está vinculada a una reserva específica.

- **User → CourtReview**: Relación N:1, un usuario puede dejar varias valoraciones.
- **Faq**: Tabla independiente que almacena preguntas frecuentes, sin relaciones externas.

En esta implementación, se han definido claves primarias (`PK`) en columnas como `id` o `court_id` para identificar de forma única cada registro dentro de las tablas. Esto nos permite garantizar que cada entidad (como usuarios, pistas o reservas) esté correctamente diferenciada y que no se generen duplicados en la base de datos.

Asimismo, se han configurado claves foráneas (`FK`), como `user_id`, `customer_id`, `court_id` y `reservation_id`, que enlazan las distintas tablas entre sí. Por ejemplo, en la tabla `Reservation`, la columna `user_id` actúa como clave foránea que apunta a la clave primaria `id` de la tabla `User`, garantizando que cada reserva esté vinculada a un usuario válido.

Gracias a este diseño de claves primarias y foráneas, hemos asegurado la integridad referencial del modelo, evitando inconsistencias o datos huérfanos. Además, esto nos ha permitido optimizar las consultas y facilitar la navegación entre entidades relacionadas dentro de la aplicación.

5.3.3. Uso de pgAdmin para desarrollo y pruebas

Para facilitar la administración y el monitoreo de la base de datos durante el desarrollo, se incorpora `pgAdmin`, un cliente web que permite:

- Navegar por las tablas, esquemas y datos almacenados.
- Ejecutar consultas SQL para inspección o pruebas.
- Crear, modificar y eliminar tablas, vistas o índices de forma visual.
- Revisar estadísticas y logs del servidor PostgreSQL.

Además, este servicio se despliega directamente desde el archivo `docker-compose.yml`, como se muestra en el Código 5.21. Se utiliza la imagen oficial `dpage/pgadmin4`, a la que se le pasan variables de entorno para definir el usuario y la contraseña iniciales (`PGADMIN_DEFAULT_EMAIL` y `PGADMIN_DEFAULT_PASSWORD`). El servicio queda accesible en el puerto `8083` del host y está configurado para depender del contenedor de base de datos `db`, asegurando que la base de datos esté levantada antes de iniciar pgAdmin.

```
pgadmin:
    image: dpage/pgadmin4
    container_name: my_pgadmin
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: admin
    ports:
      - "8083:80"
    depends_on:
      - db
```

Código 5.21: Definición del servicio pgAdmin en docker-compose.yml

5.4. Frontend en React

El frontend de la aplicación se ha desarrollado utilizando **React**, una de las bibliotecas más populares para la construcción de interfaces web interactivas y escalables. Esta sección describe cómo se ha estructurado la aplicación, los principales módulos empleados y cómo se integran entre sí para proporcionar una experiencia de usuario fluida. Cuando se despliega el frontend, lo primero que aparece al acceder es un control de acceso mediante sistema de autenticación por login, que permite que solo los usuarios registrados puedan acceder a las funcionalidades internas. Según el usuario que inicie sesión, el sistema distinguirá entre uno de los roles existentes: **administradores (operator)** y **clientes (customer)**.

Los clientes acceden a una interfaz optimizada para la reserva de pistas y la consulta de información, presentada en un tema claro. Por su parte, los administradores disponen de una interfaz con un tema oscuro y acceso a paneles adicionales de gestión. Estos paneles permiten administrar clubes, usuarios, pistas y contenidos relevantes del sistema.

Además, los administradores cuentan con la capacidad de acceder a la vista interna de cada club, lo que les permite gestionar sus configuraciones, pistas, reservas y usuarios asociados. Aunque esta vista se asemeja a la de los clientes, mantiene el tema oscuro y ofrece opciones adicionales de personalización y control, facilitando así las tareas de administración sin necesidad de abandonar su contexto administrativo.

5.4.1. Estructura general y punto de entrada

El punto de entrada principal del frontend está definido en el archivo `index.js`, como vemos en el Código 5.22, donde se configura e inicializa toda la aplicación React. El proceso comienza con la llamada a `ReactDOM.render`, que es la función encargada de renderizar la estructura de componentes React dentro de un elemento del DOM (Document Object Model), que representa la estructura jerárquica de nodos HTML en la página. En este caso, el árbol de React se inserta en el elemento con `id=root"` en el archivo `index.html`, permitiendo montar la interfaz de usuario completa en el navegador.

A continuación, se describe brevemente cada uno de los elementos que componen este árbol:

- **Provider (Redux)**: Permite envolver toda la aplicación con el proveedor de Redux, proporcionando acceso al estado global desde cualquier componente. `Redux` es una librería para la gestión del estado de la aplicación que permite centralizar y controlar de forma predecible los datos compartidos entre componentes. Al usar `Provider`, aseguramos que todos los componentes descendientes puedan conectarse fácilmente al estado global y reaccionar a los cambios de datos, lo que facilita la escalabilidad y el mantenimiento de la aplicación.
- **TranslationProvider**: Habilita la internacionalización y carga las traducciones necesarias para permitidor cambiar el idioma a cualquier de los soportados.
- **BrowserRouter**: Define el enrutador basado en el historial del navegador, gestionando las rutas y permitiendo la navegación entre vistas. Este componente pertenece a la librería `react-router-dom` y permite que la aplicación sea una SPA (Single Page Application) manejando las transiciones de página sin recargar el navegador. En este caso, se le pasa la propiedad `basename={process.env.PUBLIC_URL}`, lo que permite especificar un prefijo común para todas las rutas de la aplicación (en nuestro caso, simplemente /").

- **MUIsticaProvider**: Establece los estilos y el tema visual personalizado de la aplicación, usando la biblioteca Material-UI (MUI), que es un popular conjunto de componentes de interfaz de usuario para React basado en las directrices de diseño de Google Material Design. Dentro de este proveedor, destaca el uso de `ThemeProvider`, que aplica un tema dinámico basado en el rol del usuario (como `customer` u `operator`), y `CssBaseline`, que normaliza los estilos por defecto del navegador para garantizar una apariencia uniforme en todos los navegadores.

- **SnackbarProvider**: Es un componente que nos permite gestionar las notificaciones emergentes tipo snackbar, permitiendo mostrar mensajes breves y contextuales al usuario (confirmaciones, advertencias o errores).

- **ErrorBoundary**: Implementa un límite de errores que envuelve componentes hijos y captura cualquier fallo inesperado que ocurra durante el renderizado o en los métodos del ciclo de vida. Esto evita que errores locales provoquen el colapso completo de la aplicación, mostrando en su lugar un mensaje de error controlado para mejorar la resiliencia y experiencia de usuario.

- **ScrollToTop**: Es un componente utilitario que fuerza el scroll al inicio de la página cada vez que se cambia de ruta en la aplicación. Esto garantiza una navegación coherente, evitando que el usuario quede posicionado en medio o al final de una página anterior al saltar a una nueva vista.

- **Routes / App**: Define la configuración de rutas y renderiza el componente principal `App`, que contiene todas las vistas y páginas de la aplicación. El componente `Routes` de React Router se encarga de gestionar el enrutamiento declarativo dentro de la aplicación, mientras que `Route` especifica qué componente debe renderizarse para un determinado patrón de ruta. En este caso, la ruta `/*"` actúa como un comodín que captura todas las rutas definidas internamente dentro de `App`, permitiendo que dicho componente gestione la estructura y distribución de las diferentes páginas de la plataforma. Este enfoque organiza el enrutamiento en capas, manteniendo limpio y modular el archivo `index.js`.

- **HealthCheck**: Ejecuta comprobaciones de salud del backend enviando peticiones periodicas para asegurarse que tenemos conexión con él, mostrando alertas si el servidor no está disponible.

Esta estructura modular permite mantener la aplicación organizada, además de facilitar la escalabilidad y el mantenimiento, con responsabilidades bien separadas entre cada proveedor o componente.

```
ReactDOM.render(
  <Provider store={store}>
    <TranslationProvider>
        <BrowserRouter basename={process.env.PUBLIC_URL}>
          <MUIsticaProvider>
            <SnackbarProvider>
              <ErrorBoundary>
                <ScrollToTop />
                <Routes>
                  <Route element={<App />} path="/*" />
                </Routes>
              </ErrorBoundary>
              <HealthCheck />
```

```
14              </SnackbarProvider>
15            </MUIsticaProvider>
16          </BrowserRouter>
17      </TranslationProvider>
18    </Provider>,
19    document.getElementById("root")
20  );
```

Código 5.22: Archivo principal del frontend: index.js

5.4.2. Componente principal `App`

El componente `App` actúa como el núcleo del frontend, definiendo la estructura principal de rutas, vistas y reglas de acceso. Este componente es el encargado de renderizar las diferentes páginas según la ruta actual y el rol del usuario, asegurando que la navegación y la seguridad estén correctamente gestionadas.

El `App` utiliza el componente `Routes` de React Router para configurar todas las rutas disponibles en la aplicación, incluyendo rutas públicas, privadas, de cliente, de administrador y de administrador operando como cliente. Cada ruta está asociada a un componente React específico que se carga de forma perezosa (`lazy loading`) para optimizar el rendimiento.

El árbol de rutas está envuelto por:

- `RouterLayout`: Define la estructura visual común para todas las páginas.
- `RequireAuth`: Protege las rutas que requieren autenticación, verificando el rol del usuario y bloqueando el acceso no autorizado.
- `RedirectWhenLoggedIn`: Redirige a los usuarios ya autenticados lejos de las páginas públicas como el login.
- `AsCustomerLayout`: Ofrece a los administradores la posibilidad de gestionar clubes concretos mediante una vista similar a la del cliente, pero dedicada a tareas de administración.

Además, se utiliza `useSelector` de Redux para obtener el rol del usuario logueado (`customer` u `operator`) y `useMemo` para decidir dinámicamente qué dashboard renderizar al inicio. Por ejemplo, un cliente será dirigido al `CustomerDashboard` y un operador al `OperatorDashboard`.

Cada componente es envuelto en un `Suspense` con la función `withSuspensor` de forma que mostrará un indicador de carga mientras los componentes se cargan dinámicamente. Esto mejora la experiencia de usuario al evitar bloqueos con pantallas en blanco durante la carga.

Finalmente, el conjunto de rutas está organizado en bloques según el rol:

- Rutas públicas: login.
- Rutas para clientes: reservas, noticias, FAQs, coaching, área de miembros, etc.
- Rutas para operadores: gestión de usuarios, clubes y reservas.
- Rutas de operador como cliente: dashboard, reservas y módulos internos del club.

En resumen, el componente `App` centraliza la configuración de navegación y acceso, garantizando una arquitectura limpia, modular y fácil de mantener, fundamental para la escalabilidad del frontend.

5.4.3. Componente `App` y gestión avanzada de rutas

El componente `App` es el núcleo de la aplicación React, encargado de definir la navegación, la carga de páginas y el control de acceso según los roles de usuario. Este componente ha sido diseñado cuidadosamente para garantizar modularidad, rendimiento y seguridad.

Carga diferida (*lazy loading*): En primer lugar, los componentes de página se importan usando `React.lazy()` como se puede ver en Código 5.23. Esto permite dividir el código en bloques separados que se cargan dinámicamente cuando se necesitan, en lugar de incluir todo el código de la aplicación en un único paquete inicial. Gracias a esto, se mejora el tiempo de carga inicial y se optimiza la experiencia del usuario.

```
1 const Faqs = lazy(() => import("pages/Faqs"));
2 const OperatorCustomerAdd = lazy(() => import("pages/OperatorCustomerAdd
    "));
```

Código 5.23: Importación de componente con lazy

Obtención del rol de usuario: A continuación, se obtiene el rol del usuario autenticado usando `useSelector`, un hook proporcionado por Redux para acceder al estado global de la aplicación. Esto permite determinar qué vistas y permisos corresponden a ese usuario:

```
1 const { role } = useSelector(getMe);
```

Código 5.24: Obtención del rol del usuario autenticado

Selección dinámica del dashboard: Se usa el hook `useMemo` para seleccionar dinámicamente qué componente debe actuar como página principal (`Home`) según el rol del usuario, como se muestra en el Código 5.25. El hook `useMemo` permite memorizar un valor calculado (en este caso, el componente correspondiente al dashboard) y solo lo recalcula si las dependencias (aquí, el `role`) cambian, mejorando así el rendimiento y evitando cálculos innecesarios.

```
1 const Home = useMemo(() => {
2   const homes = { operator: OperatorDashboard, customer:
       CustomerDashboard };
3   return homes[role];
4 }, [role]);
```

Código 5.25: Selección dinámica del dashboard con useMemo

Definición de las rutas principales: El árbol de navegación se define usando `<Routes>` y `<Route>` de React Router como se puede observar en el Código 5.26. La primera capa envuelve toda la aplicación en `RouterLayout`, que establece la estructura general del sitio:

```
1 <Routes>
2   <Route element={withSuspensor(<RouterLayout />)} path={ROUTES.home}>
3     ...
4   </Route>
```

```
</Routes>
```

Código 5.26: Definición de rutas anidadas utilizando React Router

Bloques internos de rutas: Dentro de `RouterLayout`, se definen bloques de rutas según su tipo:

- `RedirectWhenLoggedIn`: Si un usuario autenticado intenta acceder al login, lo redirige automáticamente.
- `RequireAuth`: Protege rutas privadas, permitiendo el acceso solo a roles autorizados como `customer` u `operator`.
- `AsCustomerLayout`: Habilita una vista especial para que los administradores accedan a las interfaces específicas de un cliente.

Además, gracias a que cada componente está envuelto con la función `withSuspensor`, se muestra un `LinearProgress`, como podemos ver en Código 5.27 mientras se carga, lo que garantiza una experiencia de usuario fluida.

```
const withSuspensor = (comp) => <Suspense fallback={<LinearProgress
    />}>{comp}</Suspense>;
```

Código 5.27: Función que envuelve componentes en Suspense con un indicador de carga LinearProgress

Uso de rutas centralizadas: El objeto `ROUTES` define de forma centralizada todos los caminos de la aplicación. Esto simplifica la gestión de rutas y evita errores por nombres duplicados o inconsistentes, permitiendo definir URLs dinámicas con parámetros como `:customerId`. Esta estrategia hace que toda la navegación sea más mantenible y clara.

En el Código 5.28 se muestra cómo se estructuran las rutas dentro del componente `App`, que organiza el acceso en bloques diferenciados según el rol del usuario:

- Bloques públicos accesibles a cualquier visitante (como `/login`).
- Bloques protegidos para clientes (`customer`) con rutas como `/faqs`.
- Bloques para administradores (`operator`) con rutas de gestión como `/operatorCustomerAdd`.
- Un bloque especial para administradores actuando como cliente (`AsCustomerLayout`), que permite acceder a vistas específicas del cliente pero manteniendo el contexto de operador.

```
<Routes>
  <Route element={withSuspensor(<RouterLayout />)} path={ROUTES.home}>
    <Route element={<RedirectWhenLoggedIn />}>
      <Route element={withSuspensor(<Login />)} path={ROUTES.login} />
    </Route>
    <Route element={<RequireAuth roles={["customer"]} />}>
      <Route element={withSuspensor(<Faqs />)} path={ROUTES.faqs} />
    </Route>
    <Route element={<RequireAuth roles={["operator"]} />}>
      <Route element={withSuspensor(<OperatorCustomerAdd />)} path={
          ROUTES.operatorCustomerAdd} />
```

```
11      </Route>
12      <Route element={<RequireAuth roles={["operator"]} />}>
13        <Route element={<AsCustomerLayout />}>
14          <Route element={withSuspensor(<Faqs />)} path={ROUTES.
                asCustomerFaqs} />
15        </Route>
16      </Route>
17    </Route>
18  </Routes>
```

Código 5.28: Resumen del componente App

En resumen, `App` actúa como un orquestador que combina navegación, control de acceso, carga asíncrona de páginas y personalización según el rol del usuario, lo que permite mantener una arquitectura limpia y escalable.

5.4.4. Gestión de autenticación con Redux Toolkit

La autenticación del frontend se gestiona usando **Redux Toolkit**, una librería que simplifica el manejo del estado global en aplicaciones React. Para ello, se define un *slice*, una estructura que agrupa estado, acciones y reducers relacionados en un único módulo cohesivo.

Un *slice* en Redux Toolkit representa una porción del estado global de la aplicación junto con las funciones (`reducers`) que lo modifican. En este caso utilizamos el `authSlice` como ejemplo, que gestiona toda la lógica relacionada con la autenticación: tokens, roles, identificación del usuario y estado de login.

El archivo `authSlice.js` tiene una estructura que define:

- Un `initialState` que marca los valores iniciales, como el token de acceso, el rol del usuario y si está autenticado (`isLoggedIn`).
- Un conjunto de `reducers` que contienen la lógica para modificar ese estado, como se muestra en el Código 5.29.

```
1  const authSlice = createSlice({
2    name: "auth",
3    initialState,
4    reducers: {
5      setMe: (state, action) => {
6        const { role, customer_id: customerId } = action.payload;
7        state.role = role;
8        state.customerId = customerId;
9        state.isLoggedIn = true;
10     },
11     setCredentials: (state, action) => {
12       const { access, refresh, rememberMe } = action.payload;
13       state.access = access;
14       localStorage.setItem("accessToken", access);
15       rememberMe
16         ? localStorage.setItem("refreshToken", refresh)
17         : sessionStorage.setItem("refreshToken", refresh);
18     },
19     logOut: (state) => {
```

```
      state.role = undefined;
      state.isLoggedIn = undefined;
      localStorage.setItem("refreshToken", "");
      sessionStorage.setItem("refreshToken", "");
    },
  },
});
```

Código 5.29: Fragmento del authSlice para manejar autenticación

Principales reducers implementados:

- `setMe`: Almacena en el estado global los datos básicos del usuario extraídos del backend, como `role` y `customerId`, y marca al usuario como autenticado.
- `setCredentials`: Guarda el token de acceso y lo persiste en `localStorage` o `sessionStorage`, según si el usuario elige la opción de "recordarme".
- `logOut`: Limpia todos los datos del usuario y elimina los tokens guardados, cerrando efectivamente la sesión.

Selectores: Se definen funciones auxiliares (*selectors*) como:

- `getToken`: Obtiene el token de acceso actual.
- `getMe`: Recupera todo el estado de autenticación.
- `getRole`: Recupera únicamente el rol del usuario.

Flujo general de autenticación:

1. Al iniciar sesión, el backend devuelve un token y los datos del usuario.
2. El frontend guarda esta información en el estado global usando `setMe` y `setCredentials`.
3. Los componentes consumen el estado global usando el hook `useSelector(getMe)` para mostrar la interfaz adecuada según el rol del usuario.
4. Al cerrar sesión, `logOut` limpia el estado y los tokens.

Este sistema permite que toda la aplicación React esté sincronizada con el estado de autenticación, garantizando que las vistas, menús y permisos mostrados sean siempre consistentes con el perfil del usuario.

5.4.5. Gestión de peticiones HTTP con RTK Query

En este frontend hemos implementadpp **RTK Query**, una herramienta incluida en Redux Toolkit, para gestionar las peticiones HTTP de forma eficiente. RTK Query simplifica la definición de endpoints, el manejo de estados de carga, la caché automática y la integración con el estado global de Redux.

El archivo `serviceApiSlice.js` define el objeto base `serviceApiSlice` usando `createApi`, que configura:

- La URL base de todas las peticiones (`SERVICE_API_URL`).
- Los encabezados comunes (como el token de acceso extraído del estado de autenticación).
- Los tipos de etiquetas (`tagTypes`) para el control de caché.

Este slice actúa como contenedor principal al que otros módulos pueden inyectar endpoints específicos según sus necesidades, como se observa en el Código 5.30.

```
export const serviceApiSlice = createApi({
  tagTypes: Object.values(SERVICE_TAGS),
  reducerPath: "service_api",
  baseQuery: configBaseQuery({
    baseQuery: serviceBaseQuery,
    reachableKey: "isServiceReachable",
  }),
  endpoints: () => ({}),
});
```

Código 5.30: Definición del slice base para peticiones

Sobre el `serviceApiSlice` base, se construyen *API Slices* especializados para diferentes dominios funcionales, como reservas o usuarios. Estos módulos utilizan la función `injectEndpoints` para añadir nuevos endpoints de forma modular.

Por ejemplo, en `bookingApiSlice.js`, se define un slice específico para las operaciones relacionadas con reservas (*bookings*). Este slice contiene mutaciones y consultas relacionadas con reservas y se conecta automáticamente con la configuración y caché del slice principal.

```
export const boookingAPiSlice = serviceApiSlice.injectEndpoints({
  endpoints: (builder) => ({
    createReservation: builder.mutation({
      query: ({ user_id, court_id, reservation_time, customer_id }) =>
          ({
        url: `/reservations/`,
        method: "POST",
        body: { user_id, court_id, reservation_time, customer_id },
      }),
      providesTags: [SERVICE_TAGS.Reservations],
    }),
  }),
});
```

Código 5.31: Definición del endpoint createReservation en bookingApiSlice

Los endpoints se crean con el objeto `builder`, que da acceso a métodos en función del tipo de petición HTTP que vayamos a implementar. El método `builder.mutation` se utiliza para definir operaciones que modifican datos en el servidor (como `POST`, `PUT`, `PATCH` o `DELETE`). Si queremos definir operaciones de solo lectura (`GET`), usamos en su lugar `builder.query`. Esto permite a RTK Query optimizar la caché y distinguir claramente entre consultas y modificaciones de datos, facilitando el control de dependencias y refresco automático.

RTK Query genera automáticamente *hooks* personalizados para interactuar con cada endpoint definido. El nombre de estos hooks se construye siguiendo una convención automática: primero se antepone el prefijo `use`, luego el nombre del endpoint (en este caso, `CreateReservation`)

y finalmente el sufijo `Mutation` o `Query` dependiendo del tipo de operación. Esto significa que al definir el endpoint `createReservation` usando `builder.mutation`, el hook generado será `useCreateReservationMutation`. El prefijo `use` es un estándar en React para indicar que se trata de un *hook*, lo que permitirá a las herramientas de desarrollo detectar su uso correctamente dentro de componentes funcionales. Este *hook* proporciona una función para lanzar la petición y un objeto con los estados de carga, error y datos, simplificando la interacción con la API, como podemos ver en el Código 5.32.

```
const [createReservation] = useCreateReservationMutation();
await createReservation({
  user_id: userId,
  court_id: selectedCourt,
  reservation_time: selectedDateTime,
  customer_id: parseInt(customerId)
});
```

Código 5.32: Uso del hook useCreateReservationMutation

5.4.6. Interfaz de usuario: pantallas principales

La aplicación cuenta con una interfaz moderna y adaptada a cada tipo de usuario. Existen dos temas visuales definidos de forma dinámica:

- **Tema claro**: se aplica automáticamente a los usuarios con rol `customer`.
- **Tema oscuro**: reservado para los usuarios `operator` (administradores o gestores).

La asignación del tema se hace mediante Redux Toolkit en el estado global de la aplicación. En el archivo `authSlice.js` se guarda al usuario autenticado y todos sus datos, incluido el campo `role`, y a la hora de mostrar un estilo visual o dirigir al usuario hacia unas pantallas u otras se coteja la información guardada en este estado. Esta lógica se implementa en el archivo `App.js` mediante el hook `useMemo`, como se ilustra en el código de la figura 5.33.

```
const Home = useMemo(() => {
  const homes = {
    operator: OperatorDashboard,
    customer: CustomerDashboard,
  };
  return homes[role];
}, [role]);
```

Código 5.33: Selección dinámica del dashboard en función del rol

5.4.6.1. Pantalla de login

La pantalla de login, mostrada en la figura 5.5, es la puerta de entrada a la aplicación. Utiliza un diseño centrado, con un fondo degradado que se adapta al tema activo. El componente principal responsable es `Login.js`, que renderiza un encabezado con el logo y el título de bienvenida, seguido del componente funcional `<LoginForm />`, encargado de gestionar la lógica de autenticación del usuario.

El componente `LoginForm.js` encapsula toda la lógica del proceso de login:

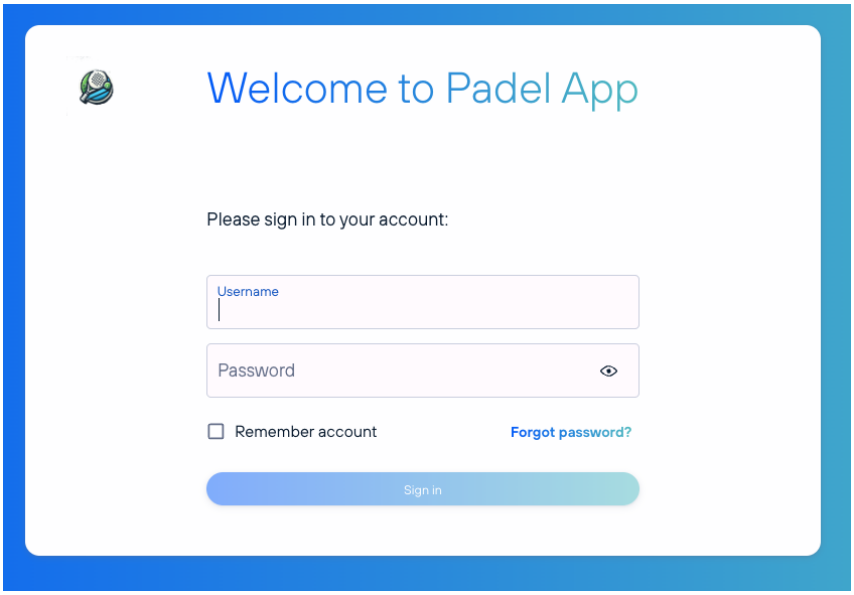

Fig. 5.5: Pantalla de inicio de sesión con fondo degradado y logo superior.

- Inicializa los campos del formulario (`login`, `password`, `rememberMe`) con estado local mediante `useMemo`.
- Gestiona eventos como mostrar/ocultar la contraseña o abrir diálogos informativos si el usuario olvida sus credenciales.
- Valida los datos introducidos con `Yup`, y en caso de error, lo notifica con un componente `<Alert>`.
- Si los datos son correctos, utiliza el hook `useSignIn1Mutation` para realizar una petición `POST` al backend (Código 5.34).
- Nada más reciibir el token de autenticación, se lanza la acción `dispatch(setCredentials(...))`, que actualiza el estado global del usuario y redirige al dashboard correspondiente.

```
signIn1: builder.mutation({
  query: ({ login, password }) => {
    const formData = new FormData();
    formData.append('name', login);
    formData.append('password', password);
    return {
      url: `/login/`,
      method: "POST",
      body: formData,
    };
  },
  invalidatesTags: [ACCOUNTS_TAGS.Me],
})
```

Código 5.34: Petición de autenticación al backend con FormData

En caso de olvidar la contraseña, el usuario puede pulsar sobre el enlace correspondiente, lo que abre un `Dialog` informativo con el mensaje de ayuda y la dirección de contacto del soporte técnico. Este comportamiento está implementado en el componente `LoginForm`, tal como se muestra en el código de la figura 5.35.

```
<DialogContentText>
  {t("forgotMessage")}
  <br />
  {t("forgotContact")} <a href={`mailto:${supportEmail}`}>{supportEmail
     }</a>
</DialogContentText>
```

Código 5.35: Diálogo de recuperación de contraseña

Así, la pantalla de login no sólo permite el acceso a la plataforma, sino que gestiona adecuadamente la experiencia del usuario en escenarios de error o recuperación de credenciales. Si el proceso de autenticación tiene éxito, el usuario es redirigido automáticamente al panel correspondiente, como veremos a continuación.

5.4.6.2. Pantallas para usuarios cliente

Tras el inicio de sesión, si el usuario tiene el rol `customer`, accede directamente al **CustomerDashboard**, cuya vista inicial se muestra en la figura 5.6. En ella se presentan los clubes de pádel disponibles, cada uno representado como una tarjeta visual generada con imágenes personalizadas creadas mediante *Sora*, el generador de imágenes de OpenAI [26]. Para conseguir una apariencia atractiva y contextual, las imágenes fueron generadas mostrando monumentos icónicos de las ciudades donde se ubican los clubes, integrados visualmente con una pista de pádel en primer plano. Este enfoque permite al usuario identificar fácilmente el club por su localización y refuerza la ambientación visual de la aplicación.

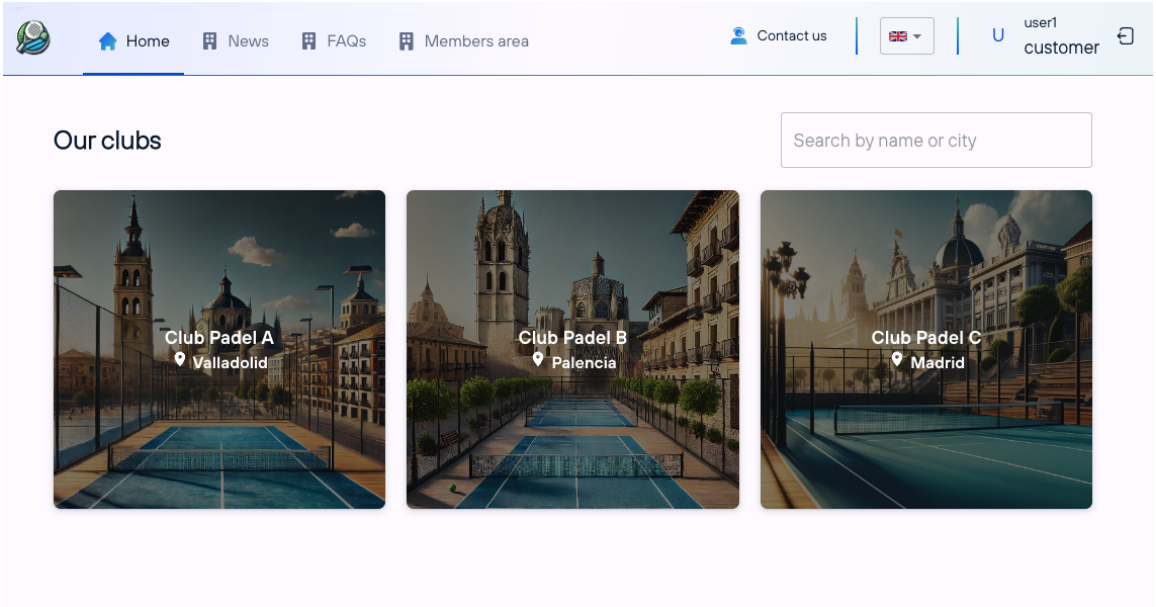

Fig. 5.6: Vista principal del panel de usuario cliente con búsqueda y listado de clubes

Desde esta pantalla, el usuario puede buscar clubes por nombre o ciudad mediante un campo de texto que realiza una búsqueda dinámica. El componente React responsable de esta funcionalidad es `Dashboard.js`, el cual se apoya en el estado global gestionado con Redux y en las peticiones asincrónicas proporcionadas por RTK Query. En concreto, se utiliza la llamada `useCustomersQuery()` para obtener la lista completa de clubes asociados al sistema.

Cuando el usuario selecciona uno de los clubes, la interfaz cambia al componente `ClubDetails.js`, que proporciona una vista detallada del club. Como se observa en la figura 5.7, se muestra una imagen de fondo con un botón en su interior, y en la parte inferior, un cuadro con el clima actual de la ciudad donde se encuentra ubicado. Esta información meteorológica se obtiene en tiempo real a partir de una API externa, consultada mediante una mutación que envía las coordenadas geográficas del club.

Esta integración de la información meteorológica añade valor a la experiencia de usuario, permitiendo prever condiciones desfavorables (lluvia, viento, humedad) que puedan influir en su decisión a la hora de reservar una pista.

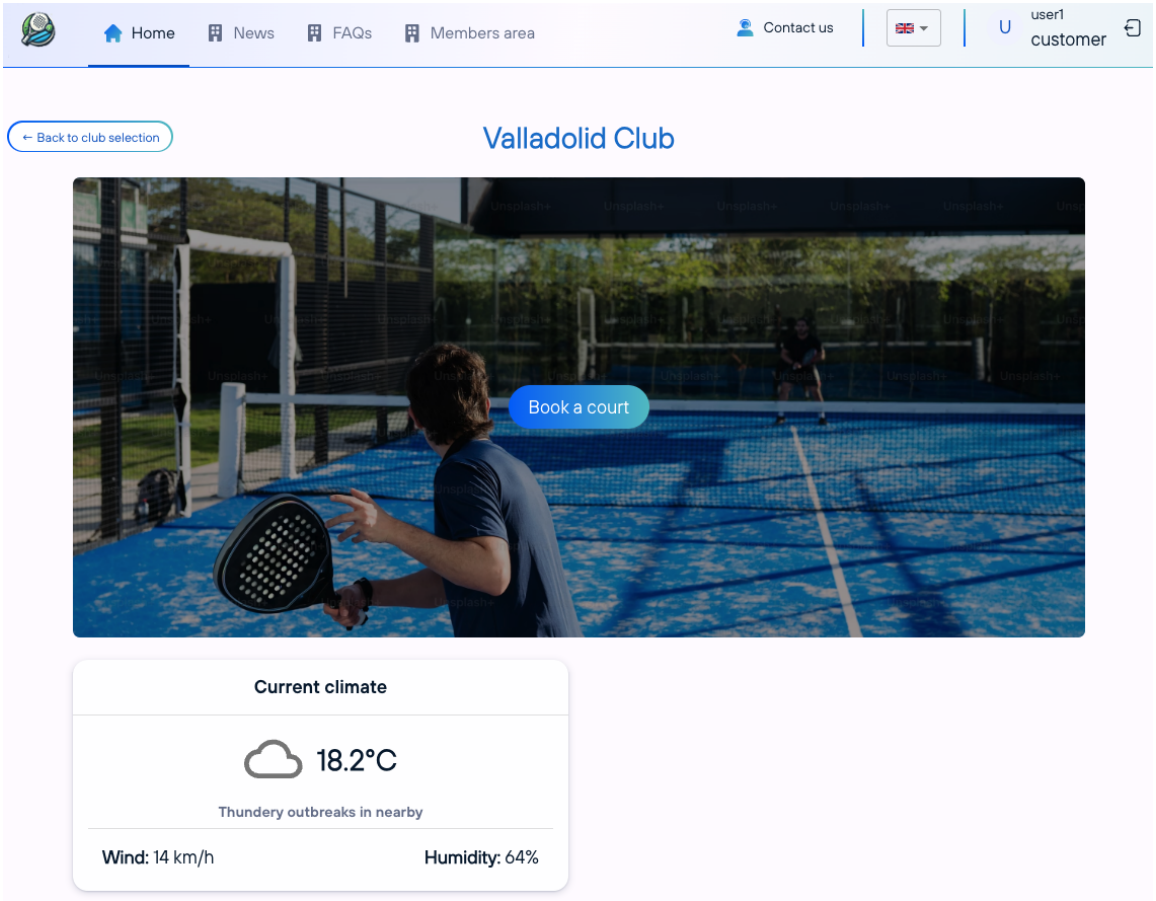

Fig. 5.7: Detalles del club seleccionado y visualización del clima actual

Además del clima, el botón contenido en el interior de la imagen principal permite al usuario acceder directamente a la página de reserva de pistas. Esta transición entre componentes se realiza mediante `useNavigate` de React Router, que preserva el contexto del club seleccionado (nombre, ubicación y `customerId`). De esta forma hace fácil la reserva de este apartado, evitando que tenga que introducir más datos.

Proceso de reserva de pista

La pantalla de reservas permite al usuario realizar una reserva personalizada según sus preferencias. Si el usuario ha accedido desde el componente `ClubDetails`, el club ya se encuentra preseleccionado; en caso contrario, la interfaz permite seleccionar el club manualmente desde un desplegable.

La figura 5.8 muestra una vista general del proceso completo de reserva, incluyendo la selección del horario y de la pista deseada. Este componente corresponde a `Booking.js` y condensa en una única interfaz todas las decisiones que debe tomar el usuario para formalizar una reserva.

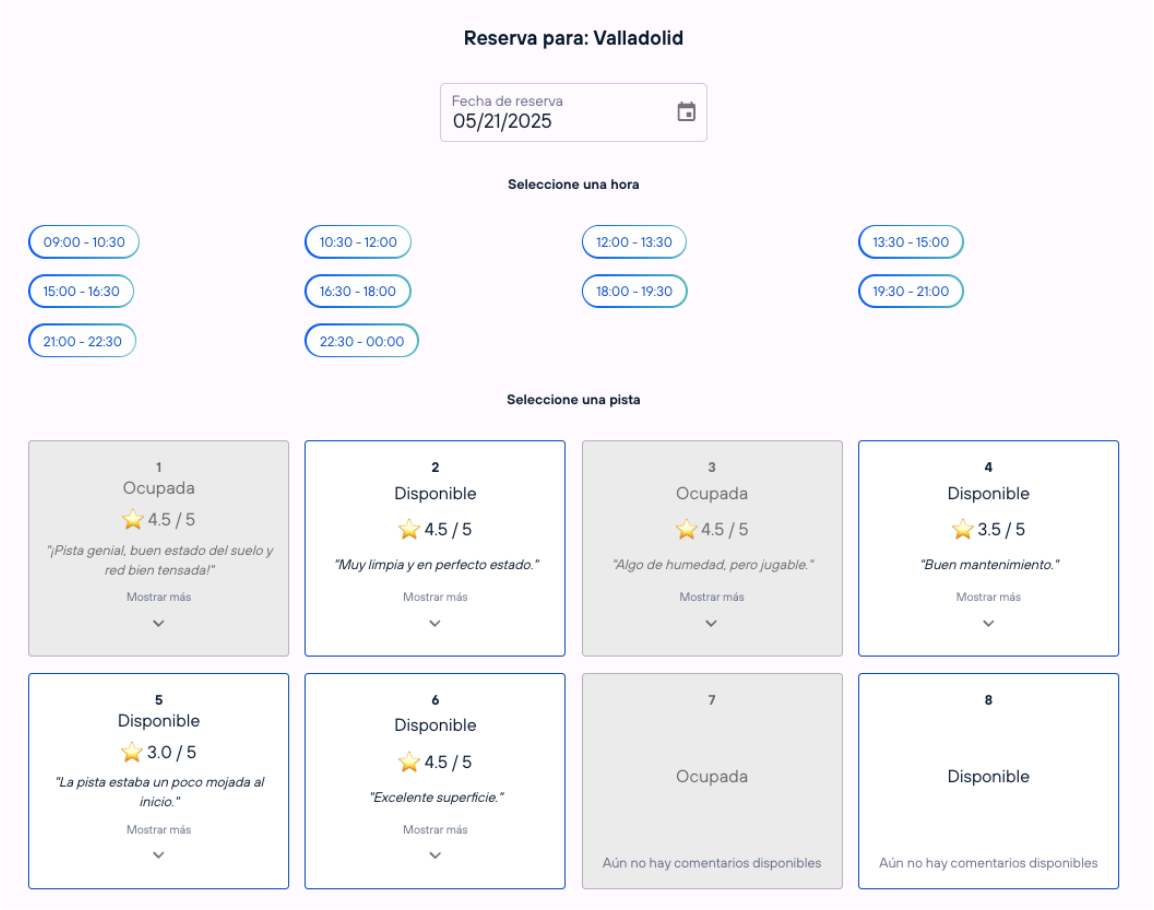

Fig. 5.8: Vista general del sistema de reservas con selección de hora y pista

A continuación, el usuario debe elegir la fecha en la que desea reservar, así como un tramo horario. Las franjas disponibles están predefinidas en intervalos de 90 minutos, duración estándar para una partida de pádel. Una vez seleccionados estos parámetros, se realiza una consulta a la API para obtener la disponibilidad de pistas en dicho club, fecha y horario.

```
availableCourts: builder.query({
  query: ({ customer_id, date }) => ({
    url: `/customers/${customer_id}/available-courts/`,
    params: { date },
  }),
  providesTags: [SERVICE_TAGS.Reservations],
}),
```

Código 5.36: Consulta de pistas disponibles

La respuesta de esta consulta incluye información muy valiosa: por cada pista se indica si está disponible o no, su nota media (basada en valoraciones anteriores) y un máximo de cinco comentarios recientes.

Como puede observarse en la figura 5.9, cada pista se representa mediante una tarjeta donde se muestra su estado de disponibilidad, valoración numérica y el comentario más reciente. El

usuario tiene la posibilidad de hacer clic en `Mostrar más` para expandir la tarjeta y visualizar el histórico de valoraciones recientes. Este comportamiento dinámico mejora notablemente la experiencia del usuario, permitiéndole tomar decisiones más informadas antes de realizar una reserva.

Fig. 5.9: Visualización expandida de reviews asociadas a una pista

Este sistema de puntuación y opiniones representa el principal factor diferencial de la aplicación respecto a otras plataformas de gestión deportiva. No todas las pistas son iguales, y en deportes como el pádel, pequeñas diferencias pueden impactar notablemente en la experiencia de juego. Por ejemplo:

- Algunas pistas pueden estar situadas en zonas con mal drenaje, haciendo que el bote de la pelota se vea afectado en condiciones húmedas.
- Otras pistas, aunque cubiertas, pueden estar expuestas lateralmente a la lluvia o al viento, lo cual puede dificultar el juego en días de climatología adversa.
- En ciertos horarios, la iluminación o la orientación solar puede influir en la visibilidad.

Gracias al sistema de valoraciones, los usuarios pueden beneficiarse de la experiencia de otros y reservar la pista que mejor se adapte a sus necesidades y preferencias, mejorando así la satisfacción general y evitando posibles frustraciones por condiciones no deseadas.

Una vez seleccionada la pista deseada, se habilita el botón `Reservar pista`. Al pulsarlo, aparece un diálogo de confirmación como el mostrado en la figura 5.10, indicando la pista y el horario elegidos. Si el usuario acepta, se realiza la reserva mediante una llamada POST a la API. Si la operación tiene éxito, se muestra un segundo cuadro de diálogo confirmando la reserva (figura 5.11).

Fig. 5.10: Diálogo de confirmación de reserva

Además, como parte del sistema de notificaciones del backend, se envía automáticamente un correo electrónico al usuario confirmando la reserva realizada. En él se indican los datos relevantes: club, pista, fecha y hora. Un ejemplo de dicho correo puede verse en la figura 5.12.

Fig. 5.11: Reserva confirmada correctamente

Fig. 5.12: Email de confirmación de reserva enviado al usuario

Área de miembros

Tras completar una reserva con éxito (figura 5.11), el usuario tiene la opción de acceder directamente a sus reservas mediante el botón `Ir a mis reservas`. Esta acción redirige al apartado `Members Area`, accesible también desde el menú de navegación superior. Esta sección está dividida en dos pestañas principales: **Perfil** y **Reservas**, como se muestra en la figura 5.13.

En la pestaña `Reservas`, el usuario puede consultar el historial de sus reservas futuras y pasadas. Es posible ordenarlas cronológicamente (ascendente o descendente), visualizar detalles como el clima estimado para el momento de la reserva, o cancelar una reserva si fuera necesario porque no se va a poder asistir o porque, por ejemplo, un cambio en la meteorología de la pista en la fecha de la reserva va a impedir jugar. También se incluye una paginación configurable, tal como se muestra en la figura.

Cada tarjeta de reserva muestra información detallada sobre el club, la pista, la fecha y la hora, acompañada de una previsión meteorológica que incluye la condición climática, temperatura, viento y humedad. Esta información favorece la transparencia y facilita la planificación, permitiendo al usuario anticiparse a condiciones adversas.

En la pestaña `Perfil`, el usuario puede consultar sus datos personales (nombre y correo electrónico) en una tarjeta visual que aparece centrada en la pantalla. Esta tarjeta está acompañada por una animación decorativa, obtenida mediante la librería `Lottie` de React. Este detalle aporta un toque visual moderno y atractivo a la sección de perfil, mejorando la experiencia de usuario (figura 5.14).

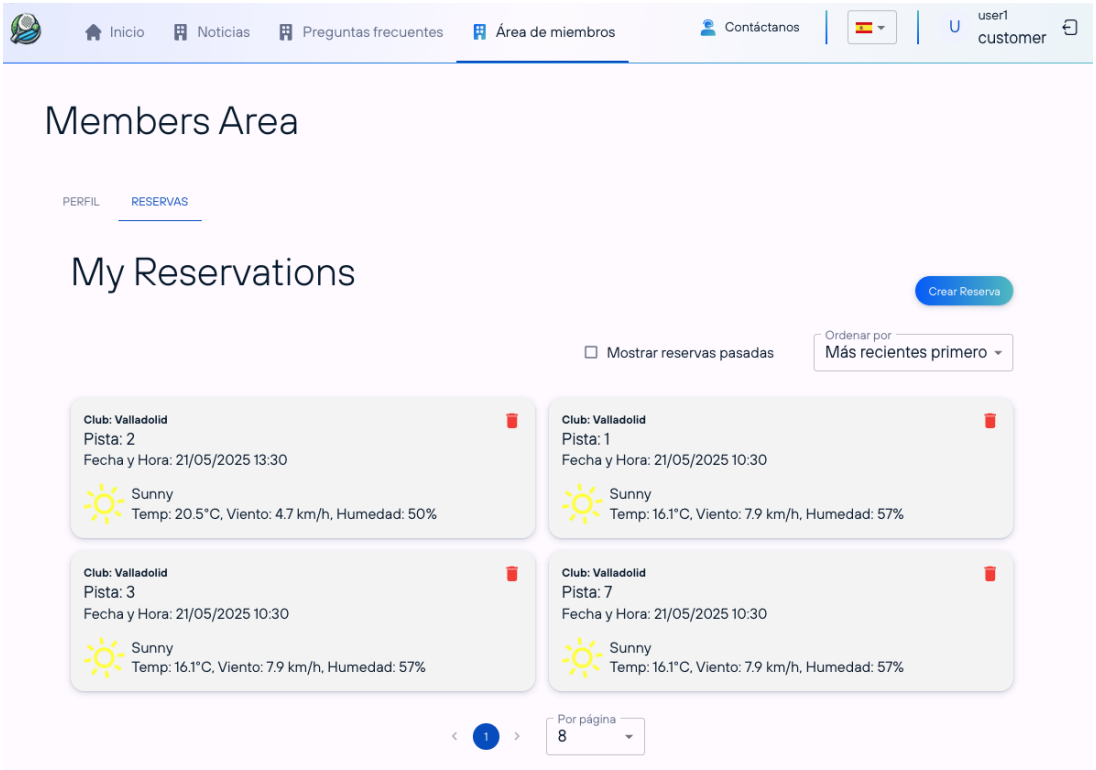

Fig. 5.13: Área de miembros con pestañas para perfil y reservas

El botón `Editar perfil` abre un cuadro de diálogo modal que permite modificar tanto el nombre como el email, y opcionalmente introducir una nueva contraseña. Este formulario utiliza la mutación `useUpdateUserByIdMutation`, la cual realiza una petición PUT al endpoint correspondiente del backend con los nuevos datos:

```
updateUserById: builder.mutation({
  query: ({ userId, name, password, email }) => ({
    url: `/user/${userId}`,
    method: "PUT",
    headers: {
      'Content-Type': 'application/json',
    },
    body: { name, password, email },
  }),
  invalidatesTags: [ACCOUNTS_TAGS.Users],
}),
```

Código 5.37: Actualización de usuario

Este diálogo se muestra en la figura 5.15, y garantiza una experiencia sencilla y directa para que el usuario mantenga sus datos siempre actualizados. En caso de éxito, el sistema actualiza la vista y muestra un mensaje temporal de confirmación.

En conjunto, esta área proporciona al usuario cliente una experiencia completa de autogestión, centralizando sus reservas, sus datos y facilitando la toma de decisiones mediante información contextual clave como la meteorología o las valoraciones de las pistas.

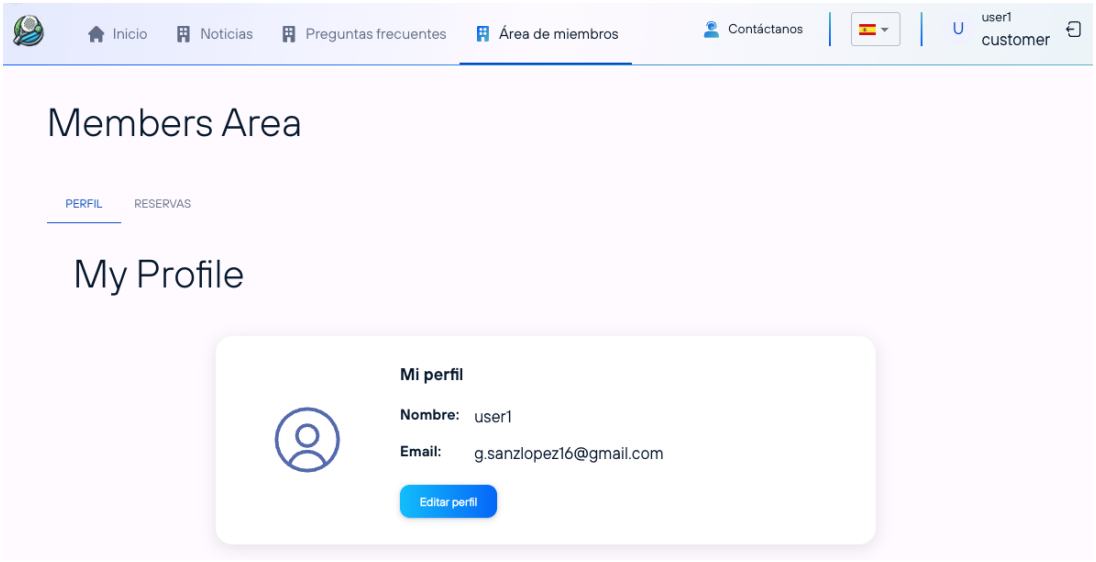

Fig. 5.14: Visualización del perfil del usuario con animación decorativa

Fig. 5.15: Cuadro de diálogo para edición del perfil

Noticias de pádel

Dentro del menú superior de navegación se encuentra el apartado `Noticias`, el cual ofrece al usuario una vista actualizada de noticias relevantes sobre el mundo del pádel, como se muestra en la figura 5.16.

Esta funcionalidad se implementa en el componente `PadelNews.js`, y emplea el hook `useGetPadelNewsQuery()` para consumir una API externa de noticias deportivas, concretamente filtrando por el término `premier padel`. Además, se ofrece al usuario la posibilidad de:

- Seleccionar una fecha de inicio para visualizar noticias recientes a partir de ese momento.
- Ordenar los artículos por antigüedad (más recientes o más antiguos).
- Navegar cómodamente mediante paginación.

Una funcionalidad destacable es la validación de fechas: si el usuario intenta seleccionar una fecha demasiado antigua (más de un mes), se activa un `Snackbar` que informa de que esta funcionalidad es de pago. Esto introduce de forma elegante una limitación comercial dentro de la interfaz sin romper la experiencia de usuario.

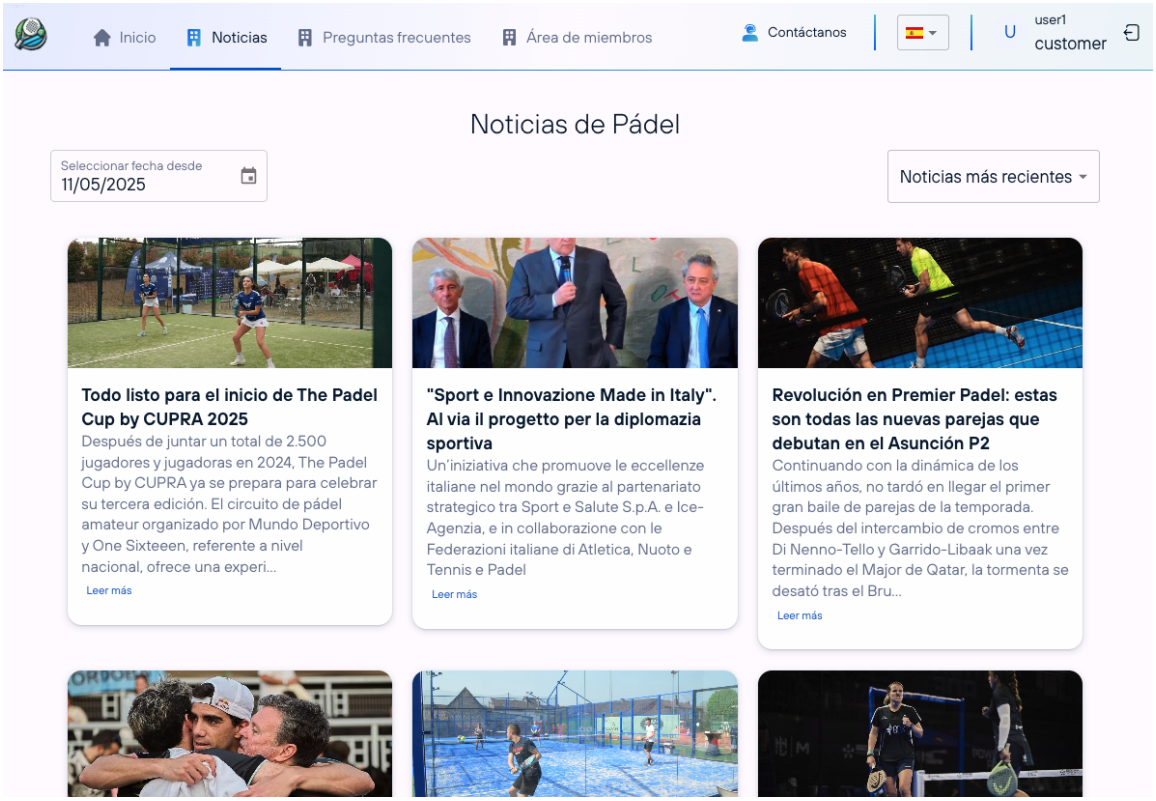

Fig. 5.16: Sección de noticias sobre pádel

Las noticias se muestran en tarjetas con imagen, título, descripción y un botón `Leer más` que abre el artículo original en una nueva pestaña.

Preguntas frecuentes (FAQs)

La sección de `Preguntas Frecuentes`, accesible también desde el menú de navegación, busca resolver de forma rápida las dudas más comunes que pueden tener los usuarios, como se aprecia en la figura 5.17.

Desde el componente `Faqs.js` se hace una petición para obtener el listado de preguntas comunes desde el backend con el hook `useGetFaqsQuery()`. Aquí se utiliza el componente `Accordion` para mostrar tarjetas expandibles que enseñan la respuesta a la pregunta.

Además, se ha añadido una breve introducción que invita al usuario a contactar en caso de no encontrar la respuesta deseada, completando así una funcionalidad de ayuda clara y accesible.

5.4.6.3. Pantallas para usuarios administrador (rol `operator`)

Cuando un usuario con rol `operator` accede al sistema tras el inicio de sesión, es redirigido automáticamente al **Operator Dashboard**, que muestra un resumen de los clubes de pádel bajo su gestión (figura 5.18).

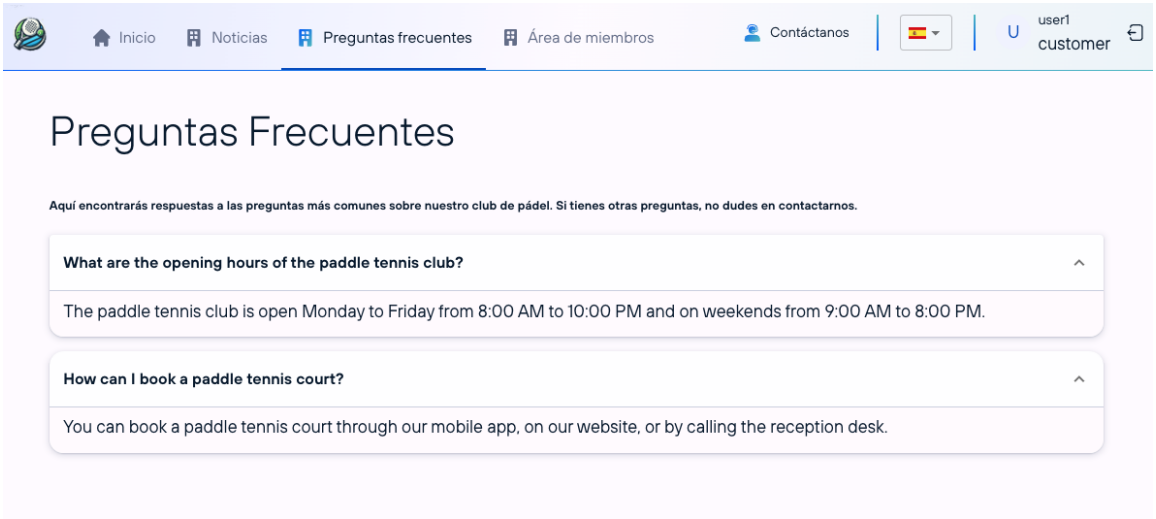

Fig. 5.17: Vista de la sección de preguntas frecuentes

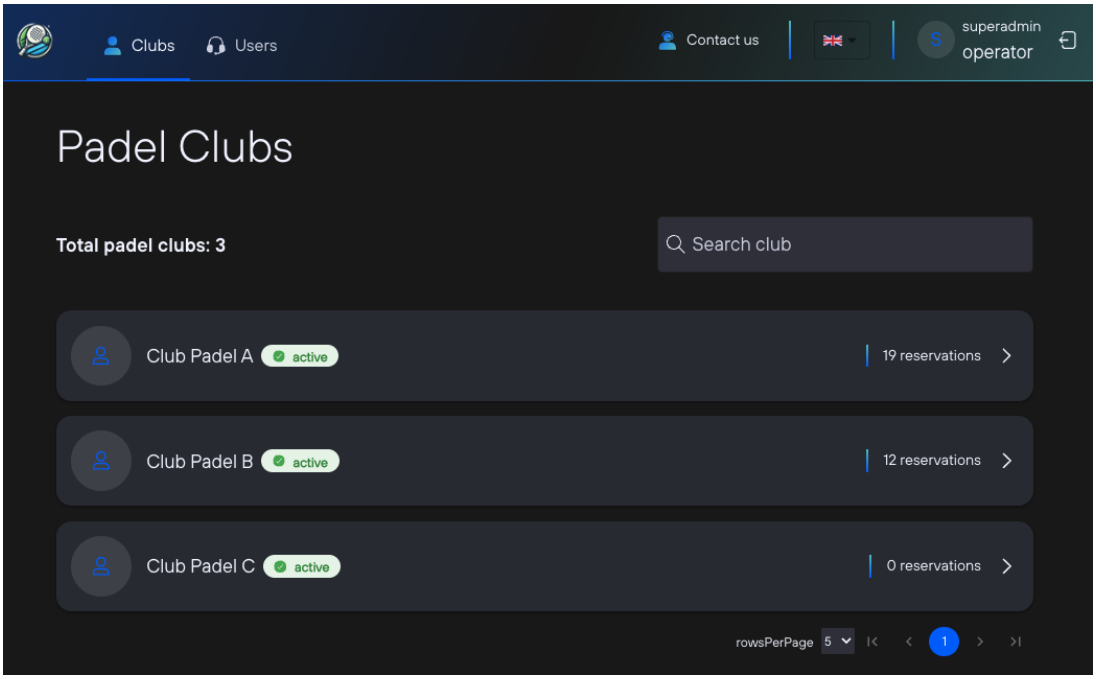

Fig. 5.18: Panel de administración con resumen de clubes de pádel

Cada club se presenta en forma de tarjeta, mostrando el nombre, el estado y el número total de reservas. La interfaz incluye una barra de búsqueda para filtrar clubes por nombre o estado. Esta funcionalidad se implementa mediante el componente `SearchBar` reutilizable, que permite búsquedas dinámicas sin recargar la página.

Si el operador selecciona uno de los clubes, accede al **Dashboard de gestión de pistas**, una página con información en tiempo real sobre la ocupación de pistas en ese club concreto (figura 5.19).

En la parte superior se muestra un resumen general con:

- Número total de pistas ocupadas.

Fig. 5.19: Estado actual de las pistas en el club seleccionado

- Número total de pistas disponibles.
- Total de reservas activas en el club (reservas futuras programadas).

Este resumen se actualiza automáticamente cada minuto, utilizando un temporizador (`setInterval`) que actualiza la hora del sistema sin necesidad de recargar la página.

Debajo, se muestra el estado de cada pista individualmente. Las pistas aparecen representadas con tarjetas coloreadas en verde (disponible) o rojo (ocupada), facilitando una visión inmediata de la situación actual del club. Esta vista es especialmente útil para los administradores en tiempo real, por ejemplo, para anticiparse a posibles sobrecargas, incidencias o cancelaciones.

El diseño está pensado para que, con solo un vistazo, se puedan tomar decisiones operativas sin necesidad de acceder a cada reserva individualmente, lo cual aporta eficiencia en la gestión diaria del club.

Al igual que los usuarios cliente, el operador también dispone de acceso a otras secciones informativas de la aplicación. Desde la barra de navegación superior puede alternar entre distintas pestañas como `News` y `FAQs`, las cuales le permiten visualizar el contenido público de la plataforma desde una perspectiva administrativa.

Sección de noticias (News)

Esta vista permite al operador acceder al mismo contenido que vería un usuario cliente en la sección de noticias (figura 5.20), pero adaptado al tema oscuro que caracteriza la interfaz de administración.

Desde aquí, el operador puede verificar fácilmente que las noticias públicas se están cargando

Fig. 5.20: Vista de noticias desde el panel de operador con tema oscuro

correctamente, que las imágenes están bien formateadas, y que el filtrado por fecha u orden de popularidad funciona como se espera. Esta capacidad de validación rápida resulta especialmente útil para detectar errores visuales o comprobar si las noticias externas (obtenidas desde una API pública) están adecuadamente integradas.

Sección de preguntas frecuentes (FAQs)

Del mismo modo, la pestaña `FAQs` permite al operador revisar en tiempo real las preguntas frecuentes disponibles para los usuarios (figura 5.21).

Fig. 5.21: Vista de preguntas frecuentes desde la perspectiva del operador

Esta funcionalidad es especialmente valiosa ya que permite al operador comprobar que las

FAQs están correctamente traducidas, formateadas y visibles para los clientes. También facilita el trabajo de supervisión y mantenimiento de contenido sin necesidad de alternar entre cuentas de distinto rol.

En conjunto, estas vistas "espejo" permiten a los operadores asumir un rol de control de calidad visual y funcional, validando directamente cómo se presenta la información en la aplicación pública. Esto es coherente con una filosofía de diseño centrada en roles múltiples, donde cada usuario puede tener visibilidad sobre lo que afecta a su área de responsabilidad, incluso si no lo edita directamente.

Gestión de reservas desde el panel del operador

Por último, el operador dispone de una pestaña específica para la **gestión de reservas**, accesible desde la barra de navegación mediante el botón `Reservas`. En esta vista, mostrada en la figura 5.22, se presenta una tabla dinámica con todas las reservas registradas por los usuarios del club seleccionado.

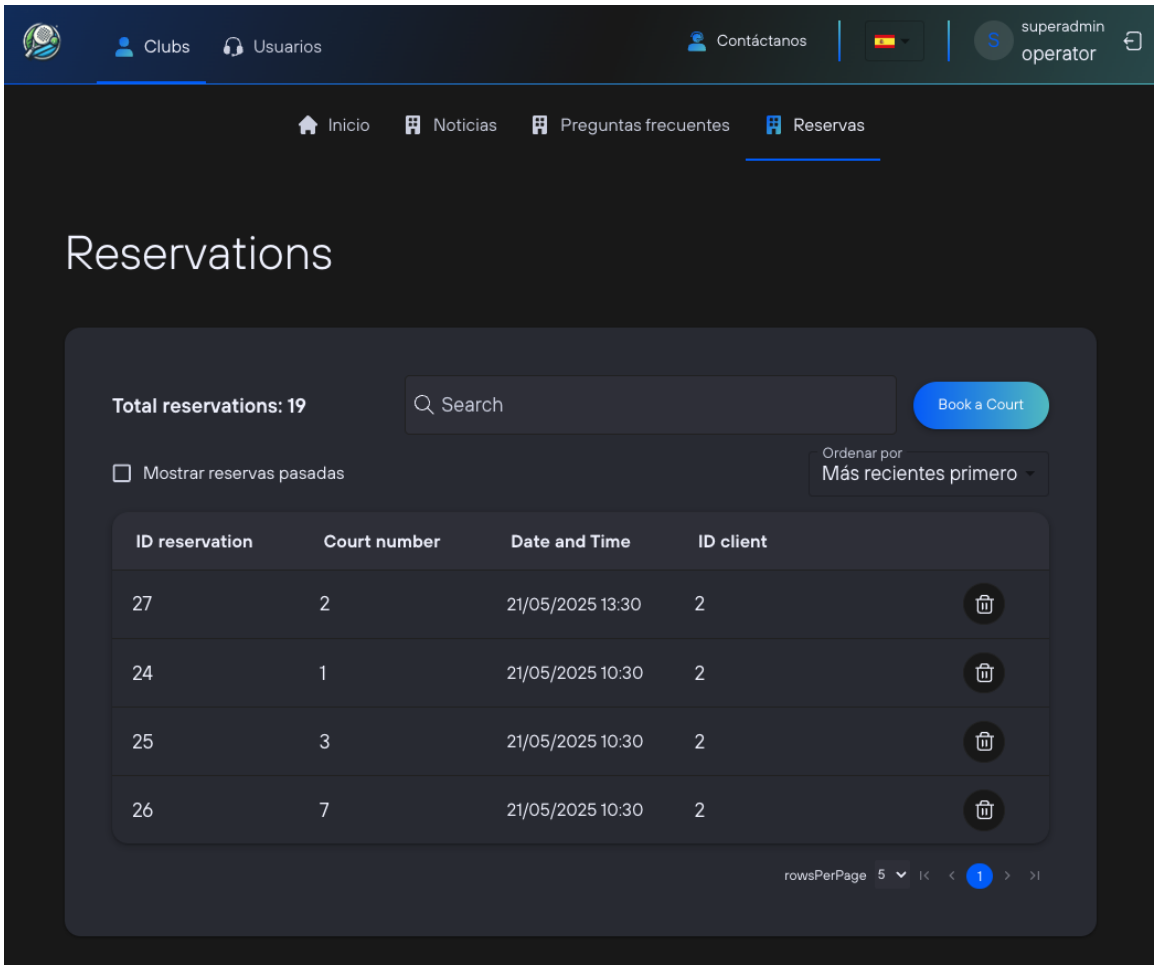

Fig. 5.22: Panel de gestión de reservas en modo operador

Esta herramienta proporciona funcionalidades clave para la administración efectiva de las reservas:

- Visualización del número total de reservas activas y pasadas.

- Búsqueda de reservas por cualquier campo (ID, número de pista, cliente, etc.).
- Ordenación configurable (más recientes primero, más antiguas, etc.).
- Posibilidad de filtrar únicamente las reservas futuras o incluir también las históricas mediante una casilla de verificación.
- Eliminación directa de reservas a través del icono de papelera.

Cada fila de la tabla muestra información clave como el **ID de la reserva**, **número de pista**, **fecha y hora programadas** y el **ID del cliente** que realizó la reserva. Esta estructura facilita al administrador identificar fácilmente cualquier reserva que requiera modificación o cancelación, ya sea por razones organizativos, de mantenimiento, o a petición del propio cliente.

Esta vista centraliza toda la información crítica relacionada con las reservas, agilizando las tareas de supervisión y proporcionando un entorno de gestión claro, ordenado y eficaz. Junto con el dashboard de pistas en tiempo real, constituye el núcleo operativo para la administración diaria del club.

Gestión de usuarios desde el panel de operador

El último apartado accesible para el operador desde la barra de navegación superior es el de `Usuarios`. Esta vista, representada en la figura 5.23, permite al administrador gestionar los usuarios del sistema de forma centralizada y eficaz.

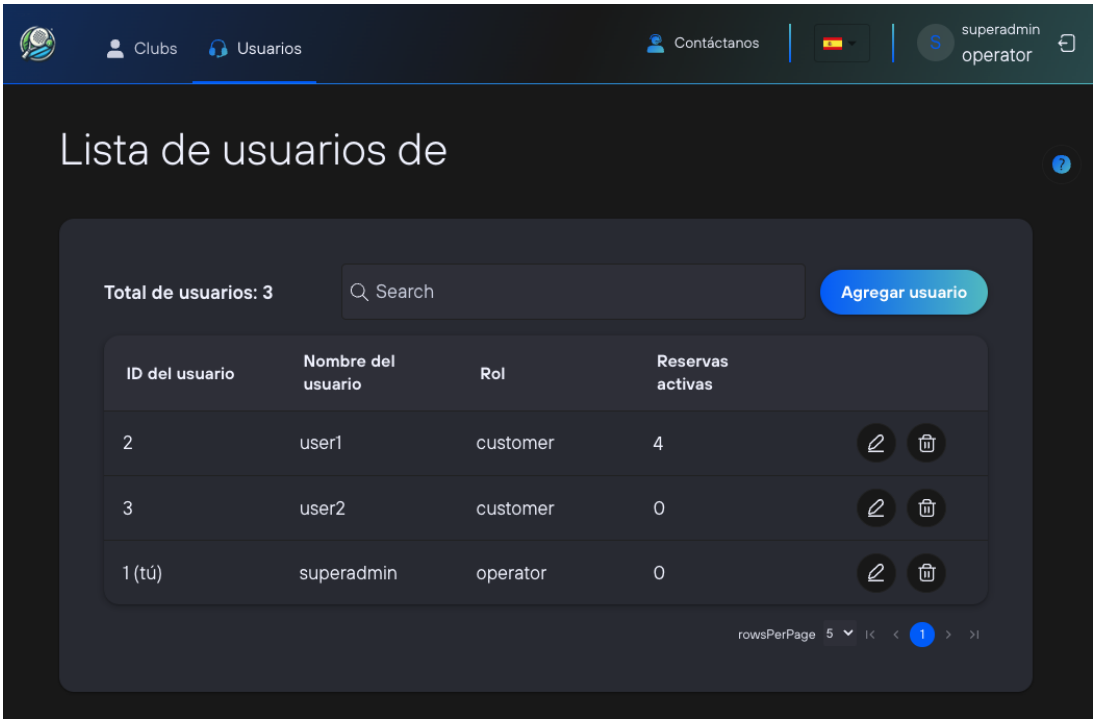

Fig. 5.23: Panel de gestión de usuarios con acciones de edición y borrado

La tabla muestra para cada usuario los siguientes campos clave:

- **ID del usuario**: identificador único en el sistema.

- **Nombre de usuario**: alias utilizado en la aplicación.
- **Rol**: que puede ser `customer` o `operator`.
- **Reservas activas**: número de reservas pendientes asociadas al usuario.

Además de consultar esta información, el operador puede realizar acciones administrativas esenciales como:

- **Buscar usuarios** mediante una barra de búsqueda integrada.
- **Editar los datos** de cualquier usuario existente mediante el icono de lápiz.
- **Eliminar usuarios** si fuera necesario, con el botón de papelera, siempre teniendo en cuenta las reservas activas asociadas.
- **Crear nuevos usuarios**, tanto del tipo cliente como administrador, usando el botón `Agregar usuario`.

Esta herramienta resulta particularmente valiosa en contextos donde se gestionan múltiples clubes o sedes, ya que facilita al operador mantener bajo control el acceso de los distintos perfiles a la aplicación.

En conjunto, esta vista completa la suite de herramientas disponibles para los usuarios con rol `operator`, quienes disponen de una consola de administración visual, clara y efectiva para gestionar clubes, pistas, reservas, contenido informativo y usuarios en tiempo real.

5.5. Despliegue con Docker y Nginx

Una vez finalizada la implementación del sistema, el despliegue completo se realiza mediante contenedores Docker organizados a través de un archivo `docker-compose.yml`. Tal como se anticipó en la sección 4.6, la arquitectura consta de cuatro servicios principales: **frontend**, **backend**, **base de datos** y **pgAdmin**, ejecutados de forma aislada pero conectados a través de una red interna común.

Para construir la aplicación frontend, se ejecuta el comando:

```
npm run apps:deploy
```

Código 5.38: Generación de ficheros estáticos

Este comando se apoya en varios scripts definidos en el `package.json` (Código 5.39), y tiene como objetivo compilar la aplicación React y generar los ficheros estáticos optimizados para producción. Estos ficheros —HTML, JavaScript minificado, CSS y recursos— se depositan en una estructura dentro de `src/deploy/public/{APP}`.

```
"scripts": {
    "start": "react-app-rewired start",
    "build": "GENERATE_SOURCEMAP=false react-app-rewired build",
    "apps:deploy": "cross-env node ./commands/appsNpmCommand.js --run=
       build && node ./commands/createNginxConfigs.js && node ./commands
       /createServeConfigs.js",
```

```
    "eslint": "eslint"
  }
```

Código 5.39: Script de despliegue definido en el package.json

Una vez generados, estos archivos se copian dentro de un contenedor `nginx`, usando el siguiente Dockerfile específico que vemos en el Código 5.40.

```
FROM nginx:alpine
ARG APP
WORKDIR /usr/share/nginx/html
COPY src/deploy/public/${APP} .
COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 8081
```

Código 5.40: Dockerfile para frontend con Nginx

Este contenedor servirá la aplicación frontend a través del puerto `8081`. La configuración avanzada de `nginx.conf` (listado 5.41) garantiza un buen rendimiento, seguridad y control de caché. Además, está preparada para servir correctamente aplicaciones SPA (Single Page Applications) como las construidas con React.

```
server {
  listen 8081 default_server;
  root /usr/share/nginx/html;
  location / {
    add_header Content-Security-Policy "...";
    try_files $uri /index.html =404;
  }
  ...
}
```

Código 5.41: Fragmento de configuración avanzada de Nginx

Por otro lado, el backend desarrollado con FastAPI se construye usando el siguiente Dockerfile que vemos en el Código 5.42.

```
FROM python:3.11.9-slim
RUN apt-get update && apt-get install -y libpq-dev gcc
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY ./app /app
COPY ./wait-for-it.sh /wait-for-it.sh
RUN chmod +x /wait-for-it.sh
EXPOSE 8000
CMD ["/wait-for-it.sh", "db:5432", "--", "uvicorn", "main:app", "--host
    ", "0.0.0.0", "--port", "8000"]
```

Código 5.42: Dockerfile para el backend Python

Este contenedor expone el servicio FastAPI en el puerto `8000`, conectado a la base de datos PostgreSQL a través de la variable de entorno `DATABASE_URL`.

La base de datos y la herramienta de administración `pgAdmin` se integran usando imágenes preconstruidas de Docker Hub [27]:

- `postgres:alpine` para el contenedor `postgres_db`.
- `dpage/pgadmin4` para `my_pgadmin`, expuesto en el puerto `8083`.

La orquestación completa se lleva a cabo a través del archivo `docker-compose.yml`, cuyo contenido se presenta en el Código 5.43. En este archivo se especifican los cuatro servicios fundamentales: base de datos, backend, frontend y pgAdmin, la herramienta de administración de la base de datos. Cada servicio está configurado con sus respectivas rutas de construcción, variables de entorno y puertos expuestos.

```
version: '3.8'
services:
  db:
    build:
      context: ./backend/postgres
    container_name: postgres_db
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: dbname
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"

  app:
    build: ./backend
    container_name: padel_backend
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      DATABASE_URL: "postgresql://user:password@db/dbname"
      TZ: Europe/Madrid
    volumes:
      - /etc/timezone:/etc/timezone:ro
      - /etc/localtime:/etc/localtime:ro

  pgadmin:
    image: dpage/pgadmin4
    container_name: my_pgadmin
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: admin
    ports:
      - "8083:80"
    depends_on:
      - db

  frontend:
    build:
```

```
      context: ./front-padel
      dockerfile: Dockerfile.deploy-app
      args:
        APP: padel-app
    container_name: padel_frontend
    ports:
      - "8081:8081"
    depends_on:
      - app

volumes:
  postgres_data:
```

Código 5.43: Archivo docker-compose.yml con definición de servicios

Una vez definido este archivo, basta con ejecutar el siguiente comando (Código 5.44) para compilar las imágenes y levantar los servicios en sus respectivos contenedores:

```
docker-compose up --build
```

Código 5.44: Comando Docker Compose para levantar servicios.

Este comando levanta los cuatro contenedores definidos como vemos en la Figura 5.24:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
221a0490dc98	padel-app-tfg-frontend	"/docker-entrypoint.…"	21 seconds ago	Up 20 seconds	80/tcp, 0.0.0.0:8081->8081/tcp	padel_frontend
bd856cae3d40	padel-app-tfg-app	"/wait-for-it.sh db:…"	22 seconds ago	Up 20 seconds	0.0.0.0:8000->8000/tcp	padel_backend
85d59d157cc5	dpage/pgadmin4	"/entrypoint.sh"	22 seconds ago	Up 20 seconds	443/tcp, 0.0.0.0:8083->80/tcp	my_pgadmin
9c7fbee1ef03	padel-app-tfg-db	"docker-entrypoint.s…"	22 seconds ago	Up 20 seconds	0.0.0.0:5432->5432/tcp	postgres_db

Fig. 5.24: Contenedores activos tras despliegue

Resumen de puertos expuestos por servicio:

- `frontend`: puerto `8081`, aplicación React servida por Nginx.
- `backend`: puerto `8000`, API en FastAPI.
- `db (PostgreSQL)`: puerto `5432`, acceso a base de datos.
- `pgAdmin`: puerto `8083`, herramienta de administración visual.

Gracias a esta arquitectura en contenedores, se consigue un sistema modular, portable y fácil de levantar o reiniciar. En entornos reales de desarrollo o producción, esta metodología acelera enormemente el ciclo de vida del software y facilita tareas de integración continua, pruebas o mantenimiento.

Capítulo 6

Conclusiones

El desarrollo de esta plataforma de reservas para clubes de pádel ha permitido dar respuesta a una necesidad real mediante una solución tecnológica sólida, moderna y orientada a mejorar significativamente la experiencia de los usuarios. El sistema diseñado permite gestionar reservas, usuarios, pistas y clubes de forma centralizada, integrando tecnologías actuales como FastAPI en el backend, React en el frontend y contenedores Docker para facilitar el despliegue y la escalabilidad del servicio.

Una de las claves del proyecto ha sido superar las funcionalidades básicas esperadas en este tipo de plataformas, incorporando elementos innovadores que aportan un valor diferencial frente a otras soluciones comerciales existentes. En particular, se han introducido dos mejoras reseñables que mejoran tanto la operativa del club como la satisfacción y fidelización de los usuarios.

Por un lado, se ha implementado un sistema de reseñas asociado a cada pista individual, permitiendo que los jugadores valoren su experiencia tras cada reserva. Este enfoque, más granular que los sistemas de puntuación globales a nivel de club que utilizan otras plataformas, ofrece una información más precisa y útil para otros usuarios a la hora de elegir qué pista reservar. Además, constituye una herramienta valiosa para los gestores del club, ya que permite identificar rápidamente aquellas instalaciones que requieren mejoras o mantenimiento, fomentando así un ciclo de calidad y mejora continua.

Por otro lado, el sistema incorpora una funcionalidad inteligente de notificaciones meteorológicas personalizadas. Con el objetivo de reducir cancelaciones de última hora o experiencias insatisfactorias causadas por el mal tiempo, la aplicación envía de forma automática un correo electrónico al usuario 24 horas antes de su reserva. Este mensaje no solo recuerda la cita, sino que incluye un análisis actualizado de la previsión meteorológica para la ubicación y hora concreta de la reserva, comparándola con la previsión registrada en el momento en que se realizó la misma. En caso de que existan cambios significativos —como un aumento del viento, una bajada de temperatura o la aparición de precipitaciones— el usuario es notificado de forma clara y comprensible, lo que le permite tomar decisiones informadas, como reprogramar o cancelar su reserva.

La automatización de estas tareas se ha implementado mediante procesos asíncronos que se ejecutan periódicamente, garantizando así su funcionamiento sin intervención manual. Asimismo, se ha añadido un tercer tipo de correo que se envía una vez finalizada la reserva, agradeciendo al usuario su visita e invitándolo a dejar una valoración. Esta última comunicación cierra el ciclo completo de experiencia digital en torno a la reserva, fomentando la interacción post-servicio y

alimentando el sistema de reseñas.

En definitiva, la plataforma desarrollada no solo cumple con los requisitos funcionales de un sistema de reservas eficiente, sino que introduce mejoras reales en la relación entre los clubes y sus usuarios. La atención al detalle, la automatización contextual y la orientación al usuario han sido constantes a lo largo del proyecto, permitiendo alcanzar una solución completa, moderna y diferenciadora. Estas características, junto a una arquitectura tecnológica escalable y fácilmente mantenible, sitúan esta aplicación como una opción competitiva y con gran potencial de evolución en el ámbito de la gestión deportiva digital.

Trabajos futuros

A pesar de que la plataforma desarrollada ofrece una solución completa y plenamente funcional, existen diversas líneas de mejora que podrían explorarse en el futuro para ampliar su alcance y sofisticación. Desde un punto de vista técnico, una posible evolución sería la incorporación de una pasarela de pagos que permitiera a los usuarios abonar sus reservas directamente desde la propia aplicación, automatizando así la gestión económica del club y reduciendo la necesidad de interacción manual. También se podría mejorar la experiencia de acceso mediante la integración de sistemas de autenticación social, permitiendo a los usuarios iniciar sesión con cuentas de Google u otras plataformas externas, lo que simplificaría el proceso de registro e inicio de sesión. Finalmente, sería conveniente incluir en el proyecto una infraestructura de integración y despliegue continuos (CI/CD), que facilite la validación automática del código mediante pruebas para después hacer su despliegue, garantizando así la calidad del software y mejorando los ciclos de desarrollo.

Desde una perspectiva funcional, también existen múltiples posibilidades de evolución. Una de ellas sería la incorporación de reservas grupales o recurrentes, permitiendo a los usuarios programar sesiones periódicas sin necesidad de repetir el proceso manualmente. También sería interesante ampliar la plataforma con un módulo de gestión de torneos o eventos, en el que los usuarios puedan inscribirse y consultar horarios y emparejamientos. Adicionalmente, se podrían añadir funcionalidades contextuales más avanzadas, como la predicción de si una pista tendrá sol o sombra en función del día del año, la franja horaria y la orientación específica de la pista, ofreciendo al usuario información útil para tomar decisiones más ajustadas a sus preferencias. Otra mejora relevante sería la recomendación automática del club más cercano a través del uso de la geolocalización del dispositivo del usuario, mejorando así la accesibilidad y personalización del servicio. Por último, dotar a los gestores del club de un sistema de informes analíticos, con estadísticas de uso, niveles de satisfacción y rendimiento económico, permitiría una toma de decisiones más informada y una gestión más estratégica de las instalaciones.

En conjunto, estas posibles líneas de mejora muestran que la plataforma no solo es funcionalmente rica en su estado actual, sino que dispone de una base sólida y flexible sobre la que seguir construyendo, adaptándose a nuevos contextos de uso y mejorando progresivamente tanto la experiencia del usuario como la eficiencia operativa del club.

Referencias

[1] Playtomic. *Playtomic - Encuentra dónde y con quién jugar al pádel y tenis*. 2025. URL: https://playtomic.io/ (última visita el 8 de abril de 2025).

[2] Reservaplay. *Reservaplay - Gestión total de instalaciones deportivas*. 2025. URL: https://reservaplay.com/ (última visita el 8 de abril de 2025).

[3] Clupik. *Clupik - Software para clubes y entidades deportivas*. 2025. URL: https://www.clupik.com/ (última visita el 8 de abril de 2025).

[4] Sebastián Ramírez. *FastAPI - The Modern Web Framework for Python*. 2025. URL: https://fastapi.tiangolo.com/ (última visita el 10 de marzo de 2025).

[5] JetBrains. *The State of Developer Ecosystem 2024: Python Section*. 2024. URL: https://blog.jetbrains.com/pycharm/2024/12/the-state-of-python/.

[6] Saurabh Singh. *Legacy Companies in the AI Era: Why FastAPI Is the Key to Python Integration*. Towards Dev. 2024. URL: https://medium.com/towardsdev/legacy-companies-in-the-ai-era-why-fastapi-is-the-key-to-python-integration-2be089d8fb8c.

[7] Redux Toolkit. *RTK Query - Redux Toolkit*. Documentación oficial de RTK Query. 2025. URL: https://redux-toolkit.js.org/rtk-query/overview (última visita el 8 de abril de 2025).

[8] Meta (Facebook). *React - A JavaScript library for building user interfaces*. 2025. URL: https://react.dev/ (última visita el 10 de marzo de 2025).

[9] State of JS. *State of JS 2023 — Developer Survey Results*. Consultado en abril de 2025. 2023. URL: https://2023.stateofjs.com/en-US/libraries/front-end-frameworks/.

[10] State of JS. *State of JS 2024 — Front-end Frameworks*. Accedido en abril de 2025. 2024. URL: https://2024.stateofjs.com/en-US/libraries/front-end-frameworks/.

[11] PostgreSQL Global Development Group. *PostgreSQL Documentation*. 2025. URL: https://www.postgresql.org/docs/ (última visita el 10 de marzo de 2025).

[12] Stack Overflow. *Stack Overflow Developer Survey 2024*. Consultado en abril de 2025. 2024. URL: https://survey.stackoverflow.co/2024/#technology-databases.

[13] Timescale Inc. *The State of PostgreSQL 2024*. Consultado en abril de 2025. 2024. URL: https://www.timescale.com/state-of-postgres/2024.

[14] DB-Engines. *DB-Engines Ranking — April 2025*. Ranking actualizado mensualmente. 2025. URL: https://db-engines.com/en/ranking.

[15] Docker Inc. *Docker Documentation*. 2025. URL: https://docs.docker.com/ (última visita el 10 de marzo de 2025).

[16] Nginx. *Nginx Documentation*. 2025. URL: https://nginx.org/en/docs/ (última visita el 10 de marzo de 2025).

[17] *Stack Overflow Developer Survey 2023*. https://survey.stackoverflow.co/2023/. Consultado en abril de 2025.

[18] *Usage statistics and market share of Nginx*. https://w3techs.com/technologies/details/ws-nginx. Consultado en abril de 2025.

[19] Mike Bayer. *SQLAlchemy Documentation*. Documentación oficial del ORM SQLAlchemy. 2025. URL: https://docs.sqlalchemy.org/ (última visita el 8 de abril de 2025).

[20] Samuel Colvin. *Pydantic Documentation*. Documentación oficial de Pydantic. 2025. URL: https://docs.pydantic.dev/ (última visita el 8 de abril de 2025).

[21] Miro. *Miro - Visual Collaboration Platform*. Herramienta utilizada para el diseño del diagrama de modelo de datos. 2025. URL: https://miro.com (última visita el 9 de abril de 2025).

[22] M. Jones, J. Bradley y N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. Consultado en abril de 2025. 2015. DOI: 10.17487/RFC7519. URL: https://www.rfc-editor.org/info/rfc7519.

[23] WeatherAPI. *WeatherAPI - Weather Forecast, Current Weather and Historical Weather Data*. 2025. URL: https://www.weatherapi.com/docs/ (última visita el 8 de abril de 2025).

[24] MUI Team. *Material UI - React Components for Faster Web Development*. 2025. URL: https://mui.com/ (última visita el 10 de abril de 2025).

[25] Jan Mühlemann et al. *i18next - Internationalization Framework for JavaScript*. 2025. URL: https://www.i18next.com/ (última visita el 10 de abril de 2025).

[26] OpenAI. *Sora - Video and Image Generation by OpenAI*. Disponible en: https://openai.com/sora. 2024.

[27] Docker Inc. *Docker Hub*. Última visita el 19 de mayo de 2025. 2024. URL: https://hub.docker.com/.

Anexo A

Repositorio en GitHub

Con el objetivo de facilitar la evolución del proyecto y permitir futuras contribuciones por parte de otros desarrolladores o equipos interesados, el código fuente completo de la aplicación, incluyendo tanto el backend desarrollado con `FastAPI` como el frontend implementado en `React`, se encuentra disponible públicamente en un repositorio de GitHub.

Además del código, el repositorio incluye documentación técnica, instrucciones para su despliegue mediante `Docker` y ejemplos de configuración necesarios para su ejecución en un entorno local o de producción.

Se puede acceder al repositorio a través del siguiente enlace:

`https://github.com/guillecxb/padel-app`