



MÓDULO PROYECTO

CFGS Desarrollo de Aplicaciones
Multiplataforma
Informática y Comunicaciones

Desarrollo de aplicación multiplataforma para aficionados a los juegos de mesa a través de frontend en React Native y backend basado en microservicios con NodeJS y Docker. “LUDOKEEPER”

Tutor individual: Juan Antonio Alonso Velasco

Tutor colectivo: Cristina Silván Pardo

Año: 2025

Fecha de presentación: 12/06/2025

Nombre y Apellidos: Álvaro Cilleruelo Sinovas

Email: alvaro.cilsin@educa.jcyl.es



Tabla de contenido

1	Identificación proyecto	4
2	Organización de la memoria	5
3	Descripción general del proyecto	7
3.1	Objetivos	7
3.2	Cuestiones metodológicas	7
3.3	Entorno de trabajo (tecnologías de desarrollo y herramientas).....	8
4	Descripción general del producto.....	11
4.1	Visión general del sistema: límites del sistema, funcionalidades básicas, usuarios y/o otros sistemas con los que pueda interactuar.....	11
4.2	Descripción breve de métodos, técnicas o arquitecturas(m/t/a) utilizadas.....	12
4.3	Despliegue de la aplicación indicando plataforma tecnológica, instalación de la aplicación y puesta en marcha.....	13
5	Planificación y presupuesto	15
5.1	Planificación	15
5.2	Presupuesto.....	15
6	Documentación Técnica: análisis, diseño, implementación y pruebas.....	17
6.1	Especificación de requisitos	17
6.2	Análisis del sistema	20
6.3	Diseño del sistema:	21
6.3.1	Diseño de la Base de Datos	21
6.3.2	Diseño de la Interfaz de usuario.	22
6.3.3	Diseño de la Aplicación.	23
6.4	Implementación:	23
6.4.1	Entorno de desarrollo.	23
6.4.2	Estructura del código.	24
6.4.3	Cuestiones de diseño e implementación reseñables.....	27

6.5	Pruebas.....	27
7	Manuales de usuario	29
7.1	Manual de usuario.....	29
7.2	Manual de instalación	29
8	Conclusiones y posibles ampliaciones	30
9	Bibliografía	31
10	Anexos	33

1 Identificación proyecto

En este proyecto se ha buscado desarrollar una aplicación que plasme los conocimientos adquiridos en el Grado Superior de Desarrollo de Aplicaciones Multiplataforma, ampliados mediante la investigación de tecnologías y procedimientos relevantes.

Para ello, se ha optado por la creación de una aplicación multiplataforma llamada **Ludokeeper** para la cual se ha utilizado React Native con Expo, debido a su versatilidad, su amplia comunidad y soporte, y su base en JavaScript y React. La aplicación gestionará la lógica de negocio y el acceso a datos a través de un backend API desarrollado con Node.js con Express y/o Fastify en función de las necesidades, lo que permite utilizar el mismo lenguaje en ambos programas y garantiza un buen rendimiento y escalabilidad. Además, como norma general se empleará TypeScript, lenguaje de programación el cual es un superconjunto de JavaScript al que añade tipado estricto. Con el objetivo de mejorar el desarrollo, se ha decidido adoptar una arquitectura de microservicios y utilizar Docker. Esto facilita la integración de tecnologías como Keycloak para la gestión de usuarios.

Finalmente, el último paso ha sido dotar al proyecto de un tema específico para su presentación. Tomando inspiración de un interés personal, la aplicación incluye herramientas útiles para aficionados a los juegos de mesa, como son la gestión de inventario, el registro de partidas, el catálogo de juegos, el simulador de dados o el control del tiempo.

2 Organización de la memoria

La memoria del proyecto se organiza en diez capítulos principales, cada uno de los cuales aborda aspectos específicos del proyecto desde su concepción hasta su implementación y conclusiones. A continuación, se describe en detalle la estructura de la memoria:

1 - Identificación del proyecto

Incluye la portada con los datos identificativos del alumno, el título del proyecto, el ciclo formativo cursado y el centro educativo.

2 - Organización de la memoria

Describe la estructura del documento, los apartados en los que se divide y su propósito dentro del contexto del proyecto.

3 - Descripción general del proyecto

Explica los objetivos principales y secundarios, la metodología empleada durante el desarrollo, el entorno de trabajo y las tecnologías utilizadas.

4 - Descripción general del producto

Presenta una visión funcional del sistema: sus límites, funcionalidades, tipos de usuarios y arquitectura general. También se justifica la elección de técnicas y herramientas empleadas, así como el modo de despliegue.

5 - Planificación y presupuesto

Detalla la planificación temporal de las tareas mediante un cronograma y realiza una estimación del coste del proyecto considerando desarrollo, software, hardware y otros factores.

6 - Documentación técnica

Abarca el análisis, diseño, implementación y pruebas del sistema. Incluye la especificación de requisitos funcionales y no funcionales, los diagramas E/R, el diseño de la interfaz de usuario, la organización del código, y el conjunto de pruebas realizadas para verificar su correcto funcionamiento.

7 - Manuales de usuario e instalación

Proporciona las instrucciones necesarias para el uso de la aplicación por parte del usuario final, así como para su instalación y puesta en marcha en un entorno local o remoto.

8 - Conclusiones y posibles ampliaciones

Recoge una reflexión final sobre el proceso, los aprendizajes obtenidos, los retos superados y plantea posibles mejoras o funcionalidades futuras para el sistema.

9 - Bibliografía y webgrafía

Enumera las fuentes consultadas durante el desarrollo, incluyendo documentación oficial, tutoriales, artículos técnicos y otros recursos relevantes.

10 - Anexos

Incluye documentación complementaria como el código fuente de los microservicios, capturas de pantalla, esquemas de datos y cualquier otro material relevante no incluido en los apartados anteriores.

3 Descripción general del proyecto

3.1 Objetivos

Con este proyecto se ha buscado, a nivel educativo, afianzar y demostrar las nociones adquiridas a lo largo de los módulos del grado, además de ampliar este conocimiento, a nivel tanto personal como profesional. La metodología que se ha seguido para esto ha sido analizar los conocimientos aplicables al desarrollo e investigar nuevos, aplicando en partes del desarrollo los que fuesen apropiados.

Por otro lado, se ha perseguido el objetivo de crear una aplicación que es útil tanto para uso propio como para una posible futura publicación. Además, esta puede servir como base para futuros desarrollos, gracias a la escalabilidad y versatilidad de las tecnologías y métodos elegidos.

3.2 Cuestiones metodológicas

Para la realización de este proyecto se ha utilizado un **modelo en cascada con retroalimentación**, una versión adaptada del modelo clásico en cascada que permite volver a fases anteriores si se detectan errores o si se desea mejorar alguna decisión previa.

Esta metodología es especialmente útil en entornos académicos o en proyectos individuales, ya que proporciona una estructura clara y ordenada del desarrollo, al mismo tiempo que ofrece la flexibilidad necesaria para corregir o mejorar sin comprometer el conjunto.

El flujo de trabajo se ha dividido en las siguientes etapas:

1. Análisis

Se definieron los requisitos funcionales y no funcionales del sistema. Se estudió qué datos debía manejar la aplicación (juegos, partidas, usuarios), qué interacciones debía ofrecer, y cómo se organizaría el sistema en microservicios.

2. Diseño

Se elaboraron los diagramas de base de datos (modelo entidad-relación), los wireframes de las pantallas principales y la arquitectura del sistema, distribuyendo los servicios por áreas (autenticación, inventario, catálogo, partidas).

3. Implementación

Se desarrolló el backend modularizado usando NodeJS (con Express y Fastify), así como el frontend con Expo para React Native. También se configuró Keycloak como sistema de autenticación centralizado.

4. Pruebas

Se realizaron pruebas tanto unitarias como funcionales para asegurar la calidad del sistema. Se detectaron errores que motivaron volver a fases anteriores para ajustar modelos de datos, endpoints o flujos de validación.

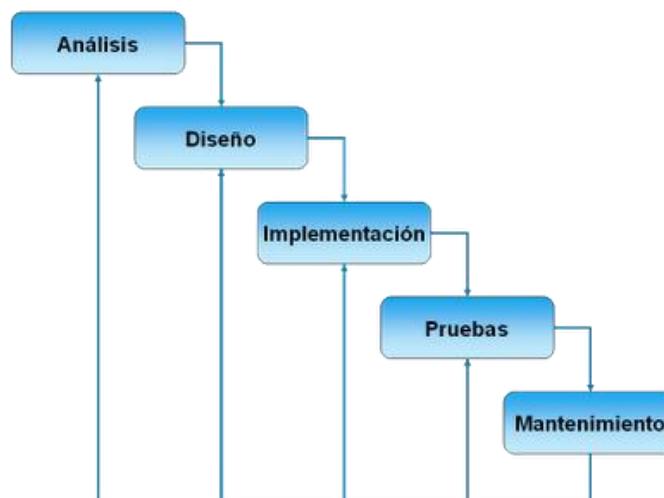
5. Despliegue

Se preparó un sistema automatizado de despliegue local mediante un script en PowerShell que levanta todos los microservicios con Docker y arranca el cliente Expo.

6. Documentación

Se redactó esta memoria y los manuales asociados, facilitando la entrega, comprensión y reutilización futura del proyecto.

Gracias a la posibilidad de retroalimentación en cada fase, se logró un desarrollo más estable y robusto, a pesar del tiempo limitado disponible para su realización.



3.3 Entorno de trabajo (tecnologías de desarrollo y herramientas)

El proyecto se ha desarrollado siguiendo un **modelo distribuido basado en microservicios**, con separación clara entre cliente, servidores de negocio y servicios de autenticación. Cada microservicio puede desplegarse de forma independiente y se comunica con los demás mediante peticiones HTTP autenticadas mediante tokens.

El frontend actúa como **cliente multiplataforma** que consume distintas APIs las cuales constan de un catálogo de juegos, un inventario personal, un registro de partidas y un control de usuarios con un sistema de autenticación centralizado con **Keycloak**.

Las herramientas y tecnologías más destacables que se han utilizado a lo largo del desarrollo han sido:

JavaScript: Utilizado como lenguaje principal en todo el desarrollo. Es el lenguaje base de React Native y de NodeJS, fácil de integrar con librerías de UI y muy extendido en el desarrollo web.

TypeScript: Utilizado en el frontend y en la mayoría de los microservicios como mejora a JavaScript. Aporta tipado estático sobre este, lo que mejora la mantenibilidad, escalabilidad y ayuda a prevenir errores en tiempo de desarrollo.

Node.js: Entorno de ejecución para JavaScript en el backend. Se ha utilizado como base para los microservicios, dada su eficiencia y ecosistema.

Express: Entorno de trabajo para el desarrollo de APIs en Node.js. Se ha utilizado exclusivamente en el microservicio de autenticación (IAM), ya que es un framework sencillo y muy flexible. Para este microservicio, la prioridad era una configuración clara y directa, ya que toda la lógica gira en torno a proxy de tokens y validaciones simples.

Fastify: Framework alternativo a Express, enfocado en el rendimiento. Se ha utilizado en los microservicios de catálogo, inventario y partidas, por ser un framework más moderno, orientado al rendimiento, con soporte nativo para tipos y un ecosistema más optimizado para APIs REST. Su sistema de plugins y su integración con herramientas modernas como Zod lo hacen ideal para servicios más complejos.

Keycloak: Plataforma de gestión de identidades y autenticación. Se utiliza para el inicio de sesión, control de acceso y gestión de roles de los usuarios. Se ha configurado para trabajar con PostgreSQL como base de datos externa.

MongoDB: Base de datos NoSQL orientada a documentos. Permite almacenar datos estructurados en formato JSON de forma flexible sin depender de una estructura fija en el modelo.

PostgreSQL: Base de datos relacional utilizada por Keycloak para almacenar de forma persistente los datos de autenticación, usuarios, roles, sesiones activas y configuración. Frente a MongoDB, permite una mayor consistencia.

React Native con Expo: Framework para el desarrollo de aplicaciones móviles multiplataforma (Web/iOS/Android) usando TypeScript. Al estar basado en React, framework web, permite emplear las bases de este para el desarrollo móvil. Expo es un conjunto de herramientas y servicios para React Native que simplifica el desarrollo y pruebas.

Docker: Plataforma de contenedores utilizada para encapsular y ejecutar los microservicios, bases de datos y Keycloak de forma aislada.

Docker Compose: Herramienta para definir y ejecutar múltiples contenedores de Docker con un solo archivo de configuración.

PowerShell: Lenguaje de scripting para el sistema operativo Windows utilizado para automatizar el entorno de desarrollo local, levantando los microservicios y el frontend con un único script.

Zod: Librería de validación de esquemas en JavaScript/TypeScript, utilizada para validar las entradas del backend.

zod-to-json-schema: Conversor de esquemas Zod a esquemas compatibles con OpenAPI (Swagger), facilitando la documentación automática de la API.

Vitest: Herramienta moderna para la ejecución de pruebas unitarias y de integración en proyectos Node.js.

Visual Studio Code: Entorno de desarrollo (IDE) principal utilizado para escribir el código tanto del frontend como del backend.

Git y GitHub: Sistema de control de versiones y repositorio remoto. Permite el control del código, la colaboración y el seguimiento del progreso del proyecto.

Figma: Plataforma de diseño de interfaces y prototipado. Se utilizó para crear wireframes y definir el diseño visual antes de la implementación.

4 Descripción general del producto

4.1 Visión general del sistema: límites del sistema, funcionalidades básicas, usuarios y/o otros sistemas con los que pueda interactuar.

LudoKeeper es una aplicación móvil que permite a los usuarios controlar todo lo que abarca a su relación con los juegos de mesa. El sistema se ha diseñado con una arquitectura de microservicios que separa claramente los distintos dominios de la aplicación:

- Inventario del usuario: permite añadir juegos (propios o desde un catálogo), editarlos, organizarlos y eliminarlos.
- Registro de partidas: permite guardar información de cada sesión de juego (fecha, jugadores, duración, observaciones...).
- Explorador de juegos: catálogo público con información general de juegos de mesa (categoría, mecánicas, editorial, número de jugadores...).
- Autenticación centralizada: gestión de identidad y roles de usuario a través de Keycloak.

Los límites del sistema se abarcan en:

- Es una aplicación cliente-servidor distribuida, donde el cliente es una app móvil desarrollada en React Native con Expo.
- El sistema está diseñado para ejecutarse de forma local (modo desarrollo) utilizando Docker y Docker Compose.
- La autenticación y control de acceso se realiza a través de un servidor Keycloak, que gestiona la seguridad de las API privadas mediante tokens JWT.
- El sistema está preparado para evolucionar y escalar, permitiendo alojarlo en servidores remotos o plataformas cloud si se desea en un futuro.

Las funcionalidades básicas que se pueden abstraer serían:

- Autenticación de usuarios: Registro, inicio de sesión, cierre de sesión. Control de roles (usuario o administrador). Gestión de tokens JWT a través de Keycloak.
- Gestión de inventario: Añadir juegos desde un catálogo público o crearlos manualmente. Modificar datos personalizados (nombre, descripción, número de jugadores...). Eliminar juegos del inventario. Filtrar y buscar por nombre, número de jugadores, fuente de datos, etc.
- Registro de partidas: Guardar partidas jugadas con datos como fecha, duración, jugadores, resultado, comentarios... Vincular cada partida a un juego del inventario. Consultar el historial de partidas por juego o por fecha.

- Explorar juegos: Consultar una base de datos pública de juegos de mesa. Filtrar por categoría, mecánica, editorial, año, número de jugadores... Visualizar detalles de cada juego.
- Herramientas adicionales (complementarias): Temporizador y cronómetro de partida. Simulador de dados (con animaciones y elección de tipo de dado).

El sistema está diseñado para un único tipo de usuario principal: el jugador registrado.

Este usuario tiene acceso, previa autenticación, a su inventario, sus partidas y a las herramientas y funcionalidades de la aplicación.

Adicionalmente, se contempla el rol administrador, usado internamente para restringir el acceso a operaciones críticas de la aplicación (como añadir o modificar juegos públicos del catálogo). Este control se gestiona desde Keycloak mediante asignación de roles.

Los sistemas con los que interactúa son:

- Frontend: aplicación móvil desarrollada en React Native con Expo.
- Backend distribuido: compuesto por varios microservicios escritos en Node.js.
- Keycloak: servicio de autenticación independiente que gestiona sesiones y roles de usuario.
- Base de datos MongoDB: almacenamiento principal de datos de inventario, partidas y catálogo.
- Base de datos PostgreSQL: persistencia de usuarios y configuración del sistema de autenticación (Keycloak).

El sistema se ha diseñado como multiplataforma, y puede ejecutarse en cualquier sistema operativo al estar completamente contenerizado mediante Docker. La parte frontal funciona tanto en Windows como en Android y, pese a no haberse podido probar, en iOS. En Windows y Linux se podría exportar de forma nativa mediante plugins.

4.2 Descripción breve de métodos, técnicas o arquitecturas(m/t/a) utilizadas.

En el desarrollo del frontend se ha utilizado una arquitectura modular basada en separación de responsabilidades, con el objetivo de mantener el proyecto escalable, mantenable y fácilmente *testable*.

La organización se ha estructurado en torno a módulos funcionales, agrupando archivos y lógica según su dominio (autenticación, inventario, partidas, herramientas, catálogo...). Esta aproximación permite aislar las funcionalidades y reutilizar componentes y lógica con facilidad. Esta arquitectura está pensada para proyectos en evolución, con múltiples microservicios, funcionalidades y vistas. Al separar claramente la lógica de red, el estado, la presentación y los componentes, se facilita la escalabilidad, la detección de errores, y la colaboración entre desarrolladores.

Para el desarrollo de los microservicios backend he optado por una arquitectura modular y desacoplada, basada en buenas prácticas de separación de responsabilidades. Cada microservicio (inventario, catálogo y partidas) es una aplicación independiente escrita con Node.js y estructurada de forma coherente y escalable. Esta estructura promueve alta mantenibilidad gracias a la separación de capas; escalabilidad, facilitando añadir nuevas funcionalidades o microservicios sin romper los existentes; reutilización y testeo, al tener claramente definida la lógica de negocio y los servicios; y documentación automática gracias a la integración de Swagger con zod-to-json-schema.

Cada microservicio sigue una estructura por capas que diferencia entre rutas, controladores, servicios, validaciones y modelos de datos. Esta separación permite que el código sea más fácil de entender, extender y probar. Además, se ha aplicado validación centralizada con Zod, lo que permite unificar la validación de entrada de datos, la generación de tipos TypeScript y la creación automática de la documentación OpenAPI.

Se han elegido tecnologías adaptadas a cada necesidad. Por ejemplo, Express se ha utilizado en el microservicio de autenticación por su simplicidad y amplia compatibilidad conmiddlewares existentes, mientras que Fastify ha sido utilizado en los microservicios de catálogo, inventario y partidas por su mejor rendimiento, integración directa con Zod y soporte nativo para la generación de documentación Swagger.

Todos los servicios están contenedezados con Docker y orquestados con Docker Compose, lo que permite desplegar el ecosistema completo del proyecto con un único comando y asegurar la portabilidad entre entornos. Se ha creado un script automatizado en PowerShell para facilitar el arranque del sistema en nuevos equipos, incluyendo la verificación de herramientas necesarias, el arranque de servicios y la inicialización del cliente Expo.

Para garantizar la calidad del software, se han implementado pruebas unitarias y de integración en los microservicios mediante el framework Vitest. En los servicios que emplean MongoDB se ha utilizado MongoMemoryServer, lo que permite ejecutar las pruebas contra una base de datos en memoria, sin necesidad de dependencias externas. Esto asegura una retroalimentación rápida durante el desarrollo y evita errores en fases tempranas.

4.3 Despliegue de la aplicación indicando plataforma tecnológica, instalación de la aplicación y puesta en marcha

La aplicación ha sido desarrollada y probada en entornos basados en **Windows 11 y Manjaro Linux**, utilizando como entorno de ejecución Node.js para los microservicios y Expo CLI para el frontend multiplataforma. Todos los servicios backend están contenerizados mediante **Docker** y se gestionan mediante **Docker Compose**. Para la autenticación se emplea **Keycloak**, que se ejecuta en un contenedor propio junto a una base de datos **PostgreSQL**. El cliente funciona tanto en emuladores Android/iOS como en dispositivos físicos conectados mediante Expo Go y en cualquier navegador web con soporte de JavaScript.

Para facilitar el despliegue del sistema en nuevos entornos, se ha desarrollado un **script automatizado en PowerShell** que realiza las siguientes acciones:

- Verifica que las herramientas necesarias estén instaladas y avisa en caso contrario.
- Comprueba que Docker Desktop esté en ejecución y lo inicia si es necesario.
- Renombra automáticamente los archivos .env.test a .env si no existe el archivo de entorno principal.
- Lanza los contenedores de cada microservicio (IAM, keycloak, catalog, inventory, match) con docker-compose up --build.
- Instala las dependencias del frontend React Native si no existen (npm install).
- Inicia el cliente Expo para ejecutar la aplicación en modo desarrollo.

Este proceso garantiza que un entorno limpio pueda ejecutar todo el sistema en pocos minutos sin intervención manual. Además, el uso de .env.test y variables centralizadas asegura que el despliegue sea coherente entre entornos de desarrollo, pruebas y producción manteniendo la seguridad.

Una vez completado el script de despliegue, el entorno queda completamente funcional:

- El backend responde a través de endpoints protegidos con JWT emitidos por Keycloak.
- El frontend puede iniciar sesión, comunicarse con los microservicios y consumir todos los datos.
- La interfaz de usuario se abre automáticamente en Expo Developer Tools, permitiendo acceder desde un emulador o mediante escaneo del QR desde un dispositivo móvil.

Todo el ecosistema está diseñado para ser portable, automatizable y reproducible, facilitando el trabajo en equipo y la puesta en marcha del proyecto en entornos nuevos o distribuidos.

5 Planificación y presupuesto

5.1 Planificación

El desarrollo del proyecto se ha organizado a lo largo de 3 meses (marzo, abril y mayo), siguiendo una planificación flexible, pero con hitos definidos. A continuación, se describe el reparto de tareas aproximado por semanas planificado al principio del desarrollo. Debido a la complejidad y la necesidad de investigación y aprendizaje continuo se ha acabado prolongando a junio.

Marzo

Semana 1–2: Diseño del proyecto, definición de requisitos funcionales, planificación general.

Semana 3–4: Configuración de Keycloak y sistema de autenticación IAM, pruebas con JWT y Docker.

Abril

Semana 1: Desarrollo del microservicio de inventario (estructura, validación, controladores, Swagger).

Semana 2: Implementación de pruebas automatizadas (Vitest + MongoMemoryServer), depuración y ajustes.

Semana 3: Desarrollo del microservicio de catálogo (estructura similar, con control de roles).

Semana 4: Implementación del microservicio de partidas (match).

Mayo

Semana 1: Desarrollo del frontend con React Native y Expo. Pantallas de login, registro e inventario.

Semana 2: Herramientas internas (temporizador, datos, buscador, filtros).

Semana 3: Integración frontend–backend, autenticación y gestión de partidas.

Semana 4: Despliegue, redacción de memoria y pruebas finales.

El flujo de trabajo ha seguido una metodología iterativa e incremental, con revisiones continuas para adaptarse a los cambios de prioridad y detectar errores en fases tempranas. Se ha utilizado control de versiones con Git y desarrollo modular para facilitar la integración progresiva.

5.2 Presupuesto

A continuación, se desglosa el coste estimado del proyecto, considerando el tiempo de desarrollo y los recursos técnicos utilizados:

- Coste de desarrollo: 50 horas estimadas (excluyendo tiempo de aprendizaje ya que lo considero como beneficio propio) a 10€/h (valor de mercado para desarrolladores junior en España) da 500€.

- Coste de software: 0€ ya que se ha usado software gratuito y pruebas temporales gratuitas de versiones de pago.
- Coste de hardware: Ordenador y teléfono personal de desarrollo sin inversión adicional.
- Coste de hosting: Durante el desarrollo, se ha utilizado entorno local con contenedores Docker.
- Otros: Formación autodidacta y documentación sin coste directo, pero entre tiempo y cursos se podría valorar en 100€.

Presupuesto total estimado del proyecto: **600€**

6 Documentación Técnica: análisis, diseño, implementación y pruebas.

6.1 Especificación de requisitos

A continuación, se enumeran los requisitos funcionales del sistema, agrupados por microservicio o módulo de funcionalidad. Cada uno de estos requisitos define una acción que el usuario puede realizar o una característica esperada del sistema.

1. Requisitos funcionales del sistema de autenticación (IAM + Keycloak)

RF01: El usuario podrá registrarse proporcionando nombre, apellidos, email, *username* y contraseña.

RF02: El usuario podrá iniciar sesión con sus credenciales y obtener un token JWT.

RF03: El sistema permitirá renovar el token de acceso mediante el *refresh token*.

RF04: El backend podrá validar el JWT recibido para proteger los endpoints privados.

RF05: El administrador podrá asignar o revocar roles (por ejemplo, admin) a través del microservicio IAM.

RF06: El sistema gestionará los errores de autenticación (credenciales incorrectas, expiración, token inválido...).

2. Requisitos del microservicio de inventario

RF07: El usuario podrá consultar todos los juegos registrados en su inventario personal.

RF08: El usuario podrá añadir juegos personalizados o importarlos desde el catálogo.

RF09: El usuario podrá editar los datos de un juego de su inventario (nombre, descripción, jugadores, etc.).

RF10: El usuario podrá eliminar juegos de su inventario.

RF11: El sistema permitirá aplicar filtros en las búsquedas (por número de jugadores, fuente, etc.).

3. Requisitos del microservicio de catálogo

RF12: El sistema ofrecerá una API pública para consultar juegos de mesa registrados en el catálogo.

RF13: Los juegos del catálogo incluirán nombre, imagen, editorial, mecánicas, categorías, año, etc.

RF14: El usuario podrá filtrar juegos por mecánica, número de jugadores, año o editorial.

RF15: Solo los usuarios con rol admin podrán añadir o modificar juegos del catálogo.

4. Requisitos del microservicio de partidas

RF16: El usuario podrá registrar una partida indicando los juegos utilizados, jugadores participantes, fecha y resultados.

RF17: El usuario podrá consultar el historial de partidas jugadas.

RF18: El sistema permitirá filtrar partidas por juego, fecha o jugadores.

RF19: El usuario podrá editar o eliminar partidas registradas.

5. Requisitos del frontend (aplicación React Native)

RF20: El usuario podrá iniciar sesión y registrarse desde la aplicación.

RF21: El usuario podrá visualizar y gestionar su inventario.

RF22: El usuario podrá registrar nuevas partidas desde la app.

RF23: El usuario podrá acceder a herramientas útiles como lanzador de datos, temporizador y cronómetro.

RF24: El usuario podrá buscar juegos dentro del catálogo general.

RF25: El diseño adaptará su apariencia a modo claro y oscuro según preferencias del usuario.

RF26: El usuario recibirá retroalimentación visual y mensajes de error en formularios.

RF27: El sistema manejará errores de red o sesión expirada, proponiendo volver a iniciar sesión.

A continuación, se enumeran los requisitos no funcionales del sistema, agrupados por categoría técnica o de calidad. Cada uno de estos requisitos define una característica que afecta al comportamiento general, la calidad, o la experiencia de uso del sistema.

1. Requisitos de seguridad

RNF01: La comunicación entre el cliente móvil y los microservicios deberá realizarse mediante HTTPS en entornos de producción.

RNF02: Todos los endpoints protegidos del backend deberán requerir un token JWT válido emitido por Keycloak.

RNF03: Las operaciones sensibles, como la creación o modificación de juegos del catálogo, solo estarán disponibles para usuarios con rol admin.

2. Requisitos de escalabilidad y arquitectura

RNF04: La arquitectura del sistema deberá permitir escalar cada microservicio de forma independiente.

RNF05: El microservicio del catálogo deberá estar optimizado para manejar un gran volumen de datos de forma eficiente.

3. Requisitos de portabilidad

RNF06: El sistema deberá poder ejecutarse en cualquier equipo que tenga Docker Desktop y Node.js instalado.

RNF07: La aplicación móvil deberá ser compatible tanto con dispositivos Android como iOS mediante Expo, además de web.

4. Requisitos de mantenibilidad

RNF08: El código deberá estar estructurado por capas y módulos, siguiendo buenas prácticas de separación de responsabilidades.

RNF09: Las validaciones de datos deberán centralizarse con Zod para facilitar su mantenimiento y reutilización.

RNF10: El sistema deberá incluir documentación técnica generada automáticamente mediante Swagger y zod-to-json-schema.

5. Requisitos de fiabilidad

RNF11: Los microservicios backend deberán estar cubiertos por pruebas unitarias y de integración automatizadas.

RNF12: Las pruebas no deberán depender de servicios externos y deberán ejecutarse en bases de datos en memoria (MongoMemoryServer).

RNF13: El sistema deberá contar con un script automatizado para facilitar su despliegue y minimizar errores humanos.

6. Requisitos de usabilidad

RNF14: La interfaz de la aplicación deberá ser clara, intuitiva y adaptada al contexto de juegos de mesa.

RNF15: El diseño visual deberá incluir modo claro y oscuro, y proporcionar retroalimentación visual en formularios y errores.

7. Requisitos de compatibilidad

RNF16: El sistema deberá funcionar correctamente en distintos navegadores y versiones del sistema operativo donde se ejecute Docker.

RNF17: El cliente Expo deberá funcionar en múltiples plataformas sin necesidad de cambios en el código base.

6.2 Análisis del sistema

El análisis del sistema parte de los requisitos funcionales previamente definidos, detallando con mayor profundidad las operaciones permitidas, las entidades implicadas y las validaciones necesarias. El sistema se estructura en torno a cuatro grandes dominios funcionales: autenticación, inventario, catálogo y partidas.

A continuación, se describe el análisis de cada uno de ellos:

Autenticación y gestión de usuarios (IAM + Keycloak)

- Alta de usuario: El usuario debe proporcionar nombre, apellidos, email, username y contraseña. Validaciones: campos obligatorios, formato de email, contraseña con complejidad mínima.
- Login: El usuario se autentica con username y contraseña, obteniendo un token JWT y un refresh token. Validaciones: credenciales correctas, usuario habilitado.
- Renovación de token: Si el token ha expirado, puede renovarse mediante el refresh token desde el frontend.
- Roles: La gestión de permisos se realiza mediante roles (user, admin) definidos en Keycloak. Los endpoints protegidos utilizan middleware para validar el rol y los permisos asociados.

Inventario personal de juegos

- Consulta del inventario: El usuario autenticado podrá obtener una lista de todos los juegos guardados en su inventario. Soporte para filtros por tipo de juego (custom o catalog), número de jugadores, etc.
- Alta de juego personalizado: Permite añadir un juego desde cero, especificando nombre, descripción, jugadores, duración, imagen y fecha de adquisición. Validaciones: campos obligatorios, rango de jugadores válido, imagen en línea opcional.
- Importación desde el catálogo: Permite añadir juegos existentes del catálogo al inventario del usuario. Validaciones: gameld del catálogo debe existir.
- Edición de juego: El usuario puede editar los datos personalizados de cualquier juego de su inventario. Restricción: solo sobre sus propios juegos.
- Eliminación de juego: El usuario puede eliminar cualquier juego de su inventario. Confirmación necesaria en el frontend.

Catálogo general de juegos

- Consulta pública del catálogo: Cualquier usuario puede acceder a la lista de juegos, filtrando por nombre, editorial, año, mecánicas o categorías.

- Alta de juego (solo admin): Permite registrar juegos nuevos en el catálogo. Validaciones: nombre obligatorio, número de jugadores, año, mecánicas/categorías predefinidas.
- Edición de juego del catálogo (solo admin): El administrador puede actualizar cualquier campo de un juego registrado.
- Consulta de detalles: Cualquier usuario puede acceder a los detalles completos de un juego específico.

Registro de partidas jugadas

- Registro de partida: El usuario puede guardar una partida, indicando: juego utilizado, jugadores, fecha, duración y resultados. Validaciones: mínimo un jugador, juego debe existir (propio o de catálogo).
- Historial de partidas: Listado de partidas jugadas por el usuario. Posibilidad de filtrar por juego, fecha o participantes.
- Edición o eliminación de partida: Se pueden actualizar los datos registrados o eliminar una partida existente.

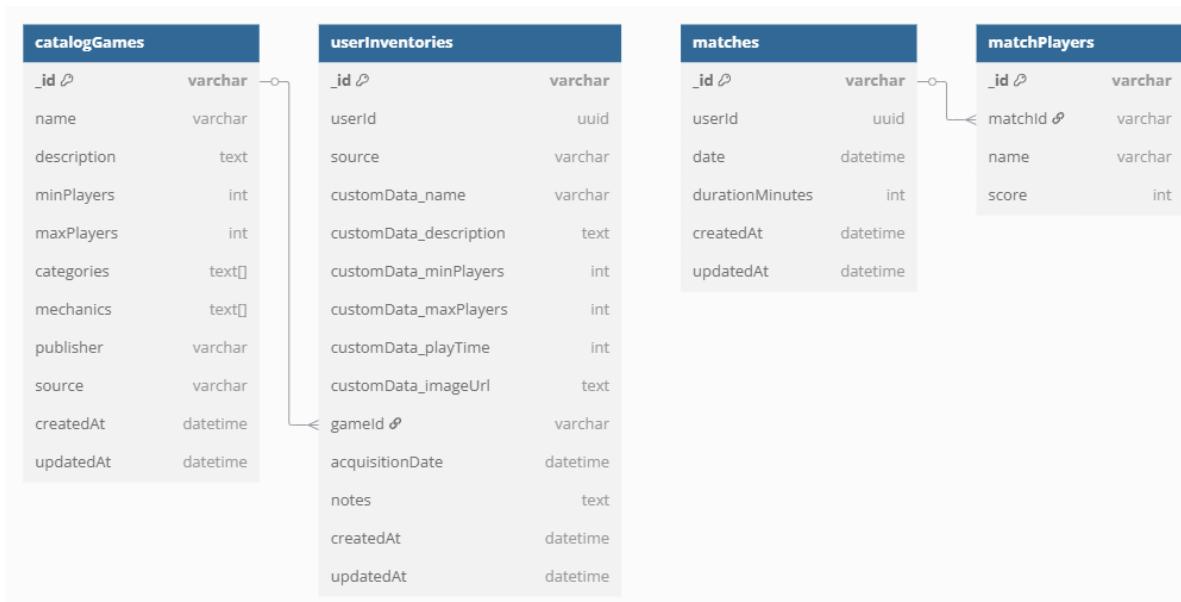
6.3 Diseño del sistema:

6.3.1 Diseño de la Base de Datos

El diseño de las bases de datos se ha llevado a cabo siguiendo una transición lógica desde un modelo Entidad-Relación (E/R) a un modelo relacional adaptado al paradigma NoSQL, dado que los microservicios utilizan MongoDB como sistema gestor de base de datos. Pese a su naturaleza flexible, se ha definido un esquema lógico claro para cada colección con el objetivo de garantizar la validación de datos, la escalabilidad y la interoperabilidad entre microservicios.

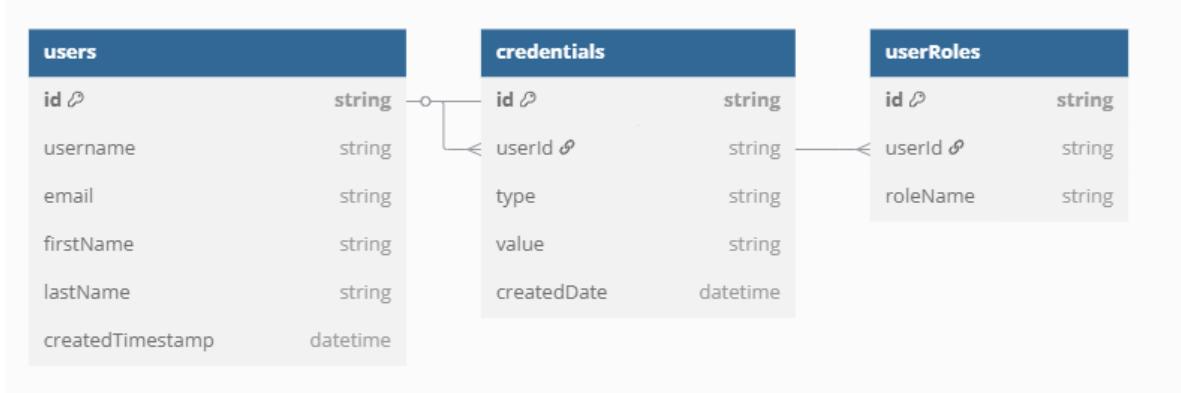
El sistema cuenta con cuatro bases de datos diferenciadas, una por cada microservicio:

- Base de datos del microservicio de inventario: Gestiona los juegos de mesa que cada usuario tiene en su colección personal. Cada documento puede referenciar a un juego del catálogo o incluir datos personalizados.
- Base de datos del microservicio de Catálogo: Contiene la información global sobre los juegos de mesa, accesible públicamente para todos los usuarios. Cada entrada incluye nombre, editorial, número de jugadores, mecánicas, categorías y una imagen.
- Base de datos del microservicio de Partidas: Permite registrar partidas jugadas por los usuarios, asociadas a un juego del catálogo o del inventario. También se almacenan los jugadores, duración, fecha y resultado de la partida.



1Las tablas no se relacionan directamente, pero si lo hacen los microservicios

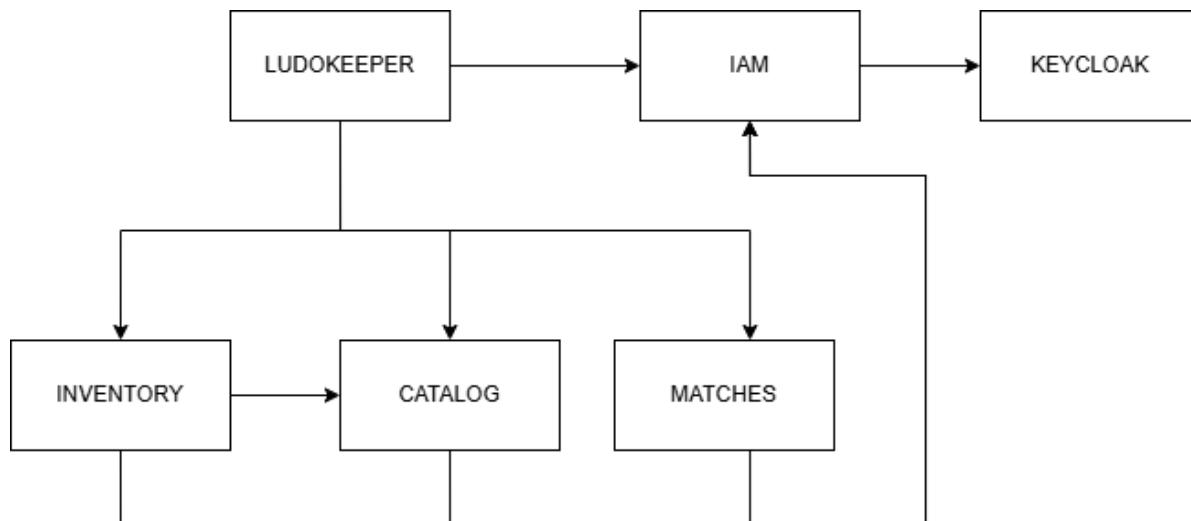
- Base de datos del microservicio de IAM (Keycloak): Aunque este microservicio utiliza una base de datos gestionada por Keycloak, forma parte del sistema y es clave en la autenticación y autorización. Se monta en PostgreSQL para asegurar la integridad de los datos.



2Aproximación de las tablas que genera Keycloak con la configuración actual

6.3.2 Diseño de la Interfaz de usuario.

6.3.3 Diseño de la Aplicación.



6.4 Implementación:

6.4.1 Entorno de desarrollo.

El entorno de desarrollo ha sido cuidadosamente configurado para garantizar una experiencia fluida, reproducible y profesional durante todo el ciclo de vida del proyecto. Se ha optado por tecnologías modernas, herramientas ampliamente utilizadas en la industria y una configuración homogénea para todos los microservicios y el frontend.

El proyecto está organizado bajo una arquitectura de microservicios con una raíz común del repositorio que incluye:

`backend/` → Contiene los microservicios de **inventario, catálogo, partidas y autenticación (IAM)**.

`keycloak/` → Contiene la configuración personalizada de Keycloak como proveedor de identidad.

`frontend/ludokeeper/` → Aplicación móvil desarrollada con React Native y Expo.

Cada servicio es independiente, con su propio Dockerfile, archivo docker-compose.yml, y su entorno de desarrollo definido.

En cuanto a tecnologías, estas son las más destacadas:

Tecnologías Backend (microservicios)

Node.js – Entorno de ejecución JavaScript/TypeScript

TypeScript – Tipado estático para JS

Express – Framework backend usado en el microservicio de autenticación

Fastify – Framework backend ligero y eficiente usado en inventario, catálogo y partidas

Zod – Validación de esquemas de datos

Swagger (Fastify Swagger) – Documentación automática de APIs

zod-to-json-schema – Conversión de esquemas Zod a Swagger

MongoDB – Base de datos NoSQL para inventario, catálogo y partidas

PostgreSQL – Base de datos usada por Keycloak

Tecnologías de autenticación

Keycloak – Servidor de identidad y autorización (OpenID Connect, OAuth 2.0)

JWT (JSON Web Tokens) – Autenticación entre servicios y usuarios

Frontend y mobile

React Native – Desarrollo de apps móviles multiplataforma

Expo – Entorno simplificado para React Native

React Navigation – Navegación estructurada en la app

React Hook Form + Yup/Zod – Manejo de formularios y validación

Axios – Cliente HTTP para conexión con APIs

Testing

Vitest – Testing unitario y de integración en backend

MongoMemoryServer – Simulación de base de datos Mongo en memoria para tests

DevOps y herramientas

Docker – Contenerización de microservicios

Docker Compose – Orquestación local de todos los servicios

Visual Studio Code – Editor de código principal

Git + GitHub – Control de versiones y colaboración

PowerShell – Scripting de automatización (build.ps1)

6.4.2 Estructura del código.

Estructura principal



```
.  
|   └── backend  
|       ├── IAM/          # Servicio de autenticación (con proxy a Keycloak)  
|       ├── inventory/    # Gestión del inventario de juegos del usuario  
|       ├── catalog/      # Catálogo global de juegos (fuente pública o manual)  
|       └── match/        # Registro y gestión de partidas  
|   └── keycloak/        # Configuración y realm para Keycloak  
└── frontend/  
    └── ludokeeper/      # App móvil Expo  
└── public_key.pem     # Clave pública JWT compartida  
└── run.ps1            # Script para iniciar el proyecto (Windows)  
└── run.sh             # Script para iniciar el proyecto (Linux/macOS)  
└── README.md          # Este archivo
```

Frontend

```
src/  
├── api/                # Servicios para conectarse a los microservicios  
│   ├── auth.ts  
│   ├── catalog.ts  
│   ├── inventory.ts  
│   └── match.ts  
├── components/  
│   ├── form/            # Inputs, selects, formularios reutilizables  
│   ├── games/           # GameCard, barra de búsqueda y filtros  
│   └── layouts/         # Modales: detalles, formularios, etc.  
├── config/              # URLs de APIs  
├── constants/           # Íconos, navegación...  
├── hooks/               # useInventory, useAuthForm, useMatch...  
├── store/               # authStore, themeStore (Zustand)  
├── styles/              # Estilos globales, colores, tipografías  
├── utils/               # Funciones auxiliares como getGameInfo  
├── validations/         # Validaciones con Yup  
├── types/               # Tipado global  
└── app/  
    ├── (auth)/          # Login y registro  
    ├── (root)/          # Pantallas principales: inventario, catálogo, menú  
    ├── (tools)/          # Herramientas: datos, temporizador, cronómetro  
    └── (other)/          # Ajustes, acerca de
```

Backend (ejemplo inventory)

```

src/
├── controllers/          # Lógica de los endpoints (controladores HTTP)
│   └── inventoryController.ts
├── middlewares/         # Middlewares como autenticación JWT
│   └── auth.ts
├── models/               # Modelos Mongoose para MongoDB
│   └── userInventory.ts
├── routes/               # Rutas Fastify y su documentación Swagger
│   └── inventoryRoutes.ts
├── schemas/              # Validaciones Zod + conversión a JSON Schema
│   └── inventorySchemas.ts
├── services/             # Lógica de negocio desacoplada
│   └── inventoryService.ts
└── test/
    └── inventory/
        ├── helpers/      # Setup de entorno de pruebas y mocks
        │   ├── authMock.ts
        │   ├── db.ts
        │   └── setupTestApp.ts
        ├── inventory.test.ts      # Pruebas de integración (rutas)
        └── inventoryService.test.ts # Pruebas unitarias (servicios)
├── types/                # Tipado global o personalizado para Fastify
│   └── index.ts
├── .env                  # Variables de entorno
├── Dockerfile            # Imagen del microservicio
├── docker-compose.yml     # Orquestación (inventario + MongoDB)
├── package.json / tsconfig.json / vitest.config.ts
├── public_key.pem        # Clave pública para verificación de JWT
├── setup.ts              # Setup global de tests con MongoMemoryServer
└── README.md             # Documentación del microservicio

```

Backend (IAM)

```

src/
├── config/          # Configuración de Keycloak y sesión
│   └── keycloak.js
├── controllers/    # Lógica de endpoints (register/login)
│   └── auth.controller.js
├── middlewares/    # Validación de datos con Joi
│   └── validate.js
├── routes/          # Definición de rutas de autenticación y roles
│   ├── auth.routes.js
│   └── roles.routes.js
├── services/         # Acceso a la API de Keycloak desde Node.js
│   └── keycloak.service.js
├── validators/       # Esquemas Joi
│   └── auth.validator.js
├── docs/             # Swagger JSON para documentación OpenAPI
│   └── swagger.json
├── server.js          # Punto de entrada de la app
├── .env               # Variables de entorno
├── Dockerfile          # Imagen de Docker
├── docker-compose.yml  # Orquestación de contenedor
└── README.md          # Documentación del microservicio

```

6.4.3 Cuestiones de diseño e implementación reseñables.

6.5 Pruebas.

Durante el desarrollo del proyecto se han llevado a cabo diferentes tipos de pruebas, tanto manuales como automatizadas, con el objetivo de garantizar la fiabilidad, la calidad del código y el correcto funcionamiento del sistema en su conjunto.

Las pruebas se han clasificado en las siguientes categorías:

Pruebas unitarias

Se han implementado pruebas unitarias para los servicios de los microservicios backend, con el objetivo de comprobar de forma aislada la lógica de negocio. Estas pruebas se han desarrollado usando el framework Vitest, que permite una integración fluida con TypeScript y ofrece resultados rápidos y claros.

Para evitar depender de una base de datos real, se ha utilizado MongoMemoryServer, una herramienta que simula una instancia en memoria de MongoDB, permitiendo realizar pruebas rápidas y sin necesidad de conexión a un entorno externo.

Ejemplo de servicios probados: Crear, actualizar y eliminar juegos del inventario. Validación de esquemas Zod. Conversión entre juegos del catálogo y personalizados

Pruebas de integración automatizadas

Además de los servicios, se han probado los endpoints completos (rutas + lógica) para asegurarse de que las respuestas HTTP son correctas, y que la integración entre controladores, servicios y base de datos se comporta como se espera.

Estas pruebas también se han realizado con Vitest, lanzando peticiones reales simuladas con herramientas como supertest o llamadas a los servidores Fastify en modo test.

Aspectos cubiertos: Autenticación con tokens válidos e inválidos. Acceso protegido a rutas privadas. Filtros, parámetros y validaciones de entradas. Comprobación de errores y status codes.

Pruebas manuales en frontend y Postman

Durante el desarrollo de la aplicación móvil, se han realizado pruebas manuales en dispositivos físicos y emuladores utilizando Expo Go. Estas pruebas han permitido verificar la experiencia de usuario, la navegación entre pantallas, la persistencia del estado, y la comunicación con los microservicios vía API.

Paralelamente, se han utilizado colecciones de Postman para validar el comportamiento de los endpoints de cada microservicio de manera manual y estructurada, especialmente útil para verificar: Flujo de login y obtención de tokens JWT. Registro y modificación de juegos. Visualización y registro de partidas. Pruebas con filtros, búsquedas y errores.

Pruebas de seguridad y validación

Se han tenido en cuenta aspectos básicos de seguridad en los microservicios, como: Validación estricta de entradas con Zod. Verificación de autenticación y roles mediante JWT. Pruebas de rutas sin token, con token mal formado o con permisos insuficientes. Comprobación de errores controlados en el backend

Pruebas multiplataforma (Expo)

El frontend desarrollado con Expo se ha probado en diferentes entornos: Simulador Android local. Dispositivo físico Android (Expo Go). Navegador web (modo desarrollo de Expo). Estas pruebas han permitido validar la correcta visualización, el funcionamiento de los formularios, el modo claro/oscuro y el acceso a funcionalidades internas como modales o navegación entre pantallas.



IES RIBERA DE CASTILLA

LUDOKEEPER

7 Manuales de usuario

7.1 *Manual de usuario*

7.2 *Manual de instalación*

8 Conclusiones y posibles ampliaciones

De este proyecto saco dos conclusiones claras:

- Ha sido una experiencia gratificante en cuanto al aprendizaje.
- El desarrollo de software tiene muchísimos detalles que aprender, y es muy importante forjar una buena base.

En cuanto ha posibles mejoras podría destacar bastantes ya que, al haberme centrado en hacer pequeñas implementaciones interesantes, hay mucho en lo que ahondar pero principalmente destaco la accesibilidad ya que es algo muy importante y muy extenso en lo que me gustaría haber profundizado.

9 Bibliografía

Node.js (entorno de ejecución JavaScript): <https://nodejs.org/es/docs/>

TypeScript (tipado para JavaScript): <https://www.typescriptlang.org/docs/>

Express.js (framework web para Node): <https://expressjs.com/es/>

Fastify (framework web rápido para Node): <https://www.fastify.io/docs/latest/>

Zod (validación de datos en TypeScript): <https://zod.dev/>

zod-to-json-schema (para Swagger): <https://github.com/StefanTerdell/zod-to-json-schema>

MongoDB (base de datos NoSQL): <https://www.mongodb.com/docs/>

Vitest (testing moderno para TypeScript): <https://vitest.dev/>

Expo (desarrollo con React Native): <https://docs.expo.dev/>

React Native: <https://reactnative.dev/docs/getting-started>

React Navigation (navegación móvil): <https://reactnavigation.org/docs/getting-started>

React Hook Form (formularios): <https://react-hook-form.com/>

Keycloak (gestión de usuarios y autenticación): <https://www.keycloak.org/documentation/>

JWT (JSON Web Tokens): <https://jwt.io/introduction>

Docker (contenedorización): <https://docs.docker.com/>

Docker Compose: <https://docs.docker.com/compose/>

Postman (test manual de APIs): <https://learning.postman.com/docs/getting-started/introduction/>

Swagger/OpenAPI (documentación de APIs): <https://swagger.io/docs/specification/about/>

Midudev – Canal de referencia en desarrollo web moderno: <https://www.youtube.com/@midudev>

(Especialmente útil para entender Node, TypeScript, monorepos y buenas prácticas)

Fazt Code – Tutoriales completos sobre backend con Node y Express:

<https://www.youtube.com/@FaztTech>

Codigo Alex – Canal orientado a microservicios con Node.js, Mongo y Docker:

<https://www.youtube.com/@CodigoAlex>

HolaMundo – Tutoriales prácticos de programación y arquitectura:

<https://www.youtube.com/@holamundoDev>

Diego De Granda – Cursos introductorios a React Native y Expo:

<https://www.youtube.com/@diegodegranda>

Nico Dev – Explicaciones claras sobre arquitectura, microservicios y testing:

https://www.youtube.com/@nico_dev

Keycloak en Español (varios canales) – Uso de Keycloak en proyectos reales

MongoDB University (YouTube) – Explicaciones oficiales sobre esquemas, queries y buenas prácticas:

<https://www.youtube.com/@MongoDB>

dbdiagram.io (modelado de bases de datos): <https://dbdiagram.io/>

Draw.io / Diagrams.net (diagramas UML y ER): <https://www.diagrams.net/>

ShadCN/UI (componentes para React): <https://ui.shadcn.dev/>

Lucide Icons (íconos SVG): <https://lucide.dev/icons/>

GitHub Docs: <https://docs.github.com/>

ChatGPT (asistencia técnica y generación de documentación): <https://chat.openai.com/>



IES RIBERA DE CASTILLA

LUDOKEEPER

10 Anexos