Aplicación web para la gestión de eventos deportivos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Carlos Daniel Pontón Valladares
Tutor:
Ginés García Mateos

Facultad
de Informática
UMU

1 de Julio de 2026

Aplicación web para la gestión de eventos deportivos

Diseño, implementación y despliegue

Autor
Carlos Daniel Pontón Valladares

Tutor
Ginés García Mateos

Facultad
de Informática
UMU

Grado en Ingeniería Informática

UNIVERSIDAD
DE MURCIA

Murcia, 1 de Julio de 2026

Agradecimientos

Quisiera expresar mi agradecimiento a mi tutor, el Dr. Ginés García Mateos, por su orientación y supervisión a lo largo de este proyecto. También quiero dar las gracias a mi familia y amigos, muy especialmente a mis padres y mis hermanas, por ser un pilar fundamental. Sin su paciencia, apoyo incondicional y compañía durante todos estos años, este logro no habría sido posible. Finalmente, una mención especial para mis dos amigos peludos, la mejor compañía durante las incontables horas de desarrollo de este trabajo.

Resumen

El sedentarismo representa un grave problema de salud pública, fuertemente agravado por las barreras logísticas y sociales que dificultan la organización de actividades deportivas. Para superar este obstáculo, este Trabajo de Fin de Grado presenta una aplicación web para la gestión de encuentros deportivos. El proyecto hace énfasis en el enfoque técnico, aplicando altos estándares de calidad de *software* y diseño arquitectónico para asegurar la escalabilidad y la vida útil del sistema a largo plazo.

Este documento describe el proceso completo seguido para la construcción de la plataforma. El recorrido comienza con un análisis del estado del arte, que proporciona una visión general de las soluciones existentes en el mercado, culminando con la implementación y el despliegue del prototipo funcional final.

El objetivo principal de este trabajo es construir una solución web robusta para la gestión de eventos deportivos. Para lograr una arquitectura sólida, el núcleo del sistema se apoya en la Arquitectura Hexagonal, DDD, CQRS y una estrategia de *testing* para asegurar la calidad y el correcto funcionamiento de sus componentes. Por su parte, la interfaz de usuario se ha diseñado priorizando la *responsividad* y la optimización frente a los motores de búsqueda (SEO). En este contexto, el *stack tecnológico* seleccionado para este proyecto está compuesto por NestJS, Next.js y PostgreSql. Además, la solución se despliega en un entorno de producción (`https://tfg.cponton.com/`) mediante la combinación de Docker, Caddy y Vercel.

Dada la naturaleza dinámica del proyecto y la incertidumbre propia de los procesos de desarrollo de software, la construcción del sistema se ha guiado por la metodología ágil Kanban. Este enfoque nos ha permitido adaptarnos al constante cambio de los requisitos y la aparición continua de nuevos retos técnicos. En consecuencia, el sistema se ha diseñado de forma iterativa, sometiendo el planteamiento inicial a un proceso continuo de refinamiento. Para ilustrar y guiar esta implementación, este trabajo se apoya en diagramas de casos de uso y diagramas de secuencia para describir y justificar las decisiones tomadas. En definitiva, el desarrollo ha seguido un proceso metódico aplicando los enfoques y tecnologías planteadas.

Finalmente, el presente documento concluye con un repaso general donde se valida el cumplimiento de los objetivos propuestos y se adopta un enfoque crítico para analizar las posibles áreas de mejora. Por este motivo, se enumeran las principales deficiencias encontradas. Por otro lado, dado que el producto resultante no es más que un MVP del diseño global, se enumeran también las vías de investigación futura para extender y mejorar la solución construida.

Extended Abstract

Introduction

Today, a sedentary lifestyle is a major public health issue. According to the World Health Organization (WHO), increasing physical activity could prevent millions of deaths every year [1], a challenge that aligns with Sustainable Development Goal (SDG) 3 [2]. However, reports like the Eurobarometer [3] show that almost half of the European population does not exercise regularly due to a lack of time or interest. This project suggests that one of the underlying causes of this demotivation is organizational friction: the complex logistics required to coordinate schedules, facilities, and participants, often made worse by the lack of an active social circle.

To address this issue, this Bachelor's Thesis proposes the design and implementation of a specialized web platform to break down these entry barriers. It aims to centralize the management of amateur sporting events and make it easier for athletes to connect.

At the same time, the project takes a critical approach to system quality. Aware of the high cost of technical debt in the industry [4], the solution avoids rushing the delivery of features. Instead, backed by Martin Fowler's Design Stamina Hypothesis [5], it invests more effort in the initial stages of architectural design. The goal is to build a scalable and maintainable system that guarantees a long lifecycle.

State of the art

The state of the art analysis reveals a mature and diversified ecosystem of applications aimed at sports event management, although none fully resolve current organizational needs.

Pioneering tools like Timpik [6] stand out for their robust business logic in handling unforeseen events such as last-minute dropouts, but they heavily penalize the user experience with an obsolete and unintuitive web interface. Leading racket sports platforms such as Playtomic [7] offer an excellent booking system and a precise player leveling algorithm. However, they operate within a closed ecosystem dependent on private facilities, excluding the use of public spaces. Other alternatives like CeleBreak [8] manage to eliminate organizational friction through a centralized model where the user simply pays and plays. Nevertheless, this imposes an economic barrier to entry and removes the freedom for athletes to create and manage their own matches. From a more community-driven perspective, Strava [9] shines for its massive social and gamification

component, but it features a primarily retrospective approach that relegates the organization and management of future events to the background. Finally, general-purpose solutions like Meetup [10] feature powerful geolocated discovery algorithms, but their lack of specialization makes them inefficient for connecting a strictly athletic audience.

Consequently, this Bachelor's Thesis emerges as a response to the demand that current offerings fail to meet, aiming to provide a more accessible and functional alternative.

Goals analysis and methodology

The purpose of the proposed project is the design and implementation of a web application for the management of amateur sporting events. In this context, the aim is to build a functional prototype and lay the foundations for an extensible and scalable system, upon which future enhancements and new features can be added. Furthermore, the user interface aims to be responsive and highly optimized for search engines.

This primary goal has been divided into multiple sub-objectives: preliminary analysis, study of the methodology and technical justification of technologies, system design, system development, and finally, the deployment and validation of the application.

Likewise, these sub-objectives have been broken down into a series of tasks to facilitate time management and the evaluation of objective fulfillment.

Software Development Methodology

Given the uncertainty inherent in defining requirements in modern software development, rigid traditional approaches prove counterproductive. To address this need for flexibility, the project adopts the agile Kanban methodology [11]. This iterative approach is ideal for building a Minimum Viable Product (MVP), as it prioritizes the continuous delivery of value and allows for rapid adaptation to potential changes or new technical challenges. Furthermore, to complement this agile philosophy and facilitate the visualization and organization of development tasks, the Trello platform [12] was employed to visually organize tasks into columns based on their execution status: TO-DO, Doing, Reviewing, and Done.

Technology evaluation

The backend constitutes the core of the project, requiring a scalable and secure framework with support for automated testing and strict typing. System communication is achieved by exposing a Representational State Transfer (REST) Application Programming Interface (API) [13], ensuring high compatibility and low coupling integration. After evaluating Express as an alternative, NestJS [14] was selected as the

primary technology due to its modular structure and dependency injection system—fundamental characteristics for implementing a Hexagonal Architecture. Additionally, to maximize system performance while retaining all the architectural advantages of the framework, Fastify [15] was configured as the underlying HTTP adapter. Finally, to ensure maximum reliability in the delivery of emails required for authentication flows, the third-party email service Postmark [16] was integrated.

For the data persistence layer, NoSQL options were discarded in favor of a relational model. PostgreSQL [17] was the selected technology due to its transactional robustness, efficiency with structured data (`JSONB`), and, fundamentally, the integration of the PostGIS extension [18], an indispensable tool for resolving geospatial queries for events. Its administration from the backend is managed via TypeORM [19], allowing the domain logic to remain entirely decoupled, strictly following Domain-Driven Design (DDD) principles.

Regarding the user interface, after evaluating alternatives like Nuxt, Next.js [20] was chosen for its industry maturity, native support for Server-Side Rendering (SSR), and its excellent capacity for Search Engine Optimization (SEO).

Development environment & deployment

The system followed a hybrid deployment strategy where the backend, frontend, and database are hosted on independent environments:

- **Backend:** Containerized with Docker [21] on a dedicated Virtual Private Server (VPS), employing Caddy [22] as a reverse proxy to automate Transport Layer Security (TLS) certificates. The database operates on an independent VPS within the same network to ensure maximum security and low latency.
- **Frontend:** Hosted on Vercel [23], leveraging its infrastructure to optimize rendering (SSR) and global distribution via its Content Delivery Network (CDN).
- **Security and Monitoring:** Cloudflare [24] acts as an edge network to protect against malicious access and optimize traffic. Sentry [25] is used in both environments to capture errors in real time.

Furthermore, the workflow was managed using Git [26] & GitHub [27] for version control, Docker to simulate the database environment locally, and Postman [28] to perform manual tests on the API.

System design & architecture

The system design was initially structured under a monolithic architecture to expedite the delivery of the MVP, although its technical foundations guarantee a natural

evolution towards a distributed architecture that favors future scalability. Internally, the backend core is based on the Hexagonal Architecture proposed by Alistair Cockburn [29], which isolates business logic from infrastructure details through a system of ports and adapters. This separation is reinforced by DDD principles [30], modeling the system using entities, value objects, and aggregates within bounded contexts to protect business rules. Additionally, the Command Query Responsibility Segregation (CQRS) pattern [31] was integrated to separate and optimize read and write operations.

To support this architectural framework, design patterns such as dependency injection and *Unit of work* were implemented to guarantee atomic operations, along with *Pessimistic locking* strategies to protect data integrity in high-contention scenarios. To ensure development quality, a testing strategy based on Mike Cohn's test pyramid [32] was designed, applying unit, integration, and End-to-End (E2E) tests, focused strictly on the Authentication module due to the time constraints inherent in developing the MVP.

Finally, this entire theoretical approach was materialized and documented within the project through the exhaustive definition of use cases, business rules, class diagrams for each bounded context, and a global entity-relationship (ER) model, laying a solid foundation prior to the implementation phase.

System development

Rather than detailing individual use cases, the backend implementation is best understood by analyzing the generic flow of a write request (*Command*). The lifecycle of a protected request begins at the security layer, where a *Guard* intercepts the call before it reaches the web controller. If the user is not authenticated, the system rejects the request, throwing a 401 (*Unauthorized*) error and aborting execution following the *fail fast* pattern. Once this barrier is cleared, the request reaches the controller, which delegates execution to the corresponding use case in the application layer. This layer orchestrates the operation, relying on the domain to validate business rules and utilizing the Unit of work pattern to ensure atomic persistence in the database, ultimately returning the response to the client.

This flow is significantly altered for read operations (*Queries*) thanks to the implementation of CQRS. In these cases, the request bypasses the domain layer, avoiding the costly instantiation of entities, and directly targets optimized database projections. At the business logic level, one of the most notable aspects of the implementation is the modeling of the sports system, which was designed based on composition rather than inheritance. This grants the system core enormous flexibility to incorporate new athletic disciplines with customized rules in the future.

Regarding the user interface, frontend development is built upon three fundamental pillars. First, a global authentication context (*Auth Context*) that reactively manages the user's session state. Second, a dedicated services layer that abstracts and centrali-

zes asynchronous communication with the API. And finally, a robust form validation system that guarantees data integrity on the client side before issuing any request to the server.

Finally, the solution was materialized through a secure multi-environment deployment. The backend infrastructure is hosted in Virtual Private Clouds (VPCs) on DigitalOcean [33], isolating services at the network level. The ecosystem runs via Docker containers orchestrated alongside Caddy, which acts as a reverse proxy and web server, automating the management of security certificates and the efficient routing of traffic.

Conclusions and future directions

This project has validated the design of a clean architecture oriented towards maintainability using Hexagonal Architecture and DDD, whose steep learning curve was compensated by the resulting robustness. As the system scaled, CQRS was successfully integrated to optimize read queries in the Activities module without compromising domain coherence during write operations.

Regarding quality assurance, testing proved to be a valuable tool, but unsustainable in a highly volatile environment. Continuous technical changes made test maintenance an unacceptable overhead, highlighting that in phases of high uncertainty, it must be applied with caution. Furthermore, frontend development revealed that maintaining separate repositories favors code duplication, and the overuse of external dependencies drastically harms the bundle size and SEO. Ultimately, the objectives have been met, resulting in a solid MVP that lays the groundwork for future iterations.

System limitations

Despite the success of the prototype, several limitations arising from deliberate concessions and time constraints have been identified:

- **Architectural and code debt:** Includes code duplication due to the lack of *Shared Packages*, coupling between the Auth and User contexts, and the use of unspecialized write models. Additionally, the domain contains an external dependency (**Big.js** [34]).

- **Security and concurrency risks:** Potential Denial of Service (DoS) via database connection exhaustion during *bcrypt* verification, and a minimal TOCTOU risk in the user creation flow.

- **Frontend and Quality Assurance:** Technical difficulties in synchronizing the React session state across multiple tabs, suboptimal API error presentation, and test coverage strictly limited to the Authentication module.

Future directions

The designed architecture supports the expansion of the system far beyond the current MVP. The main lines of future work are:

- **Mitigation of limitations:** Priority resolution of the technical deficiencies listed above, including the implementation of JSON Web Token (JWT) revocation mechanisms to strengthen session security.
- **Evolution to a distributed architecture:** Transitioning towards event-driven communication, eventual consistency, and asynchronous background workers.
- **System feature expansion:** Implementation of the remaining designed features, including a reporting and penalty module, sports facilities management, direct booking integration, a community reputation system, and demographic filters.
- **Route capability enrichment:** Integration with APIs to enrich routes with elevation, distance, and terrain data, among others, in the background.
- **Frontend optimization:** Replacing external dependencies with custom native utilities to optimize the bundle size and load times.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

En el contexto actual donde las nuevas tecnologías gobiernan sobre la vida de las personas, el sedentarismo emerge como un grave problema de interés mundial. Según apunta la Organización Mundial de la Salud (OMS) en sus Directrices sobre Actividad Física y Hábitos Sedentarios, cada año podrían evitarse entre cuatro y cinco millones de muertes si los niveles de actividad física aumentaran entre la población [1]. En consecuencia, fomentar la práctica habitual del deporte es una necesidad prioritaria que se alinea con el Objetivo de Desarrollo Sostenible (ODS) 3 que apunta a garantizar una vida sana y promover el bienestar para todos en todas las edades [2].

Sin embargo, la erradicación de la vida sedentaria no es un reto trivial dado que la actividad física frecuente se enfrenta a una serie de barreras de entrada. De acuerdo con el Eurobarómetro sobre deporte y actividad física, el 45% de la población europea no practica deporte de forma regular debido a la falta de tiempo, la desmotivación y el desinterés, entre otras razones [3].

Por tanto, este proyecto nace para dar respuesta a un problema crítico que no cuenta con una solución estandarizada para afrontarlo. La falta de tiempo y la falta de interés evidencian que la fricción organizativa puede ser un elemento diferencial. Mientras que las preferencias del entorno también influyen en los hábitos saludables de las personas. En este contexto, la aplicación planteada pretende resolver estos obstáculos mediante una plataforma que elimina la barrera organizativa y conecta a un público interesado en el deporte.

Por otro lado, también hay que darle importancia al desarrollo de software basado en el análisis y diseño de código mantenible y escalable a lo largo del tiempo. El código de baja calidad o la acumulación de deuda técnica suponen un coste de hasta 1,52 billones de dólares a las empresas americanas [4]. Estos datos sugieren que construir software de menor calidad a cambio de una puesta en marcha prematura puede suponer grandes perjuicios en etapas futuras de los proyectos.

1.1 Logística organizativa como barrera de la actividad física

Como se comentó anteriormente, la falta de tiempo e interés suponen las dos razones principales que provocan que una gran parte de la población no practique deporte de forma habitual. Aunque no podemos darlo por hecho, es posible que estas barreras se deban a la logística requerida para la organización de encuentros deportivos. Si

Figura 1.1: Hipótesis de la Resistencia del Diseño. Fuente: Martin Fowler [5]

bien muchos deportes pueden practicarse sin preparativos previos, existen otros que requieren esfuerzos para gestionar los calendarios, a los participantes, el equipo e incluso las instalaciones necesarias. Por lo tanto, la complejidad organizativa se plantea como una causa probable de la falta de tiempo e interés general en el deporte.

Añadido a esto, el entorno social de cada persona puede influir en sus hábitos. Mientras que un entorno activo puede generar interés en el deporte, un entorno sedentario puede provocar rechazo. La ausencia de un círculo social interesado en el deporte y la incompatibilidad horaria solo agravan esta barrera.

En este contexto, un producto especializado puede ser la solución a este problema, permitiendo conectar a deportistas, centralizando la gestión de eventos y rompiendo la barrera organizativa. De este modo, la solución planteada también puede servir como una herramienta para la iniciación en nuevas modalidades deportivas.

1.2 Importancia de un buen diseño

Debido al auge de las metodologías ágiles, los procesos de desarrollo de software han cambiado drásticamente. Como resultado, las fases de análisis y diseño se reducen al máximo o desaparecen para agilizar la entrega de valor. No obstante, esta práctica limita la capacidad de evolución de un proyecto, provocando su abandono prematuro en favor de nuevas soluciones y la aparición de código *legacy*, conocido por ser extremadamente complejo y caro de mantener.

Por este motivo, invertir recursos en el diseño y la arquitectura de un producto software resulta fundamental. Martin Fowler menciona que el buen diseño de software no solo es necesario, sino que permite avanzar más rápido y por más tiempo, aumentando el ciclo de vida del sistema [5]. Si bien omitir la fase de diseño permite acelerar la entrega de funcionalidad acumulativa, este ritmo de desarrollo va perdiendo eficacia a medida que se avanza en el tiempo. Por su parte, el desarrollo basado en un buen diseño es penalizado por la sobrecarga inicial de esta fase, alargando la entrega de valor

inicial, pero garantiza una velocidad de entrega sostenida. Esta hipótesis puede verse representada en la Figura 1.1. Como se observa, la fase de diseño empieza a compensar tan pronto como se alcanza la línea de rentabilidad (*design-payoff line*).

Finalmente, el reporte *The Cost of Poor Software Quality in the US* [4] de 2022 reafirma la teoría de Fowler, añadiendo el factor económico y evidenciando el gran desperdicio de recursos que supone para las empresas tecnológicas el desarrollo de software deficiente. En definitiva, la construcción de una arquitectura sólida desde las etapas iniciales de un proyecto supone un sobrecoste inicial que es compensado con creces en el largo plazo.

2 Estado del arte

En este capítulo se analizarán y compararán distintas aplicaciones existentes relacionadas con la gestión de eventos deportivos.

2.1 Proyectos relacionados

Presentamos algunas de las aplicaciones más relevantes y conocidas relacionadas con la gestión de eventos deportivos. También, analizamos sus características, su enfoque para resolver el problema y, finalmente, indicaremos sus aspectos positivos y negativos.

2.1.1 Timpik

Timpik [6] es una de las aplicaciones pioneras en la gestión de eventos deportivos amateur en España. Se centra, principalmente, en la gestión y organización técnica. Permite a los usuarios crear actividades indicando la información básica del encuentro. Es una de las plataformas más desarrolladas, ya que cuenta con listas de espera para actividades de alto interés, un sistema de valoraciones, y gestión de botes económicos para el pago de pistas. En la Figura 2.1 se muestra la interfaz visual de los detalles y la gestión de asistencia de un evento típico en la aplicación.

Figura 2.1: Interfaz visual de un evento en Timpik.

Como aspectos positivos, destaca su robusta *lógica de negocio*, que resuelve eficientemente problemas comunes como las bajas de última hora. Sin embargo, su principal

aspecto negativo es la interfaz de usuario obsoleta y poco intuitiva que presenta su versión web. Si bien su aplicación móvil intenta resolver este inconveniente con una línea de diseño más actual, la experiencia en su plataforma web sigue estando muy penalizada.

2.1.2 Playtomic

Playtomic [7] es la plataforma líder actual en los deportes de raqueta como el pádel y el tenis. Uno de sus aspectos positivos es su sistema de gestión de instalaciones deportivas, una de las vías futuras propuestas en este trabajo. En la Figura 2.2 se observa la pantalla de búsqueda, donde los usuarios pueden filtrar y reservar pistas en clubes asociados.

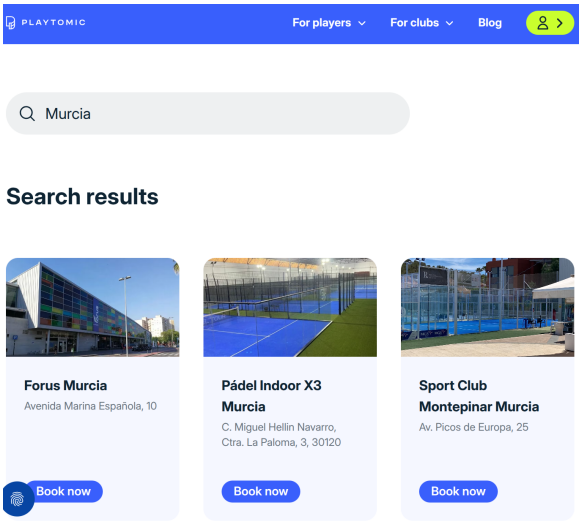


Figura 2.2: Interfaz visual del apartado de búsqueda de instalaciones en Playtomic.

Otro de sus fortalezas radica en su algoritmo de nivelación de jugadores muy preciso, el cual garantiza que los partidos públicos sean competitivos y equilibrados. Sin embargo, uno de sus principales puntos negativos tiene que ver con su ecosistema cerrado, limitado y dependiente de instalaciones deportivas privadas. No ofrece soporte ni flexibilidad para organizar eventos en espacios públicos, lo que excluye a un amplio sector de deportistas amateur.

2.1.3 CeleBreak

CeleBreak [8] es una aplicación que busca solucionar la dificultad de jugar al fútbol en grandes ciudades cuando no se dispone de un equipo completo. La plataforma sigue un modelo centralizado, donde los usuarios no tienen el control sobre las actividades, sino que dependen totalmente de los eventos deportivos que organiza el equipo que

opera la aplicación. En la Figura 2.3 se detalla la vista de un partido oficial organizado por la plataforma, mostrando las plazas disponibles y el coste individual.

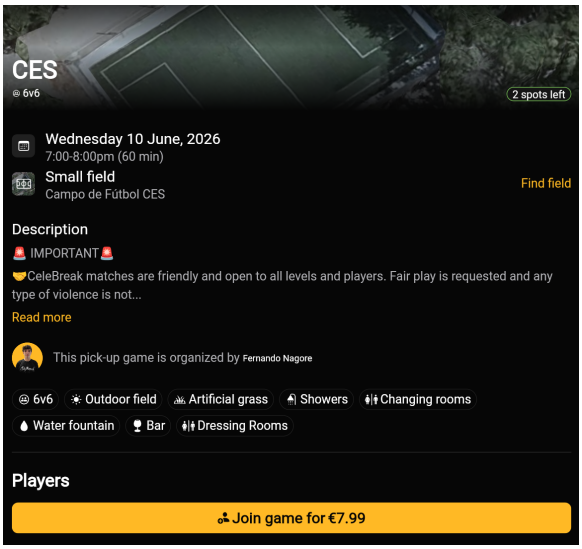


Figura 2.3: Interfaz visual de un partido en Celebreak.

Este producto elimina por completo la fricción organizativa, lo cual es un punto a favor. Un usuario simplemente paga su cuota, acude al campo y juega. Sin embargo, esto también puede ser visto como una faceta negativa, puesto que su modelo de pago aumenta la barrera de entrada para deportistas amateur y su sistema centralizado elimina la posibilidad de que los usuarios creen y gestionen sus propias actividades.

2.1.4 Strava

Strava [9] es la red social para deportistas al aire libre más grande del mundo. Esta plataforma es mucho más que una simple red social; permite la monitorización por GPS y el registro asíncrono de actividades físicas. En la Figura 2.4 se expone la sección de *Clubs*, donde los usuarios pueden unirse y convocar salidas grupales.

El mayor aspecto positivo de Strava es su inmenso componente social y de gamificación. Esta última característica es una de las principales razones del crecimiento y la amplia comunidad con la que cuenta Strava.

Su aspecto negativo, desde el punto de vista de la gestión de eventos, es que su enfoque es retrospectivo, ya que está diseñada, principalmente, para mostrar actividades que ya se han realizado. De aquí nacen sus funciones de clasificación y registro de entrenamientos. Si bien cuenta con un gestor de eventos deportivos, no es una función principal y no está muy desarrollada.

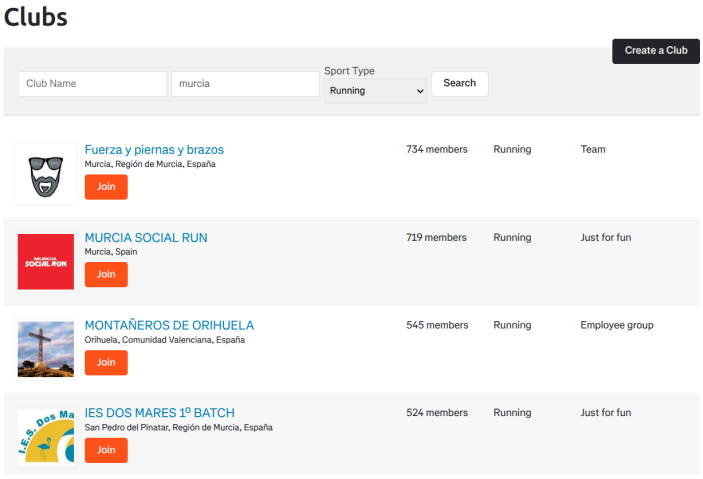


Figura 2.4: Interfaz visual del apartado *Clubs* de Strava.

2.1.5 Meetup

Meetup [10] no es una plataforma específica para deportes. Se centra en grupos y eventos, de cualquier índole. Su principal enfoque es conectar personas con intereses comunes, utilizando su función de búsqueda basada en la localización del usuario y sus foros de discusión. En la Figura 2.5 se presenta el listado dinámico de actividades geolocalizadas recomendadas según las preferencias previas del usuario.

Su principal aspecto positivo es su potente algoritmo de descubrimiento, permitiendo a un usuario encontrar grupos y actividades de su interés en su misma área. Sin embargo, su enfoque generalista resulta ineficiente para conectar y atraer a un público deportista.

Figura 2.5: Interfaz visual de la búsqueda de eventos en Meetup.

2.2 Conclusión

Tras el análisis del estado del arte, hemos evidenciado la existencia de un ecosistema maduro y diversificado de aplicaciones orientadas a la gestión deportiva. Sin embargo, este estudio refleja que ninguna de las soluciones actuales resuelve en su totalidad las necesidades planteadas. Mientras que algunas plataformas presentan deficiencias en su experiencia de usuario, otras carecen de flexibilidad en su modelo organizativo o imponen barreras de entrada que dificultan la práctica habitual de deporte.

Por otro lado, este análisis ha aportado un conocimiento valioso que ha servido como referencia técnica para definir y desarrollar nuestra solución.

Así pues, este Trabajo de Fin de Grado surge como respuesta a la demanda que la oferta actual no logra cubrir, buscando ofrecer una alternativa más accesible y funcional.

3 Análisis de objetivos y metodología

3.1 Objetivos del proyecto

El propósito del proyecto que se plantea es el diseño e implementación de una aplicación web que permita la gestión de eventos deportivos de carácter amateur. En este contexto, se pretende construir un prototipo funcional y sentar las bases para un sistema extensible y escalable, sobre el que se puedan añadir mejoras y nuevas características.

La aplicación web deberá permitir al usuario crear eventos deportivos, solicitando la información que describa, de forma precisa, las características del encuentro. Asimismo, deberá ofrecer la posibilidad de buscar actividades de otros usuarios con el mismo nivel de detalle, de manera que un deportista pueda identificar su nivel de afinidad con cada evento de forma rápida y sencilla.

Por otro lado, la interfaz adoptará un enfoque *responsivo* para garantizar una experiencia óptima en diferentes tipos de dispositivos, tanto móviles como de escritorio. Además, sabiendo que la viabilidad de una plataforma de estas características se sustenta en contar con un gran volumen de usuarios recurrentes, es necesario que la aplicación web esté fuertemente *optimizada frente a los motores de búsqueda* (SEO).

En virtud de lo anterior, se han planteado los siguientes subobjetivos, estructurados de forma que faciliten la medición de su cumplimiento y la organización temporal del proyecto.

- **Subobjetivo 1. Análisis preliminar**
    1. Estudio del estado del arte en las aplicaciones de gestión de eventos deportivos.
    2. Análisis y definición de los objetivos.
- **Subobjetivo 2. Metodología y justificación técnica**
    1. Estudio y evaluación de las metodologías de trabajo.
    2. Estudio y justificación de las tecnologías para el desarrollo del frontend.
    3. Estudio y justificación de las tecnologías para el desarrollo del backend.
    4. Estudio y justificación de los enfoques y arquitecturas para el desarrollo del backend.

- **Subobjetivo 3. Diseño del sistema**
  1. Análisis y definición de los requisitos.
  2. Definición del alcance inicial del *Producto Mínimo Viable* (MVP).
  3. Diseño del diagrama de clases del sistema.
  4. Definición de los casos de uso del sistema.
  5. Diseño de la base de datos.
- **Subobjetivo 4. Desarrollo del sistema**
  1. Desarrollo del módulo de autenticación de usuarios.
  2. Desarrollo de pruebas automatizadas para el módulo de autenticación de usuarios.
  3. Desarrollo del módulo de eventos deportivos.
  4. Diseño y maquetación de la interfaz de usuario.
  5. Integración del frontend y el backend.
- **Subobjetivo 5. Despliegue y validación**
  1. Contenerización del proyecto y sus dependencias.
  2. Despliegue del backend.
  3. Despliegue del frontend.
  4. Validación de la aplicación frente al tutor del proyecto.
  5. Documentación del proyecto.

3.2 Metodología

En el contexto del desarrollo de software moderno, la fase de elicitación de requisitos suele enfrentarse a un alto grado de incertidumbre. A diferencia de los enfoques tradicionales en cascada, que definen las especificaciones de manera rígida en las etapas iniciales, la creación de aplicaciones dinámicas exige una gran flexibilidad. Por ello, intentar anticipar y bloquear exhaustivamente todas las funcionalidades desde el inicio de este proyecto resultaría ineficaz y contraproducente frente al descubrimiento de nueva información o de retos técnicos durante el desarrollo.

Para hacer frente a este escenario, se ha adoptado **Kanban** [11] como metodología de trabajo. Esta técnica ágil se ajusta idóneamente a entornos de desarrollo iterativo y de alta incertidumbre, como es la construcción de un MVP. A diferencia de las planificaciones cerradas que pueden estancar el avance del proyecto, Kanban prioriza la entrega continua de valor, limita la cantidad de trabajo en curso y facilita la adaptación inmediata ante nuevos requisitos o imprevistos.

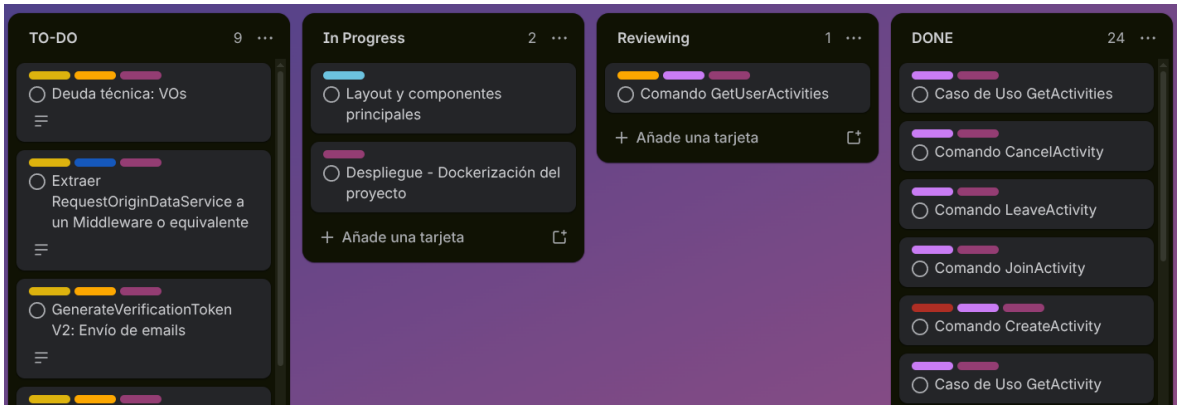


Figura 3.1: Tablero de Trello utilizado para la gestión de las tareas de desarrollo.

La aplicación práctica de esta metodología requiere una herramienta que permita visualizar el estado del proyecto en todo momento. En este sentido, la plataforma web **Trello** [12] encaja perfectamente con la filosofía Kanban. Tal y como se ilustra en la Figura 3.1, su interfaz basada en tableros, columnas y tarjetas facilita el seguimiento del flujo de trabajo de manera intuitiva. Específicamente, el tablero se ha estructurado en cuatro columnas que representan el ciclo de vida de cada tarea:

- **TO-DO**: Agrupa las tareas pendientes que aún no han sido iniciadas. Constituye el trabajo pendiente del proyecto.
- **In-Progress**: Contiene las tareas en fase de desarrollo activo. Siguiendo las directrices de Kanban, se ha establecido un límite máximo de dos tareas simultáneas para mantener el foco y evitar cuellos de botella.
- **Reviewing**: Aunque habitualmente se destina a la revisión de código por pares (*Code Review*), en el marco de este proyecto individual se ha adaptado como una fase de validación manual y pruebas del producto.
- **DONE**: Recopila las tareas finalizadas y validadas, representando el trabajo completado y el valor entregado al sistema.

3.3 Planificación

En la Figura 3.2 se ilustra la planificación temporal del proyecto mediante un diagrama de Gantt, estructurada en tareas y semanas. Cabe destacar el solapamiento de las fases de análisis y desarrollo, un claro reflejo de la naturaleza iterativa de la metodología ágil adoptada. Asimismo, se observa una alta concurrencia de trabajo durante la semana 21. Este comportamiento responde a la fase de validación del sistema con el tutor, la aplicación de las correcciones pertinentes y el despliegue final de la aplicación en producción.

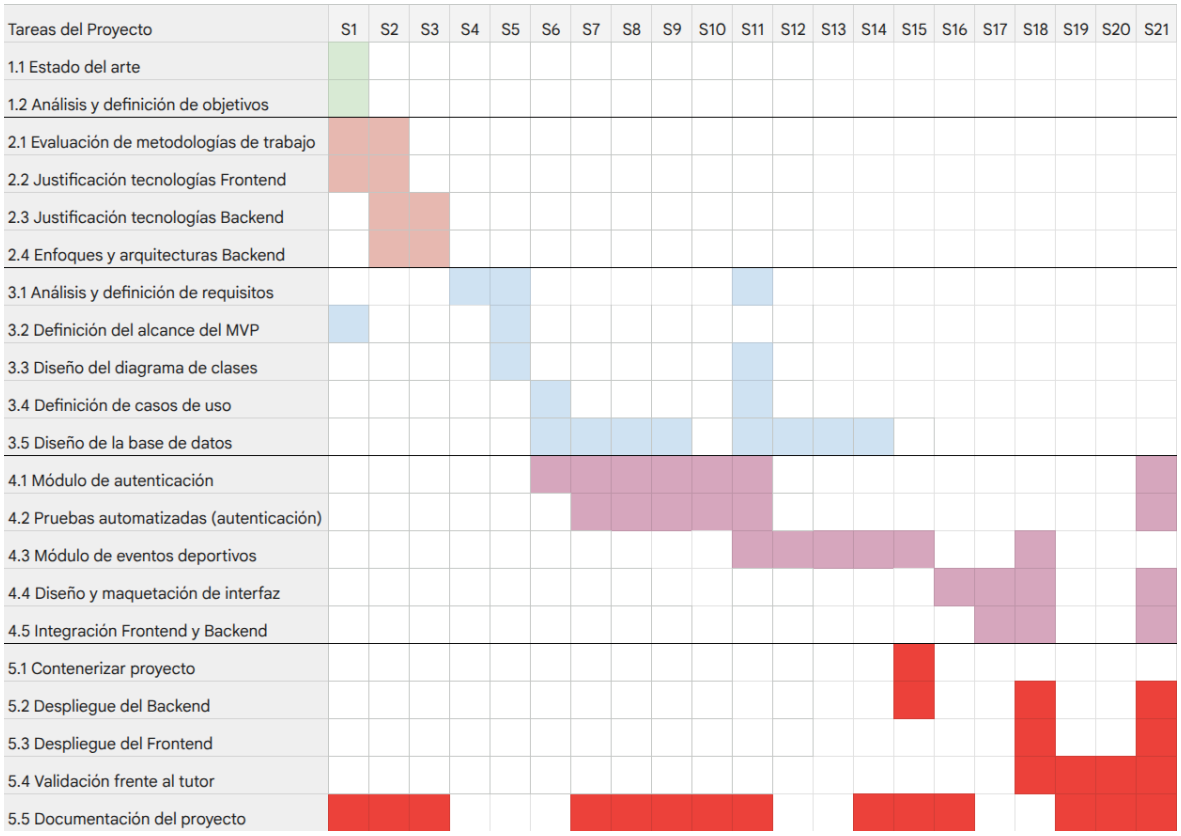


Figura 3.2: Diagrama de Gantt de la planificación temporal del proyecto.

4 Evaluación y selección de tecnologías

4.1 Backend

El backend representa la pieza central del proyecto y sobre la cual recaerá el mayor peso del desarrollo. Este componente es el responsable de gestionar la lógica de negocio, interactuar con la base de datos y garantizar que las operaciones se realizan correctamente y de forma segura. Por este motivo, resulta fundamental escoger un *framework* que ofrezca escalabilidad, buen rendimiento, seguridad y una documentación extensa. Además, atendiendo a los requisitos de calidad del software de este proyecto, el soporte para pruebas automáticas (testing) y el uso de *tipado estático* estricto son condiciones indispensables. Esta característica resulta vital para detectar errores durante el desarrollo, mejorar la legibilidad del código y sentar las bases de un sistema mantenible a largo plazo.

La comunicación con el exterior se realizará exponiendo una **API REST** [13]. Se ha elegido este protocolo porque cumple perfectamente con los requisitos técnicos del sistema: comunicación sin estado, alta compatibilidad, baja complejidad de integración y un coste de mantenimiento reducido. Cabe recalcar que el empleo de una API REST simplemente dicta las reglas para la comunicación entre cliente y servidor, sin acoplar el backend a una infraestructura concreta, lo que abre las puertas a futuras evoluciones del sistema.

En la Tabla 4.1 se muestra una comparativa entre las dos tecnologías más representativas del ecosistema Node.js: Express [35] y NestJS [14]. Aunque ambas son opciones válidas, se ha seleccionado **NestJS** como la herramienta idónea, en base a las necesidades del proyecto. A pesar de presentar una curva de aprendizaje más pronunciada que Express, NestJS ofrece un sistema de inyección de dependenciass y una estructura modular que resultan óptimos para implementar una Arquitectura Hexagonal, permitiendo construir una aplicación totalmente desacoplada y *testable*. A esta ventaja se le suma la inclusión de utilidades integradas que aceleran el desarrollo: *pipe*s para la validación de datos, guards para la seguridad y la posibilidad de instalar filtros de excepciones para centralizar la gestión de errores.

Un aspecto técnico interesante de NestJS es que se ejecuta como una capa superior sobre Express, actuando como una capa de abstracción. No obstante, NestJS ofrece la posibilidad de intercambiar la tecnología subyacente. Para este proyecto, se ha configurado **Fastify** [15] como adaptador HTTP. Esta decisión permite mantener todas las

Criterio de evaluación	Express	NestJS
Estabilidad y escalabilidad	Alta.	Muy alta.
Soporte de arquitecturas	Tradicional (REST). Otras arquitecturas requieren configuración manual.	Excelente. Soporte nativo para REST, GraphQL, WebSockets y microservicios.
Soporte TypeScript [36]	Posible mediante configuración externa.	Nativo y estricto por defecto.
Curva de aprendizaje	Muy baja.	Pronunciada.
Rendimiento	Bueno.	Excelente (con Fastify).
Seguridad	Básica por defecto. Requiere librerías externas.	Estructura robusta mediante utilidades integradas.
Documentación y ecosistema	Gigantesco, al ser el estándar histórico.	Muy amplio, estructurado y moderno.
Soporte Testing	Básico.	Integrado de serie.

Tabla 4.1: Comparativa entre los marcos de trabajo para el backend: Express y NestJS.

ventajas arquitectónicas y de experiencia de desarrollo que ofrece NestJS, al tiempo que el sistema se beneficia del alto rendimiento que caracteriza a Fastify. Además, la implementación de cualquier adaptador es completamente transparente al código de la aplicación.

Por otro lado, respecto a la delegación de servicios externos en la infraestructura de notificaciones, los flujos de autenticación del sistema requieren la verificación de la identidad del usuario mediante el envío de *token*s seguros por correo electrónico. Para garantizar la fiabilidad de estas comunicaciones, se ha optado por integrar **Postmark** [16] como servicio de correo en lugar de gestionar una infraestructura propia.

4.2 Capa de Persistencia y Base de Datos

La base de datos es el componente encargado de almacenar, organizar y garantizar la coherencia de la información utilizada por el sistema. Su papel es fundamental, ya que el rendimiento, la escalabilidad y la seguridad de la aplicación dependen en gran medida de la tecnología utilizada en esta capa. Por tanto, resulta de vital importancia seleccionar una solución que se adecúe a los requisitos funcionales y técnicos del proyecto.

Criterio de evaluación	PostgreSQL	MySQL[37]	MongoDB [38]
Integridad referencial	Excelente. Soporte estricto de *FK*.	Alta pero con restricciones.	Nula.
Rendimiento (read/write)	Excelente equilibrio.	Alto en consultas, menor en escrituras.	Muy alto. Penalizada en *joins*.
Escalabilidad	Alta.	Alta.	Excelente (*sharding*).
Tipos de datos soportados	Excelente. Soporte nativo avanzado y altamente ampliable mediante extensiones.	Alto. Incluye soporte JSON, aunque las funciones más avanzadas dependen del motor.	Reducido, aunque con gran flexibilidad estructural.

Tabla 4.2: Comparativa de los sistemas de gestión de bases de datos evaluados.

Para la elección de la base de datos se evaluaron los dos principales enfoques: bases de datos relacionales (**SQL**) y bases de datos no relacionales (**NoSQL**). En la Tabla 4.2 se expone una comparativa detallada con las opciones analizadas. Dado que la base del sistema reside en las relaciones entre entidades y que el soporte para transacciones complejas es un requisito técnico innegociable, se descartó el enfoque NoSQL en favor de un modelo relacional.

La elección final recayó sobre **PostgreSQL** [17]. Si bien MySQL resulta una opción muy sólida y extendida en la industria tecnológica, las necesidades específicas de la plataforma decantaron la balanza hacia esta solución. Por un lado, ofrece un excelente equilibrio de rendimiento en operaciones concurrentes de lectura y escritura. Además, su avanzado sistema de extensiones permite integrar herramientas como **PostGIS** [18], una característica clave para resolver las consultas geoespaciales requeridas en la búsqueda de eventos deportivos cercanos.

Asimismo, para optimizar el almacenamiento de datos estructurados variables, se han empleado columnas de tipo `JSONB` indexadas mediante Índices Invertidos Generalizados (**GIN**), los cuales permiten acelerar drásticamente las consultas dinámicas sobre estas estructuras. Adicionalmente, PostgreSQL destaca por su solidez y seguridad en escenarios transaccionales exigentes, donde el sistema mantiene un excelente rendimiento en las operaciones de escritura que suelen requerir la inserción coordinada en varias tablas (usuarios, eventos, inscripciones) y la consecuente actualización

simultánea de múltiples índices sin romper la consistencia.

La gestión de una base de datos resulta especialmente compleja si no se acompaña de herramientas adecuadas. Para interactuar con este entorno desde el backend se ha utilizado un Mapeador Objeto-Relacional (**ORM**), concretamente **TypeORM** [19]. Más allá de su compatibilidad con NestJS y su eficiente sistema de migraciones para controlar los cambios de esquema entre entornos, el motivo de mayor peso es la capacidad de esta herramienta para separar la definición de las tablas de las entidades de negocio. Esto ayuda a mantener el dominio del producto totalmente desacoplado, puro y alineado con las directrices dictadas por el Desarrollo Dirigido por el Dominio (**DDD**).

4.3 Frontend

La capa de presentación de este proyecto cuenta con unos requisitos muy específicos de responsividad y SEO. En este sentido, resulta esencial el control de la consistencia del enrutamiento para evitar contenido duplicado. Por tanto, es necesario encontrar un marco de trabajo que proporcione características como: *renderización* en el lado del servidor (**SSR**), generación estática (**SSG**), capacidades de revalidación incremental (**ISR**), soporte de TypeScript y un ecosistema fiable y maduro.

En el panorama actual, existe un gran número de tecnologías que reúnen estas características deseadas de forma parcial o total. Sin embargo, este análisis se limitará a las dos más representativas del ecosistema JavaScript: Next.js [20] y Nuxt [39]. En la Tabla 4.3 se ilustra una comparativa entre estos dos marcos de trabajo, reflejando características técnicas muy similares y equivalentes. Ante esta igualdad, el factor decisivo recae sobre la experiencia previa en el ecosistema **React** [40]. El dominio de las herramientas durante el desarrollo permite que la curva de aprendizaje y la fricción inicial se reduzcan drásticamente, garantizando la entrega de un MVP de calidad y dentro de los plazos establecidos. Así pues, **Next.js** es la tecnología seleccionada para construir la interfaz de usuario del proyecto.

Para agilizar la construcción del frontend sin renunciar a la personalización, se hará uso de **shadcn/ui** [41], una colección de componentes accesibles y listos para integrar. El estilizado de estos componentes y del resto de la aplicación se llevará a cabo mediante **Tailwind CSS** [42], un marco de trabajo basado en clases de utilidad CSS. Finalmente, **next-translate** [43] se utilizará para la internacionalización del sistema.

Por otro lado, se integrará la API de **Google Maps** [44] para visualizar mapas en la interfaz de usuario, así como para obtener sugerencias de lugares a partir de la entrada del usuario.

Criterio de evaluación	Next.js (React)	Nuxt (Vue)
Renderización	Soporte nativo para SSR, SSG e ISR.	Soporte nativo para SSR, SSG e ISR.
Optimización SEO	Muy alta. Control granular de metadatos.	Alta. Generación automática de metadatos.
Estabilidad y escalabilidad	Muy alta. Respaldado por Vercel, diseñado para aplicaciones empresariales.	Alta. Diseño modular ideal para proyectos de mediana y gran envergadura.
Soporte TypeScript	Soporte nativo.	Soporte nativo.
Curva de aprendizaje	Moderada, hereda la fricción inicial de React.	Suave, hereda la baja fricción inicial de Vue.
Adopción industrial	Masiva. Estándar actual.	Extensa.
Documentación y ecosistema	Muy amplio.	Amplio.

Tabla 4.3: Comparativa entre los marcos de trabajo para el frontend: Next.js y Nuxt.

4.4 Entorno de desarrollo y despliegue

Para garantizar la viabilidad de la plataforma en producción y optimizar la gestión de los recursos, se ha optado por una estrategia de despliegue híbrida y desacoplada. La infraestructura del sistema se distribuye de la siguiente manera:

- **Backend**: La lógica de la API se ejecuta de forma aislada en un servidor privado virtual (VPS). Para evitar inconsistencias entre entornos, la aplicación está totalmente contenerizada con **Docker** [21]. Como punto de acceso se emplea un contenedor de **Caddy** [22] que actúa como proxy inverso. Se seleccionó Caddy por encima del estándar Nginx para reducir la fricción inicial de configuración, ya que automatiza por completo la obtención y renovación de los certificados TLS. Por último, la base de datos se ha desplegado en un VPS independiente pero integrado dentro de la misma red, garantizando el aislamiento completo de los datos, mayor seguridad y una latencia mínima en la comunicación.

- **Frontend**: Se despliega de manera independiente en la plataforma **Vercel** [23]. Esto permite delegar por completo la optimización del SSR y la distribución global en la red de distribución de contenido (CDN) de la plataforma.

- **Seguridad y enrutamiento**: Se utiliza la red perimetral de **Cloudflare** [24] para enrutar todo el tráfico hacia la plataforma. Funciona como un proxy inverso de alto nivel que protege el sistema contra accesos maliciosos y reduce la latencia de las peticiones globales a través de su extensa CDN.
- ***Observabilidad***: Se ha configurado el servicio de monitorización **Sentry** [25] tanto en el frontend como en el backend con el objetivo de centralizar y capturar excepciones en tiempo real, facilitando la detección rápida de errores en producción.

Por otro lado, para facilitar la construcción del producto final, se han utilizado las herramientas que se detallan a continuación:

- **Git [26] y GitHub [27]**: Herramientas esenciales para el control de versiones, la gestión de ramas y el respaldo seguro del código fuente del proyecto.
- **Docker**: Utilizado en el entorno local para levantar de forma inmediata la base de datos PostgreSQL, asegurando que el desarrollo replique fielmente las condiciones del servidor de producción.
- **Postman [28]**: Plataforma empleada para ejecutar pruebas manuales contra los diferentes *endpoint*s de la API durante el ciclo de vida del desarrollo.

5 Análisis y diseño del sistema

5.1 Arquitectura del sistema

En esta sección se explican brevemente los enfoques y patrones arquitectónicos utilizados para la construcción del backend de este proyecto.

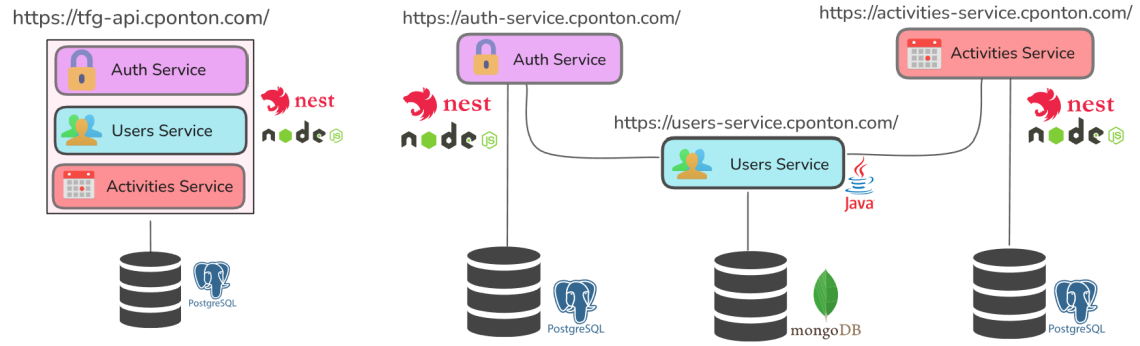
Es conveniente apuntar que, para la construcción del MVP, se ha seguido un diseño monolítico, agrupando todos los componentes del sistema en un mismo proyecto. Si bien su modelo centralizado supone un único punto de fallo y presenta ciertas limitaciones de escalabilidad, la adopción de este enfoque no obedece a una entrega apresurada, sino a la decisión técnica de evitar la sobreingeniería y la complejidad operativa inherente a los sistemas distribuidos en esta etapa inicial.

Por otro lado, cabe enfatizar que la solución implementada, gracias a su fuerte modularidad interna, está preparada para evolucionar de forma natural hacia una arquitectura distribuida, lo cual aportaría una mayor escalabilidad y resiliencia frente a fallos. En la Figura 5.1 se ilustra gráficamente la diferencia entre ambos enfoques, contextualizando la arquitectura actual y apuntando hacia el diseño ideal.

5.1.1 Arquitectura Hexagonal

El concepto de Arquitectura Hexagonal fue propuesto por Alistair Cockburn en el artículo *The Hexagonal (Ports & Adapters) Architecture* [29] publicado en 2005. En dicho escrito, Cockburn describía una arquitectura basada en puertos y adaptadores, con el objetivo de separar el interior de la lógica de negocio del exterior, entendiendo por exterior todo aquello que no forma parte del núcleo del sistema. Esta separación permite que el corazón de la aplicación interactúe con la infraestructura sin generar acoplamiento, aumentando la mantenibilidad y la *testabilidad*. En la Figura 5.2 se muestra una representación gráfica de la Arquitectura Hexagonal.

- **Puertos:** Se refieren a las interfaces o contratos que define el dominio para la comunicación con el exterior. Por un lado, tenemos los **puertos de entrada**, que definen las operaciones que el exterior puede solicitar al núcleo. Por su parte, los **puertos de salida** representan las dependencias requeridas por el sistema.
- **Adaptadores:** Se refieren a los componentes que posibilitan la comunicación con el entorno. Los **adaptadores de entrada** actúan como traductores entre la infraestructura y el núcleo, mientras que los **adaptadores de salida** implementan los contratos definidos por los puertos de salida del dominio.



(a) Arquitectura monolítica (actual).

(b) Arquitectura distribuida (ideal).[1]

Figura 5.1: Comparativa de las arquitecturas evaluadas para el backend.

5.1.2 Desarrollo Dirigido por el Dominio

El Desarrollo Dirigido por el Dominio (DDD) es un enfoque de desarrollo de software propuesto por Eric Evans en su libro *Domain-Driven Design: Tackling Complexity in the Heart of Software* [30]. Este planteamiento complementa a la Arquitectura Hexagonal, ya que define su interpretación particular de lo que forma parte del núcleo del sistema y lo que conforma el exterior. Para Evans, el sistema se compone de cuatro capas, donde las capas más interiores no deben tener dependencias hacia las exteriores.

- **Capa de Dominio**: Constituye el corazón del sistema, donde residen las reglas de negocio, entidades y objetos de valor. Esta capa ignora completamente la existencia del resto y corresponde al núcleo de la Arquitectura Hexagonal.
- **Capa de Aplicación**: Se encarga de coordinar las operaciones del sistema delegando la lógica de negocio al dominio. Aquí se definen los *casos de uso* u operaciones que el sistema es capaz de ejecutar. Corresponde a los puertos de entrada.
- **Capa de Infraestructura**: Contiene las implementaciones técnicas necesarias para que el sistema funcione (bases de datos, APIs externas, etc.), actuando como los adaptadores de salida en la Arquitectura Hexagonal.
- **Capa de Presentación**: Es la responsable de la comunicación del núcleo del sistema con los clientes finales. Su función principal es interpretar y transformar las peticiones entrantes, delegando la ejecución de las operaciones en la capa de aplicación. Asimismo, se encarga de dar formato y devolver el resultado correspondiente al cliente.

[1] La ilustración representa la flexibilidad de la arquitectura, permitiendo que cada servicio utilice un stack tecnológico independiente. Las herramientas y bases de datos mostradas son meramente ilustrativas. La elección definitiva para cada servicio requiere un proceso de evaluación individualizado.

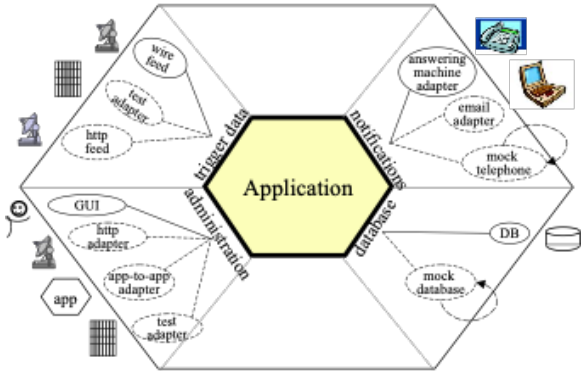


Figura 5.2: Representación visual de la arquitectura hexagonal. Fuente: Alistair Cockburn [29].

Cabe destacar que la capa de presentación, en la integración actual de DDD con la Arquitectura Hexagonal, es considerada como parte del exterior del hexágono. Los elementos de esta capa, como los controladores web, la interfaz gráfica o los procesos en segundo plano (*workers*), son tratados simplemente como adaptadores de entrada.

Por otro lado, el DDD propone una serie de conceptos clave para el modelado del sistema que se explican a continuación.

- **Entidades**: Objetos con identidad propia que mantienen su continuidad a lo largo de su ciclo de vida, incluso si sus atributos cambian. Representan conceptos del negocio.
- **Objetos de Valor (*Value objects*)**: Objetos inmutables que no poseen identidad propia. Se definen y se comparan exclusivamente por el valor de sus atributos.
- **Agregados (*Aggregates*)**: Son conjuntos de entidades y objetos de valor que se pueden tratar como una sola unidad. Su función principal es garantizar la consistencia transaccional, permitiendo que los datos del agregado sean coherentes ante los cambios.
- **Lenguaje Ubicuo**: Vocabulario común y preciso compartido por expertos del dominio y el equipo de desarrollo. Su objetivo es eliminar ambigüedades.
- **Contexto Delimitado (*Bounded context*)**: Límite lógico donde un modelo de dominio es válido y coherente. Permite que un mismo término adquiera significados o reglas de negocio diferentes en distintos contextos, asegurando la consistencia del *Lenguaje Ubicuo* en todo el sistema.

5.1.3 Command Query Responsibility Segregation

El patrón *Command Query Responsibility Segregation* (CQRS), ideado y popularizado por Greg Young, según apunta Martin Fowler [31], establece una separación

arquitectónica entre las operaciones de escritura y las de lectura. La idea fundamental reside en utilizar modelos distintos para actualizar la información (comandos) y para leerla (consultas), permitiendo que estas últimas se realicen sobre proyecciones optimizadas para maximizar la eficiencia y reducir la latencia.

Mientras que el DDD se encarga de proteger las invariantes del dominio durante las operaciones de escritura, CQRS optimiza la recuperación de información. En la lectura, el sistema evita el uso de los modelos de dominio, ya que la instanciación de entidades y value objects para validar reglas de negocio supondría una carga computacional innecesaria. Así, CQRS complementa al DDD permitiendo que las operaciones de lectura se enfoquen exclusivamente en la eficiencia.

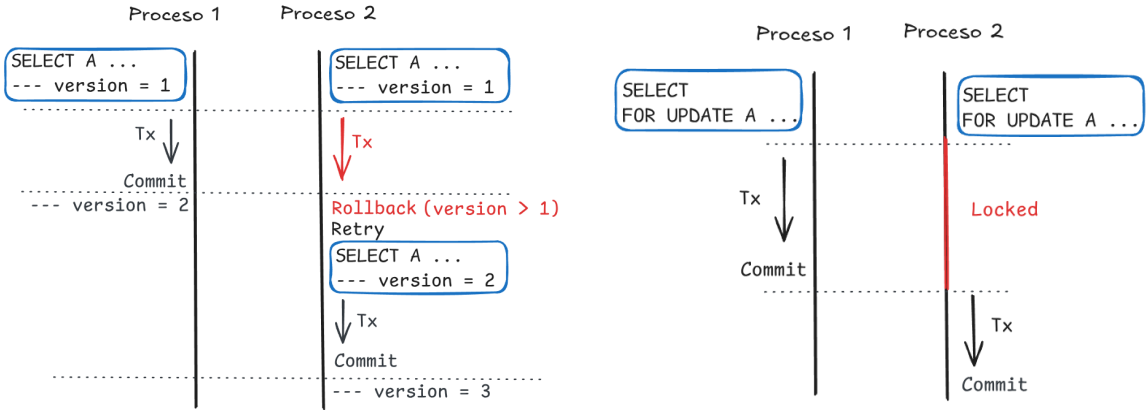
No obstante, es preciso destacar que CQRS no constituye una solución universal. Como indica Fowler [31], su implementación debe evaluarse cuidadosamente, ya que en ciertas aplicaciones o bounded contexts puede introducir una complejidad arquitectónica excesiva que no se justifica por los beneficios obtenidos.

5.1.4 Patrones de apoyo

- **Inyección de Dependencias**: Patrón orientado a la inversión de control. La responsabilidad de instanciar un objeto se delega a un componente externo, reduciendo el acoplamiento fuerte entre clases. La dependencia es suministrada (inyectada) en tiempo de ejecución.
- **Fail fast**: Principio de diseño que dicta que una operación debe interrumpirse inmediatamente al detectar un error o estado inválido. Desde el punto de vista de la seguridad, se impide propagar datos inválidos al resto del sistema. En cuanto al rendimiento, se evita el gasto innecesario de recursos computacionales.
- ***Notification pattern***: Técnica que permite acumular errores durante una operación, generalmente de validación de datos, evitando detener el proceso ante el primer fallo detectado.
- ***Result pattern***: Patrón que permite el tratamiento consistente de la respuesta de cualquier operación. Encapsula el valor de éxito o los detalles del error.
- **Unit of work**: Patrón que garantiza la atomicidad de las transacciones, haciendo un seguimiento de los cambios realizados sobre diversos objetos durante una operación de negocio.

5.1.5 Manejo de la concurrencia

En sistemas de alto rendimiento existe una alta probabilidad de que varios procesos accedan a un mismo registro simultáneamente con la intención de modificarlo. El control de concurrencia se encarga de garantizar la coherencia de los datos, evitando



(a) Control de concurrencia optimista (*Optimistic Locking*).

(b) Control de concurrencia pesimista (*Pessimistic Locking*).

Figura 5.3: Comparativa de los mecanismos de control de concurrencia a nivel de base de datos. Fuente: Elaboración propia.

problemas tales como condiciones de carrera o vulnerabilidades derivadas del tiempo de comprobación y uso (TOCTOU). En la Figura 5.3 se ilustra el comportamiento de las dos técnicas de bloqueo a nivel de base de datos evaluadas. Es preciso recalcar que estos métodos de control de concurrencia son utilizados a nivel de registro.

- ***Optimistic locking***: Utiliza un campo de versión, generalmente numérico, en la tabla para rastrear las modificaciones de un registro. El sistema lee esta versión inicialmente y, al intentar actualizar la fila, compara los valores. Si la versión actual en la base de datos es mayor a la leída, significa que otro proceso modificó el registro durante ese intervalo. Esto revierte la transacción, descartando los cambios.

  Esta técnica resulta muy eficiente en escenarios de **baja contención**, puesto que los procesos no se bloquean. Su principal desventaja es que, en caso de colisión, requiere implementar una lógica de reintentos a nivel de aplicación.

- **Pessimistic locking**: Se basa en el bloqueo explícito de un registro durante el transcurso de una transacción, garantizando que no pueda ser modificado por otros procesos hasta que dicha transacción finalice. Es el enfoque ideal para escenarios de **alta contención**, ya que evita que los procesos realicen trabajo que acabará siendo descartado.

  Mientras un proceso adquiere el bloqueo (*lock*), los demás quedan a la espera de su liberación. Aunque facilita la puesta en marcha al no requerir reintentos a nivel de aplicación, puede provocar cuellos de botella o problemas de rendimiento si se prevén demasiadas colisiones simultáneas.

  Este es el tipo de bloqueo adoptado para las operaciones críticas del sistema.

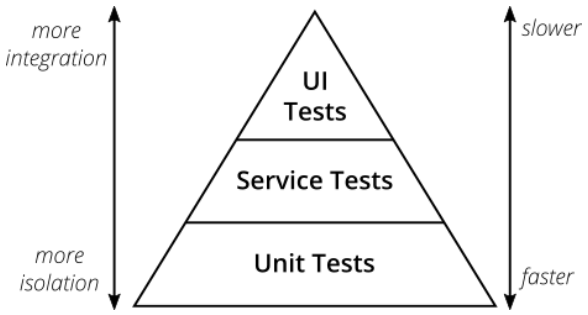


Figura 5.4: Representación visual de la pirámide de tests. Fuente: Ham Vocke [32]

5.1.6 Testing

Para garantizar la calidad del código y verificar el correcto funcionamiento del sistema, se ha diseñado una estrategia de pruebas basada en la pirámide de tests de Mike Cohn [32], ilustrada en la Figura 5.4. Este modelo defiende la necesidad de construir una base amplia de pruebas unitarias debido a su excelente relación coste/utilidad, reducir el volumen de las pruebas de integración y limitar al mínimo las pruebas *End-to-End* (E2E). Esto se debe a que el coste de desarrollo y la fragilidad del test aumentan a medida que se escala en la pirámide.

- **Pruebas unitarias:** Evalúan el comportamiento de piezas individuales del código de forma aislada, garantizando que su lógica interna sea correcta.
- **Pruebas de integración:** Verifican la interacción entre varias unidades individuales, asegurando el correcto funcionamiento y la comunicación entre ellas.
- **Pruebas E2E:** Validan flujos de negocio completos simulando el comportamiento real del sistema. Abarcan desde la petición inicial en el controlador web hasta la persistencia en la base de datos o la llamada a servicios externos.

No obstante, debido a las limitaciones temporales que acarrea cualquier proyecto de software, no ha resultado viable alcanzar una cobertura global del sistema. Por lo tanto, la estrategia de testing se ha limitado al módulo de autenticación.

5.2 Requisitos de la aplicación

Para definir los requisitos funcionales del proyecto se han utilizado una serie de diagramas de casos de uso que reflejan las acciones que puede ejecutar cada actor en cada uno de los contextos de la aplicación. A continuación se detallan los diferentes requisitos funcionales, así como los requisitos técnicos y reglas de negocio identificados para este proyecto.

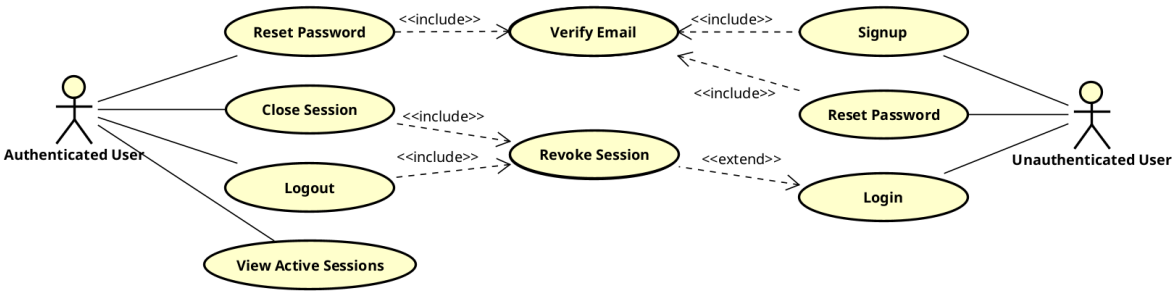


Figura 5.5: Diagrama de casos de uso del contexto autenticación.

5.2.1 Autenticación (Bounded Context Auth)

En la Figura 5.5 se muestra el diagrama de casos de uso que ejemplifica las interacciones del usuario con el bounded context *Auth*.

Casos de uso

- **CdU-01. Signup** - Un usuario no autenticado puede registrarse en el sistema indicando su nombre, email, nombre de usuario y una contraseña.
- **CdU-02. Reset Password** - Un usuario, autenticado o no, puede recuperar su contraseña verificando su identidad e indicando su nueva contraseña.
- **CdU-03. Login** - Un usuario no autenticado puede iniciar sesión en el sistema indicando su dirección de correo y su contraseña.
- **CdU-04. Close Session** - Un usuario autenticado puede cerrar una sesión (diferente a la actual) indicando su identificador.
- **CdU-05. Logout** - Un usuario autenticado puede cerrar su sesión actual.
- **CdU-06. View Active Sessions** - Un usuario autenticado puede ver el listado de sus sesiones activas.
- **CdU-07. Revoke Session (*Sub-caso de uso*)** - El sistema puede invalidar una sesión en respuesta a una petición del usuario (include) o en base a una política del sistema (extend).
- **CdU-08. Verify Email (*Sub-caso de uso*)** - El sistema puede verificar la identidad del usuario.

Reglas de negocio

- **RN-01. Creación de la cuenta de usuario** - Un usuario no se puede crear con ningún estado intermedio (ej: pendiente). Solo se puede registrar tras completar la verificación de identidad.
- **RN-02. Recuperación de contraseña** - Un usuario puede recuperar su contraseña tras completar la verificación de identidad.
- **RN-03. Política de contraseñas** - Durante el flujo de recuperación de contraseña, la nueva contraseña del usuario no puede coincidir con la actual.
- **RN-04. Política de sesiones** - El usuario puede tener hasta un máximo de 6 sesiones activas al mismo tiempo. El flujo de inicio de sesión puede revocar una o más sesiones (las más antiguas) para cumplir con esta política.

Requisitos técnicos

- **RT-01. Token de acceso** - El acceso a recursos protegidos se gestiona mediante *JSON Web Tokens* (JWT) de corta duración (15 minutos).
- **RT-02. Token de renovación (*Refresh Token*)** - La renovación de sesión se gestiona mediante tokens opacos de alta entropía (48 caracteres) y de larga duración (15 días). Estos tokens se procesan mediante el algoritmo **HMAC-SHA256** utilizando una clave secreta (*pepper*) para su posterior almacenamiento seguro.
- **RT-03. Seguridad de tokens** - Los tokens viajan en *cookies* con la siguiente configuración: `HttpOnly=true;SameSite=lax;Secure=true`. Los tokens de acceso no se almacenan en ningún tipo de almacenamiento (*storage*) del navegador.
- **RT-04. Verificación de identidad** - La identidad del usuario se comprueba mediante *One-Time Passwords* (OTP) o tokens temporales de 8 dígitos numéricos de corta duración (15 minutos) enviados a la dirección de correo del usuario.
- **RT-05. Seguridad de datos sensibles** - Las credenciales de usuario y los códigos OTP se almacenan de forma segura usando el algoritmo *bcrypt*.
- **RT-06. Metadatos y datos identificativos** - El sistema captura y procesa metadatos del usuario, como la IP y el *User-Agent*, con fines de auditoría. Los datos identificativos (IP) se procesan mediante el algoritmo **HMAC-SHA256** utilizando una clave secreta (*pepper*) para su posterior almacenamiento seguro.
- **RT-07. Estrategia de *logging*** - El sistema registra eventos usando metadatos y datos identificativos del usuario (IP) con fines de auditoría. Los datos se almacenan un máximo de 15 días hasta su destrucción.

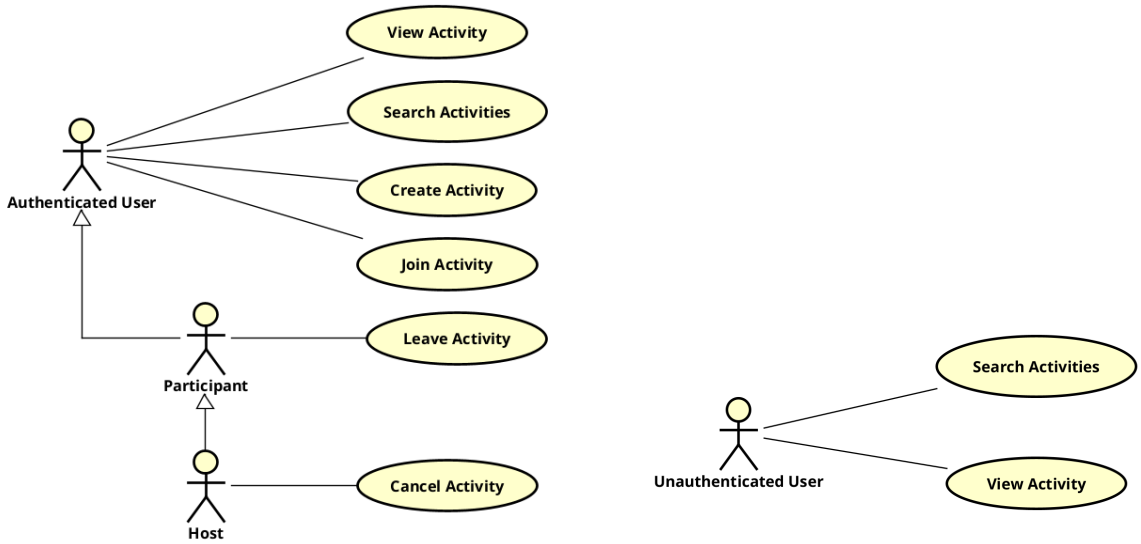


Figura 5.6: Diagrama de casos de uso del contexto actividades.

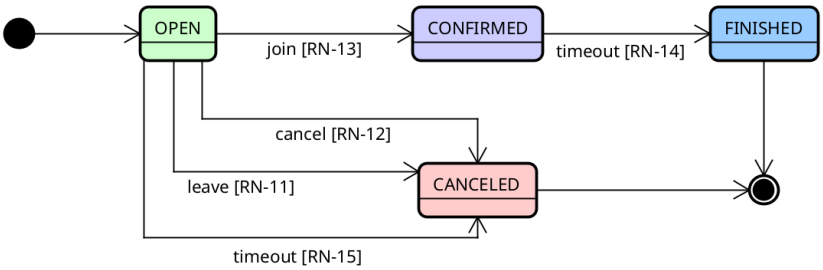


Figura 5.7: Diagrama de estados de una actividad.

- **RT-08. Usabilidad de la interfaz** - Los flujos de autenticación utilizan pantallas de tipo *wizard* con hasta 3 pasos, con el fin de asistir al usuario y mejorar su experiencia.

5.2.2 Actividades (Bounded Context Activities)

Por su parte, la Figura 5.6 ilustra el esquema de casos de uso, detallando las acciones que cada actor puede realizar dentro del bounded context *Activities*.

Adicionalmente, en la Figura 5.7 se muestra el diagrama de estados de una Actividad, incluyendo sus relaciones con las reglas de negocio.

Casos de uso

- **CdU-09. View Activity** - Un usuario, autenticado o no, puede visualizar los detalles de una actividad.

- **CdU-10. Search Activities** - Un usuario, autenticado o no, puede buscar actividades seleccionando una ubicación en un mapa y una serie de filtros tales como el deporte, uno o varios estados de la actividad, el número de plazas libres, uno o varios niveles y el orden en el que visualizar las actividades.
- **CdU-11. Create Activity** - Un usuario autenticado puede crear una actividad indicando un deporte, un título, una descripción (opcional), una fecha de celebración y una configuración de la actividad. La configuración de la actividad es variable y dependerá del deporte seleccionado.
- **CdU-12. Join Activity** - Un usuario autenticado puede unirse a una actividad cuyo aforo no esté completo, no se haya celebrado aún y permanezca en estado **OPEN**.
- **CdU-13. Leave Activity** - Un participante puede abandonar una actividad cuando esta no se haya celebrado, su tiempo de baja no se haya alcanzado y permanezca en estado **OPEN**.
- **CdU-14. Cancel Activity** - Un anfitrión puede cancelar una actividad cuando el número de participantes (excluyendo al anfitrión) sea cero y la actividad no se haya celebrado.

Reglas de negocio

- **RN-08. Inscripciones** - Un usuario puede inscribirse en una actividad cuando esta permanece en estado OPEN, su aforo no está completo y no se ha superado la hora de inicio (incluyendo un periodo de gracia de 10 minutos).
- **RN-09. Bajas** - Un usuario puede abandonar una actividad solo si esta permanece en estado OPEN y restan más de 60 minutos hasta el inicio de la actividad.
- **RN-10. Sucesión del anfitrión** - El anfitrión de una actividad puede abandonarla. Si hay más participantes en ella, el sistema designa como nuevo anfitrión al participante más antiguo.
- **RN-11. Cancelación por orfandad** - El anfitrión de una actividad puede abandonarla. Si no hay más participantes en ella, el sistema cancela automáticamente la actividad, quedando sin anfitrión ni participantes.
- **RN-12. Cancelaciones** - El anfitrión de una actividad solo puede cancelar una actividad si el número de participantes, exceptuando al propio anfitrión, es cero.
- **RN-13. Confirmación automática** - Una actividad que se encuentre en estado OPEN transiciona automáticamente al estado CONFIRMED cuando se produce una inscripción exitosa y el número de participantes es igual a la capacidad mínima de la actividad.

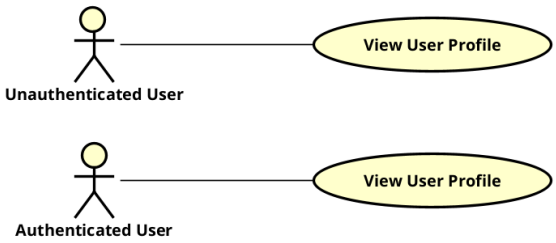


Figura 5.8: Diagrama de casos de uso del contexto usuarios.

- **RN-14. Finalización automática** - Una actividad que se encuentre en estado CONFIRMED transiciona automáticamente al estado FINISHED una vez que concluye su tiempo de duración programado desde la hora de inicio. En caso de que la duración no sea especificada, el tiempo es de 5 horas desde la hora de inicio.
- **RN-15. Cancelación automática** - Una actividad que se encuentre en estado OPEN transiciona automáticamente al estado CANCELED una vez que se alcanza su hora de inicio.
- **RN-16. Creación de actividades** - Un usuario puede crear una actividad con un mínimo de 90 minutos de antelación hasta un máximo de 30 días.

Requisitos técnicos

- **RT-09. Rendimiento de las búsquedas** - La búsqueda de actividades se restringe a un punto geográfico y un radio máximo de búsqueda (hasta 50km), con el fin de aprovechar los índices geoespaciales PostGIS y optimizar el tiempo de respuesta.
- **RT-10. Usabilidad de la interfaz de creación** - La creación de una actividad utiliza pantallas de tipo wizard con hasta 4 pasos, con el fin de asistir al usuario y mejorar su experiencia.

5.2.3 Usuarios (Bounded Context Users)

La Figura 5.8 muestra un diagrama de casos de uso que ilustra la única interacción que un usuario, ya sea autenticado o no, puede llevar a cabo con el bounded context *Users*.

Casos de uso

- **CdU-15. View User Profile** - Un usuario, autenticado o no, puede visualizar los detalles del perfil de un usuario.

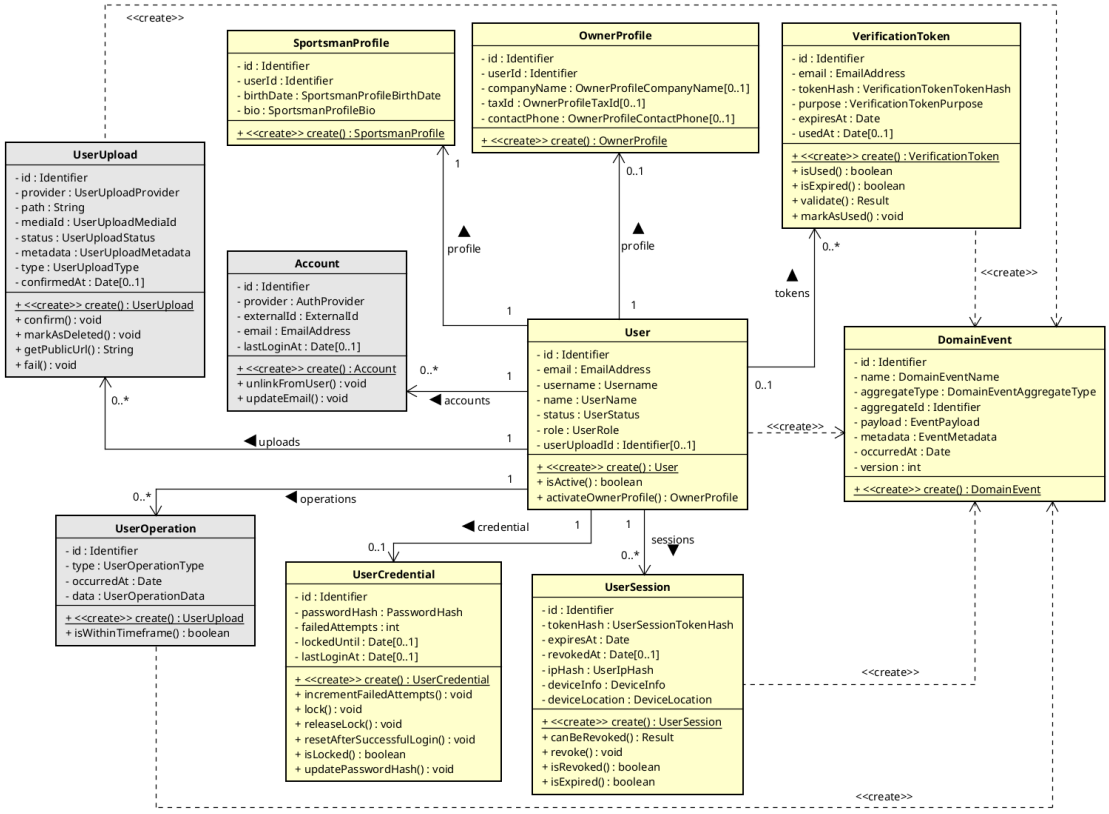


Figura 5.9: Diagrama de clases del contexto de autenticación.

5.3 Modelo del dominio

En un proyecto donde el dominio es la piedra angular del desarrollo, resulta fundamental representar gráficamente las entidades que lo componen, sus atributos y las relaciones que se establecen entre ellas. Partiendo de esta idea, el modelo de dominio representa la base de conocimiento sobre el problema tratado, facilitando la comprensión de las reglas de negocio y sirviendo de guía para el diseño del sistema.

5.3.1 Autenticación (Bounded Context Auth)

En la Figura 5.9 se ilustra el diagrama de clases del bounded context *Auth*. Tal como se aprecia, el eje central es la clase `User`, la cual se relaciona con todas las demás entidades y sirve como agregado raíz para la gran mayoría de las operaciones de autenticación. Por otro lado, el conjunto de clases con fondo amarillo corresponde a los elementos que forman parte del MVP desarrollado, mientras que las clases con fondo gris complementan el diseño global del sistema. Con el fin de asegurar la calidad del producto y cumplir con los plazos de desarrollo, se ha decidido posponer la implementación de estas últimas para futuras líneas de trabajo del proyecto.

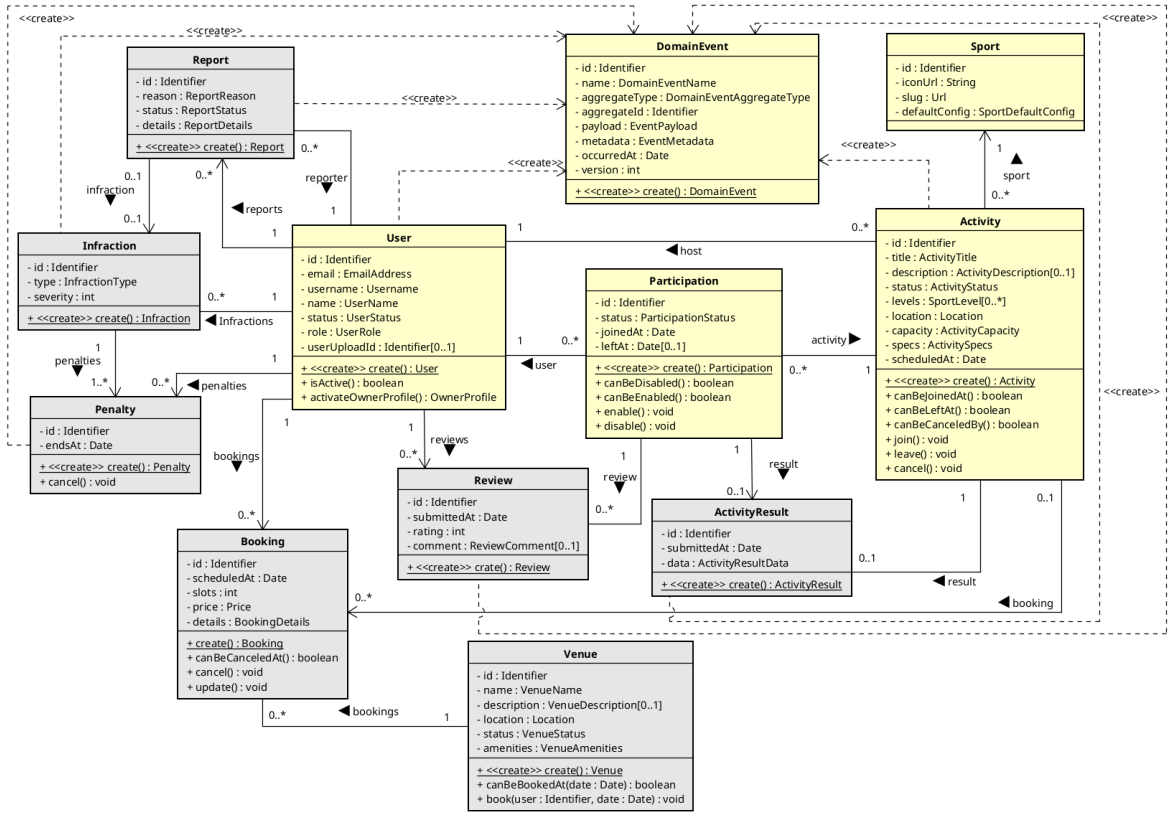


Figura 5.10: Diagrama de clases del contexto de actividades.

5.3.2 Actividades (Bounded Context Activities)

Por su parte, la Figura 5.10 muestra el diagrama de clases del bounded context *Activities*. El pilar de este esquema es la clase `Activity`, la cual actúa como el agregado raíz de todos los flujos de este contexto. Del mismo modo, se mantiene la distinción entre clases con fondo amarillo y gris bajo el mismo criterio comentado anteriormente. Cabe destacar que, aunque en el diagrama se muestra una relación directa de las clases `Penalty`, `Infraction` y `Report` con `User`, esto se ha modelado así por pura simplicidad visual. En realidad, el diseño contempla una relación polimórfica que les permite vincularse con otras entidades del sistema, tales como `Booking`, `Venue` o la propia `Activity`.

5.4 Modelo de la base de datos

El modelo de datos refleja fielmente la lógica del sistema desarrollado. En la Figura 5.11 se ilustra el diagrama Entidad-Relación (ER), detallando las tablas, sus atributos y la multiplicidad de sus conexiones.

Como se mencionó en apartados anteriores, se ha empleado PostgreSQL como siste-

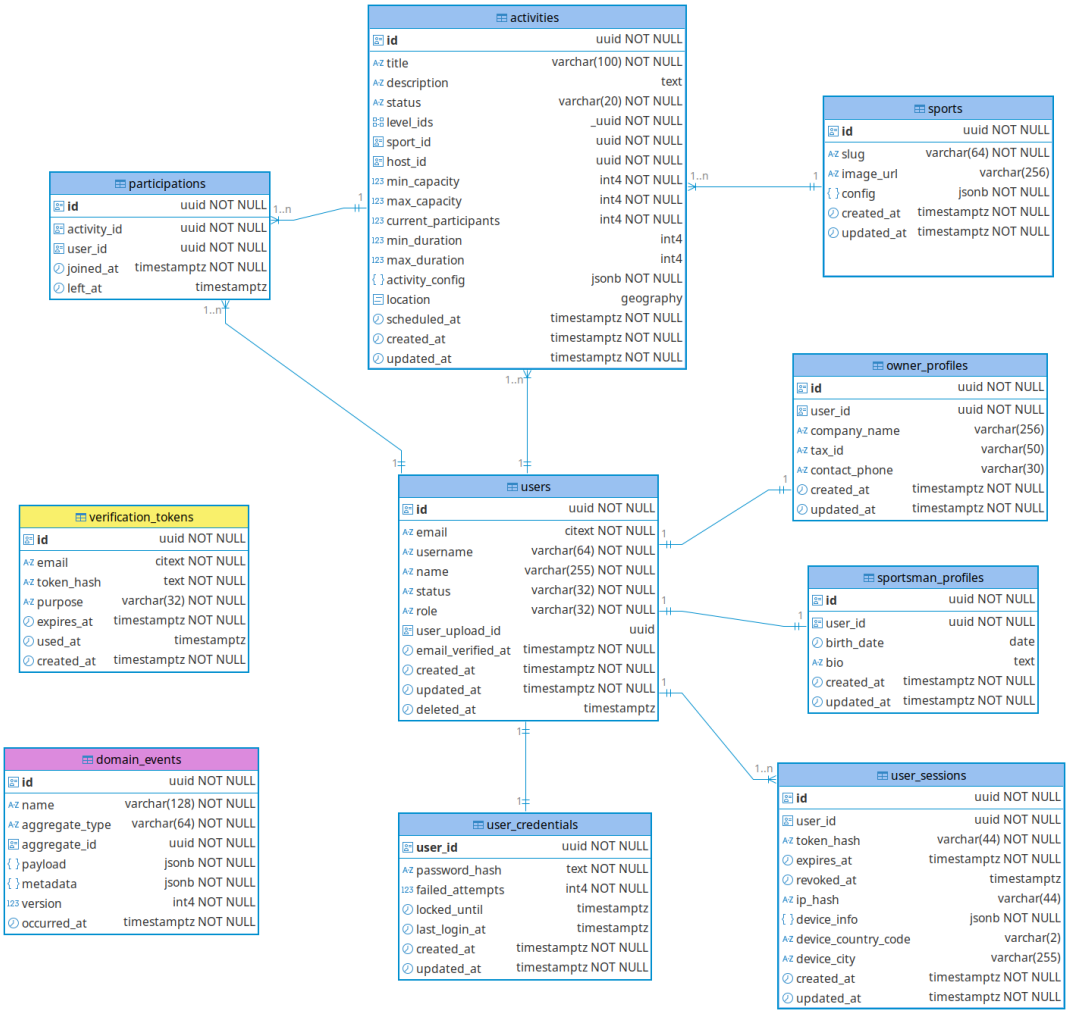


Figura 5.11: Modelo Entidad-Relación de la base de datos.

ma gestor de bases de datos, junto con el ORM TypeORM para facilitar su interacción desde el backend. La estructura de las tablas se ha construido utilizando la API de migraciones de TypeORM. Si bien esta herramienta permite generar migraciones de forma automática, se ha optado por una redacción manual de las mismas con el objetivo de mantener un control granular sobre la evolución de la base de datos.

Por otro lado, resulta conveniente dejar de manifiesto que el diagrama no refleja la realidad en su totalidad debido a las limitaciones que presenta el software con el que fue generado (DBeaver). En consecuencia, es preciso matizar los siguientes puntos:

- **Relaciones lógicas sin clave foránea:** Según se observa en el diagrama ER, la tabla `verification_tokens` no tiene ninguna relación con las demás entidades. Sin embargo, la tabla está vinculada a `users` a través de la columna `email`. Dado que el sistema exige que el usuario verifique su dirección de correo antes de crear

su cuenta, no es posible consolidar esta relación mediante una clave foránea.

- **Relaciones polimórficas:** La tabla `domain_events` implementa un diseño polimórfico, pudiendo estar vinculada a diferentes tablas. Dado que estas relaciones no se establecen a través de una clave foránea, el diagrama no es capaz de representarlas. En su lugar, las tablas se relacionan mediante el uso combinado de las columnas `aggregate_type` y `aggregate_id`. En la práctica, la tabla está relacionada con: `users`, `activities`, `verification_tokens` y `user_sessions`.

Finalmente, cabe destacar la aplicación de técnicas de *desnormalización de datos* en el esquema relacional. Este procedimiento introduce redundancia de forma intencionada con el objetivo de optimizar los tiempos de respuesta en consultas complejas. Un ejemplo representativo de esta técnica es la columna `current_participants` en la tabla `activities`. Este dato desnormalizado evita la necesidad de ejecutar funciones de agregación constantes sobre la tabla `participations`, reduciendo la carga de procesamiento y mejorando significativamente el rendimiento en las operaciones de lectura y búsqueda de actividades.

6 Implementación del sistema

Este capítulo aborda los detalles de implementación del sistema. Con el objetivo de simplificar la lectura y facilitar la comprensión de la arquitectura desarrollada, se ha acotado el análisis del backend a tres casos de estudio que ilustran los flujos más representativos de la plataforma: *LoginUser*, *CreateActivity* y *GetActivities*.

Para sustentar la explicación técnica de estas operaciones, nos apoyaremos en el uso de diagramas de secuencia. Un diagrama de secuencia es una herramienta visual que permite representar la comunicación entre los distintos actores involucrados en el proceso, incluyendo el intercambio cronológico de mensajes entre ellos. Por tanto, resulta un recurso idóneo para demostrar la división de responsabilidades y el flujo de la información a través de las distintas capas del sistema.

Antes de profundizar en los detalles técnicos de cada operación, resulta conveniente facilitar los puntos de acceso al resultado práctico de este proyecto. El código fuente ha sido liberado para su consulta, y el producto final se encuentra desplegado en un entorno de producción. A continuación, se detallan los enlaces correspondientes:

- **Plataforma web:** `https://tfg.cponton.com/`
- **Repositorio Frontend:** `https://github.com/carlosp1604/lobi-app-web`
- **Repositorio Backend:** `https://github.com/carlosp1604/lobi-app-back`

6.1 Caso de Estudio 1: *Login*

En la Figura 6.1 se muestra el diagrama de secuencia del caso de uso *LoginUser*. Cabe destacar que se ha optado por agrupar tanto las entidades de dominio como los repositorios participantes por mera simplicidad visual. Asimismo, resulta conveniente señalar que se ha incluido la capa a la que pertenece cada pieza con el fin de mostrar la división de responsabilidades y evidenciar cómo fluye la información a través del sistema de principio a fin.

6.1.1 Implementación del flujo

El flujo de inicio de sesión trasciende la simple consulta a la base de datos y la validación de credenciales en texto plano. Este proceso se orquesta estratégicamente para proteger al sistema frente a entradas inválidas o maliciosas, preservar la coherencia

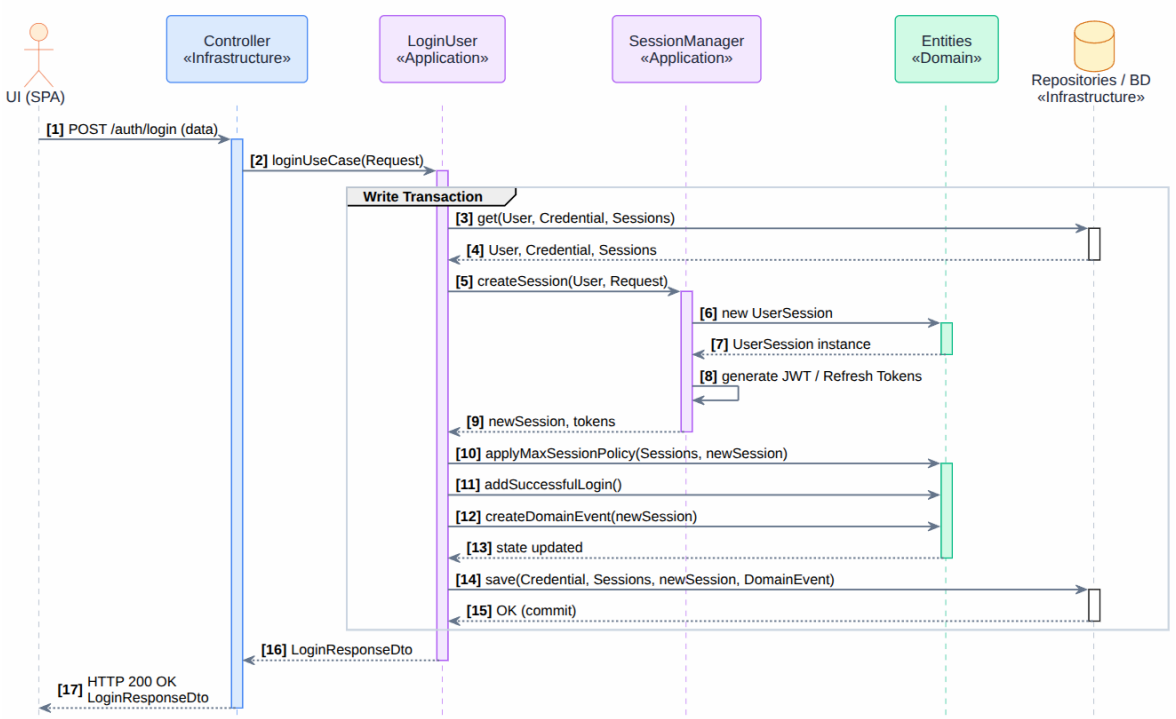


Figura 6.1: Diagrama de secuencia del caso de uso *LoginUser*.

de la información, evitar problemas derivados de la concurrencia y añadir una capa de abstracción que impide la filtración de datos sensibles. Tomando como referencia el diagrama de secuencia, el flujo se divide en las siguientes fases clave:

- **Recepción y validación de la petición HTTP:** El primer componente en intervenir es el manejador de ruta de API, actuando como adaptador de entrada. Su función es interpretar la petición y delegar la ejecución en la capa de aplicación. Adicionalmente, realiza una validación temprana utilizando las utilidades integradas de NestJS. Para evitar redundancias, esta verificación se limita a comprobar la estructura del cuerpo, rechazando inmediatamente las solicitudes malformadas y delegando las reglas complejas a las capas internas.

- **Validación de los datos en Aplicación:** Una vez que la solicitud alcanza el caso de uso, se ejecuta una validación profunda apoyándose en los value objects. Esta etapa comprende una verificación secuencial, abortando la ejecución en el instante en el que se detecta un dato inválido, tal y como dicta el patrón fail fast.

- **Apertura del contexto transaccional:** Superada la validación, el caso de uso procede a orquestar la operación. Dado que el inicio de sesión requiere múltiples accesos a la base de datos, un fallo intermedio podría generar un estado inconsistente. Para mitigar este riesgo, se emplea el patrón unit of work. Este

componente envuelve la ejecución en una transacción atómica, de modo que si cualquiera de los pasos fracasa, todos los cambios previos se revierten.

- **Delegación en el dominio y políticas de sesión:** Dentro de la transacción, las entidades se recuperan de forma independiente mediante sus respectivos repositorios, optimizando el rendimiento y evitando añadir complejidad a los modelos de dominio. Durante la lectura del agregado raíz de la operación, *User*, se aplica un bloqueo pesimista (pessimistic locking) en la base de datos, evitando condiciones de carrera si otro proceso intenta actualizar el estado del mismo agregado. Tras validar criptográficamente la credencial, se delega en el servicio *SessionManager* la emisión de los tokens. Posteriormente, utilizando las entidades de dominio, se aplican las reglas del negocio, evaluando la política de límite de sesiones concurrentes y revocando conexiones antiguas si es necesario. En respuesta a estas acciones, las entidades de dominio registran sus respectivos eventos.

- **Confirmación de la transacción:** Cuando se han aplicado todas las reglas y la coordinación ha finalizado, el unit of work confirma la transacción. Esto persiste el nuevo estado del agregado y libera el bloqueo sobre el registro, permitiendo que otros procesos continúen su ejecución. Finalmente, el caso de uso utiliza traductores especializados para devolver los datos definidos en su contrato.

- **Respuesta HTTP:** La respuesta del caso de uso se encapsula mediante el result pattern, proporcionando un control de flujo predecible sin recurrir al complejo manejo de excepciones. En caso de éxito, el manejador de ruta de API devuelve los tokens de acceso en una respuesta `200 OK`. Cabe recalcar que estos tokens también se incluyen en las *cookies* con una configuración pensada para garantizar la seguridad. En caso de fallo, este componente traduce el error en la excepción HTTP correspondiente. Aquí cabe destacar la aplicación de la técnica de **ofuscación de errores**, que consiste en agrupar errores de diferente naturaleza bajo un error genérico para no dar pistas útiles a posibles atacantes.

6.1.2 Cobertura de Tests

Para garantizar la fiabilidad y la robustez del sistema, se ha implementado una estrategia basada en la pirámide de pruebas, tal como se detalló en el capítulo anterior. Dada la criticidad que supone el flujo de inicio de sesión, no se han escatimado recursos, asegurando una cobertura exhaustiva de todos los componentes involucrados a través de los siguientes niveles:

- **Pruebas unitarias:** Cubren las entidades de dominio, los servicios, los repositorios, los value objects, el caso de uso y el manejador de ruta de API. En este nivel base, se ha alcanzado una cobertura del 100%.

- **Pruebas de integración:** Abarcan el caso de uso y los repositorios, incluyendo un escenario específico para verificar la correcta gestión de la concurrencia en base de datos.
- **Pruebas E2E:** Validan el flujo completo del endpoint, abarcando desde la recepción de la petición HTTP hasta la validación de la respuesta retornada.

En este contexto, resulta fundamental destacar un principio básico de la pirámide de pruebas: evitar el solapamiento de casos. El objetivo es que cada prueba se enfoque exclusivamente en validar un aspecto concreto del componente. Si bien eludir la redundancia implica que la confianza de los niveles superiores recae sobre la eficacia de los inferiores, esto no se considera un defecto de diseño, sino una implementación fidedigna de la pirámide de tests. Cada nivel constituye los cimientos del inmediatamente superior.

Entrando en los detalles de cada nivel, en la suite de **pruebas unitarias** destaca el uso del patrón metodológico *Arrange, Act, Assert* (AAA). En esta fase, el objetivo es validar la correcta orquestación del sistema, comprobando mediante dobles de prueba (*mocks*) que las dependencias han sido llamadas con los parámetros adecuados y que el resultado cumple el contrato establecido.

Por su parte, las **pruebas de integración** mantienen esta misma estructura conceptual, pero introducen comprobaciones directas sobre el estado de la base de datos. El propósito primordial en este punto no es reevaluar los aspectos ya cubiertos por el test unitario, sino garantizar que las piezas del sistema interactúan correctamente y que la persistencia converge hacia el estado esperado tras la operación.

Finalmente, en la cúspide de la pirámide, los **tests E2E** aseguran que la información atraviesa todas las capas de la aplicación con éxito. La validación se centra exclusivamente en confirmar que el cuerpo de la respuesta HTTP y las *cookies* de sesión devueltas son correctas. Los detalles internos de cada componente quedan al margen en esta fase, asumiendo con total seguridad que su correcto funcionamiento ya ha sido avalado por las pruebas de los niveles inferiores.

6.2 Caso de Estudio 2: *CreateActivity*

En la Figura 6.2 se puede observar el diagrama de secuencia del comando *CreateActivity*. En este flujo intervienen los mismos actores que en el caso anterior, por lo que, para evitar redundancias, el análisis se centrará en los aspectos diferenciales de esta operación.

Cabe destacar que el uso del término *comando* atiende a la filosofía aplicada en este diseño. Mientras que *LoginUser* sigue un enfoque basado únicamente en DDD, en este caso se complementa con el patrón CQRS. Si bien este patrón modifica sustancialmente el flujo de las operaciones de lectura (*queries*), en el caso de los comandos de escritura no presenta cambios estructurales significativos con respecto a un caso de uso tradicional.

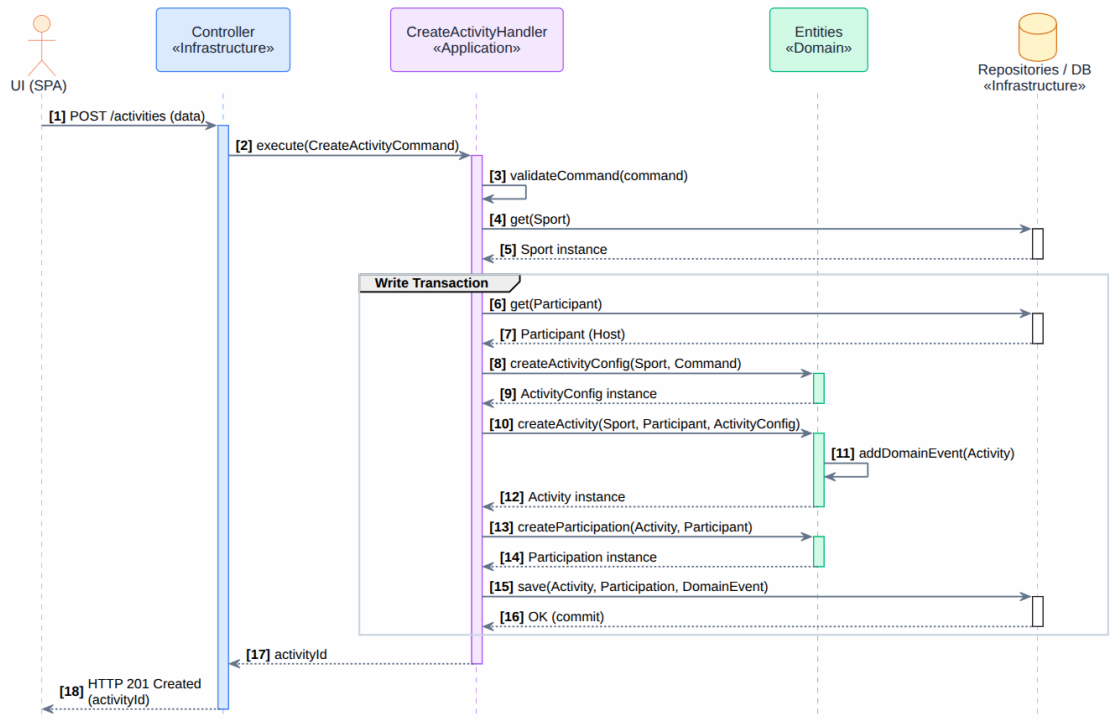


Figura 6.2: Diagrama de secuencia del comando *CreateActivity*.

- **Guard de Autenticación:** Un guard es un concepto específico del entorno de NestJS que consiste en un bloque de código ejecutado inmediatamente antes del manejador de ruta de API. Aunque es posible agregar cualquier tipo de lógica a estas funciones, en este caso se utiliza para extraer el token de acceso de la *cookie* recibida en la petición HTTP. Adicionalmente, realiza validaciones criptográficas, estructurales y temporales: comprueba que la firma del token JWT sea válida, verifica los datos del *payload* y se asegura de que no haya caducado. Un error en este componente desencadena una respuesta `401 Unauthorized`, deteniendo el flujo de inmediato sin llegar a alcanzar el manejador.

- **Acumulación de errores:** Para optimizar la experiencia de usuario en el manejo de formularios, resulta adecuado acumular los errores de validación para presentarlos agrupados, en lugar de rechazar la petición ante el primer dato inválido encontrado. Para lograrlo, el manejador del comando hace uso del notification pattern. De este modo, el endpoint se encarga de recopilar y anexar todos los detalles relativos a los fallos de validación en la respuesta de error correspondiente.

Además, este flujo presenta una complejidad técnica añadida: la aplicación soporta la creación de actividades con información variable de acuerdo con el deporte seleccionado. Para solventar este reto, se ha diseñado una solución basada en polimorfismo, tal y como se detalla en el siguiente apartado.

Sistema de deportes basado en composición Dada la naturaleza altamente variable de las reglas y características de cada deporte, se han evaluado dos estrategias arquitectónicas para su implementación:

- **Basada en herencia:** Consiste en definir una clase base abstracta y extenderla en múltiples subclases para añadir la información específica de cada deporte. Aunque funcional, esta solución presenta serios problemas de extensibilidad, ya que añadir nuevas reglas provocaría una explosión de clases y anclaría cada modelo a la rigidez inherente de la herencia.
- **Basada en composición:** Propone que la entidad base del deporte se limite a reunir características universales. Para añadir reglas, métricas o requisitos, se le inyectan *plugins* reutilizables y personalizables. De esta forma, cada deporte evoluciona de manera independiente. En un sistema diseñado para abarcar un amplio catálogo de disciplinas con distintos niveles de desarrollo, esta flexibilidad resulta esencial.

Bajo este modelo de composición, la solución se sustenta en dos conceptos clave:

- ***Specs* (Especificaciones):** Representan verdades intrínsecas o características universales de un deporte que pueden, o no, ser personalizadas por los usuarios. Por ejemplo, la configuración de participantes de una actividad.
- ***Capabilities* (Capacidades):** Definen la información orientada a la creación de actividades. Indican qué datos específicos de cada deporte se pueden o deben configurar al organizar un encuentro. Por ejemplo, la duración del evento.

Dado que ambos conceptos presentan naturalezas tan diferentes, se han modelado como conjuntos de datos independientes.

Ciclo de vida de una *Capability*/*Spec* Si bien la composición resuelve la rigidez estructural, introduce un reto técnico significativo a la hora de gestionar el ciclo de vida de estos objetos. Puesto que cada *spec* o *capability* presenta una estructura de datos diferente, el sistema debe ser capaz de procesar colecciones de datos polimórficas.

El uso de sentencias condicionales masivas (`if/switch`) para cubrir cada variante no es una solución viable debido a que genera código repetitivo y viola el Principio de Diseño Abierto/Cerrado (OCP). Para solventar este problema se utilizan factorías en dos capas de la arquitectura:

- **Capa de Aplicación (Validación, Esquemas y Traductores):** Definir estáticamente todos los posibles datos de un deporte en el DTO de entrada es inviable. En su lugar, el caso de uso extrae los datos crudos y delega en una factoría para obtener el validador adecuado para cada *plugin*. Además, este mismo validador tiene la responsabilidad de preparar y exponer los esquemas que

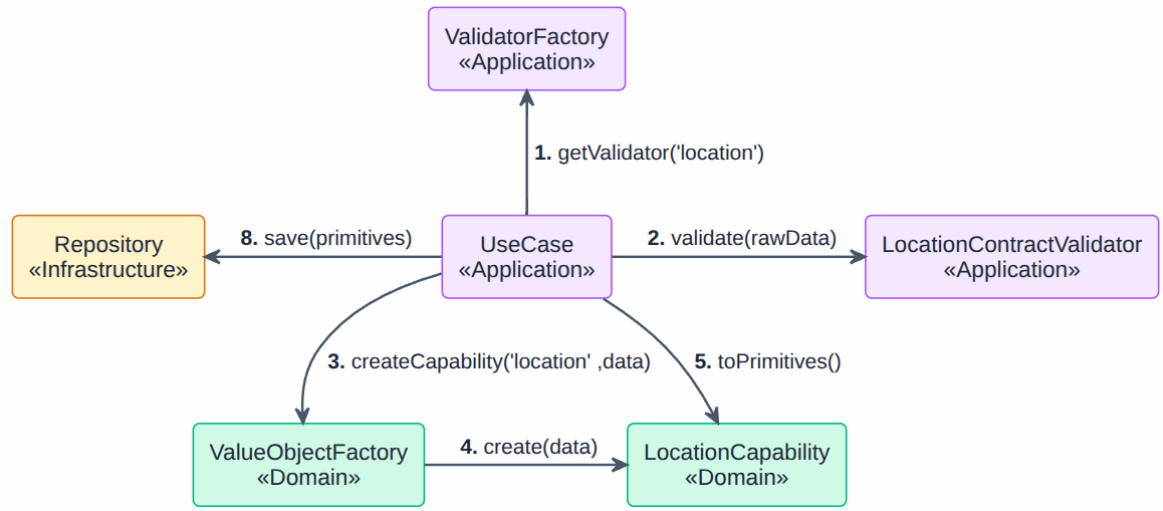


Figura 6.3: Diagrama de comunicación del ciclo de vida de una *Capability* durante el flujo *CreateActivity*.

acepta cada *capability* o *spec* hacia el exterior. Por último, el sistema dispone de una factoría especializada para obtener los traductores que se encargarán de formatear los datos adecuadamente para que el frontend los pueda representar.

- **Capa de Dominio (Value objects):** Las *capabilities* y *specs* se han modelado como value objects. La naturaleza de los datos deportivos facilita su comparación por valor, justificando la ausencia de identidad propia. La factoría de dominio es la responsable de construir estos objetos, los cuales gestionan su propia validación. Finalmente, la persistencia se logra gracias a que cada uno de estos objetos de valor implementa la interfaz `Serializable`. De esta forma, son capaces de exponer sus datos internos sin romper el encapsulamiento.

Gracias a este diseño, el sistema es capaz de reconstruir o instanciar cualquiera de estos *plugins* en cualquier flujo. En la Figura 6.3 se muestra un diagrama de comunicación con el proceso mediante el cual el flujo *CreateActivity* valida, construye y almacena una *capability*. El proceso de reconstrucción presenta las mismas características, al igual que el flujo equivalente de los *specs*. Por último, para mejorar la experiencia de desarrollo, se han implementado registros estáticos para estas factorías. Aunque esta decisión introduce un ligero concepto de infraestructura, es un compromiso técnico aceptado conscientemente a cambio de aprovechar todo el potencial del tipado estático del compilador y el autocompletado.

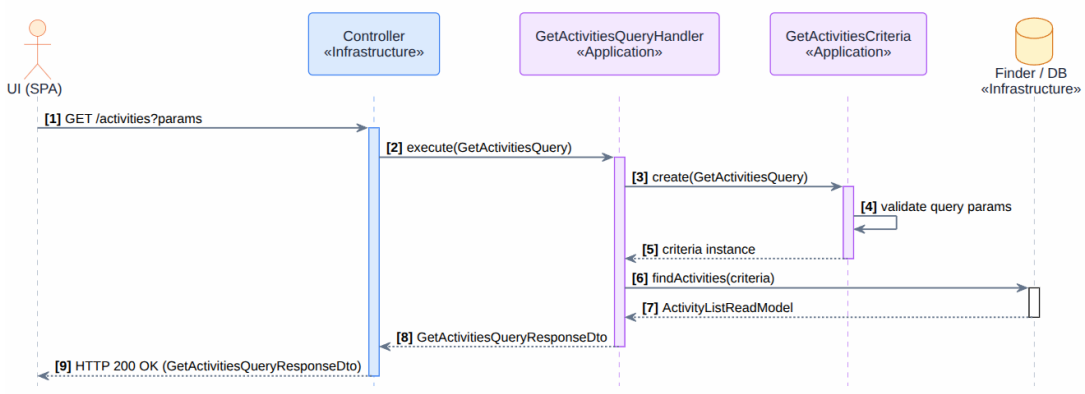


Figura 6.4: Diagrama de secuencia de la consulta *GetActivities*.

6.3 Caso de Estudio 3: *GetActivities*

La Figura 6.4 ilustra el diagrama de secuencia de la consulta *GetActivities*. A pesar de que este esquema se asemeja a los gráficos anteriores, se puede identificar una diferencia clave: la ausencia de la capa de dominio. Esto se debe al uso del patrón CQRS, el cual se enfoca en optimizar el rendimiento de las lecturas evitando la sobrecarga computacional que supone utilizar las entidades de dominio. Cabe aclarar que, aunque los value objects sí participan en la validación inicial de los parámetros, el dominio no interviene en el núcleo de la operación.

Para analizar este flujo, el análisis se centrará en sus aspectos particulares y diferenciales con respecto a las operaciones analizadas previamente:

- **Autenticación opcional:** El problema de las consultas *N+1* (*N+1 Queries*) puede degradar el rendimiento del sistema. Para evitar este obstáculo, el manejador de ruta de API incorpora una instrucción que impide al guard de autenticación lanzar una excepción `401` si la autenticación falla. Esto permite a la consulta ejecutarse tanto si la sesión es anónima como si existe un usuario acreditado. En este último caso, la consulta aprovecha para recuperar los datos que relacionan al usuario autenticado con cada actividad devuelta.

- **Validación de la entrada:** La verificación de los parámetros sigue utilizando los value objects. Sin embargo, en esta operación aparece una nueva pieza: el patrón *Criteria*. Este componente se encarga de validar la entrada y preparar los filtros de búsqueda. Cabe considerar que la paginación y la ordenación de resultados no conciernen a la lógica de negocio, sino que constituyen mecanismos de presentación y rendimiento. Por lo tanto, el *Criteria* representa un concepto de la capa de aplicación.

- **Ejecución de la consulta:** En esta etapa aparecen los *finders* como nuevos actores. Mientras que un repositorio se encarga de reconstruir agregados del do-

minio, los *finders* recuperan modelos de lectura a partir de proyecciones de la base de datos optimizadas para consultas. Estos modelos de lectura normalmente se definen en la capa de aplicación, por lo que su formato es cercano a los DTO finales.

Puesto que el dominio no interviene en este proceso, surge un nuevo desafío habitual en sistemas CQRS: la duplicación de las reglas de negocio en las consultas. Para mitigar este problema, se hace uso de especificaciones reutilizables que el *finder* aplica durante la consulta.

- **Devolución de los resultados:** Aunque los modelos de lectura obtenidos por la consulta pueden servir como respuesta final, el sistema de deportes basado en composición necesita lógica extra. Es necesario delegar en los traductores de la capa de aplicación para transformar los *plugins* presentes en la configuración de cada actividad en la información requerida por el frontend. En este paso también se integran los datos que relacionan al usuario autenticado con cada actividad.

6.4 Frontend

La interfaz de usuario constituye uno de los componentes principales del producto final. Su desarrollo ha consistido, principalmente, en el maquetado de los diferentes elementos que conforman el frontend. No obstante, a continuación se detallan los tres aspectos que han supuesto verdaderos retos técnicos.

6.4.1 Auth Context

La autenticación del usuario es uno de los flujos más críticos del frontend. En un framework basado en el ecosistema React [40], una de las soluciones más adecuadas para abordar este aspecto es mediante los *contexts*. Esta herramienta permite dar soporte a estados globales que necesitan ser accedidos desde diferentes puntos de la interfaz. Además, se hace uso de *reducers* para organizar la lógica del contexto y mejorar la legibilidad del código. Este componente supone la base del resto de procesos que soporta el cliente, por lo que es fundamental que su implementación cubra los supuestos más críticos detectados.

En la Figura 6.5 se muestra un diagrama que ilustra los estados y las transiciones posibles del *auth context*.

- **Inicialización y recarga de página:** Durante el arranque, la aplicación no dispone de la información de sesión del usuario. Por ello, el *context* se inicializa en el estado *loading*, que representa una situación de transición. Para permitir que la aplicación arranque o se recupere tras un refresco de página, se utiliza el *localStorage*. Esta utilidad permite persistir el estado de la sesión. Para evitar riesgos de seguridad, se almacena únicamente la información básica del usuario

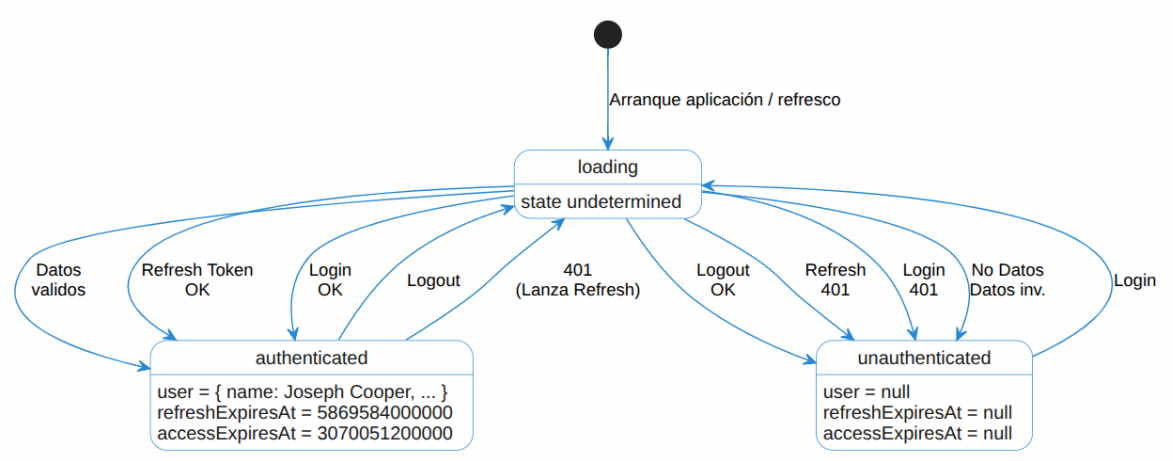


Figura 6.5: Diagrama de estados del contexto de autenticación del frontend.

junto con la fecha de expiración de los tokens. Por lo tanto, en el arranque se buscan los datos en el almacenamiento. En caso de existir y ser considerados válidos, la aplicación transita al estado *authenticated*. De lo contrario, el *context* consulta la validez del token de renovación y ejecuta una operación de renovación de sesión. En cualquier otro caso, el *context* pasa al estado *unauthenticated*.

- **Inicio de sesión:** Esta operación altera el estado del contexto. Si es exitosa, actualiza los datos del usuario en *localStorage* y cambia el estado a *authenticated*. En caso contrario, se borran los datos almacenados con el fin de evitar futuros estados inconsistentes y se asigna el estado *unauthenticated*.

- **Renovación de sesión:** Para cubrir esta operación se evaluaron las estrategias proactiva y reactiva. La estrategia proactiva se basa en un *timer* que lanza una petición de renovación cuando se aproxima la expiración del token de acceso. Por su parte, la estrategia reactiva consiste en lanzar la renovación cuando el frontend recibe un `401 Unauthorized` como respuesta a una petición autenticada. Este último enfoque es el que se ha implementado finalmente. Para ello ha sido necesario configurar *retries* e *interceptors* en el cliente API. De esta forma, las peticiones de renovación de sesión se ejecutan de forma automática al detectar respuestas `401`. En la práctica, una operación de renovación actualiza el estado del mismo modo que un inicio de sesión.

- **Cierre de sesión:** Este flujo está diseñado para ser idempotente, de forma que sucesivas ejecuciones resulten en el mismo estado. La operación borra los datos almacenados y marca el estado de la sesión como *unauthenticated*.

6.4.2 Servicios y validación

El acceso a los servicios ofrecidos por el backend se realiza a través de tres clientes API con objetivos diferentes. Por un lado, el cliente denominado *público* sirve para hacer peticiones a recursos no protegidos. Por ello, no envía credenciales en las solicitudes y no implementa *retries* ni *interceptors*. Por su parte, el cliente *protegido* se encarga de dar acceso a los endpoints protegidos, incluyendo credenciales en las peticiones e implementando *retries* e *interceptors* para lanzar peticiones de renovación de sesión ante respuestas `401`. Finalmente, el *auth context* hace uso de un cliente exclusivo para acceder a los flujos de autenticación. Para esta labor incluye credenciales, pero no implementa ningún mecanismo de reintentos o interceptores.

Por otro lado, la interfaz se protege frente a respuestas inválidas o inesperadas utilizando el result pattern y la librería de validación **Zod** [45]. Además, se han identificado varios estados para el resultado de la llamada a un servicio:

- **Éxito:** El servicio ha obtenido un resultado exitoso y los datos tienen el formato esperado.
- **Error por validación de datos:** La API devuelve un código de éxito, pero la estructura de los datos no pasa la validación.
- **Error controlado:** El backend devuelve un código de error esperado y los datos de la respuesta superan la validación.
- **Error por validación de la respuesta de error:** El servicio recibe una respuesta de error, pero la validación del cuerpo de la respuesta falla.
- **Error inesperado:** El servidor devuelve un código de error inesperado.
- **Error de red:** La comunicación con la API no es posible.

La Figura 6.6 muestra un diagrama de secuencia donde interviene el *auth context* junto con un servicio. Con el objetivo de evitar la sobrecarga visual, se ha omitido el resto de participantes. Tal como se observa, este diagrama describe el flujo de una operación que falla inicialmente por un error de autenticación y solicita al contexto que ejecute una renovación de sesión. Se ilustra, también, el cambio de estado durante la operación de renovación y la ejecución del *retry* para repetir la solicitud inicial una vez que el contexto de autenticación ha renovado la sesión exitosamente.

6.4.3 Formularios dinámicos

Una de las funciones de la aplicación permite agregar información variable a cada actividad en función del deporte escogido. En este escenario, la complejidad de los formularios puede aumentar hasta el punto de volverse insostenible. Para evitar este

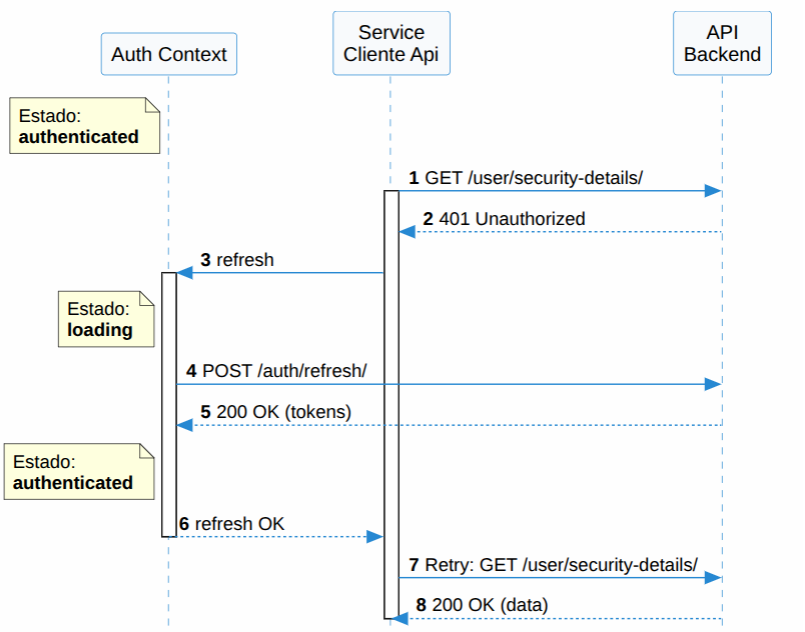


Figura 6.6: Diagrama de secuencia de una operación que requiere una renovación de sesión y un reintento.

inconveniente, se ha utilizado la librería especializada en formularios **React Hook Form** [46] junto con la librería de validación **Zod**.

La combinación de estas dos tecnologías resulta en un potente gestor de formularios. Por un lado, React Hook Form ofrece un excelente rendimiento incluso en escenarios donde los campos se añaden o quitan dinámicamente. Por su parte, Zod permite añadir esquemas de validación complejos para comprobar la validez de los datos introducidos por el usuario. Además, estas dos herramientas ofrecen utilidades para *Cross Field Validation* (validación cruzada), un proceso que permite validar reglas que relacionan más de un elemento del formulario a la vez.

A continuación, se detallan los procesos de agregar o eliminar un campo del formulario:

- **Agregar un campo nuevo:** El campo se agrega al formulario y a su esquema de validación. Además, se le asigna un valor por defecto. Tras esta acción, se ejecuta una validación para actualizar el estado global. Este nuevo campo agregado pasa a ser obligatorio.

- **Eliminación de un campo:** Se elimina el campo del formulario y de su esquema de validación. Posteriormente, se lanza una validación para actualizar su estado.

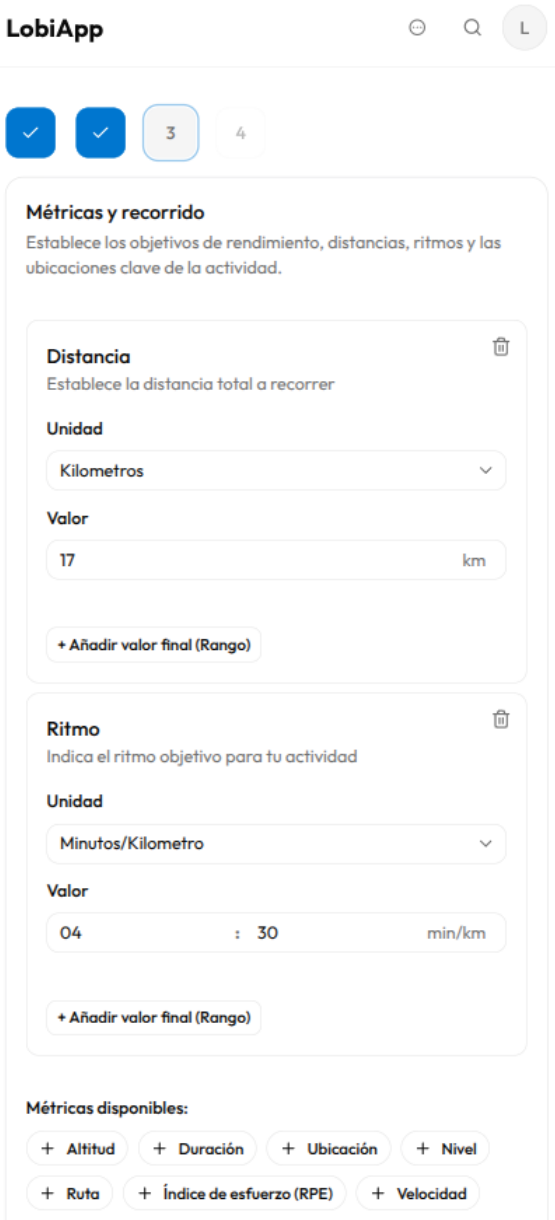
6.4.4 Interfaz de usuario

A continuación se exponen las vistas principales de la aplicación web. Esta selección de capturas proporciona una perspectiva general del diseño del frontend, evidenciando el resultado visual del proyecto.

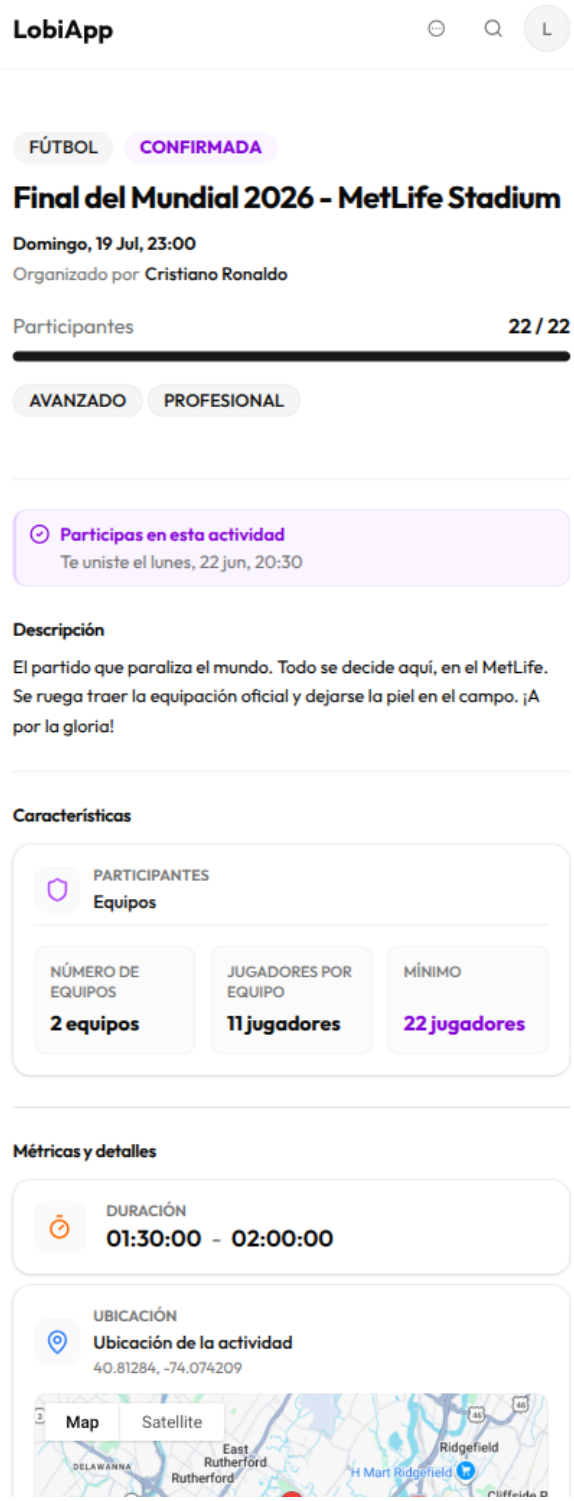
(a) Página de resultados del buscador.

(b) Perfil de usuario.

Figura 6.7: Vistas principales de la interfaz de usuario (continúa en la siguiente página).



(c) Paso de elección de capacidades en el formulario de crear una actividad.



(d) Visualización de una actividad.

Figura 6.7: Vistas principales de la interfaz de usuario (continuación).

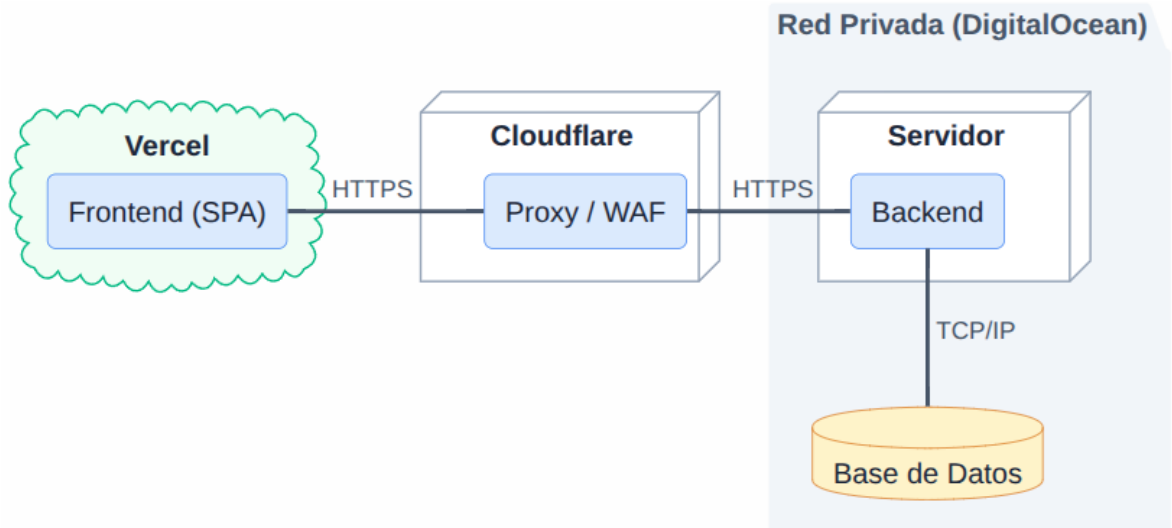


Figura 6.8: Topología de red del despliegue.

6.5 Despliegue

En la Figura 6.8 se ilustra la topología del despliegue global del proyecto. Como aspecto a destacar, cabe señalar el uso de la plataforma **DigitalOcean** [33] como proveedor de alojamiento y el aprovechamiento de sus *Virtual Private Clouds* (VPC) para desplegar el backend y la base de datos en instancias independientes bajo la misma red, reduciendo la latencia de la comunicación y aumentando la seguridad. Por su parte, el frontend se despliega en **Vercel** [23], aprovechando la alta automatización y la práctica ausencia de complejidad operativa que caracteriza a esta plataforma.

6.5.1 Contenerización del proyecto

Durante el despliegue de un proyecto es habitual enfrentarse a problemas clásicos como la compleja configuración inicial de la infraestructura y las inconsistencias provocadas por las variaciones de versiones entre distintos entornos. Para solucionar estas cuestiones, se ha empaquetado el código fuente del backend junto con sus dependencias en un contenedor de **Docker**. Además, se incluye la configuración para desplegar un proxy inverso con **Caddy**. Esta herramienta permite redirigir las conexiones hacia la aplicación y automatizar la gestión de certificados TLS, necesarios para ofrecer conexiones seguras al cliente.

Con este enfoque se logran solventar los problemas mencionados. Se consigue reducir al máximo la configuración inicial del servidor y garantizar la igualdad de condiciones de los entornos sobre los que se despliegue el sistema.

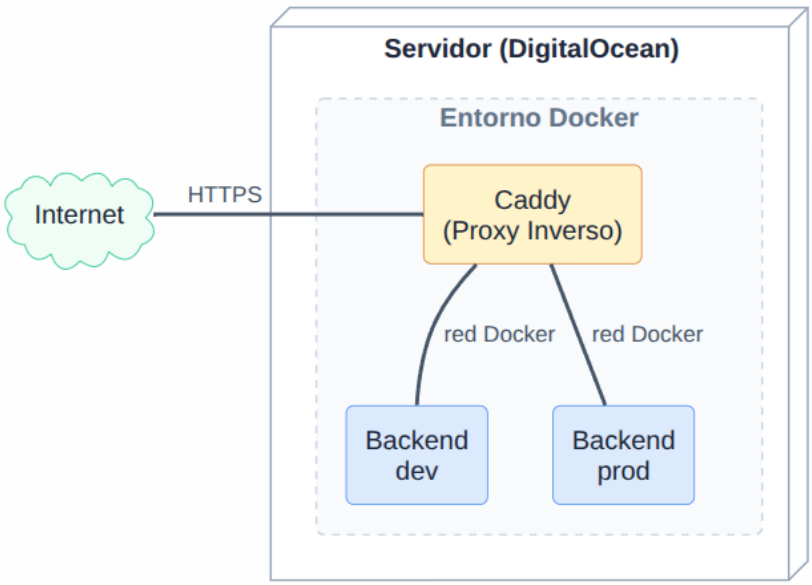


Figura 6.9: Despliegue multi-entorno en el backend con Docker.

6.5.2 Despliegue multi-entorno

En la actualidad, las aplicaciones suelen requerir múltiples entornos para soportar los estrictos procesos de validación y control de calidad. Por ello, se ha optado por desplegar el sistema en dos entornos independientes sobre el mismo servidor. Por un lado, el entorno de desarrollo dará soporte a pruebas manuales bajo condiciones controladas. Por su parte, el entorno de producción dará servicio a los usuarios finales.

En la industria es posible encontrar configuraciones más complejas y sofisticadas que suelen incluir más entornos y el uso de un servidor dedicado para cada uno de ellos. Sin embargo, se considera que la solución planteada es suficiente para cubrir las demandas de la etapa inicial del proyecto.

Aprovechando las capacidades de Docker, se ha configurado el proxy inverso Caddy para soportar el autodescubrimiento [47]. De esta forma, es posible desplegar varias instancias del proyecto en el mismo servidor. Las nuevas instancias, siempre que se use la configuración planteada, serán aceptadas por Caddy de forma automática.

En la Figura 6.9 se muestra una representación gráfica de los entornos de desarrollo y producción desplegados sobre el mismo VPS.

7 Conclusiones y vías futuras

7.1 Conclusiones

Este proyecto de fin de grado ha permitido diseñar un sistema utilizando tecnologías modernas y metodologías que componen una arquitectura limpia, orientada a la mantenibilidad y a la extensibilidad a largo plazo de la solución. La adopción de la Arquitectura Hexagonal junto con el enfoque de desarrollo DDD ha supuesto una gran dificultad inicial por la pronunciada curva de aprendizaje que requieren estos planteamientos arquitectónicos. Si bien algunos conceptos fueron fácilmente asimilables, otros no lo fueron tanto, ya que implicaron un cambio de paradigma a la hora de pensar en la estructura del código.

Por otro lado, a medida que la complejidad del proyecto aumentaba, se empezó a evidenciar la necesidad de complementar las metodologías iniciales con un patrón como CQRS. Mientras que en el módulo de autenticación dominan los flujos de escritura que utilizan el dominio para imponer las reglas de negocio, el módulo de actividades presenta características que requieren soporte de consultas optimizadas. En este contexto, CQRS surgió como una técnica muy efectiva para mejorar el rendimiento y reducir la complejidad de las operaciones de lectura, al tiempo que se sigue protegiendo la coherencia de los datos mediante las entidades del dominio.

Por su parte, el diseño de pruebas automatizadas en este trabajo ha arrojado dos conclusiones muy valiosas. En primer lugar, se comprobó de primera mano que esta práctica permite aumentar la calidad del sistema, justificando el tiempo de desarrollo requerido para conseguir una cobertura sólida. Sin embargo, también se ha podido constatar que su aplicación en un entorno de alta incertidumbre resulta contraproducente. En este caso concreto, la constante aparición de nuevos retos técnicos supuso el abandono progresivo de los tests automatizados. Las refactorizaciones necesarias para mantener las pruebas operativas tras los continuos cambios estructurales se volvieron una tarea insostenible. Esta experiencia demuestra que el testing debe aplicarse estratégicamente, evitando que se convierta en un sobrecoste temporal y en una fuente de frustración durante el desarrollo.

Adicionalmente, la construcción del frontend ha evidenciado que el rendimiento y la responsividad de la interfaz son elementos primordiales para mejorar la experiencia del usuario. Además, se ha descubierto que, en proyectos donde el backend y el frontend residen en repositorios independientes, la duplicación de código puede desencadenar un grave problema de mantenibilidad. Aunque compartir lógica básica es una práctica habitual, a medida que el proyecto escala, la sincronización de estos fragmentos puede

resultar inmanejable. Asimismo, se ha observado que el uso excesivo de dependencias externas aumenta drásticamente el *bundle size* (tamaño del paquete) del cliente, lo que afecta negativamente tanto a la experiencia de usuario como al SEO, ya que incrementa el tiempo de carga inicial.

En definitiva, los objetivos planteados han sido cumplidos satisfactoriamente. La exposición a retos técnicos y arquitectónicos del mundo real ha aportado una experiencia invaluable que consolida los conocimientos adquiridos durante el Grado. Si bien el sistema final no es perfecto, su sólida base arquitectónica representa una excelente oportunidad para seguir iterando, desarrollando nuevas características y adquiriendo nuevos conocimientos.

7.2 Limitaciones del sistema

En esta sección se enumerarán las limitaciones del sistema que se han identificado durante o tras su desarrollo. Adicionalmente, se diferenciará entre deficiencias provocadas por concesiones deliberadas y aquellas derivadas de la curva de aprendizaje de las tecnologías y metodologías utilizadas.

- **Duplicación de código:** Los *Shared Packages* (librerías compartidas) son una herramienta que busca evitar el código duplicado entre proyectos. No obstante, la implementación de esta solución suponía una sobrecarga temporal inasumible para este trabajo.
- **Acoplamiento de los bounded context *Auth* y *Users*:** Esta deficiencia del sistema radica en que el contexto de autenticación depende directamente del contexto de usuarios. La solución ideal pasa por introducir un concepto intermedio llamado *Identity* para representar al usuario en el módulo *Auth* y eliminar así la dependencia.
- **Modelos de escritura especializados:** Esta limitación deliberada se debe a que los procesos requieren de modelos de escritura con la menor cantidad de información necesaria para realizar una tarea. En su lugar, por motivos de agilidad, se ha utilizado un único modelo global para todas las operaciones.
- **Dependencia externa en el Dominio:** El complejo manejo de los números de coma flotante en JavaScript [48] forzó el uso de la librería **Big.js** [34] en el dominio. Para mitigar los efectos adversos de esta decisión arquitectónica, la dependencia se encapsuló en un value object.
- **Sistema de deportes basado en composición:** Representa una limitación potencial a futuro. A medida que aumenten las características soportadas por la aplicación, es posible que la complejidad de las factorías exija la aplicación del Principio de Inversión de Dependencias, lo que rompería el soporte nativo del compilador para el tipado estático y el autocompletado.

- **Presentación de errores de la API:** Durante la implementación del patrón de notificación, no se resolvió correctamente la identificación unívoca de cada fallo posible. Esta restricción técnica dificulta la correcta presentación de los errores al usuario final.
- **Filtración de responsabilidad en el logging:** Se ha identificado de forma errónea al componente encargado de registrar los eventos del sistema en algunos puntos específicos del código.
- ***DoS* por agotamiento de conexiones a la base de datos:** El uso del algoritmo *bcrypt* para procesar los tokens de verificación alarga drásticamente la duración de las transacciones. En escenarios de alta concurrencia, esto puede provocar una denegación de servicio por agotamiento del *pool* de conexiones.
- **TOCTOU en el flujo *CreateUser*:** Si bien es un riesgo mínimo, la estrategia óptima implicaría adoptar el enfoque *Easier to Ask Forgiveness than Permission*, el cual dicta ejecutar la operación sin comprobar previamente las colisiones y reaccionar únicamente en caso de fallo.
- ***Auth Context* multipestaña:** Concesión deliberada asumida debido a la alta complejidad técnica que supone sincronizar el estado de un contexto de React a través de múltiples pestañas del navegador simultáneamente.
- **Cobertura de tests:** Tal y como se comentó en el capítulo de implementación, la estrategia de pruebas se ha limitado al módulo de autenticación. Esta restricción degrada la garantía de calidad global del producto.

7.3 Vías futuras

El diseño del sistema desarrollado contempla muchas más características de las que finalmente han sido incluidas en el prototipo de la aplicación. Por ello, a continuación se listan las principales líneas de investigación para seguir ampliando la funcionalidad de la plataforma.

- **Subsanación de limitaciones:** Antes de incorporar nuevas funciones al sistema, se considera imprescindible abordar las deficiencias detectadas, garantizando que no deriven en problemas críticos a largo plazo.
- **Arquitectura distribuida y procesamiento en segundo plano:** Para que el sistema alcance su máximo potencial de escalabilidad, la aplicación debe evolucionar hacia una arquitectura distribuida donde los diferentes módulos se comuniquen mediante eventos y el estado de las entidades se base en la *consistencia eventual*. Esta capacidad facilitará la introducción de procesos (*workers*)

en segundo plano para tareas asíncronas, como el envío de notificaciones o la actualización automática de actividades.

- **Módulo de reportes:** Este contexto tiene como objetivo mejorar la confianza en la plataforma, permitiendo registrar reportes y aplicar sanciones a los usuarios en caso de absentismo o comportamiento inapropiado.
- **Módulo de gestión de instalaciones deportivas:** Proporcionará a los propietarios de recintos deportivos las herramientas necesarias para gestionar y promocionar sus instalaciones directamente desde la aplicación.
- **Módulo de reservas:** Íntimamente ligado al punto anterior, brindará a los usuarios la posibilidad de reservar instalaciones de forma integrada durante o después de la creación de una actividad.
- **Módulo de valoraciones y resultados:** Se perfila como el componente más ambicioso del proyecto. Sentará las bases del ecosistema social de la aplicación, permitiendo a los usuarios construir una reputación basada en sus actuaciones. Además, el registro de resultados funcionará como una bitácora personal, documentando las marcas obtenidas en cada evento.
- **Preferencias de usuario y gestión de perfil:** La personalización de preferencias es un aspecto crucial para optimizar la experiencia de usuario. En paralelo, un sistema avanzado de perfiles se erige como otro de los pilares fundamentales para potenciar el componente social del sistema.
- ***Capability* de Ruta:** Actualmente se cuenta con varias métricas que el usuario puede configurar al crear una actividad, y el objetivo es ampliar este catálogo. La capacidad de Ruta permitirá trazar el recorrido esperado durante la práctica deportiva. El principal atractivo técnico de esta *capability* será su proceso de enriquecimiento de datos en segundo plano: tras indicar una ruta en el mapa, el backend consultará una API especializada (como Google Maps) para extraer e integrar perfiles de elevación, distancias y tipos de terreno.
- **Filtros demográficos (Edad y Género):** Permitirán a los anfitriones restringir la participación para fomentar espacios deportivos más homogéneos y seguros, reduciendo así la barrera psicológica de practicar deporte con desconocidos.
- **Soporte de revocación de tokens:** Dada la naturaleza sin estado de los JWT, su revocación representa un desafío técnico. El objetivo será analizar e integrar mecanismos adicionales que solventen este punto débil del sistema.
- **Desarrollo de componentes y utilidades propias para el frontend:** Como se expuso en las conclusiones, mantener un control estricto del *bundle size* de la interfaz es vital. Para lograrlo, resulta necesario desarrollar componentes nativos y utilidades propias que sustituyan progresivamente a las dependencias externas.

Bibliografía

[1] Organización Mundial de la Salud. Directrices de la OMS sobre actividad física y hábitos sedentarios. Technical report, Organización Mundial de la Salud, 2020. URL https://iris.who.int/handle/10665/337004/. Accedido: 19-06-2026.

[2] Naciones Unidas. Objetivo ods 3: Garantizar una vida sana y promover el bienestar para todos en todas las edades, 2015. URL https://www.un.org/sustainabledevelopment/es/health/. Accedido: 19-06-2026.

[3] European Commission. Eurobarometer survey: Sport and physical activity. Technical report, Directorate-General for Education, Youth, Sport and Culture, 2022. URL https://europa.eu/eurobarometer/surveys/detail/2668/. Accedido: 19-06-2026.

[4] Herb Krasner. The Cost of Poor Software Quality in the US: A 2022 Report. Technical report, Consortium for Information & Software Quality (CISQ), 2022. URL https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2022-report/. Accedido: 19-06-2026.

[5] Martin Fowler. Design stamina hypothesis, 2007. URL https://martinfowler.com/bliki/DesignStaminaHypothesis.html. Accedido: 19-06-2026.

[6] Timpik Technologies S.L. Timpik: Plataforma para organizar partidos y encontrar jugadores, 2026. URL https://www.timpik.com/. Accedido: 10-06-2026.

[7] Playtomic S.L. Playtomic: Plataforma de reserva de pistas de pádel y tenis, 2026. URL https://playtomic.io/. Accedido: 10-06-2026.

[8] CeleBreak S.L. Celebreak: Aplicación para la organización de partidos de fútbol, 2026. URL https://celebreak.com/. Accedido: 10-06-2026.

[9] Strava Inc. Strava: Red social y aplicación de seguimiento para deportistas, 2026. URL https://www.strava.com/. Accedido: 10-06-2026.

[10] Bending Spoons US Inc. Meetup: Plataforma para encontrar y organizar eventos locales, 2026. URL https://www.meetup.com/. Accedido: 10-06-2026.

[11] Atlassian LLC. ¿Qué es la metodología Kanban?, 2026. URL https://www.atlassian.com/es/agile/kanban/. Accedido: 08-06-2026.

[12] Atlassian LLC. Trello: Plataforma de gestión de tareas y proyectos. URL `https://trello.com/es/`. Accedido: 08-06-2026.

[13] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Tesis doctoral, University of California, Irvine, 2000. URL `https://ics.uci.edu/~fielding/pubs/dissertation/top.htm`. Origen del estilo arquitectónico REST.

[14] Kamil Mysliwiec. Nestjs: Framework progresivo de node.js para aplicaciones del servidor, 2026. URL `https://nestjs.com/`. Accedido: 09-06-2026.

[15] OpenJS Foundation. Fastify: Framework web de alto rendimiento para node.js, 2026. URL `https://fastify.dev/`. Accedido: 09-06-2026.

[16] ActiveCampaign LLC. Postmark: Servicio de envío de correo electrónico transaccional, 2026. URL `https://postmarkapp.com/`. Accedido: 10-06-2026.

[17] PostgreSQL Global Development Group. Postgresql: Sistema de gestión de bases de datos relacionales de código abierto, 2026. URL `https://www.postgresql.org/`. Accedido: 09-06-2026.

[18] OSGeo Foundation. Postgis: Extensión espacial para la base de datos postgresql, 2026. URL `https://postgis.net/`. Accedido: 20-06-2026.

[19] TypeORM. Typeorm: Mapeador objeto-relacional (orm) para typescript y javascript, 2026. URL `https://typeorm.io/`. Accedido: 09-06-2026.

[20] Vercel Inc. Next.js: Framework de react para el desarrollo web, 2026. URL `https://nextjs.org/`. Accedido: 09-06-2026.

[21] Docker Inc. Docker: Plataforma de contenerización de aplicaciones, 2026. URL `https://www.docker.com/`. Accedido: 09-06-2026.

[22] Caddy Server. Caddy: Servidor web con soporte https automático, 2026. URL `https://caddyserver.com/`. Accedido: 09-06-2026.

[23] Vercel Inc. Vercel: Servicio de despliegue y alojamiento web, 2026. URL `https://vercel.com/`. Accedido: 09-06-2026.

[24] Cloudflare Inc. Cloudflare: Soluciones de rendimiento y seguridad web, 2026. URL `https://www.cloudflare.com/`. Accedido: 09-06-2026.

[25] Functional Software Inc. Sentry: Plataforma de monitorización de rendimiento y rastreo de errores, 2026. URL `https://sentry.io/`. Accedido: 09-06-2026.

[26] Git. Git: Sistema de control de versiones, 2026. URL `https://git-scm.com/`. Accedido: 09-06-2026.

[27] GitHub Inc. Github: Plataforma de alojamiento de código y desarrollo colaborativo, 2026. URL `https://github.com/`. Accedido: 09-06-2026.

[28] Postman Inc. Postman: Plataforma para el desarrollo y pruebas de apis, 2026. URL `https://www.postman.com/`. Accedido: 09-06-2026.

[29] Alistair Cockburn. The hexagonal (ports & adapters) architecture, sep 2005. URL `https://alistair.cockburn.us/hexagonal-architecture/`. Accedido: 10-06-2026.

[30] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2003. ISBN 978-0321125217.

[31] Martin Fowler. Cqrs, 2011. URL `https://martinfowler.com/bliki/CQRS.html`. Accedido: 10-06-2026.

[32] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum.* Addison-Wesley Professional, 2009. ISBN 9780321579362.

[33] DigitalOcean LLC. Digitalocean: Proveedor de servicios de computación en la nube, 2026. URL `https://www.digitalocean.com/`. Accedido: 18-06-2026.

[34] MikeMcl. Big.js: Librería de javascript para el manejo de números de coma flotante, 2026. URL `https://mikemcl.github.io/big.js/`. Accedido: 20-06-2026.

[35] OpenJS Foundation. Express: Framework de aplicaciones web para node.js, 2026. URL `https://expressjs.com/en/`. Accedido: 20-06-2026.

[36] Microsoft Corporation. Typescript: Superset de javascript con tipado estático, 2026. URL `https://www.typescriptlang.org/`. Accedido: 09-06-2026.

[37] Oracle Corporation. Mysql: Sistema de gestión de bases de datos relacionales, 2026. URL `https://www.mysql.com/`. Accedido: 20-06-2026.

[38] MongoDB Inc. Mongodb: Base de datos nosql orientada a objetos, 2026. URL `https://www.mongodb.com/`. Accedido: 20-06-2026.

[39] Nuxt Contributors. Nuxt: Framework de aplicaciones web basado en vue.js, 2026. URL `https://nuxt.com/`. Accedido: 20-06-2026.

[40] Meta Open Source. React: Librería de javascript para la construcción de interfaces de usuario, 2026. URL `https://react.dev/`. Accedido: 09-06-2026.

[41] shadcn. shadcn/ui: Colección de componentes de interfaz reutilizables, 2026. URL `https://ui.shadcn.com/`. Accedido: 09-06-2026.

[42] Tailwind Labs Inc. Tailwind CSS: Framework de CSS, 2026. URL https://tailwindcss.com/. Accedido: 09-06-2026.

[43] Aral Roca. next-translate: Plugin de internacionalización para aplicaciones next.js, 2026. URL https://github.com/aralroca/next-translate/. Accedido: 09-06-2026.

[44] Google LLC. Google maps platform: Servicios de mapas, rutas y análisis geoespacial, 2026. URL https://mapsplatform.google.com/. Accedido: 20-06-2026.

[45] Colin McDonnell. Zod: Librería de validación de esquemas con inferencia de tipos estáticos, 2026. URL https://zod.dev/. Accedido: 18-06-2026.

[46] React Hook Form Contributors. React hook form: Librería para la gestión de formularios en react, 2026. URL https://react-hook-form.com/. Accedido: 18-06-2026.

[47] Lucas Lorentz. caddy-docker-proxy: Caddy como proxy inverso para docker, 2026. URL https://github.com/lucaslorentz/caddy-docker-proxy/. Accedido: 22-06-2026.

[48] ECMA International. Ecmascript® language specification: Numbers and dates, 2026. URL https://tc39.es/ecma262/#sec-numbers-and-dates/. Accedido: 17-06-2026.

[49] Ham Vocke. The practical test pyramid, feb 2018. URL https://martinfowler.com/articles/practical-test-pyramid.html. Accedido: 08-06-2026.

Glosario

backend	Parte de una aplicación web que gestiona la lógica de negocio, el acceso a datos y la comunicación con otros sistemas.
bounded context	Límite lógico donde un modelo de dominio es válido y coherente.
condición de carrera	Anomalía de los sistemas concurrentes, derivado de una mala sincronización de procesos. Provoca que el resultado de una operación dependa del orden temporal, generando indeterminismo..
consulta geoespacial	Operaciones para almacenar y consultar datos de ubicaciones o geometrías en una base de datos.
dominio	En , representa la actividad de negocio que el software busca resolver o automatizar.
endpoint	Punto de acceso mediante el cual un cliente puede interactuar con los recursos o servicios que expone una API.
escalabilidad	Capacidad de un sistema para manejar cargas de trabajo crecientes manteniendo su rendimiento, adaptando sus recuros de forma eficiente.
escalable	Sinónimo de escalabilidad. *Ver también:* escalabilidad.
extensibilidad	Capacidad de un sistema de software para admitir nuevas funcionalidades o cambios.
extensible	Sinónimo de extensibilidad. *Ver también:* extensibilidad.
fail fast	Principio de diseño que dicta que una operación debe interrumpirse inmediatamente al detectar un error o estado inválido.
filtro de excepciones	En NestJS, mecanismo que centraliza la captura y gestión de errores.
framework	Conjunto de herramientas y componentes reutilizables que sirven de base para facilitar la construcción de aplicaciones web.

frontend	Parte de una aplicación web que gestiona la interfaz de usuario y la interacción con el sistema.
guard	En NestJS, mecanismo que determina si una petición puede ser procesada por un manejador de ruta de API.
hashing	Operación matemática irreversible y determinista que transforma una entrada en una ristra de caracteres de salida de tamaño fijo..
integridad referencial	Propiedad de las bases de datos relacionales que asegura la integridad de las relaciones entre las tablas. Es decir, evita que un registro de una tabla apunte a otro registro inexistente.
internacionalización	Proceso de diseñar una aplicación para que soporte múltiples idiomas y contextos culturales.
inyección de dependencias	Patrón de diseño en el que un objeto recibe sus dependencias desde el exterior, normalmente a través de un contenedor.
lock	Mecanismo de concurrencia que evita el acceso simultáneo de varios procesos a un mismo recurso.
logging	Mecanismo mediante el cual se registran eventos de diferente índole en respuesta a condiciones programadas, con fines de observabilidad.
lógica de negocio	Conjunto de reglas, algoritmos y procesos que definen como una aplicación funciona internamente.
manejador de ruta de API	Bloque de código que se ejecuta al recibir una petición de un cliente y que devuelve el recurso solicitado.
mantenibilidad	Capacidad de un sistema para ser modificado con el fin de actualizar software (por ejemplo, dependencias), corregir errores o mejorar el rendimiento.
mantenible	Sinónimo de mantenibilidad. *Ver también*: mantenibilidad.
migración	Proceso que permite aplicar y gestionar cambios en la estructura de una base de datos (tablas, columnas, relaciones) para mantener la coherencia entre el los distintos entornos.
notification pattern	Patrón de diseño utilizado para acumular múltiples errores durante un proceso, generalmente de validación de datos.

observabilidad Capacidad de un sistema para exponer información que permita conocer su estado interno, normalmente a través de métricas, registros y trazas.

optimistic locking Técnica de control de concurrencia basado en un campo de versión, sin bloqueo explícito.

pessimistic locking Técnica de control de concurrencia que bloquea un recurso durante el transcurso de una transacción.

pipe En NestJS, mecanismo que transforma y valida los datos de una petición antes de que lleguen al manejador de ruta de API.

rama En un sistema de control de versiones, línea de desarrollo independiente para trabajar en nuevas funciones o correcciones sin afectar al código principal.

renderización Proceso que transforma el código fuente de una aplicación (HTML, CSS, JavaScript) en la interfaz visual que ve el usuario en el navegador.

responsividad Capacidad de una interfaz para adaptarse automáticamente a diferentes tamaños de pantalla y dispositivos.

responsivo Sinónimo de responsividad. *Ver también*: responsividad.

result pattern Técnica que permite acumular errores durante una operación, generalmente de validación de datos, evitando detener el proceso ante el primer fallo detectado.

software Conjunto de programas y reglas informáticas para ejecutar tareas en un sistema.

stack tecnológico Conjunto de herramientas de software, como lenguajes de programación, , bases de datos, entre otros, utilizados para desarrollar un proyecto.

test Proceso controlado para verificar el comportamiento esperado de un sistema o de partes aisladas del mismo.

testabilidad Capacidad de un sistema para permitir la creación y ejecución de pruebas sobre él.

testable Sinónimo de testabilidad. *Ver también*: testabilidad.

testing Proceso de diseñar y ejecutar pruebas para comprobar el correcto funcionamiento de un sistema.

tipado estático Característica de un lenguaje de programación donde la comprobación del tipo de una variable se realiza durante la fase de compilación.

token En seguridad, cadena de caracteres habitualmente opaca, usada en procesos de autenticación para verificar la identidad de un usuario o dar acceso a un recurso.

unit of work Patrón que garantiza la atomicidad de las transacciones, haciendo un seguimiento de los cambios realizados sobre diversos objetos durante una operación de negocio.

User-Agent Cadena de texto enviada en las cabeceras HTTP que identifica a un cliente web.

value object Objeto inmutable que no posee identidad propia. Se define y se compara exclusivamente por el valor de sus atributos.

wizard Interfaz gráfica organizada en pasos secuenciales que guía al usuario en la realización de una operación compleja.

Lista de Acrónimos y Abreviaturas

AAA Siglas de *Arrange, Act, Assert*. Metodología del desarrollo de tests para estructurar pruebas unitarias.

API Siglas de *Application Programming Interface*. Conjunto de reglas y definiciones que permiten la comunicación entre sistemas de software, ofreciendo funciones y recursos a otras aplicaciones.

CDN Siglas de *Content Delivery Network*. Red de servidores distribuidos geográficamente cerca del usuario final para entregar contenido de manera más rápida y eficiente.

CQRS Siglas de *Command Query Responsibility Segregation*. Patrón arquitectónico que propone separar las operaciones de lectura de las de escritura.

CSS Siglas de *Cascading Style Sheets*. Lenguaje utilizado para dar formato y estilo al código HTML de una página web.

DDD Siglas de *Domain-Drive Design*. Enfoque de desarrollo de software que define el dominio como el núcleo del negocio.

DTO Siglas de *Data Transfer Object*. Estructura lógica usada para transferir información entre capas o sistemas.

E2E Siglas de *End-to-End*. Tipo de prueba automática que valida un flujo completo de una aplicación.

ER Siglas de *Entidad-Relación*. Herramienta de modelado que permite representar la estructura lógica de una base de datos mediante entidades y sus relaciones.

GIN Siglas de *Generalized Inverted Index*. Índice que optimiza la búsqueda en columnas de datos compuestos (p. ej: `JSONB`).

HTML Siglas de *HyperText Markup Language*. Lenguaje de marcas utilizado para estructurar un documento, comunmente, el que define una página web.

HTTP Siglas de *Hypertext Transfer Protocol*. Protocolo de comunicación para transferir información a través de la web.

ISR Siglas de *Incremental Static Regeneration*. Técnica híbrida que regenera páginas estáticas en segundo plano tras su primera generación, combinando ventajas de SSR y SSG.

JSON Siglas de *JavaScript Object Notation*. Formato de texto para el almacenamiento y el intercambio de datos estructurados.

JWT Siglas de *JSON Web Token*. Estándar abierto cuyo propósito es la transmisión segura de datos a través de objetos JSON.

MVP Siglas de *Minimum Viable Product*. Versión funcional y lista para lanzar al mercado de un sistema software, con un conjunto limitado de funciones respecto a la versión final.

NoSQL Siglas de *Not Only SQL*. Enfoque de bases de datos no relacionales que utiliza esquemas flexibles para almacenar datos.

OCP Siglas de *Open/Closed Principle*. Principio de desarrollo que dicta que un componente del sistema debe estar abierto para la extensión pero cerrado para la modificación.

ODS Siglas de *Objetivo de Desarrollo Sostenible*.

OMS Siglas de *Organización Mundial de la Salud*.

ORM Siglas de *Object-Relational Mapping*. Componente de software que facilita la interacción con una base de datos, traduciendo entre sus representaciones y las del dominio de la aplicación.

OTP Siglas de *One-Time Password*. Credencial temporal utilizada para operaciones de validación de identidad, la cual pierde su validez después de su uso.

REST Siglas de *Representational State Transfer*. Arquitectura para el diseño de APIs, centrado en la representación y manipulación de recursos.

SDG Siglas de *Sustainable Development Goals*.

SEO Siglas de *Search Engine Optimization*. Conjunto de técnicas para mejorar el posicionamiento de un sitio web en los motores de búsqueda.

SPA Siglas de *Single Page Application*. Tipo de aplicación web que carga un único documento HTML y actualiza su contenido dinámicamente..

SQL Siglas de *Structured Query Language*. Lenguaje de consulta estructurado usado en bases de datos relacionales para definir, manipular y consultar datos.

SSG Siglas de *Static Site Generation*. Técnica de renderizado que genera las páginas en tiempo de compilación, produciendo archivos HTML estáticos listos para servir.

SSR Siglas de *Server-Side Rendering*. Técnica de renderizado en la que las páginas se generan en el servidor antes de enviarse al navegador.

TLS Siglas de *Transfer Layer Security*. Protocolo criptográfico que proporciona comunicaciones cifradas en red. En un entorno web, permite comunicaciones seguras entre cliente-servidor.

TOCTOU Siglas de *Time-of-Check to Time-of-Use*. Vulnerabilidad o condición de carrera relacionada al tiempo que ocurre entre la lectura de un recurso y su uso. En este periodo de tiempo, el recurso puede ser modificado.

VPC Siglas de *Virtual Private Cloud*. Concepto relativo a la nube. Consisten en redes privadas dentro aisladas desplegadas dentro a una misma nube pública.

VPS Siglas de *Virtual Private Server*. Máquina virtual independiente que funciona dentro de un servidor compartido.

WHO Siglas de *World Health Organization*.