Informe INFO188-17

Desarrollo de Roguelike Deck-Building Game de estrategia Distribuido

Integrantes:

- Carolina Obreque Higueras
- Benjamín Parra Barbet
- Cristóbal Rebolledo Oyanedel
- Pascal Salinas Schmidt
- Renton Tapia Yefi

Profesor: Luis Veas Castillo
Curso: INFO288 - Sistemas Distribuidos
Fecha de entrega: Lunes 31 de abril del 2025

ÍNDICE

I. Introducción

Un Roguelike Deck-Building Game es un tipo de juego de cartas en el que los jugadores construyen un mazo durante el transcurso de una partida donde las decisiones estratégicas juegan un papel clave. A lo largo de cada ronda, los jugadores deben adaptarse a situaciones generadas aleatoriamente, lo que le da un alto nivel de rejugabilidad y variabilidad al juego. Con el auge de los juegos en línea, los juegos de este tipo han evolucionado hacia plataformas digitales, permitiendo que los jugadores compitan de manera remota y en tiempo real. Un problema que se podría encontrar en la creación de estos juegos es como permitir que sea eficiente, con baja latencia y alta disponibilidad, y además, permir que sea escalable.

Para abordar estos problemas, se propone implementar el juego sobre una arquitectura de sistema distribuido. Un sistema distribuido es un conjunto de nodos interconectados que trabajan de manera coordinada para ofrecer un servicio como si se tratara de una sola entidad. En este contexto, la arquitectura distribuida permitirá gestionar múltiples aspectos del juego, como la asignación de partidas, la persistencia de datos y la sincronización en tiempo real, a través de diversos servidores.

A lo largo de este informe, se presentará la solución propuesta, detallando su arquitectura y componentes clave. En la siguiente sección, se describe la oportunidad de innovación que representa este proyecto en el contexto de los juegos en línea y con esto, se aborda la solución propuesta, explicando su funcionamiento general.

Posteriormente, se incluirán diagramas que representen la arquitectura del sistema y sus componentes principales, junto con una descripción de los elementos de software utilizados. Se presentará el modelo físico de infraestructura que se usará, el modelo fundamental que muestre los requisitos no funcionales y el modelo de base de datos requerido para la gestión de los datos del juego.

Finalmente, se presentará una lista de preguntas que ayudaran a especificar mejor los requisitos del juego . También se revisarán las herramientas y bibliotecas necesarias para el desarrollo del sistema justificando la elección de cada una.

II. Oportunidad de innovación

Los juegos de cartas online ofrecen un espacio único para la creatividad y la interacción social, presentando una gran oportunidad de destacar entre la competencia. Si se introduce una mecánica de juego innovadora y una experiencia de usuario envolvente, este tipo de juego puede capturar la atención de nuevos usuarios, convirtiéndose en una opción atractiva para momentos de ocio. Dentro de este contexto, el potencial de este juego radica en su enfoque de Roguelike Deck-Building. Esta propuesta se diferencia de otros juegos en el mercado al combinar elementos de estrategia, acción y personalización dentro de un formato de juego competitivo y accesible. Esto crea una experiencia de juego diferente que puede enganchar a jugadores tanto casuales como hardcore, con un enfoque en la jugabilidad a largo plazo.

III. Solución propuesta

Para llevar a cabo esta propuesta, se plantea el diseño y desarrollo de un Roguelike Deck-Building Game de estrategia en el cual los jugadores podrán construir y mejorar su mazo para competir en partidas en línea, además deben tomar decisiones estratégicas sobre cómo y cuándo jugar sus cartas.. El sistema utilizará una arquitectura distribuida para gestionar la carga de manera eficiente, reduciendo la latencia y mejorando la tolerancia a fallos. Esto permitirá que el juego sea escalable a largo plazo y mantenga una alta disponibilidad para los jugadores.

III-1. Explicación del juego: Cada jugador comienza con un mazo de 7 cartas personalizadas. Este mazo incluye una carta de personaje y 6 cartas adicionales que pueden ser aliados o hechizos. Los personajes y aliados cuentan con atributos de ataque y vida, además de efectos especiales, mientras que los hechizos solo poseen efectos. Se pueden ver algunos ejemplos de efectos en el **Cuadro I**. El juego se desarrolla en un tablero compuesto por hexágonos, donde cada uno representa una casilla en la que los personajes pueden posicionarse. Los personajes y aliados se colocan en estos hexágonos y pueden atacar a otros personajes ubicados en las casillas adyacentes, conectadas por los de cada hexágono. El objetivo del juego es derrotar al personaje del oponente, para poder obtener mejoras para sus cartas y obtener nuevas cartas en un mapa generado aleatoriamente. Cada carta puede ejecutar acciones, y los jugadores disponen de 3 acciones por turno para jugar carta, lo que añade profundidad táctica al juego. Las acciones que los jugadores pueden realizar en una partida y sus respectivos costos se presentan en el **Cuadro II**.

Efecto	**Descripción**
± HP	Añade o quita vida a una carta.
± ATK	Añade o quita ataque a una carta
Alcance	Determina la distancia en casillas a la que una carta puede atacar a otra.
Armadura	Absorbe daño hasta ser destruida.
Vanguardia	Obliga a los enemigos a atacar esta carta antes que a otras que protege.
AoD	Efecto que influye en las casillas cercanas.
Velocidad	Define cuántas casillas puede moverse una carta en una sola acción.

Cuadro I: Efectos de las cartas

Acción	**Costo**	**Descripción**
Tirar una carta en el tablero	Entre 1 y 3	De un personaje es 0, de un aliado o hechizo se especifica en la carta.
Moverte en el tablero	1	Para los aliados y el personaje.
Atacar	1	Para los aliados y el personaje.

Cuadro II: Reglas de acciones en el tablero

IV. Diseños

IV-A. Diagrama de arquitectura

La **Figura1** muestra el diagrama de arquitectura de componentes de la solución, donde se representan las principales interacciones entre los módulos del sistema.

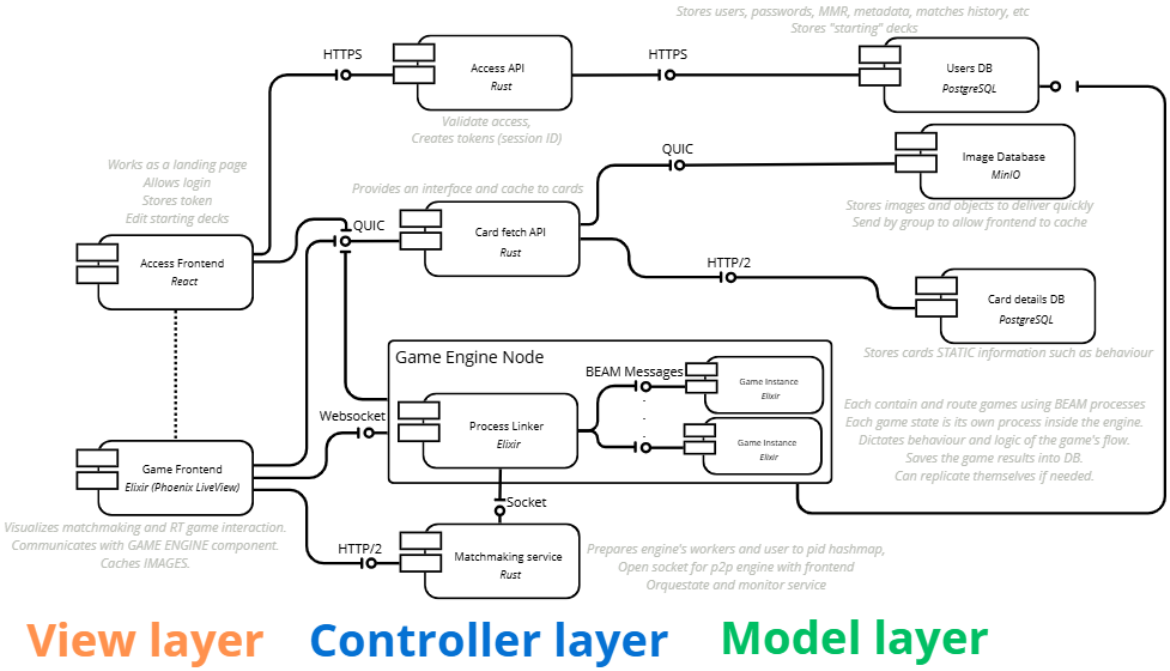

Figura 1: Diagrama de arquitectura del sistema

IV-B. Componentes de software

A continuación, se presenta la lista de los componentes de software utilizados en la solución, junto con una breve descripción de su función dentro del sistema:

- **Acces Frontend:** *React frontend Versión 18.2.0*
  Página web principal del juego que funciona como *landing page* e interfaz de acceso para los usuarios. Permite el inicio de sesión, almacena el token de autenticación y ofrece opciones para la edición de los mazos iniciales.
- **Game frontend:** *Elixir Versión 1.18.3*
  Interfaz gráfica del juego implementada con Phoenix LiveView para la visualización y manejo de interacciones en el cliente. Se encarga de visualizar el estado del matchmaking y las interacciones en tiempo real durante la partida. Además, se comunica con el componente **Game Engine** para recibir y enviar eventos del juego.
- **Access API** *Rust Versión 1.75.0*
  Valida el acceso de los usuarios y gestionar la autenticación. Genera tokens de sesión para mantener la seguridad.
- **Card fetch API** *Rust Versión 1.75.0*
  Proporciona una interfaz para acceder a la información de las cartas, gestionando su caché para optimizar la carga de datos.
- **Game Engine Node:** *Elixir Versión 1.18.3*
  Se encarga de contener y gestionar las partidas utilizando procesos BEAM, donde cada estado de juego es un proceso independiente dentro del motor. Define la lógica y el flujo del juego, asegurando el correcto comportamiento de las mecánicas. Al finalizar una partida, almacena los resultados en la base de datos. Además, los nodos deben ser replicables por el servicio de emparejamiento o Kubernetes para mejorar la escalabilidad y la tolerancia a fallos. Este cuenta con dos subcomponentes definidos a continuación
  1. **Process Linker:** Establece la comunicación entre un juego y un jugador, creando un proceso y enviando mensajes a través de PID. También proporciona información a los servicios.
  2. **Game Instance:** Cada estado de juego se almacena en un proceso de instancia del juego. Dicta el comportamiento, la lógica y la validación de las acciones dentro del juego.
- **Matchmaking Service:** *Rust Versión 1.75.0*

Abre un socket para la comunicación P2P entre el motor de juego y el frontend. Además, orquesta y monitorea el servicio para asegurar una distribución eficiente de las partidas.

- **Image Database:** *Minio Versión 2025-03-12a*
  Servicio de almacenamiento y distribución de imágenes y objetos del juego, optimizado para entregar recursos gráficos rápidamente al frontend.
- **UsersDB:** *PostgreSQL Versión 17.4*
  Base de datos dedicada al almacenamiento de información de usuarios, incluyendo nombre de usuario, contraseñas, MMR (Matchmaking Rating), metadatos y demás detalles relevantes. También almacena los mazos iniciales de los jugadores.
- **Cards details DB:** *PostgreSQL Versión 17.4*
  Base de datos relacional encargada de almacenar información sobre las cartas, incluyendo ataque, vida, habilidades y las mecánicas necesarias.

IV-C. Modelo físico

Tanto el Access Frontend como el Game Frontend se ejecutarán en la máquina del usuario, los requisitos mínimos serían:

- **CPU**: Intel(R) Core I3-10110U
- **GPU**: Intel(R) UHD Graphics (Grafica integrada)
- **RAM**: 4GB
- **Disk**: 10GB Free space (para cachear sprites e imágenes)

Los Game Engine Node correrán en un servidor NVMe Bit del proveedor Webdock que se puede observar en la **Figura 2** que cuenta con un plan mensual de 8€lo que equivale a aproximadamente CLP $8151.

El resto de componentes serán ejecutados en un servidor NVMe Nano4 del proveedor Webdock que se puede observar en la **Figura 3** que cuenta con un plan mensual de 1.08€lo que equivale a aproximadamente CLP $1010 .

La distribución de los componentes sería:

- Una maquina para la Access API y Card fetch API
- Otra maquina para Users DB, Image Database y Card details DB
- Una tercera maquina para Matchmaking service

IV-D. Modelo fundamental

El modelo fundamental describe los requisitos no funcionales de la arquitectura del sistema distribuido para el juego.

- **Tolerancia a fallos:** La arquitectura del juego implementará mecanismos de recuperación ante fallos utilizando **supervisores en Elixir** y **reinicios automáticos en Rust (tokio)** para garantizar la continuidad del servicio. Además, se empleará `Redis` para la sincronización de sesiones y almacenamiento de estados temporales en caso de fallas en el servidor principal.
- **Disponibilidad:** Se aplicará una estrategia de alta disponibilidad con balanceo de carga y múltiples instancias de servidores utilizando **Kubernetes** o **Phoenix PubSub** permitirá una distribución eficiente de eventos en tiempo real, minimizando interrupciones en la comunicación entre jugadores.
- **Escalabilidad:** La infraestructura soportará escalamiento horizontal mediante la adición de servidores y la partición de datos en `PostgreSQL`. Para el manejo de eventos en tiempo real tenemos `Phoenix` que se integrarán con Redis para distribuir la carga entre múltiples nodos.

IV-E. Modelo de base de datos

A continuación se presentan los modelos relacionales de las bases de datos que estructuran la información necesaria para el funcionamiento del juego. La **Figura 4** muestra el modelo de la base de datos que almacena los detalles de las cartas y La **Figura 5** presenta el modelo de la base de datos de usuarios, que contiene la información de cada jugador.

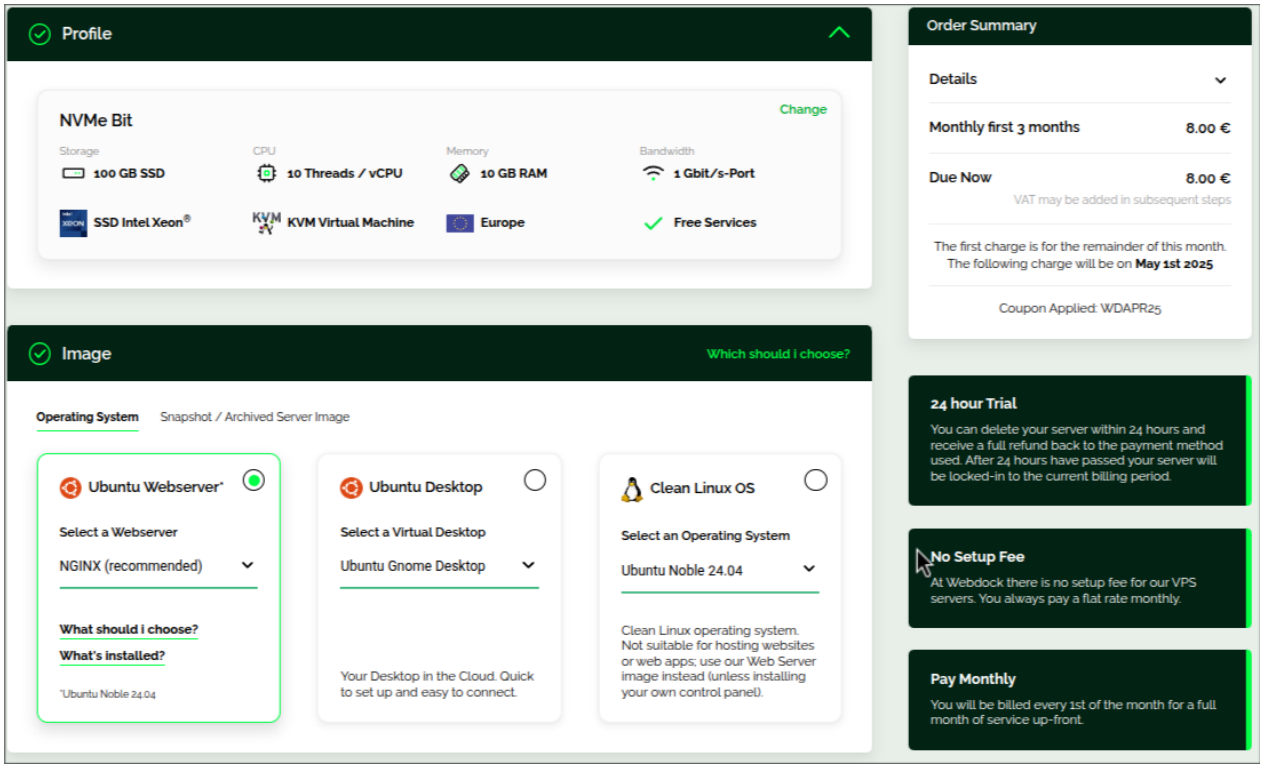

Figura 2: Servidor de juego

IV-F. Preguntas para cliente

A continuación, se presenta una serie de preguntas que tienen por objetivo definir de mejor manera los requisitos del sistema y asegurar que cumpla las expectativas del cliente.

1. ¿Cuál es el público objetivo del juego?
   resp: El público objetivo son jugadores de entre 15 y 40 años que disfrutan de juegos de estrategia, construcción de mazos y roguelikes.
2. ¿Cuántos jugadores se espera que puedan jugar simultáneamente?
   resp: Se espera que el juego soporte al menos 1000 jugadores simultáneos en una partida en línea, con un sistema de matchmaking que permita jugar contra otros usuarios de manera eficiente.
3. ¿Se requieren medidas avanzadas de seguridad?
   resp: El juego requerirá medidas de seguridad para evitar el cheating
4. ¿El juego tendrá alguna tienda para comprar personajes, cartas, skins, etc.?
   resp: Una vez completado el juego base se planea agregar una tienda de cosméticos
5. Si se cuenta con una tienda, ¿se requerirá un sistema de pago integrado?
   resp: Si se requeriría dado que necesitamos asegurar que los pagos se realicen de manera segura y libre de fallos pero eso se implementará después de que este implementado el juego base
6. ¿Cuáles son los requisitos mínimos de hardware y software para los jugadores?
   resp: Los requisitos mínimos seria un procesador Intel(R) Core I3-10110U y al menos 4GB de RAM
7. ¿Se requiere soporte para múltiples idiomas?
   resp: En etapas iniciales no se necesitara este soporte, sin embargo en futuras actualizaciones se podría considerar
8. ¿Habrá integración con redes sociales u otros servicios externos?
   resp: Nuestro juego busca ser simple y rápido, por lo que se evitaran integraciones innecesarias con otros servicios
9. ¿Se planea lanzar el juego en plataformas móviles o solo en PC?

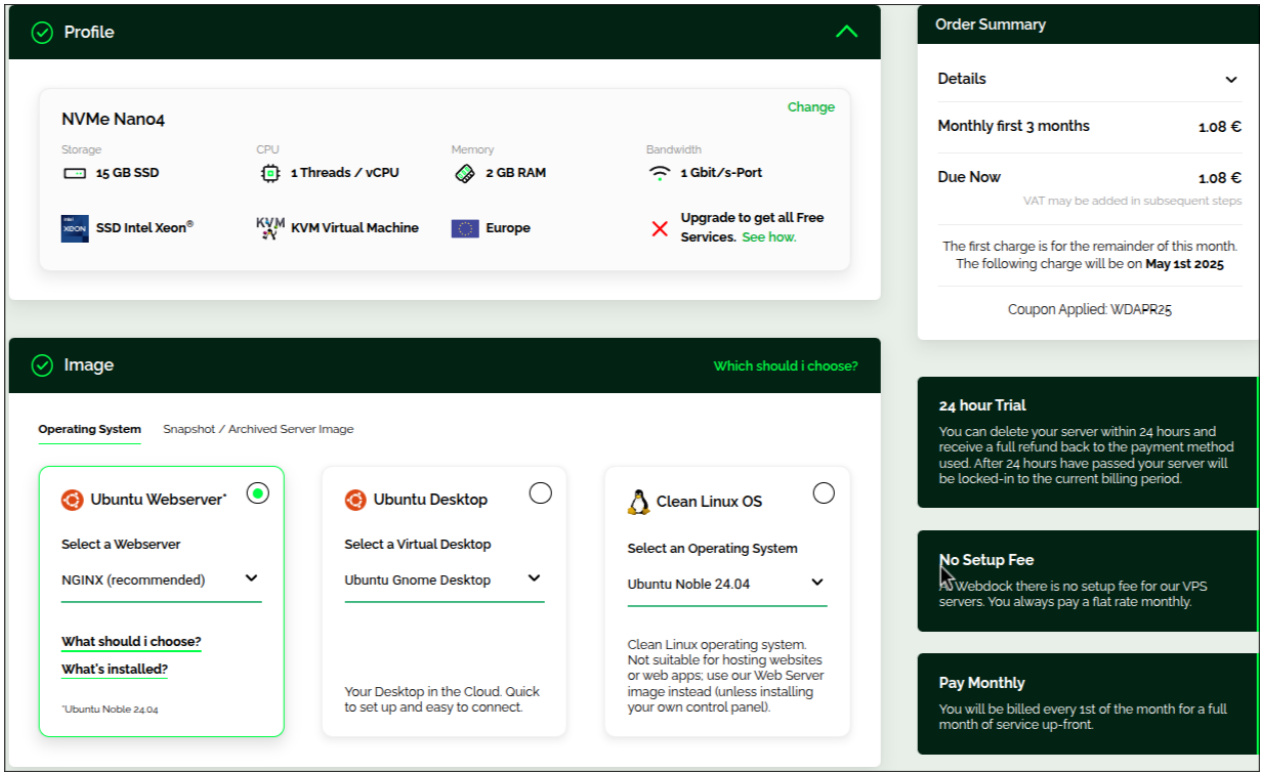

Figura 3: Servidor en la nube

resp: Si bien el juego sera lanzado para la web, pero solo con soporte para PC

10. ¿Cuántos servidores se destinarán para la infraestructura y cuáles son sus características?
    resp: Los servidores usados serán...
11. ¿Cuál es el tiempo de respuesta máximo aceptable para las interacciones en el juego?
    resp: Las interacciones deberían tener un tiempo de respuesta inferior a un segundo para garantizar una buena experiencia de usuario
12. ¿Qué mecanismos de tolerancia a fallos deben implementarse para garantizar la disponibilidad del juego en caso de problemas?
    resp: Si se cae una instancia de juego no debería afectar al resto del sistema, y volver a levantarse con un sistema para recuperar los datos de sesión

IV-G. Herramientas

Para el desarrollo del juego de cartas en línea, se utilizarán herramientas mayormente de software libre debido a su flexibilidad, escalabilidad y comunidad de soporte. La combinación de estas tecnologías permite una implementación eficiente sin costos adicionales de licencias.

- **Software Libre:** Se emplearán tecnologías de código abierto como `Elixir`, `Rust`, `React` y `PostgreSQL`. Estas herramientas han sido elegidas por su rendimiento, escalabilidad en entornos de producción ademas de curva de aprendizaje de las mismas tecnologías en comparación con otras, es ese equilibrio entre rendimiento-usabilidad lo que también intentamos buscar al elegir estas tecnologías.
- **Herramientas desarrolladas:** Se construirán sistemas personalizados para la lógica del juego y el emparejamiento de jugadores en `Elixir` y `Rust`. Esto permitirá una integración óptima con el backend y la gestión eficiente del estado de los jugadores a la hora de tener varias conexiones.
- **Servicios de terceros:** Se utilizarán soluciones como `MinIO` para almacenamiento de imágenes y `Grafana` con `Prometheus` para la monitorización del sistema. Estos servicios facilitan la observabilidad y el mantenimiento del juego.

relacional Card Details BD.png relacional Card Details BD.png relacional Card Details BD.bb relacional Card Details BD.bb

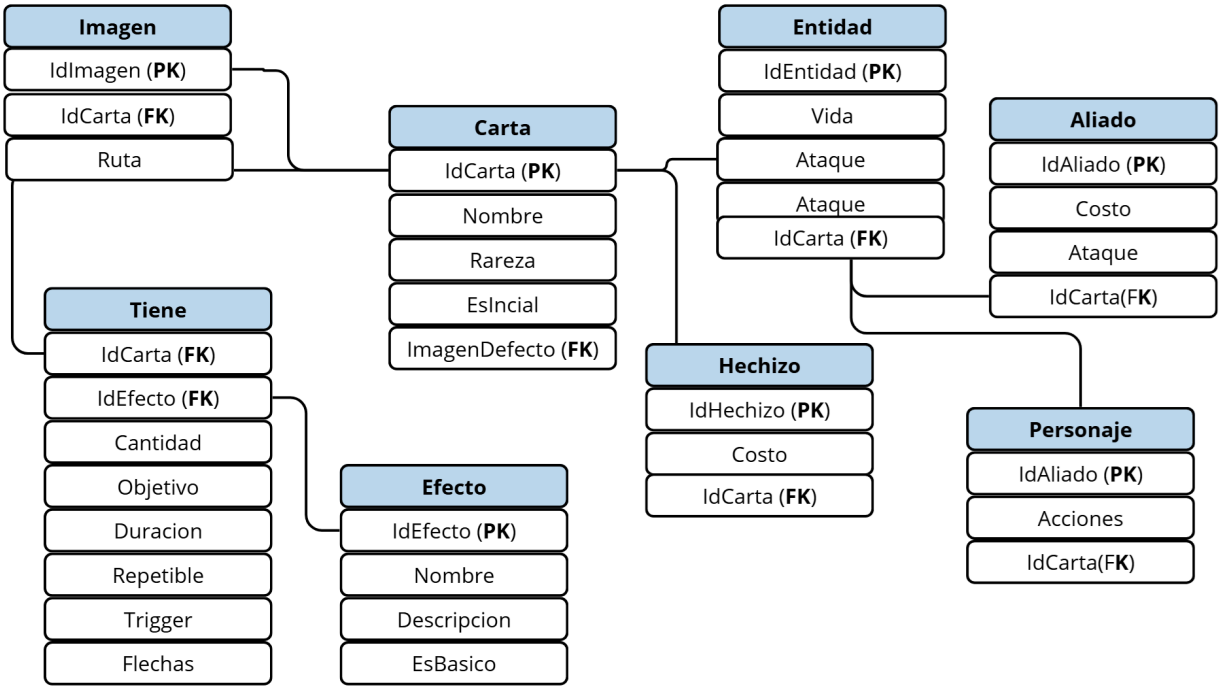

Figura 4: Card Details DB

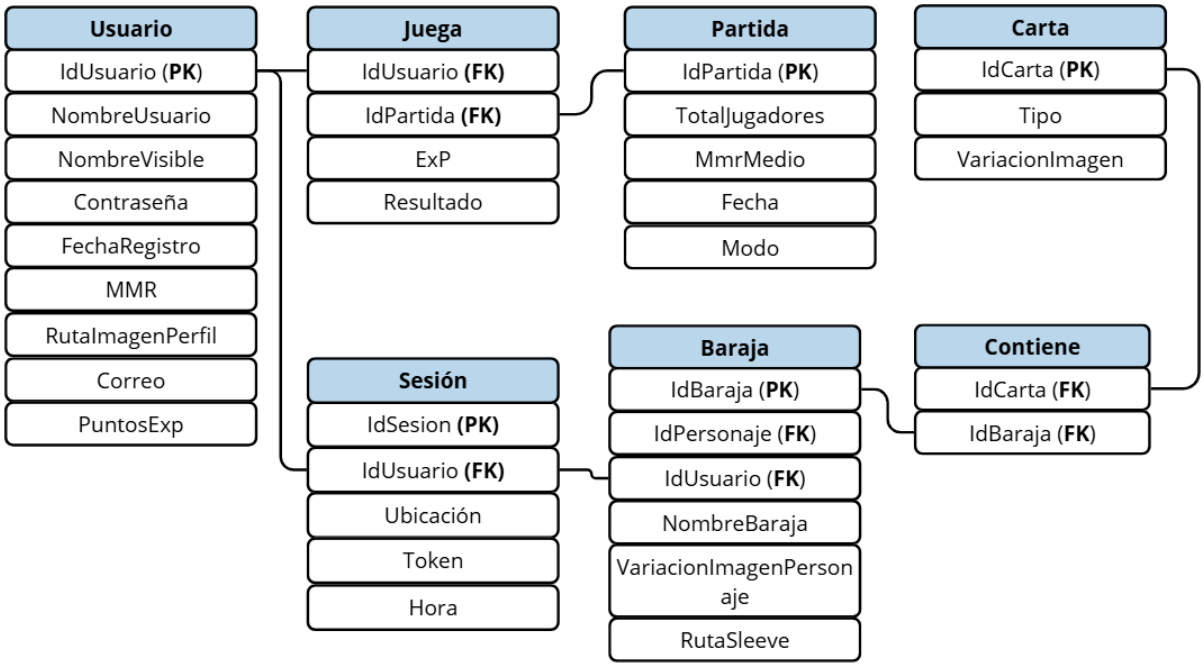

Figura 5: Users DB

El uso de estas herramientas garantiza una solución robusta, adaptable y con costos controlados, evitando la

dependencia de tecnologías propietarias.

IV-H. Resumen Bibliotecas necesarias

Para la implementación del juego de cartas en línea, se utilizarán diversas bibliotecas y frameworks según la tecnología elegida para cada componente del sistema:

- **Game Engine:** Se empleará `Elixir` para el renderizado y la lógica del juego. Estas bibliotecas permiten un rendimiento óptimo en gráficos y física ademas de ser una gran ayuda para interactividad en tiempo real con los demás jugadores.
- **Backend y comunicación en tiempo real:** Se usará `Phoenix` (Elixir) para manejar la lógica del juego en tiempo real y WebSockets. Para el servicio de emparejamiento. `axum` (Rust) proporcionarán una infraestructura escalable y eficiente apoyando lo dicho antes con Elixir.
- **Frontend:** La interfaz del juego será implementada en Elixir con `Phoenix LiveView` para la representación de gráficos en 2D, entre otras librerias para un estilo visual simple.
- **Base de datos:** `PostgreSQL` será utilizado para almacenar la información de los usuarios y cartas, mientras que MinIO servirá como sistema de almacenamiento de imágenes.
- **Monitorización y despliegue:** Se integrará `Grafana` con `Prometheus` para la observabilidad del sistema y `Loki` para la centralización de logs el cual a su vez tiene una alta con Grafana.

Estas bibliotecas han sido seleccionadas por su rendimiento, escalabilidad y compatibilidad con la arquitectura propuesta para el juego.

IV-I. Librerías necesarias por tecnología

IV-I1. Game Engine y Game Frontend: Elixir:

- `Sdtlib` - Para estructuras y funciones proporcionadas por el lenguaje
- `Phoenix LiveView` o `HTMeX` - Para interfaz visual de juegos basados en la web con interactividad en tiempo real
- `Bandit` o `Cowboy` - servidor http
- `WebSock` libreria para WebSocket
- `websock_adapter` adaptador para WebSock
- `QUICER` Quic conections

IV-I2. Matchmaking Service: Rust:

- `tokio` - Manejo de asincronía y tareas concurrentes.
- `tower` - clientes y sevidores de red
- `hyper` - implementacion de http
- `axum` o `warp` - Framework para construir APIs REST o WebSockets.
- `serde` - Serialización y deserialización de datos (JSON, BSON, etc.).
- `rustls` - Soporte para TLS (seguridad en conexiones de red).
- `chrono` - Manejo de fechas y tiempos.
- `postgres` o `tokio-postgre` - PostgreSQL integration
- `quic` - libreria para Quic

IV-I3. Página Principal: React frontend:

- `react-router-dom` - Para manejar rutas en React.
- `tailwindcss` - Para diseño rápido y estilizado.
- `framer-motion` - Para animaciones suaves.
- `axios` - Para llamadas a APIs.

IV-I4. Base de Datos para Entrega de Imágenes: MinIO:

- `minio` (SDK de MinIO) - Manejo de almacenamiento de objetos en Rust o Node.js.
- `multer` (para backend en Node.js) - Para manejo de archivos en formularios.
- `aws-sdk` - Alternativa para interactuar con MinIO usando APIs tipo S3.

IV-I5. Base de Datos de Información de Cartas y Usuarios: PostgreSQL:

- `diesel` o `sqlx` (Rust) - ORM y consultas seguras en Rust.
- `pg-promise` (Node.js) - Conexión a PostgreSQL en aplicaciones JavaScript.
- `postgres` (Rust) - Cliente PostgreSQL puro para Rust.
- `typeorm` (TypeScript) - ORM para React u otros servicios en Node.js.

IV-I6. Monitorización del Sistema: Grafana:

- `prometheus-client` (Rust) - Para enviar métricas a Prometheus/Grafana.
- `loki` - Para logging centralizado compatible con Grafana.
- `grafana-plugin-sdk` - Para desarrollar plugins personalizados de visualización.