S.I.G.P.D.

Programación Full Stack

JurassiCode

Rol	Apellido	Nombre	C.I	Email
Coordinador	Fianza	Ignacio	5.690.153-1	businessignaciofianza@gmail.com
Sub-Coordinador	Benítez	Sebastián	5.652.044-4	sebastianbenitez2505@gmail.com
Integrante 1	Fleitas	Joaquín	5.570.982-3	joacolambru7@gmail.com
Integrante 2	Paz	Tomás	5.700.344-1	tomaslautaropaz@gmail.com

Docente: Laporta, Emanuel

Fecha de culminación

10/11/2025

TERCERA ENTREGA

I.S.B.O. **3MI**

ÍNDICE

REPOSITORIO DE GitHub: 4
Análisis de PHP como lenguaje backend 4
Criterios de selección del sistema gestor de base de datos (SGBD) 6
Evaluación de frameworks frontend seleccionados (Tailwind CSS y complementarios) 8
- Tailwind CSS 8
- Laravel 9
- Justificación de la elección 10
- Complementos y herramientas adicionales 10

Selección fundamentada de herramientas para control de versiones y colaboración 11
Recomendaciones de entornos de desarrollo con sus ventajas 11
Requisitos de software (entorno objetivo en producción) 14
Incluir configuración de IDE (Visual Studio Code recomendado) y extensiones útiles 16
Diagrama entidad-relación inicial que refleje fielmente la estructura del juego 18
Entidades Principales con Atributos 19
Relaciones entre Entidades 22
Realizar la transformación del DER al modelo relacional (pasaje a tablas) 24
Aplicar el proceso de normalización hasta la Tercera Forma Normal (3FN) 26
- 1FN (Primera Forma Normal): 26
- 2FN (Segunda Forma Normal): 26
- 3FN (Tercera Forma Normal): 26

Identificar y documentar claramente claves primarias y foráneas 27
Registrar todas las decisiones de diseño con su justificación técnica 31
- Decisiones de Datos (Modelo y Normalización) 31
- Arquitectura y Backend 33
- Frontend y UX 34
- Seguridad y Buenas Prácticas 34
- Extensión Prevista (Modo Digitalizado) 35
- Despliegue y Entorno 36
- Internacionalización 36

Documentar exhaustivamente las restricciones no estructurales derivadas de las reglas del juego 37
Especificar cómo estas restricciones serán implementadas en el sistema 39
Refinar el esquema relacional normalizado (3FN), garantizando eliminación de redundancias 43
Documentar detalladamente las Restricciones No Estructurales (RNE) que representan reglas de negocio específicas 46
Presentar esquema relacional definitivo en 3FN 50

Script de inserción de prueba 52
Desarrollar script DDL final con todas las estructuras, relaciones y restricciones 54
Políticas de Mantenimiento 58
Introducción 58
Rutinas de Backup y Automatización del Sistema Operativo 58
Estrategia Documentada 58
Implementación Técnica 59
Estrategia de Versionado y Mantenimiento con GitFlow 60
Flujo de trabajo 60
Política de Releases 60
Beneficios 60
Hoja Testigo 61

REPOSITORIO DE GitHub:

https://github.com/JurassiCode/PROYECTO

Análisis de PHP como lenguaje backend

El lenguaje PHP (acrónimo recursivo de PHP: Hypertext Preprocessor) es una tecnología ampliamente utilizada en el desarrollo de aplicaciones web dinámicas. Su diseño orientado al entorno web, su larga trayectoria y su ecosistema robusto lo posicionan como una alternativa vigente frente a otros lenguajes backend como Python, JavaScript, Ruby o Java.

Una de las principales ventajas de PHP es su amplia compatibilidad con servicios de hosting, especialmente en entornos compartidos, lo que reduce los costos y simplifica el despliegue de aplicaciones. A diferencia de lenguajes como Node.js o Django, que suelen requerir configuraciones específicas o entornos dedicados, PHP puede ejecutarse en servidores tradicionales sin mayores requerimientos técnicos.

Además, su sintaxis simple y curva de aprendizaje accesible lo convierten en una excelente opción para desarrolladores principiantes. Su integración directa con HTML permite desarrollar aplicaciones web dinámicas sin necesidad de arquitecturas complejas, lo cual acelera el prototipado y facilita el mantenimiento. PHP cuenta con un ecosistema maduro, en el que destacan frameworks como

Laravel y Symfony, los cuales proporcionan herramientas modernas para enrutamiento, seguridad, ORM (Object-Relational Mapping), validaciones y más. Laravel, en particular, ofrece una sintaxis limpia, documentación extensa y una comunidad activa que contribuye constantemente con recursos y paquetes.

Otra ventaja clave es que PHP fue diseñado desde sus inicios como un lenguaje orientado a la web.

Por ello, incorpora de forma nativa funcionalidades como manejo de sesiones, envío de correos electrónicos, procesamiento de formularios y conexión a bases de datos, lo que permite construir aplicaciones completas sin necesidad de librerías externas. En cuanto a la seguridad, si bien PHP fue históricamente criticado por malas prácticas en su implementación, hoy en día los frameworks modernos incorporan medidas de protección contra amenazas comunes como inyecciones SQL, ataques XSS y CSRF, entre otras.

En conclusión, PHP continúa siendo una opción sólida y vigente en el desarrollo backend, especialmente en proyectos orientados al entorno web que requieren rapidez de desarrollo, bajo costo de infraestructura y facilidad de mantenimiento.

Criterios de selección del sistema gestor de base de datos (SGBD)

La elección del sistema gestor de base de datos (SGBD) es una decisión estratégica que impacta en múltiples aspectos del desarrollo y funcionamiento de una aplicación. En este proyecto, se seleccionó el SGBD en función de criterios técnicos objetivos, considerando las necesidades específicas de la arquitectura propuesta.

En primer lugar, se prioriza la compatibilidad con el lenguaje backend. Dado que el desarrollo se realiza en PHP, se seleccionó un SGBD que ofreciera integración directa mediante extensiones como mysqli o PDO. MySQL, ampliamente adoptado en entornos PHP, cumple con este requerimiento y permite una implementación fluida de operaciones de base de datos.

También se consideró la facilidad de uso y curva de aprendizaje. MySQL dispone de interfaces gráficas como phpMyAdmin y MySQL Workbench, que simplifican la administración y manipulación de datos, incluso para usuarios con experiencia limitada en bases de datos.

En cuanto al rendimiento, MySQL ha demostrado ser eficiente en entornos web donde predominan operaciones frecuentes de lectura y escritura. Su motor InnoDB permite garantizar integridad referencial mediante claves foráneas, lo cual es esencial en aplicaciones que manejan relaciones entre tablas, como usuarios, partidas y puntuaciones.

Además, se valoró la portabilidad y escalabilidad. MySQL puede desplegarse tanto en servidores locales como en la nube, y funciona en múltiples plataformas, lo cual favorece el crecimiento del proyecto sin necesidad de reestructuraciones importantes.

El modelo de licenciamiento fue otro aspecto considerado. MySQL es un software de código abierto bajo licencia GPL, lo cual permite su uso sin costos asociados, algo crucial en proyectos académicos o con presupuesto limitado.

Finalmente, la seguridad y el soporte comunitario también fueron elementos determinantes. MySQL permite configurar permisos por usuario, restringir accesos y definir políticas de autenticación, y cuenta con una comunidad global activa, así como abundante documentación.

En función de todos estos criterios, se concluye que MySQL es el sistema gestor de base de datos más adecuado para el presente proyecto, por su compatibilidad, robustez, eficiencia y sostenibilidad.

Evaluación de frameworks frontend seleccionados (Tailwind CSS y complementarios)

Tailwind CSS

Para el desarrollo frontend del proyecto, se seleccionó Tailwind CSS como el framework principal. Esta decisión se basó en su flexibilidad, alta personalización y la eficiencia en la creación de interfaces responsivas y modernas.

Tailwind CSS ofrece un enfoque de diseño basado en clases utilitarias, lo que permite a los desarrolladores crear diseños personalizados de forma rápida y sencilla, sin tener que escribir una gran cantidad de CSS personalizado. En lugar de usar componentes predefinidos como otros frameworks, Tailwind permite la construcción de interfaces directamente desde el HTML utilizando clases específicas para cada propiedad de estilo.

Ventajas de Tailwind CSS:

1. Alta personalización: A diferencia de los frameworks tradicionales como Bootstrap, Tailwind no impone estilos predeterminados, lo que permite un control total sobre el diseño sin restricciones.

2. Sistema de diseño responsivo: Tailwind incluye herramientas para crear interfaces que se adaptan a diferentes tamaños de pantalla con facilidad, usando sus clases sm:, md:, lg:, etc., que permiten controlar el comportamiento del diseño en dispositivos móviles, tabletas y escritorios.

3. Clases utilitarias: Facilita la creación de componentes personalizados sin necesidad de escribir CSS desde cero. Las clases como bg-blue-500, text-white, flex, entre otras, permiten hacer modificaciones de estilo directamente en el HTML.

4. Menor tamaño de archivos: Gracias a la funcionalidad de purgado de Tailwind CSS, el archivo CSS final solo incluye las clases utilizadas, lo que optimiza el

tamaño del archivo y mejora el rendimiento de carga.

Además, la documentación extensa y la gran comunidad de Tailwind CSS contribuyen a una curva de aprendizaje más rápida y al acceso a recursos adicionales como plugins y componentes reutilizables.

Laravel

Laravel es el framework de backend elegido para este proyecto. Su elección se basa en la robustez, escalabilidad y facilidad de integración con herramientas de frontend como Tailwind CSS. Laravel ofrece características como:

1. Eloquent ORM: Facilita la interacción con bases de datos mediante una sintaxis intuitiva, evitando la necesidad de escribir consultas SQL complejas.

2. Sistema de rutas y middleware: Facilita la creación de rutas y la aplicación de seguridad a través de middleware, lo que mejora la protección de las aplicaciones.

3. Autenticación y autorización integradas: Laravel incluye un sistema de autenticación muy fácil de configurar, lo que permite manejar el registro, inicio de sesión y roles de usuario de manera sencilla.

4. Facilidad de integración con frontend: Laravel se integra perfectamente con frameworks frontend como Tailwind CSS para el diseño y Vue.js o React para interactividad avanzada, si se requiere en el futuro.

5. Laravel utiliza Blade, su motor de plantillas nativo, para gestionar las vistas de forma modular y reutilizable.
   Los *layouts* en Blade permiten definir estructuras base, por ejemplo, una cabecera, navegación y pie de página comunes, que pueden ser extendidas por otras vistas del sistema. Esto facilita mantener una apariencia coherente en toda la aplicación y reduce la duplicación de código.

6. Para la parte dinámica del frontend, el equipo incorporó Alpine.js, una librería ligera inspirada en Vue.js que permite agregar interactividad directa en el HTML, ideal para componentes reactivos, menús desplegables, modales y transiciones sin necesidad de frameworks más pesados. Su integración con Laravel Blade es sencilla y potencia la experiencia de usuario sin comprometer el rendimiento.

Justificación de la elección

La combinación de Laravel y Tailwind CSS es poderosa porque ofrece lo mejor de ambos mundos: una base robusta de backend con Laravel y un sistema de diseño flexible y eficiente con Tailwind CSS. Esta elección permite desarrollar aplicaciones con una arquitectura limpia y escalable mientras se mantiene una interfaz de usuario moderna y altamente personalizable.

Laravel permite centrarse en la lógica del negocio y la estructura de la aplicación sin preocuparse por detalles complejos de seguridad y bases de datos, mientras que Tailwind CSS asegura que el frontend sea atractivo y totalmente adaptado a las necesidades del proyecto, sin limitar la creatividad o diseño.

Complementos y herramientas adicionales

Hasta el momento, no se han utilizado herramientas complementarias adicionales, ya que Tailwind CSS cubre adecuadamente todas las necesidades visuales del proyecto. Sin embargo, Laravel y Tailwind pueden ampliarse fácilmente con herramientas adicionales como Alpine.js para interactividad ligera o Livewire para interacción en tiempo real sin escribir mucho JavaScript.

Selección fundamentada de herramientas para control de versiones y colaboración

Para gestionar el control de versiones y facilitar el trabajo colaborativo durante el desarrollo del proyecto, seleccionamos las herramientas Git y GitHub. A continuación, fundamentamos esta elección

Git

Git es un sistema de control de versiones distribuido, ampliamente utilizado en el desarrollo de software. Nos permite:

- Registrar todos los cambios realizados en el código de forma ordenada y cronológica.
- Trabajar en paralelo sin sobrescribir los avances de otros integrantes, mediante ramas (branches).
- Revertir fácilmente a versiones anteriores en caso de errores o conflictos.
- Trabajar sin conexión, ya que cada desarrollador tiene una copia local completa del repositorio.
- La elección de Git responde a su eficiencia, velocidad, amplia documentación y al hecho de que es una herramienta estándar en la industria.

GitHub

GitHub es una plataforma web que complementa el uso de Git, ofreciendo:

- Almacenamiento remoto del repositorio, accesible desde cualquier lugar.
- Gestión de ramas, revisiones de código (pull requests) y seguimiento de issues o tareas.
- Visualización del historial de cambios y colaboración entre los miembros del equipo en tiempo real.
- Integraciones con otras herramientas como Trello, Slack o GitHub Actions, lo cual puede facilitar la automatización de tareas.

Optamos por GitHub por ser una de las plataformas más utilizadas en el entorno profesional y educativo, lo que también facilita el aprendizaje de buenas prácticas de desarrollo colaborativo.

Recomendaciones de entornos de desarrollo con sus ventajas

Recomendaciones de entornos de desarrollo y sus ventajas

Para el desarrollo del sistema web, se recomienda utilizar los siguientes entornos y herramientas, que se integran de forma eficiente para facilitar el desarrollo, prueba y mantenimiento del proyecto:

1. Visual Studio Code (VS Code)

Ventajas:

- Gratuito, liviano y multiplataforma.
- Amplia disponibilidad de extensiones (PHP, HTML, Live Server, Git, etc).
- Autocompletado inteligente y resaltado de sintaxis.
- Integración con terminal y control de versiones (Git).
- Vista previa en vivo y depuración integrada.

2. XAMPP / LAMP (Entorno local de servidor web)

Ventajas:

- Permite simular un servidor local con Apache, PHP y MySQL sin necesidad de internet.
- Fácil instalación y configuración.
- Incluye phpMyAdmin para gestionar bases de datos gráficamente.
- Ideal para desarrollar y probar antes de desplegar online.

3. phpMyAdmin

Ventajas:

- Interfaz web amigable para gestionar MySQL.
- Permite crear, modificar y visualizar bases de datos sin necesidad de comandos SQL complejos.
- Exporta e importa estructuras y datos fácilmente (útil para backups y migraciones).

4. Navegadores modernos (Chrome, Firefox, Edge)

Ventajas:

- Herramientas de desarrollo integradas (DevTools) para depurar código HTML, CSS y JS.
- Simulación de dispositivos móviles para test responsive.
- Buen soporte para estándares web.

5. Git + GitHub / GitLab

Ventajas:

- Control de versiones: permite ver cambios, volver atrás y colaborar en equipo sin pisarse.
- Trabajo en ramas para probar nuevas funcionalidades sin romper la versión estable.
- Historial completo de desarrollo y colaboración remota.

6. Lenguajes y tecnologías elegidas (HTML, CSS, JS, PHP y MySQL)

Ventajas:

- Son tecnologías ampliamente soportadas y con documentación disponible.
- PHP y MySQL son ideales para sistemas CRUD y login como el que usamos.
- HTML, CSS y JS permiten personalizar totalmente la experiencia de usuario.
- Código 100% controlado por el equipo, sin depender de plataformas externas.

Requisitos de software (entorno objetivo en producción)

- Fedora Server versión estable más reciente al momento del despliegue, con actualizaciones de seguridad aplicadas y soporte para arquitectura x86_64. Se recomienda utilizar un kernel Linux 6.x o superior para asegurar compatibilidad con las últimas mejoras de rendimiento y seguridad.
- Servidor web Apache HTTP Server versión 2.4 o superior, con el módulo mod_rewrite habilitado para permitir la gestión de rutas amigables de Laravel y soporte para archivos .htaccess. La configuración debe establecer el DocumentRoot apuntando a la carpeta public/ del proyecto para proteger los archivos internos y permitir que las rutas públicas sean gestionadas por Laravel. En el bloque <Directory> correspondiente a public/, se debe habilitar AllowOverride All para que el framework pueda aplicar su configuración de redirección y seguridad definida en .htaccess.
- PHP en versión mínima 8.2, siendo recomendable 8.3 para aprovechar mejoras de rendimiento y compatibilidad total con Laravel 11. Es obligatorio contar con las siguientes extensiones: pdo_mysql para la conexión segura a MySQL, mbstring para el manejo de cadenas multibyte y soporte completo de UTF-8, openssl para operaciones criptográficas y de seguridad, curl para la interacción con APIs externas, json para codificación y decodificación de datos en este formato, ctype para validaciones de caracteres, tokenizer para funcionalidades internas del framework y fileinfo para el reconocimiento de tipos de archivos. Se recomienda además habilitar opcache para mejorar el rendimiento mediante cacheo de código PHP y intl para soporte de internacionalización y localización, necesario para la versión bilingüe de la aplicación.
- Base de datos MySQL Community Server versión 8.0 o superior, utilizando el motor de almacenamiento InnoDB por su soporte de transacciones y claves foráneas. La base debe configurarse con el juego de caracteres utf8mb4 y collation utf8mb4_unicode_ci para garantizar compatibilidad con todo el rango de caracteres Unicode, incluidos emojis. Se recomienda mantener el modo SQL en estricto para reforzar la integridad de datos y prevenir inserciones o actualizaciones incompletas. Las claves foráneas deben estar habilitadas para garantizar relaciones consistentes entre tablas según el diseño normalizado del modelo relacional.
- Herramientas de desarrollo y despliegue: Git 2.x para control de versiones y trabajo colaborativo, Composer 2.x para la gestión de dependencias PHP, Node.js LTS junto con npm para compilación y gestión de assets mediante Vite (aunque en la versión actual del proyecto su uso es opcional). Como entorno de desarrollo se recomienda Visual Studio Code con extensiones específicas para PHP y Laravel, incluyendo PHP Intelephense para autocompletado y análisis, Laravel Blade Snippets para soporte de plantillas Blade y Laravel Extra Intellisense para mayor productividad en controladores, rutas y modelos.

- En el cliente, la aplicación debe ser accesible y funcional en navegadores modernos y actualizados: Google Chrome, Mozilla Firefox y Microsoft Edge en sus últimas versiones estables, asegurando compatibilidad con HTML5, CSS3 y JavaScript moderno, así como con el diseño responsivo implementado mediante Bootstrap 5.

Incluir configuración de IDE (Visual Studio Code recomendado) y extensiones útiles

Utilizamos Visual Studio Code como entorno de desarrollo principal para todo el ciclo de trabajo del proyecto. Siempre nos aseguramos de tener instalada la última versión estable disponible, ya que de esta forma contamos con las funciones más recientes y garantizamos la compatibilidad con las tecnologías que usamos, especialmente Laravel y PHP 8.4.

Nuestra configuración de Visual Studio Code está pensada para maximizar la productividad y la calidad del código. Habilitamos el formato automático al guardar (Format on Save) para mantener un estilo de código consistente sin necesidad de hacerlo manualmente. También activamos el resaltado de sintaxis y el autocompletado inteligente para PHP, Blade y JavaScript, junto con la depuración integrada que nos permite ejecutar y analizar el código directamente desde el IDE. Vinculamos Visual Studio Code con nuestro repositorio Git para que todo el control de versiones se realice dentro del mismo entorno, lo que agiliza la gestión de commits, ramas, fusiones y revisiones de cambios.

Como el entorno de ejecución final está en Fedora Server, configuramos la extensión Remote - SSH para conectarnos directamente al servidor y trabajar de forma remota. Esto nos permite abrir el proyecto que está en producción o en pruebas, editarlo en tiempo real y guardar los cambios sin transferencias manuales de archivos. También configuramos el intérprete de PHP para que apunte a la instalación real del servidor, garantizando que la ejecución y depuración sean fieles al entorno final.

Ajustamos Word Wrap para mejorar la lectura de líneas largas en plantillas Blade y archivos de configuración, y definimos reglas de sangría y espaciado que coinciden con las convenciones de Laravel y PSR-12.

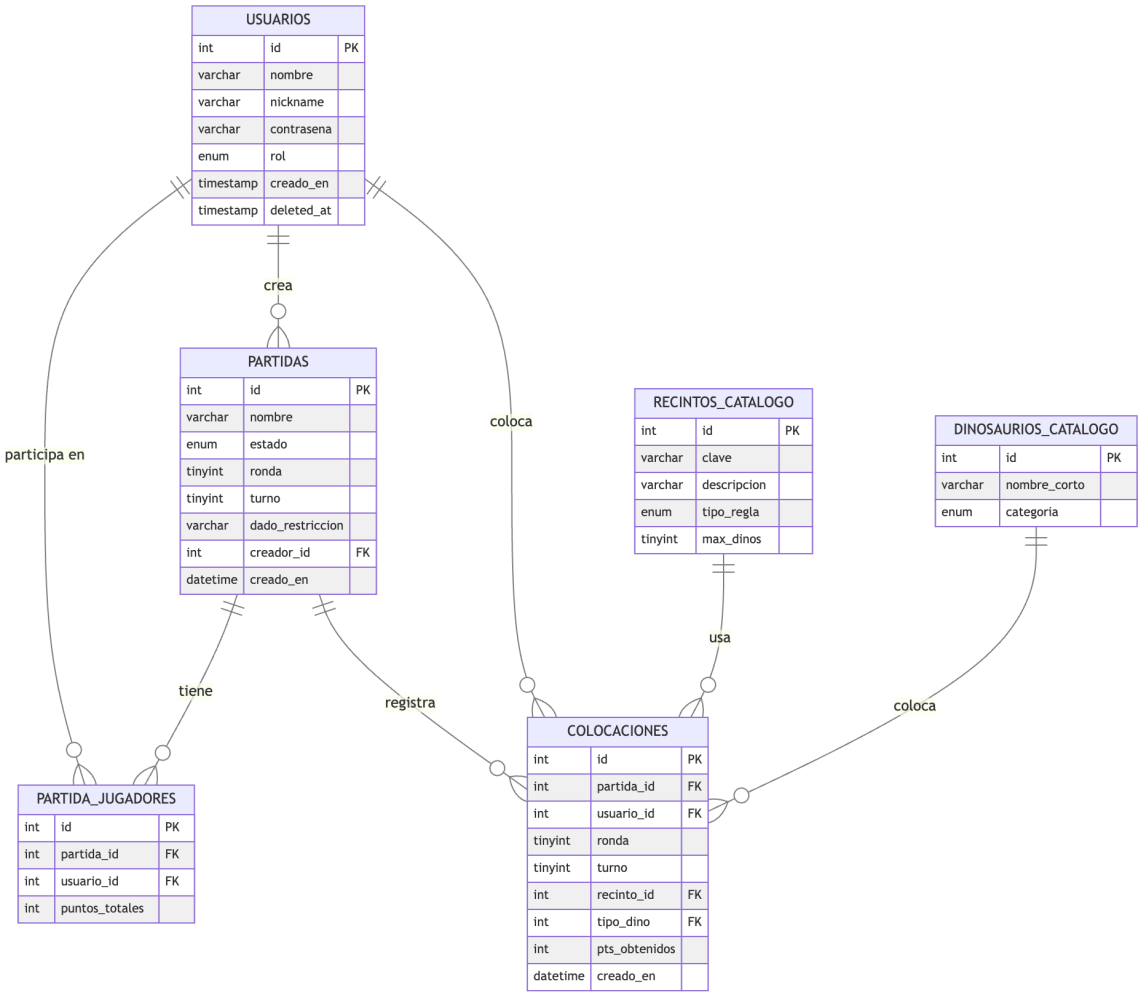
En cuanto a las extensiones que instalamos y usamos en este proyecto, cada una cumple un propósito específico:

1. PHP Intelephense – Nos da autocompletado avanzado, detección temprana de errores, navegación rápida a definiciones y documentación integrada para funciones y métodos.
2. Laravel Blade Snippets – Añade soporte específico para la sintaxis de Blade, con fragmentos predefinidos y resaltado de código optimizado.
3. Laravel Extra Intellisense – Proporciona autocompletado para rutas, controladores, vistas, facades y otros elementos propios de Laravel que el autocompletado estándar no detecta.
4. DotENV – Resalta y valida archivos .env, ayudando a detectar errores de sintaxis o variables faltantes.

5. GitLens — Git supercharged – Mejora el manejo de Git desde VS Code, mostrando historial de cambios por línea, autores, comparación de versiones y paneles avanzados para commits y ramas.
6. EditorConfig for VS Code – Asegura que todos los integrantes del equipo mantengan la misma configuración de formato de código, independientemente de su editor o sistema operativo.
7. ESLint – Analiza y corrige problemas en el código JavaScript, ayudando a mantener la calidad y consistencia del frontend.
8. Prettier – Aplica formato automático al código JavaScript, TypeScript, HTML, CSS y JSON, lo que complementa el trabajo de ESLint y mantiene un estilo uniforme.
9. Bootstrap 5 & Font Awesome Snippets – Facilita la inserción rápida de componentes y clases de Bootstrap, así como íconos de Font Awesome, para acelerar la maquetación y el desarrollo de la interfaz.
10. Path Intellisense – Autocompleta rutas de archivos y carpetas en el código, reduciendo errores de tipeo y acelerando la vinculación de recursos.
11. PHP Namespace Resolver – Ayuda a gestionar automáticamente los use en PHP, organizando e importando namespaces de forma correcta.
12. REST Client – Nos permite probar endpoints y rutas de la API directamente desde VS Code sin necesidad de abrir Postman.
13. Auto Rename Tag – Sincroniza automáticamente las etiquetas de apertura y cierre en HTML y Blade, evitando errores en la estructura.
14. Bracket Pair Color DLW – Colorea los pares de llaves, corchetes y paréntesis para facilitar la lectura de código anidado.

Esta combinación de extensiones nos permite trabajar de manera fluida en todo el proyecto JurassiDraft, desde la edición de controladores PHP hasta la maquetación de vistas y la integración con la base de datos, siempre asegurando un flujo de trabajo rápido, ordenado y estandarizado.

Diagrama entidad-relación inicial que refleje fielmente la estructura del juego



Entidades Principales con Atributos

1. **Usuarios**
    - **Atributos**:
        - id: INT (Clave primaria)
        - nombre: VARCHAR(100) (Nombre del usuario)
        - nickname: VARCHAR(50) (Nombre único de usuario)
        - contrasena: VARCHAR(255) (Contraseña cifrada)
        - rol: ENUM('admin', 'jugador') (Rol del usuario)
        - creado_en: TIMESTAMP (Fecha de creación)
        - deleted_at: TIMESTAMP (Fecha de eliminación, si es aplicable)
2. **Partidas**
    - **Atributos**:
        - id: INT (Clave primaria)
        - nombre: VARCHAR(120) (Nombre de la partida)
        - estado: ENUM('config', 'en_curso', 'cerrada') (Estado de la partida)
        - ronda: TINYINT (Número de ronda, por defecto 1)

- turno: TINYINT (Número de turno, por defecto 1)
- dado_restriccion: VARCHAR(120) (Restricción aplicada por dado)
- creador_id: INT (ID del usuario creador, clave foránea)
- creado_en: DATETIME (Fecha de creación)

3. **Partida Jugadores**
   - **Atributos**:
     - id: INT (Clave primaria)
     - partida_id: INT (ID de la partida, clave foránea)
     - usuario_id: INT (ID del jugador, clave foránea)
     - puntos_totales: INT (Puntos totales del jugador, por defecto 0)
4. **Recintos Catalogo**
   - **Atributos**:
     - id: INT (Clave primaria)
     - clave: VARCHAR(80) (Clave única del recinto)
     - descripcion: VARCHAR(255) (Descripción del recinto)
     - tipo_regla: ENUM('variedad', 'parejas', 'solitario', 'rio', 'trex', 'neutro') (Tipo de regla del recinto, por defecto 'neutro')

- max_dinos: TINYINT (Número máximo de dinosaurios permitidos en el recinto, por defecto 6)

5. **Dinosaurios Catalogo**
   - **Atributos**:
     - id: INT (Clave primaria)
     - nombre_corto: VARCHAR(60) (Nombre corto del dinosaurio, único)
     - categoria: ENUM('herbivoro', 'carnivoro', 'especial') (Categoría del dinosaurio)
6. **Colocaciones**
   - **Atributos**:
     - id: INT (Clave primaria)
     - partida_id: INT (ID de la partida, clave foránea)
     - usuario_id: INT (ID del jugador, clave foránea)
     - ronda: TINYINT (Número de ronda)
     - turno: TINYINT (Número de turno)
     - recinto_id: INT (ID del recinto, clave foránea)
     - tipo_dino: INT (ID del dinosaurio, clave foránea)

- pts_obtenidos: INT (Puntos obtenidos por la colocación, por defecto 0)
- creado_en: DATETIME (Fecha de creación)

Relaciones entre Entidades

1. **Usuarios ↔ Partidas**
   - Relación: Un usuario puede crear muchas partidas (relación uno a muchos).
   - En la tabla **partidas**, creador_id es una clave foránea que hace referencia a la tabla **usuarios**.
2. **Usuarios ↔ Partida Jugadores**
   - Relación: Un usuario puede estar en muchas partidas y en una partida puede haber muchos jugadores (relación muchos a muchos).
   - Se utiliza la tabla **partida_jugadores** como tabla intermedia, donde cada fila representa la relación entre un **usuario** y una **partida**.
   - La clave primaria de esta tabla es id y las claves foráneas son partida_id y usuario_id.
3. **Partidas ↔ Colocaciones**
   - Relación: Una partida puede tener muchas colocaciones (relación uno a muchos).

- En la tabla **colocaciones**, partida_id es una clave foránea que hace referencia a la tabla **partidas**.

4. **Usuarios ↔ Colocaciones**

   - Relación: Un usuario puede hacer muchas colocaciones en diferentes rondas (relación uno a muchos).

   - En la tabla **colocaciones**, usuario_id es una clave foránea que hace referencia a la tabla **usuarios**.

5. **Recintos Catalogo ↔ Colocaciones**

   - Relación: Un recinto puede tener muchas colocaciones (relación uno a muchos).

   - En la tabla **colocaciones**, recinto_id es una clave foránea que hace referencia a la tabla **recintos_catalogo**.

6. **Dinosaurios Catalogo ↔ Colocaciones**

   - Relación: Un dinosaurio puede estar en muchas colocaciones (relación uno a muchos).

   - En la tabla **colocaciones**, tipo_dino es una clave foránea que hace referencia a la tabla **dinosaurios_catalogo**.

Realizar la transformación del DER al modelo relacional (pasaje a tablas)

```sql
CREATE TABLE usuarios (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  nickname VARCHAR(50) NOT NULL UNIQUE,
  contrasena VARCHAR(255) NOT NULL,
  rol ENUM('admin','jugador') DEFAULT 'jugador',
  creado_en TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  deleted_at TIMESTAMP NULL
);

CREATE TABLE partidas (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(120) NOT NULL,
  estado ENUM('config','en_curso','cerrada') DEFAULT 'config',
  ronda TINYINT DEFAULT 1,
  turno TINYINT DEFAULT 1,
  dado_restriccion VARCHAR(120),
  creador_id INT NOT NULL,
  creado_en DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (creador_id) REFERENCES usuarios(id)
);

CREATE TABLE partida_jugadores (
  id INT AUTO_INCREMENT PRIMARY KEY,
  partida_id INT NOT NULL,
  usuario_id INT NOT NULL,
  puntos_totales INT DEFAULT 0,
  FOREIGN KEY (partida_id) REFERENCES partidas(id),
  FOREIGN KEY (usuario_id) REFERENCES usuarios(id)
);

CREATE TABLE recintos_catalogo (
  id INT AUTO_INCREMENT PRIMARY KEY,
  clave VARCHAR(80) UNIQUE NOT NULL,
  descripcion VARCHAR(255) NOT NULL,
  tipo_regla
ENUM('variedad','parejas','solitario','rio','trex','neutro')
```

```
DEFAULT 'neutro',
  max_dinos TINYINT DEFAULT 6
);

CREATE TABLE dinosaurios_catalogo (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nombre_corto VARCHAR(60) UNIQUE NOT NULL,
  categoria ENUM('herbivoro','carnivoro','especial') NOT NULL
);

CREATE TABLE colocaciones (
  id INT AUTO_INCREMENT PRIMARY KEY,
  partida_id INT NOT NULL,
  usuario_id INT NOT NULL,
  ronda TINYINT NOT NULL,
  turno TINYINT NOT NULL,
  recinto_id INT NOT NULL,
  tipo_dino INT NOT NULL,
  pts_obtenidos INT DEFAULT 0,
  creado_en DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (partida_id) REFERENCES partidas(id),
  FOREIGN KEY (usuario_id) REFERENCES usuarios(id),
  FOREIGN KEY (recinto_id) REFERENCES recintos_catalogo(id),
  FOREIGN KEY (tipo_dino) REFERENCES dinosaurios_catalogo(id)
```

Aplicar el proceso de normalización hasta la Tercera Forma Normal (3FN)

1FN (Primera Forma Normal):

- Todas las tablas cumplen con la **1FN**, ya que los atributos son atómicos y no contienen grupos repetidos. Cada registro tiene un valor único para cada columna.

2FN (Segunda Forma Normal):

- La tabla **partida_jugadores** no tiene dependencias parciales, ya que la clave primaria es una combinación de (partida_id, usuario_id), y todas las columnas dependen de ambas claves.

- La tabla **colocaciones** también está correctamente normalizada, ya que su clave primaria es id, y las claves foráneas no tienen dependencias parciales.

3FN (Tercera Forma Normal):

- La tabla **recintos_catalogo** y **dinosaurios_catalogo** están bien separadas para evitar dependencias transitivas.

- En **colocaciones**, las claves foráneas recinto_id y tipo_dino se refieren a las tablas **recintos_catalogo** y **dinosaurios_catalogo**, respectivamente, eliminando dependencias transitivas.

Identificar y documentar claramente claves primarias y foráneas

1. Tabla: usuarios

Claves

- **Clave primaria (PK)**:
    - id: INT (Clave primaria única para identificar a cada usuario)

2. Tabla: partidas

Claves

- **Clave primaria (PK)**:
    - id: INT (Clave primaria única para identificar cada partida)
- **Clave foránea (FK)**:
    - creador_id: INT (Clave foránea que hace referencia a la columna id de la tabla **usuarios**)

3. Tabla: partida_jugadores

Claves

- **Clave primaria (PK)**:
    - id: INT (Clave primaria única para cada registro de la tabla)
- **Claves foráneas (FK)**:

- partida_id: INT (Clave foránea que hace referencia a la columna id de la tabla **partidas**)

- usuario_id: INT (Clave foránea que hace referencia a la columna id de la tabla **usuarios**)

Relación: Una partida puede tener múltiples jugadores, y un jugador puede estar en muchas partidas.

4. Tabla: recintos_catalogo

Claves

- **Clave primaria (PK)**:
  - id: INT (Clave primaria única para identificar cada recinto)
- **Clave única (Unique Key)**:
  - clave: VARCHAR(80) (Clave única para identificar el recinto)

Relación: Los recintos están asociados a las colocaciones, pero no tienen claves foráneas aquí, ya que es una tabla de catálogo.

5. Tabla: dinosaurios_catalogo

Claves

- **Clave primaria (PK)**:
  - id: INT (Clave primaria única para identificar cada dinosaurio)

- **Clave única (Unique Key)**:
    - nombre_corto: VARCHAR(60) (Nombre corto único para identificar el dinosaurio)

Relación: Los dinosaurios están asociados a las colocaciones, pero no tienen claves foráneas en esta tabla, ya que también es un catálogo.

6. Tabla: colocaciones

Claves

- **Clave primaria (PK)**:
    - id: INT (Clave primaria única para identificar cada colocación)
- **Claves foráneas (FK)**:
    - partida_id: INT (Clave foránea que hace referencia a la columna id de la tabla **partidas**)
    - usuario_id: INT (Clave foránea que hace referencia a la columna id de la tabla **usuarios**)
    - recinto_id: INT (Clave foránea que hace referencia a la columna id de la tabla **recintos_catalogo**)
    - tipo_dino: INT (Clave foránea que hace referencia a la columna id de la tabla **dinosaurios_catalogo**)

Relación: Cada colocación está asociada a una partida, un jugador, un recinto y un dinosaurio.

Resumen de Claves:

Claves Primarias (PK):

1. **usuarios**: id
2. **partidas**: id
3. **partida_jugadores**: id
4. **recintos_catalogo**: id
   dinosaurios_catalogo: id
5. **colocaciones**: id

Claves Foráneas (FK):

1. **partidas**: creador_id → **usuarios(id)**
2. **partida_jugadores**:
   - partida_id → **partidas(id)**
   - usuario_id → **usuarios(id)**
3. **colocaciones**:
   - partida_id → **partidas(id)**
   - usuario_id → **usuarios(id)**
   - recinto_id → **recintos_catalogo(id)**
   - tipo_dino → **dinosaurios_catalogo(id)**

Registrar todas las decisiones de diseño con su justificación técnica

Decisiones de Datos (Modelo y Normalización)

1. Mantener la tabla usuarios existente
   Decisión: Conservar la tabla usuarios tal como está, mapeando contrasena como password en el modelo de Laravel.
   Justificación: Evitar cambios en producción/pruebas, reducir riesgos de incompatibilidades y facilitar la integración de la autenticación Laravel sin fricciones.

2. Modelo orientado a eventos de juego
   Decisión: Separar en entidades partidas, partida_jugadores, recintos, dinosaurios_catalogo, y colocaciones.
   Justificación: Cada entidad refleja un concepto único. Colocaciones registra cada jugada sin duplicar datos maestros, favoreciendo auditoría y consultas por partida/jugador.

3. Normalización a 3FN
   Decisión: Catálogos como recintos y dinosaurios_catalogo se mantienen independientes; se resuelve la relación N–N con partida_jugadores.
   Justificación: Elimina redundancias y anomalías de actualización, mejorando la integridad y claridad del modelo.

4. Claves técnicas + unicidades naturales
   Decisión: Usar claves primarias AUTO_INCREMENT y unicidades en claves naturales (recintos.clave, dinosaurios_catalogo.nombre_corto) y lógicas (partida_jugadores.partida_id + usuario_id).
   Justificación: Las claves técnicas simplifican relaciones y FKs, y las claves naturales preservan la consistencia de catálogos y reglas de negocio.

5. Clave primaria candidata operativa en colocaciones
   Decisión: Permitir clave técnica id, pero documentar (partida_id, usuario_id,

ronda, turno) como clave candidata.
Justificación: La clave técnica simplifica, pero la clave candidata refleja la regla del dominio (una jugada por turno) y se puede activar como UNIQUE cuando sea necesario.

6. Campo derivado pts_obtenidos
   Decisión: Almacenar pts_obtenidos en colocaciones a pesar de ser calculable.
   Justificación: Mejora el rendimiento para listados y resultados, y garantiza consistencia recalculando solo en cierres o validándolo en servicios.

7. Enums controlados en partidas.estado
   Decisión: Usar ENUM('config','en_curso','cerrada') para el estado de la partida y tipos pequeños TINYINT para ronda y turno.
   Justificación: El uso de ENUM hace explícitos los valores válidos y previene valores inválidos, y los tipos pequeños mejoran el rendimiento de índices y caché.

8. Charset/collation
   Decisión: Usar utf8mb4_unicode_ci en toda la base.
   Justificación: Compatibilidad total con Unicode y emojis, garantizando coherencia en las búsquedas y almacenamiento entre tablas.

9. Integridad y reglas de borrado/actualización
   Decisión: Integridad referencial estricta con ON UPDATE CASCADE en todas las tablas, ON DELETE CASCADE donde aplica y ON DELETE RESTRICT en referencias a usuarios y catálogos.
   Justificación: Evita registros huérfanos y asegura limpieza automática en la eliminación de partidas.

Arquitectura y Backend

10. Laravel como framework

    Decisión: Usar Laravel para ruteo, middleware, Eloquent y validaciones.

    Justificación: Alta productividad, seguridad por defecto (CSRF, XSS, bindings), estandarización y comunidad amplia.

11. Capa de negocio en services

    Decisión: Encapsular lógica de dominio en servicios (ReglasService, PuntajeService, PartidaService).

    Justificación: Controladores livianos, testeo aislado de reglas y mayor facilidad de mantenimiento a medida que evoluciona el juego.

12. Autenticación con tabla personalizada

    Decisión: Usar Auth::attempt con los campos usuario y contrasena, mutando password en el modelo.

    Justificación: Integración del login de Laravel sin cambiar la tabla y mantiene un hashing seguro (bcrypt).

13. Middleware por rol

    Decisión: Usar middleware EsAdmin para proteger las rutas de administración.

    Justificación: Control explícito de roles y seguridad clara sin depender del alias del Kernel.

Frontend y UX

14. Tailwind CSS como base

    Decisión: Usar Tailwind CSS para la estructura de layout, grid y componentes.

    Justificación: Tailwind CSS ofrece un diseño altamente personalizable, responsivo y modular sin la necesidad de sobrescribir estilos como en otros frameworks. Permite una mayor flexibilidad en la personalización del diseño con clases utilitarias directamente en HTML, reduciendo el costo de personalización de CSS y aumentando la productividad en el desarrollo. Además, se adapta fácilmente a los cambios de diseño y mejora el rendimiento mediante el uso de purge para eliminar el CSS no utilizado.

15. División por modos (Seguimiento primero)

    Decisión: Implementar primero el Modo Seguimiento (registro y validación de colocaciones y puntajes) y extender a Modo Digitalizado.

    Justificación: El primer modo cubre el alcance básico y facilita recorrer el flujo completo de puntuación antes de agregar manos, pasos y dados.

Seguridad y Buenas Prácticas

16. Consultas preparadas y sanitización

    Decisión: Usar Eloquent/Query Builder (bindings) y validaciones server-side con FormRequests.

    Justificación: Previene SQL injection y reduce la superficie de errores de entrada.

17. Sesiones en archivo en desarrollo

    Decisión: Usar SESSION_DRIVER=file en desarrollo y DB/Redis en

producción si es necesario.
Justificación: Simplificación en el entorno local; más robustez si el tráfico lo requiere en producción.

18. Autorización por pertenencia
Decisión: Validar que el usuario pertenezca a la partida para acceder a sus datos.
Justificación: Privacidad entre jugadores y control de acceso a los recursos.

Extensión Prevista (Modo Digitalizado)

19. Estado de mano y bolsa
Decisión: Usar JSON para representar manos y bolsas (contenido_json y stock_json).
Justificación: Reduce complejidad y joins; permite flexibilidad para manejar colecciones variables de dinosaurios por turno en MySQL 8.

20. Historial de restricciones (dados)
Decisión: Crear una tabla restricciones para almacenar entradas por partida, ronda, y turno.
Justificación: Permite auditar y reproducir restricciones, desacoplando el estado actual del historial.

21. Validación cruzada app + BD
Decisión: Aplicar restricciones no estructurales (dados, recintos, una jugada por turno) en services y reforzarlas con UNIQUE en colocaciones cuando se requiera.
Justificación: Flexibilidad de reglas en la app, con la opción de endurecer en la base de datos cuando se formalice la mecánica.

Despliegue y Entorno

22. Fedora Server + Apache 2.4 + PHP 8.3 + MySQL 8

    Decisión: Usar stack Linux estándar (Fedora + Apache + PHP + MySQL).

    Justificación: Facilidad de administración, alta seguridad, compatibilidad con Laravel 11 y MySQL 8.

23. Collation y zona horaria

    Decisión: Usar utf8mb4_unicode_ci y la zona horaria America/Montevideo en PHP.

    Justificación: Soporte completo de caracteres y coherencia de timestamps en el entorno local.

Internacionalización

24. ES/EN con archivos de idioma

    Decisión: Almacenar los recursos en resources/lang/es y resources/lang/en, utilizando helpers de traducción.

    Justificación: Soporta múltiples idiomas y desacopla los textos, lo que facilita la expansión a más idiomas en el futuro.

Trade-offs Explícitos

25. ENUM vs tabla de estados

    Decisión: Usar ENUM para el estado de las partidas.

    Justificación: El conjunto de valores es acotado y estable, y la consulta es más simple. Si los estados cambian en el futuro, se migrará a una tabla de soporte.

26. pts_obtenidos materializado

    Decisión: Almacenar los puntos por jugada en colocaciones.

    Justificación: Mejora el rendimiento y la auditoría, y el costo de sincronización se controla en la capa de negocio.

Documentar exhaustivamente las restricciones no estructurales derivadas de las reglas del juego

1. Estados de partida: la partida solo puede estar en uno de los estados válidos: config, en_curso o cerrada.

2. Transiciones de estado: el flujo permitido es config → en_curso → cerrada. No se puede retroceder de cerrada a en_curso sin acción de administrador.

3. Cantidad de rondas y turnos: una partida básica consta de 2 rondas, cada una de 6 turnos.

4. Participantes registrados: solo los usuarios inscritos en la partida pueden realizar jugadas en ella.

5. Orden de mesa fijo: cada jugador tiene un orden asignado al inicio y no se puede cambiar durante la partida.

6. Una jugada por turno: cada jugador debe realizar exactamente una colocación por turno.

7. Unicidad por turno: no puede haber más de una colocación con la misma combinación (partida, jugador, ronda, turno).

8. Restricción por dado: cada turno tiene una restricción que limita los recintos permitidos para todos los jugadores.

9. Obligatoriedad de cumplir la restricción: si existe al menos un movimiento válido que cumpla la restricción, el jugador debe realizarlo.

10. Capacidad de recintos: un recinto no puede recibir más dinosaurios de los permitidos por su capacidad.

11. Compatibilidad de recintos: algunos recintos aceptan solo determinados tipos de dinosaurios; se debe cumplir esta regla.

12. Validez de datos: todos los dinosaurios colocados deben existir en el catálogo y todos los recintos deben existir en el catálogo de recintos.

13. Pertenencia de jugadas: las jugadas deben estar asociadas a la partida y jugador correctos.

14. Estado para jugar: no se admiten colocaciones si la partida no está en estado en_curso.

15. Puntuación automática: cada colocación genera puntos según la regla del recinto donde se coloca.

16. Cálculo de ganador: al final de la segunda ronda, se suman los puntos de cada jugador y se determina el ganador.

17. Inmutabilidad tras cierre: no se pueden modificar jugadas ni puntajes una vez que la partida está cerrada.

18. Integridad referencial de catálogos: no se pueden eliminar recintos o dinosaurios que estén en uso en alguna colocación.

19. Acciones de administrador: solo un usuario con rol admin puede crear, eliminar o cerrar partidas y revertir jugadas.

20. Registro de auditoría: toda colocación debe tener sello de tiempo y referencia al usuario y partida para trazabilidad.

Especificar cómo estas restricciones serán implementadas en el sistema

Estados de partida (config, en_curso, cerrada)

Base de datos: el campo estado en la tabla partidas define el estado actual mediante un ENUM.
Backend: las validaciones se realizan en el PartidaController, que controla las acciones permitidas según el estado.
Frontend: la interfaz oculta o desactiva las acciones no válidas dependiendo del estado actual de la partida.

Transiciones de estado (config → en_curso → cerrada)

Base de datos: se guarda el estado actual de la partida sin forzar la transición directamente.
Backend: el PartidaController define las transiciones válidas y evita retrocesos; solo los administradores pueden revertir estados.
Frontend: los botones y flujos visibles se adaptan según la fase activa del juego.

Rondas y turnos (2 rondas, 6 turnos)

Base de datos: partidas almacena ronda y turno para seguir el progreso.
Backend: el PartidaController controla el avance entre turnos y rondas, impidiendo jugadas fuera del turno activo.
Frontend: la interfaz muestra la ronda y turno actual y bloquea acciones si no corresponde la fase.

Participantes registrados

Base de datos: partida_jugadores relaciona usuarios con partidas, garantizando unicidad por (partida_id, usuario_id).
Backend: los controladores verifican que el usuario autenticado esté registrado antes de permitir acciones.
Frontend: el panel de juego se muestra únicamente a los jugadores que pertenecen a la partida activa.

Orden de mesa fijo

Base de datos: el campo orden_mesa en partida_jugadores define la posición de cada jugador.
Backend: el PartidaController asigna el orden al inscribir jugadores y evita cambios una vez iniciada la partida.
Frontend: se renderiza la lista de jugadores en el orden establecido.

Una jugada por turno

Base de datos: la tabla colocaciones puede reforzarse con una restricción única por (partida_id, usuario_id, ronda, turno) para asegurar una sola jugada.

Backend: el ColocacionesController verifica que no exista una jugada previa para ese turno antes de guardar una nueva.
Frontend: tras confirmar la jugada, se desactiva la acción de envío para prevenir duplicaciones.

Restricción por dado

Base de datos: partidas incluye el campo dado_restriccion que indica la limitación del turno actual.
Backend: el ColocacionesController consulta esta restricción y delega su validación al ServicioPuntaje, que determina si el recinto elegido cumple con la regla.
Frontend: solo se muestran recintos válidos según la restricción activa del dado.

Capacidad de recintos

Base de datos: recintos_catalogo define la capacidad máxima con el campo max_dinos.
Backend: el ServicioPuntaje cuenta las colocaciones actuales por recinto y jugador, bloqueando jugadas que superen la capacidad.
Frontend: se muestra la cantidad de espacios disponibles por recinto en tiempo real.

Compatibilidad de recintos y dinosaurios

Base de datos: las compatibilidades se derivan de recintos_catalogo y dinosaurios_catalogo.
Backend: el ServicioPuntaje evalúa si el tipo de dinosaurio puede colocarse en el recinto correspondiente antes de guardar la jugada.
Frontend: las opciones no válidas se bloquean o se muestran con estilos deshabilitados.

Validez de datos (existencia en catálogos)

Base de datos: las claves foráneas en colocaciones aseguran la existencia de recintos y dinosaurios válidos.
Backend: los controladores usan validaciones por FormRequest para asegurar integridad antes de insertar datos.
Frontend: los formularios solo permiten seleccionar opciones provenientes de catálogos precargados.

Pertenencia de jugadas

Base de datos: las relaciones de colocaciones con partidas y usuarios garantizan correspondencia correcta.
Backend: los controladores validan que el usuario autenticado pertenezca a la partida y coincida con la jugada enviada.
Frontend: se utiliza el contexto de la partida activa para evitar IDs externos o inválidos.

Estado para jugar

Base de datos: el estado en_curso de partidas determina si se pueden realizar jugadas.
Backend: los controladores bloquean cualquier jugada o acción si la partida no se encuentra activa.
Frontend: la interfaz inhabilita o oculta la funcionalidad de juego si la partida no está en curso.

Puntuación automática por recinto

Base de datos: colocaciones mantiene el campo pts_obtenidos para registrar los puntos de cada jugada.
Backend: el ServicioPuntaje calcula los puntos obtenidos al confirmar una jugada y los guarda junto a la colocación.
Frontend: el puntaje parcial se actualiza dinámicamente tras cada colocación.

Cálculo de ganador

Base de datos: partida_jugadores conserva el campo puntos_totales como acumulado de cada jugador.
Backend: el ResultadosController realiza el cálculo final y marca la partida como cerrada al finalizar.
Frontend: se muestra una vista de resultados con el ranking general y detalle por jugador.

Inmutabilidad tras cierre

Base de datos: el estado cerrada en partidas impide seguir registrando jugadas.
Backend: los controladores rechazan cualquier intento de modificación posterior.
Frontend: las partidas cerradas se muestran solo en modo lectura.

Integridad de catálogos

Base de datos: las claves foráneas en colocaciones hacia recintos_catalogo y dinosaurios_catalogo impiden eliminar registros en uso.
Backend: los controladores capturan y comunican los errores de integridad.
Frontend: se evita la eliminación desde la interfaz cuando el elemento está referenciado en partidas activas o pasadas.

Acciones de administrador

Base de datos: los roles se definen mediante el campo rol (valores admin o jugador) en usuarios.
Backend: el middleware EsAdmin protege rutas administrativas y los controladores controlan acciones exclusivas como cerrar partidas o revertir estados.
Frontend: los botones administrativos se muestran únicamente a usuarios con rol de administrador.

Registro de auditoría

Base de datos: los campos creado_en en partidas y colocaciones registran fechas y usuarios asociados, garantizando trazabilidad.

Backend: los controladores registran cada jugada y cambio de estado en logs o historiales internos.

Frontend: se ofrece una vista de historial de jugadas ordenadas por ronda, turno y jugador.

Notas transversales de implementación

Validación de entrada: todos los controladores emplean FormRequests para validar tipos, pertenencia y rangos de datos antes de procesar la acción.

Transacciones: las operaciones que combinan múltiples pasos (guardar jugada, calcular puntos y actualizar totales) se ejecutan dentro de transacciones de base de datos para asegurar consistencia.

Concurrencia: se controla mediante verificación previa y bloqueo de doble envío en frontend.

Seguridad: se aplican Policies por pertenencia, middleware de autenticación y protección CSRF activa.

Rendimiento: los índices definidos en partidas y colocaciones optimizan consultas por estado, turno y jugador.

Internacionalización: los mensajes del sistema y validaciones se cargan desde los archivos de idioma en resources/lang/es y resources/lang/en.

Refinar el esquema relacional normalizado (3FN), garantizando eliminación de redundancias

Usuarios

Dependencias:
Los atributos nombre, nickname, contrasena, rol, creado_en y deleted_at dependen directamente de id.
El atributo rol es un valor enumerado (admin o jugador), por lo que no se repite información en otra tabla.

Justificación:
No se observan dependencias transitivas, ya que todos los atributos dependen directamente de la clave primaria id.
El uso de un campo ENUM para rol evita redundancia sin requerir una tabla adicional.

Partidas

Dependencias:
Los atributos nombre, estado, ronda, turno, dado_restriccion, creador_id y creado_en dependen directamente de id.
El atributo creador_id es una clave foránea que referencia a usuarios(id), evitando duplicar datos del creador.

Justificación:
No hay dependencias transitivas, ya que todos los atributos dependen directamente de la clave primaria id.
El uso de creador_id como clave foránea asegura la integridad y evita duplicación de información del usuario creador.

Partida_jugadores

Dependencias:
La clave primaria es id, pero existe una clave candidata compuesta por (partida_id, usuario_id).
Los atributos puntos_totales dependen de la clave primaria.

Justificación:
Los datos de los jugadores no se duplican, ya que se hace referencia a usuarios y partidas mediante claves foráneas.
La combinación (partida_id, usuario_id) garantiza la unicidad de cada jugador por

partida.
puntos_totales se mantiene como atributo derivable pero almacenado para optimizar el cálculo de resultados sin causar redundancia significativa.

Recintos_catalogo

Dependencias:
Los atributos clave, descripcion, tipo_regla y max_dinos dependen directamente de id.
El atributo clave es único y no determina ningún otro atributo no clave.

Justificación:
No hay dependencias transitivas.
Cada recinto tiene una clave única que garantiza integridad y evita duplicaciones en la descripción o reglas asociadas.

Dinosaurios_catalogo

Dependencias:
Los atributos nombre_corto y categoria dependen directamente de id.
El atributo nombre_corto es único, pero no determina ningún otro atributo no clave.

Justificación:
No existen dependencias transitivas, ya que todos los atributos dependen directamente de id.
El catálogo evita redundancias al centralizar los tipos de dinosaurios y sus categorías.

Colocaciones

Dependencias:
Incluye partida_id, usuario_id, ronda, turno, recinto_id, tipo_dino, pts_obtenidos y creado_en.
Todos los atributos dependen de la clave primaria id.
Las claves foráneas (partida_id, usuario_id, recinto_id, tipo_dino) garantizan integridad y eliminan duplicación de datos en cascada.

Justificación:
No existen dependencias transitivas, ya que todos los atributos dependen directamente de id.

No se repiten datos de jugadores, recintos o dinosaurios, y pts_obtenidos se conserva únicamente con fines de trazabilidad y cálculo de reglas.

Conclusión general

Todas las tablas cumplen con la **Tercera Forma Normal (3FN)**:

- No existen grupos repetidos ni atributos multivaluados (1FN).
- No hay dependencias parciales de atributos sobre parte de una clave compuesta (2FN).
- Se eliminaron dependencias transitivas entre atributos no clave (3FN).

El esquema garantiza **consistencia, integridad referencial y eliminación de redundancias**, manteniendo la trazabilidad completa del proceso de juego (usuarios, partidas, jugadas y puntuaciones) y permitiendo escalar el modelo sin pérdida de normalización.

Documentar detalladamente las Restricciones No Estructurales (RNE) que representan reglas de negocio específicas

Restricciones No Estructurales (RNE)

Definición

Las restricciones no estructurales son reglas de negocio que no se reflejan directamente en el modelo relacional mediante claves, tipos de datos o claves foráneas. Aun así, deben cumplirse para garantizar que el sistema respete la lógica del juego y mantenga la coherencia de los datos. Su implementación se realiza en la capa de aplicación, mediante procedimientos almacenados, triggers adicionales o validaciones específicas.

RNE 1. Inscripción de jugadores en partidas

Un usuario no puede inscribirse dos veces en la misma partida. Debe respetarse un número máximo de jugadores permitido según las reglas del juego. La asignación de orden_mesa debe ser única y secuencial dentro de cada partida.

RNE 2. Recintos por jugador y partida

Cada jugador debe tener una instancia única de cada recinto en cada partida. No se permite asignar recintos de forma duplicada a un mismo jugador. La cantidad y tipo de recintos asignados deben coincidir con las reglas oficiales del juego.

RNE 3. Colocaciones (jugadas)

Cada colocación debe referirse a un recinto existente del tablero del jugador en esa partida. No se permite colocar un dinosaurio en un recinto que no cumpla las restricciones del dado o que ya esté lleno. En un mismo turno, cada jugador puede realizar como máximo una colocación válida.

RNE 4. Puntuación

Los puntos de un jugador se calculan en base a sus colocaciones y a las reglas de puntuación de los recintos. Los puntos totales no se almacenan materializados, sino que se obtienen a partir de la tabla puntos_recinto. La actualización de los puntos se realiza únicamente en el cierre de cada ronda o al finalizar la partida, evitando inconsistencias.

RNE 5. Avance de partida

El cambio de estado de la partida debe seguir un flujo válido: config → en_curso → cerrada. No puede cerrarse una partida sin haber comenzado ni reiniciarse una vez finalizada. Solo el creador de la partida o un administrador pueden cerrarla manualmente. La transición de estado debe realizarse en un único request al finalizar un turno o lote de jugadas.

RNE 6. Auditoría de lotes de turnos
Cada cierre de turno debe registrarse en lotes_turno para mantener historial y trazabilidad. El lote debe reflejar la cantidad exacta de jugadas realizadas. Los datos de auditoría (ronda_final, turno_final, dado_final) se utilizan para consulta histórica, sin modificar el estado actual de la partida.

RNE 7. Roles y permisos
Los usuarios con rol admin pueden crear y eliminar partidas, así como gestionar jugadores. Los jugadores únicamente pueden unirse a partidas abiertas (estado config). No pueden inscribirse en partidas que ya estén en curso o cerradas.

RNE 8. Integridad temporal
No se permite registrar jugadas en partidas cerradas. Tampoco se aceptan colocaciones con fecha anterior a la creación de la partida. El campo creado_en debe reflejar siempre la secuencia real de jugadas y eventos.

Conclusión
Las Restricciones No Estructurales garantizan que el modelo en 3FN se complemente con reglas específicas del juego Draftosaurus. Estas reglas aseguran la validez de las jugadas, el correcto desarrollo de las partidas y el cumplimiento de la lógica de negocio definida. Su cumplimiento debe ser controlado desde la aplicación y, en casos necesarios, reforzado mediante procedimientos o triggers en la base de datos.

Contenido de la guía

Requisitos previos

Se requiere tener instalado PHP (versión 8.2 o superior) con las extensiones pdo_mysql, mbstring, openssl, bcmath, intl, gd, fileinfo, curl y zip. Además, es necesario Composer (versión 2.6 o superior), Node.js LTS (18 o 20) con npm, MySQL 8 y Git. En Windows se sugiere utilizar XAMPP para disponer de PHP y MySQL.
Clonación del repositorio

Para comenzar, se debe clonar el proyecto desde GitHub:

```
git clone https://github.com/JurassiCode/PROYECTO.git jurassidraft
cd jurassidraft
```

Configuración del archivo .env

Se copia el archivo de ejemplo y se genera la clave de la aplicación:

```
cp .env.example .env
php artisan key:generate
```

Luego, se editan los datos de conexión a la base de datos:

```
APP_NAME=JurassiDraft
APP_ENV=local
APP_KEY=base64:GENERADO
APP_DEBUG=true
APP_URL=http://127.0.0.1:8000

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=jurassidraft
DB_USERNAME=tu_usuario
DB_PASSWORD=tu_password

VITE_APP_NAME="JurassiDraft"
```

Instalación de dependencias

```
#Se instalan las dependencias de backend y frontend:
composer install
npm install
```

Base de datos

El sistema no utiliza migraciones ni seeders. La base de datos se crea e importa manualmente:

```
CREATE DATABASE jurassidraft CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci;

mysql -u tu_usuario -p jurassidraft < database/jurassidraft.sql
```

Ejecución del proyecto

El proyecto se levanta con un solo comando:

```
composer run dev
```

La aplicación queda disponible en http://127.0.0.1:8000, con recarga en vivo gracias a Vite.

Checklist de verificación

- Laravel responde correctamente (php artisan --version).
- La aplicación carga en http://localhost:8000.
- Tailwind refresca los estilos automáticamente.
- La base de datos importada funciona correctamente.

Presentar esquema relacional definitivo en 3FN

Claves y Dependencias Funcionales (DF) por tabla

Usuarios
PK: id
DF: id → {nombre, nickname, contrasena, rol, creado_en, deleted_at}
Candidata natural: nickname
DF: nickname → {id, …}

Partidas
PK: id
DF: id → {nombre, estado, ronda, turno, dado_restriccion, creador_id, creado_en}

Partida_jugadores
PK: id
Candidata natural: (partida_id, usuario_id)
DF: {partida_id, usuario_id} → {puntos_totales}
DF: id → {partida_id, usuario_id, puntos_totales}

Recintos_catalogo
PK: id
Candidata natural: clave
DF: id → {clave, descripcion, tipo_regla, max_dinos}
DF: clave → {id, descripcion, tipo_regla, max_dinos}

Dinosaurios_catalogo
PK: id
Candidata natural: nombre_corto
DF: id → {nombre_corto, categoria}
DF: nombre_corto → {id, categoria}

Colocaciones
PK: id
Candidata natural (operativa): (partida_id, usuario_id, ronda, turno)
DF: {partida_id, usuario_id, ronda, turno} → {recinto_id, tipo_dino, pts_obtenidos, creado_en}

Verificación por Formas Normales

1FN (Primera Forma Normal)
Todos los atributos son atómicos (sin listas ni grupos repetitivos).
Cada tabla representa una única entidad y las repeticiones (jugadas o catálogos) están correctamente separadas.
Conclusión: Todas las tablas cumplen 1FN.

2FN (Segunda Forma Normal)
Ningún atributo no clave depende solo de una parte de una clave compuesta.
Las tablas con PK simple (como id) no presentan dependencias parciales.
En las que poseen claves compuestas lógicas (como partida_jugadores o colocaciones), los atributos no clave dependen del conjunto completo de la clave.
Conclusión: Todas las tablas cumplen 2FN.

3FN (Tercera Forma Normal)
No existen dependencias transitivas entre atributos no-clave.

- usuarios: Los atributos dependen solo de id; rol es un valor enumerado y no causa dependencia transitiva.
- partidas: creador_id es una clave foránea hacia usuarios y no introduce dependencias indirectas.
- partida_jugadores: puntos_totales depende de la combinación (partida_id, usuario_id).
- recintos_catalogo y dinosaurios_catalogo: sus atributos dependen directamente de sus claves primarias.
- colocaciones: todos los atributos dependen solo de su clave primaria sin transitivas.

Conclusión: Todas las tablas cumplen 3FN.

Script de inserción de prueba

```sql
-- ========================================
-- Datos de prueba para tabla `dinosaurios_catalogo`
-- Base de datos: jurassidraft
-- ========================================

INSERT INTO dinosaurios_catalogo
(id, nombre_corto, categoria, puntos_base, nota_manual)
VALUES
(1, 'T-Rex', 'carnivoro', 5, 'DINO ROJO'),
(2, 'Stegosaurus', 'herbivoro', 1, 'DINO NARANJA'),
(3, 'Triceratops', 'herbivoro', 1, 'DINO AMARILLO'),
(4, 'Parasaurus', 'herbivoro', 1, 'DINO VERDE'),
(5, 'Brachiosaurus', 'herbivoro', 2, 'DINO CELESTE'),
(6, 'Spinosaurus', 'carnivoro', 2, 'DINO AZUL');

--  Reinicia el contador autoincremental
ALTER TABLE dinosaurios_catalogo AUTO_INCREMENT = 7;
```

```sql
-- ========================================
-- Datos de prueba para tabla `recintos_catalogo`
-- Base de datos: jurassidraft
-- ========================================

INSERT INTO recintos_catalogo
(id, clave, descripcion, tipo_regla, max_dinos, puntos_base,
puntos_condicional, nota_manual)
VALUES
(1, 'bosque_semeje', 'El bosque de la semejanza', 'semejanza', 6, 0, '0
puntos si dinosaurios iguales',
 'Debe ocuparse siempre de izquierda a derecha sin dejar huecos.'),
(2, 'prado_diferenci', 'El prado de las diferencias', 'variedad', 6, 0,
'0 puntos si dinosaurios distintos',
 'Debe ocuparse siempre de izquierda a derecha sin dejar huecos.'),
(3, 'isla_solitario', 'La isla solitaria', 'solitario', 1, 0, 'solo 1
dino en todo el parque 7 puntos',
 'Ganarás 7 puntos de victoria si ningún jugador o jugador tiene más de
1 dinosaurio.'),
(4, 'pradera_amor', 'La pradera del amor', 'parejas', 12, 0, 'por cada
```

```
pareja 5 puntos',
 'Conseguirás 5 puntos de victoria por cada pareja de dinosaurios
iguales.'),
(5, 'trio_frondoso', 'El trío frondoso', 'exactos', 3, 0, 'si hay 3
dinos, 7 puntos',
 'Ganarás 7 puntos de victoria si hay exactamente 3 dinosaurios.'),
(6, 'rey_selva', 'El rey de la selva', 'especial', 1, 0, 'si tenes mas
que el resto 7 puntos',
 'Ganarás 7 puntos de victoria si tenés más dinosaurios de este tipo que
cualquier otro jugador.'),
(7, 'rio', 'El río', 'rio', 12, 1, 'por cada dinosaurio que se tire al
río 1 punto',
 'Por cada dinosaurio que se encuentre en el Río valdrá 1 punto de
victoria.');

-- Reinicia el contador autoincremental
ALTER TABLE recintos_catalogo AUTO_INCREMENT = 8;
```

```
-- ============================================
-- Datos de prueba para tabla `usuarios`
-- Base de datos: jurassidraft
-- Contraseña: 'password'
-- ============================================

INSERT INTO usuarios (id, nombre, nickname, contrasena, rol, creado_en,
deleted_at) VALUES
(1, 'Admin Demo',    'admin_demo',
'$2y$12$/pYfVyWvF1xnZTBT2V8gnuvrmPL0xlw3OXYrgKOIImd1sFPOMasOq', 'admin',
'2025-10-16 10:21:58', NULL),
(2, 'Seba',          'seba_demo',
'$2y$12$/pYfVyWvF1xnZTBT2V8gnuvrmPL0xlw3OXYrgKOIImd1sFPOMasOq',
'jugador','2025-10-16 10:21:58', NULL),
(3, 'Tomi',          'tomi_demo',
'$2y$12$/pYfVyWvF1xnZTBT2V8gnuvrmPL0xlw3OXYrgKOIImd1sFPOMasOq',
'jugador','2025-10-16 10:21:58', NULL),
(4, 'Nacho',         'nacho_demo',
'$2y$12$/pYfVyWvF1xnZTBT2V8gnuvrmPL0xlw3OXYrgKOIImd1sFPOMasOq',
'jugador','2025-10-16 10:21:58', NULL),
(5, 'Joaco',         'joaco_demo',
'$2y$12$/pYfVyWvF1xnZTBT2V8gnuvrmPL0xlw3OXYrgKOIImd1sFPOMasOq',
'jugador','2025-10-16 10:21:58', NULL),

-- Reinicia el contador autoincremental
ALTER TABLE usuarios AUTO_INCREMENT = 8;
```

Desarrollar script DDL final con todas las estructuras, relaciones y restricciones

```
SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
START TRANSACTION;
SET time_zone = "+00:00";

DROP DATABASE IF EXISTS draftosaurus;
CREATE DATABASE IF NOT EXISTS draftosaurus
  CHARACTER SET = utf8mb4
  COLLATE = utf8mb4_unicode_ci;
USE draftosaurus;

CREATE TABLE roles (
  rol ENUM('jugador','admin') NOT NULL,
  descripcion VARCHAR(100) DEFAULT NULL,
  PRIMARY KEY (rol)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE usuarios (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  nickname VARCHAR(50) NOT NULL,
  contrasena VARCHAR(255) NOT NULL,
  rol ENUM('jugador','admin') NOT NULL DEFAULT 'jugador',
  creado_en TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  deleted_at TIMESTAMP NULL DEFAULT NULL,
  UNIQUE KEY uq_usuarios_nickname (nickname),
  KEY idx_usuarios_rol (rol),
  CONSTRAINT fk_usuarios_roles FOREIGN KEY (rol)
      REFERENCES roles (rol)
      ON UPDATE CASCADE
      ON DELETE RESTRICT
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE partidas (
  id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(120) NOT NULL,
  estado ENUM('config','en_curso','cerrada') NOT NULL DEFAULT 'config',
  ronda TINYINT UNSIGNED NOT NULL DEFAULT 1,
  turno TINYINT UNSIGNED NOT NULL DEFAULT 1,
  dado_restriccion VARCHAR(50) DEFAULT NULL,
  creador_id INT NOT NULL,
```

```
  creado_en TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  KEY idx_partidas_estado (estado),
  KEY idx_partidas_ronda_turno (ronda, turno),
  CONSTRAINT fk_partidas_creador FOREIGN KEY (creador_id)
      REFERENCES usuarios (id)
      ON UPDATE CASCADE
      ON DELETE RESTRICT
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE partida_jugadores (
  id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  partida_id BIGINT UNSIGNED NOT NULL,
  usuario_id INT NOT NULL,
  orden_mesa TINYINT UNSIGNED NOT NULL,
  puntos_totales INT NOT NULL DEFAULT 0,
  UNIQUE KEY uq_pj_partida_usuario (partida_id, usuario_id),
  UNIQUE KEY uq_pj_partida_orden (partida_id, orden_mesa),
  KEY idx_pj_partida (partida_id),
  KEY idx_pj_usuario (usuario_id),
  CONSTRAINT fk_pj_partida FOREIGN KEY (partida_id)
      REFERENCES partidas (id)
      ON UPDATE CASCADE
      ON DELETE CASCADE,
  CONSTRAINT fk_pj_usuario FOREIGN KEY (usuario_id)
      REFERENCES usuarios (id)
      ON UPDATE CASCADE
      ON DELETE RESTRICT
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE recintos_catalogo (
  id TINYINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  clave VARCHAR(50) NOT NULL,
  descripcion VARCHAR(200) DEFAULT NULL,
  UNIQUE KEY uq_recintos_clave (clave),
  KEY idx_recintos_clave (clave)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE dinosaurios_catalogo (
  id SMALLINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nombre_corto VARCHAR(50) NOT NULL,
  categoria VARCHAR(50) DEFAULT NULL,
  UNIQUE KEY uq_dino_nombre_corto (nombre_corto),
  KEY idx_dino_nombre_corto (nombre_corto),
  KEY idx_dino_categoria (categoria)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

```sql
CREATE TABLE recintos_tablero (
  id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  partida_id BIGINT UNSIGNED NOT NULL,
  usuario_id INT NOT NULL,
  recinto_id TINYINT UNSIGNED NOT NULL,
  UNIQUE KEY uq_rt_partida_usuario_recinto (partida_id, usuario_id,
recinto_id),
  KEY idx_rt_partida (partida_id),
  KEY idx_rt_usuario (usuario_id),
  KEY idx_rt_recinto (recinto_id),
  CONSTRAINT fk_rt_partida FOREIGN KEY (partida_id)
      REFERENCES partidas (id)
      ON UPDATE CASCADE
      ON DELETE CASCADE,
  CONSTRAINT fk_rt_usuario FOREIGN KEY (usuario_id)
      REFERENCES usuarios (id)
      ON UPDATE CASCADE
      ON DELETE RESTRICT,
  CONSTRAINT fk_rt_recinto FOREIGN KEY (recinto_id)
      REFERENCES recintos_catalogo (id)
      ON UPDATE CASCADE
      ON DELETE RESTRICT
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE puntos_recinto (
  id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  partida_id BIGINT UNSIGNED NOT NULL,
  usuario_id INT NOT NULL,
  recinto_id TINYINT UNSIGNED NOT NULL,
  puntos INT NOT NULL DEFAULT 0,
  actualizado_en TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  UNIQUE KEY uq_pr_partida_usuario_recinto (partida_id, usuario_id,
recinto_id),
  KEY idx_pr_partida_usuario (partida_id, usuario_id),
  CONSTRAINT fk_pr_tablero FOREIGN KEY (partida_id, usuario_id,
recinto_id)
      REFERENCES recintos_tablero (partida_id, usuario_id, recinto_id)
      ON UPDATE CASCADE
      ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE colocaciones (
  id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  partida_id BIGINT UNSIGNED NOT NULL,
  usuario_id INT NOT NULL,
```

```
  ronda TINYINT UNSIGNED NOT NULL,
  turno TINYINT UNSIGNED NOT NULL,
  recinto_id TINYINT UNSIGNED NOT NULL,
  tipo_dino SMALLINT UNSIGNED NOT NULL,
  valido TINYINT(1) NOT NULL DEFAULT 1,
  motivo_invalidez VARCHAR(200) DEFAULT NULL,
  creado_en TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  KEY idx_col_partida_usuario (partida_id, usuario_id),
  KEY idx_col_partida_ronda_turno (partida_id, ronda, turno),
  KEY idx_col_recinto (recinto_id),
  KEY idx_col_dino (tipo_dino),
  CONSTRAINT fk_col_partida FOREIGN KEY (partida_id)
      REFERENCES partidas (id)
      ON UPDATE CASCADE
      ON DELETE CASCADE,
  CONSTRAINT fk_col_usuario FOREIGN KEY (usuario_id)
      REFERENCES usuarios (id)
      ON UPDATE CASCADE
      ON DELETE RESTRICT,
  CONSTRAINT fk_col_recinto FOREIGN KEY (recinto_id)
      REFERENCES recintos_catalogo (id)
      ON UPDATE CASCADE
      ON DELETE RESTRICT,
  CONSTRAINT fk_col_dino FOREIGN KEY (tipo_dino)
      REFERENCES dinosaurios_catalogo (id)
      ON UPDATE CASCADE
      ON DELETE RESTRICT
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE OR REPLACE VIEW v_puntos_totales_por_recinto AS
SELECT
  partida_id,
  usuario_id,
  SUM(puntos) AS puntos_sumados
FROM puntos_recinto
GROUP BY partida_id, usuario_id;

ALTER TABLE usuarios ADD INDEX idx_usuarios_nombre (nombre);
ALTER TABLE partidas ADD INDEX idx_partidas_nombre (nombre);

COMMIT;
```

Políticas de Mantenimiento

Introducción

El mantenimiento de un sistema no se limita a corregir errores o actualizar versiones; también incluye la planificación y ejecución de estrategias que aseguren la continuidad operativa, la integridad de los datos y la trazabilidad del código.
En este proyecto se aplican **dos políticas fundamentales**:

1. **Rutinas de respaldo (backups y automatización del sistema).**

2. **Mantenimiento evolutivo y control de versiones mediante GitFlow y releases estables.**

Rutinas de Backup y Automatización del Sistema Operativo

Un **backup** o copia de seguridad consiste en duplicar los datos importantes del sistema para poder restaurarlos ante una pérdida, corrupción o fallo. Las **rutinas de backup** definen cómo, cuándo y dónde se guardan esas copias, asegurando que la información pueda ser recuperada de manera eficiente.

Estrategia Documentada

Objetivo: Garantizar la integridad de los datos y la disponibilidad del sistema ante cualquier eventualidad.
Frecuencia y tipos:

- **Diarios (incrementales):** se respalda la base de datos y los archivos que hayan cambiado desde el último backup completo.

- **Semanales (completos):** se realiza una copia íntegra del sistema (archivos web + base de datos).

- **Mensuales (externos):** se sincronizan las copias en un servidor remoto o NAS.

Implementación Técnica

- **Script principal:** backup.sh – genera backups automáticos con registro en logs.
- **Sincronización remota:** sync_backups.sh usa rsync para enviar copias a otro servidor o disco externo.
- **Automatización:** ambas rutinas están programadas mediante cron para ejecutarse en horarios de baja carga (03:00 y 04:00).
- **Interfaz manual:** se incluye un menú con dialog que permite:
    - Ejecutar un backup instantáneo.
    - Consultar historiales.
    - Configurar o desactivar tareas programadas.

Esta estructura cumple con la regla **3-2-1 de respaldo**, manteniendo tres copias en dos medios distintos y al menos una fuera del entorno local.

Estrategia de Versionado y Mantenimiento con GitFlow

Además del mantenimiento físico y de datos, se establece una política de **mantenimiento lógico del código** mediante el modelo **GitFlow**.
Este modelo estructura el ciclo de vida del desarrollo en ramas bien definidas:

Flujo de trabajo

- **main**: rama estable, contiene solo versiones probadas y liberadas.
- **develop**: rama principal de integración, donde se fusionan los cambios validados.
- **feature/***: ramas para nuevas funcionalidades.
- **fix/***: ramas para correcciones puntuales.
- **release/***: ramas intermedias que preparan una versión estable antes de pasar a main.

Política de Releases

Cada **versión estable** del sistema (una vez validada y testeada) genera una **release oficial**:

- Se etiqueta con un número semántico (por ejemplo, v1.0.0, v1.1.0).
- Incluye un **changelog** con los cambios y mejoras.
- Se asocia a un punto de restauración en caso de regresión.

Beneficios

- Permite mantener trazabilidad completa del proyecto.
- Facilita el rollback ante errores graves.
- Vincula el código estable con los **backups del sistema**, garantizando coherencia entre versiones de software y datos respaldados.

Hoja Testigo

Los integrantes del grupo certificamos que la presente carpeta contiene todo el material solicitado para la evaluación del proyecto en la asignatura Programación Full Stack, cumpliendo con los lineamientos establecidos por la institución.

FIANZA, IGNACIO

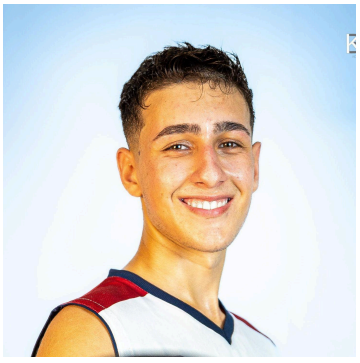
COORDINADOR

BENÍTEZ, SEBASTIÁN

SUBCOORDINADOR

FLEITAS, JOAQUÍN

INTEGRANTE 1



PAZ, TOMÁS

INTEGRANTE 2