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

15/9/2025

SEGUNDA ENTREGA

ÍNDICE

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 (Bootstrap y complementarios)

Bootstrap

Para el desarrollo frontend del proyecto se seleccionó Bootstrap como framework principal. Esta decisión se basó en su facilidad de uso, su integración rápida en cualquier entorno web, y la extensa documentación que lo acompaña.

Bootstrap ofrece un sistema de grillas responsive, lo que permite adaptar el diseño a distintos tamaños de pantalla sin necesidad de escribir reglas CSS personalizadas. También proporciona una gran variedad de componentes predefinidos (botones, formularios, tarjetas, barras de navegación, entre otros), que simplifican la construcción de interfaces visuales modernas y coherentes.

Además, incluye clases utilitarias que facilitan la personalización del diseño sin salir del HTML, lo que acelera el desarrollo y reduce la dependencia de archivos CSS personalizados.

A nivel técnico, Bootstrap fue una buena elección porque permite enfocarnos más en la lógica y funcionalidad del sistema, sin dejar de lado una apariencia profesional y ordenada.

Hasta el momento, no se han utilizado frameworks o herramientas complementarias, ya que Bootstrap cubre adecuadamente todas las necesidades visuales e interactivas del proyecto en esta etapa.

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.

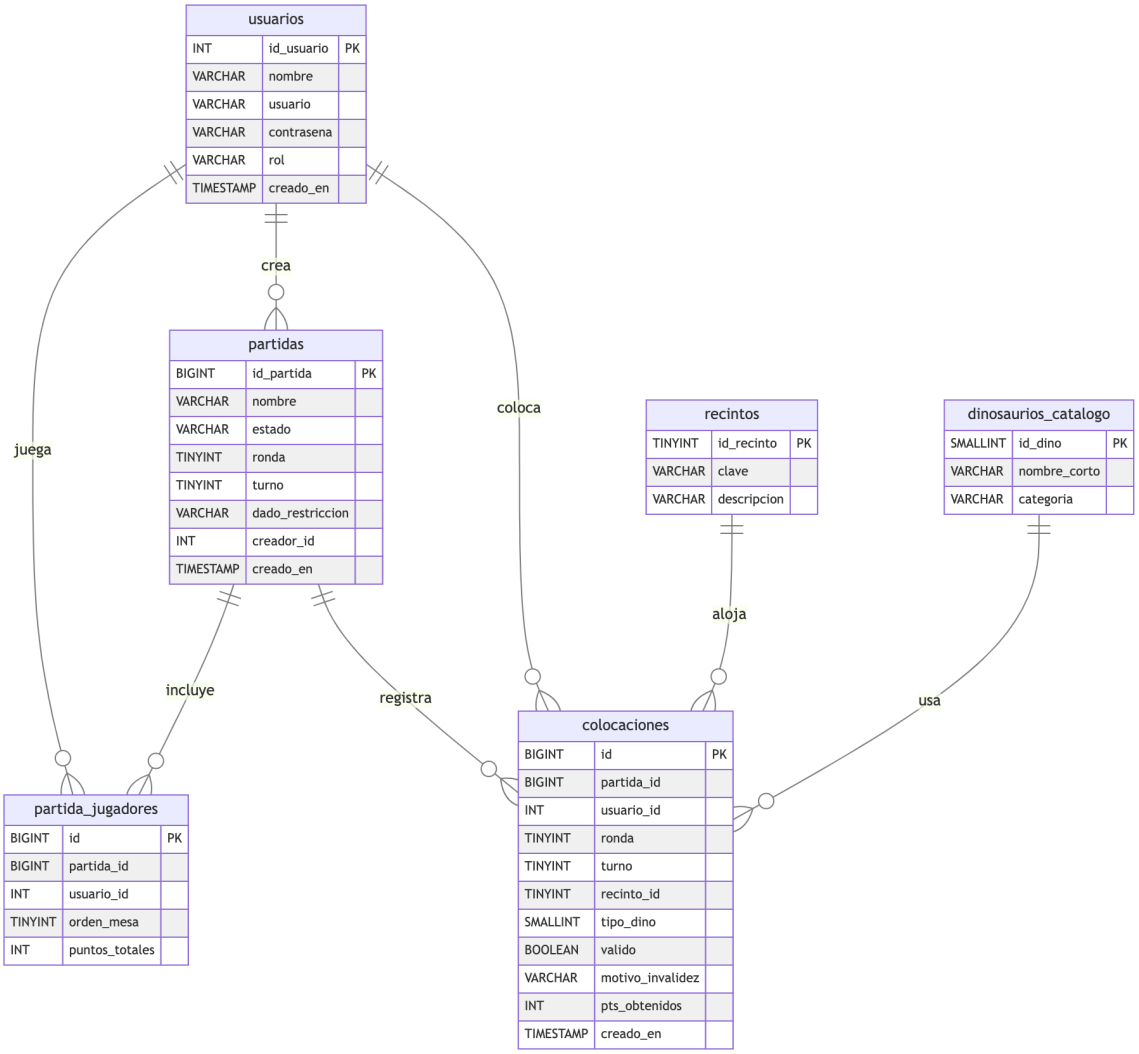
En cuanto a las extensiones que instalamos y usamos en este proyecto, cada una cumple un propósito específico:

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 (primera versión) que refleje fielmente la estructura del juego



Incluir todas las entidades principales con sus atributos y relaciones

ENTIDADES PRINCIPALES (MODO SEGUIMIENTO)

usuarios

- id_usuario (PK)
- nombre
- usuario
- contrasena
- rol [admin|jugador]
- creado_en

partidas

- id_partida (PK)
- nombre
- estado [config|en_curso|cerrada]
- ronda [1..2]
- turno [1..6]
- dado_restriccion (opcional, clave de la restricción activa)
- creador_id (FK → usuarios.id_usuario)
- creado_en

partida_jugadores

- id (PK)
- partida_id (FK → partidas.id_partida)
- usuario_id (FK → usuarios.id_usuario)
- orden_mesa
- puntos_totales

recintos

- id_recinto (PK)
- clave (única, ej: bosque, rio, prado)
- descripcion

dinosaurios_catalogo

- id_dino (PK)
- nombre_corto (único, ej: TRex, Stego)
- categoria (opcional)

colocaciones

- id (PK)
- partida_id (FK → partidas.id_partida)
- usuario_id (FK → usuarios.id_usuario)
- ronda [1..2]
- turno [1..6]
- recinto_id (FK → recintos.id_recinto)
- tipo_dino (FK → dinosaurios_catalogo.id_dino)
- valido [0/1]
- motivo_invalidez (opcional)
- pts_obtenidos
- creado_en

RELACIONES (MODO SEGUIMIENTO)

usuarios (1) — crea — (N) partidas

- partidas.creador_id referencia a usuarios.id_usuario

usuarios (1) — juega — (N) partida_jugadores

- partida_jugadores.usuario_id referencia a usuarios.id_usuario

partidas (1) — incluye — (N) partida_jugadores

- partida_jugadores.partida_id referencia a partidas.id_partida

partidas (1) — registra — (N) colocaciones

- colocaciones.partida_id referencia a partidas.id_partida

usuarios (1) — coloca — (N) colocaciones

- colocaciones.usuario_id referencia a usuarios.id_usuario

recintos (1) — aloja — (N) colocaciones

- colocaciones.recinto_id referencia a recintos.id_recinto

dinosaurios_catalogo (1) — usa — (N) colocaciones

- colocaciones.tipo_dino referencia a dinosaurios_catalogo.id_dino

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

```sql
-- CREATE DATABASE draftosaurus CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci;

-- 0) USUARIOS
CREATE TABLE IF NOT EXISTS usuarios (
  id_usuario   INT(11) NOT NULL AUTO_INCREMENT,
  nombre       VARCHAR(100) NOT NULL,
  usuario      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,
  PRIMARY KEY (id_usuario),
  UNIQUE KEY usuario (usuario)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;

-- 1) PARTIDAS
CREATE TABLE IF NOT EXISTS partidas (
  id_partida       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,    -- 1..2
  turno            TINYINT UNSIGNED NOT NULL DEFAULT 1,    -- 1..6
  dado_restriccion VARCHAR(50) NULL,
  creador_id       INT(11) NOT NULL,                       -- FK
usuarios.id_usuario
  creado_en        TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT fk_partidas_creador
    FOREIGN KEY (creador_id) REFERENCES usuarios(id_usuario)
    ON UPDATE CASCADE ON DELETE RESTRICT
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;

CREATE INDEX idx_partidas_estado ON partidas (estado);
CREATE INDEX idx_partidas_ronda_turno ON partidas (ronda, turno);
```

```
-- 2) PARTICIPANTES (relación partidas-usuarios)
CREATE TABLE IF NOT EXISTS partida_jugadores (
  id             BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  partida_id     BIGINT UNSIGNED NOT NULL,
  usuario_id     INT(11) NOT NULL,
  orden_mesa     TINYINT UNSIGNED NOT NULL,
  puntos_totales INT NOT NULL DEFAULT 0,
  CONSTRAINT fk_pj_partida
    FOREIGN KEY (partida_id) REFERENCES partidas(id_partida)
    ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT fk_pj_usuario
    FOREIGN KEY (usuario_id) REFERENCES usuarios(id_usuario)
    ON UPDATE CASCADE ON DELETE RESTRICT,
  CONSTRAINT uq_pj_partida_usuario UNIQUE (partida_id,
usuario_id),
  CONSTRAINT uq_pj_partida_orden  UNIQUE (partida_id, orden_mesa)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;

-- 3) CATÁLOGO DE RECINTOS
CREATE TABLE IF NOT EXISTS recintos (
  id_recinto  TINYINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  clave       VARCHAR(50) NOT NULL UNIQUE,
  descripcion VARCHAR(200) NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;

-- 4) CATÁLOGO DE DINOSAURIOS
CREATE TABLE IF NOT EXISTS dinosaurios_catalogo (
  id_dino      SMALLINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nombre_corto VARCHAR(50) NOT NULL UNIQUE,
  categoria    VARCHAR(50) NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;

-- 5) COLOCACIONES (jugadas)
CREATE TABLE IF NOT EXISTS colocaciones (
  id               BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  partida_id       BIGINT UNSIGNED NOT NULL,
  usuario_id       INT(11) NOT NULL,
  ronda            TINYINT UNSIGNED NOT NULL,    -- 1..2
```

```
  turno            TINYINT UNSIGNED NOT NULL,   -- 1..6
  recinto_id       TINYINT UNSIGNED NOT NULL,
  tipo_dino        SMALLINT UNSIGNED NOT NULL,
  valido           TINYINT(1) NOT NULL DEFAULT 1,
  motivo_invalidez VARCHAR(200) NULL,
  pts_obtenidos    INT NOT NULL DEFAULT 0,
  creado_en        TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT fk_col_partida
    FOREIGN KEY (partida_id) REFERENCES partidas(id_partida)
    ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT fk_col_usuario
    FOREIGN KEY (usuario_id) REFERENCES usuarios(id_usuario)
    ON UPDATE CASCADE ON DELETE RESTRICT,
  CONSTRAINT fk_col_recinto
    FOREIGN KEY (recinto_id) REFERENCES recintos(id_recinto)
    ON UPDATE CASCADE ON DELETE RESTRICT,
  CONSTRAINT fk_col_dino
    FOREIGN KEY (tipo_dino) REFERENCES
dinosaurios_catalogo(id_dino)
    ON UPDATE CASCADE ON DELETE RESTRICT
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_unicode_ci;

CREATE INDEX idx_col_part_usr ON colocaciones (partida_id,
usuario_id);
CREATE INDEX idx_col_turno    ON colocaciones (partida_id, ronda,
turno);
CREATE INDEX idx_col_recinto  ON colocaciones (recinto_id);
```

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

1) Claves y dependencias funcionales (DF) por tabla

usuarios

- PK: id_usuario
- DF básicas: id_usuario → {nombre, usuario, contrasena, rol, creado_en}; usuario → {id_usuario, …} (único)

partidas

- PK: id_partida
- DF: id_partida → {nombre, estado, ronda, turno, dado_restriccion, creador_id, creado_en}

partida_jugadores

- PK: id (surrogada)
- Candidata natural: (partida_id, usuario_id)
- DF: {partida_id, usuario_id} → {orden_mesa, puntos_totales}; id → {partida_id, usuario_id, orden_mesa, puntos_totales}

recintos

- PK: id_recinto
- Candidata natural: clave
- DF: id_recinto → {clave, descripcion}; clave → {id_recinto, descripcion}

dinosaurios_catalogo

- PK: id_dino
- Candidata natural: nombre_corto
- DF: id_dino → {nombre_corto, categoria}; nombre_corto → {id_dino, categoria}

colocaciones

- PK: id (surrogada)
- Candidata natural (operativa): (partida_id, usuario_id, ronda, turno) si se fuerza “1 colocación por jugador y turno”
- DF: {partida_id, usuario_id, ronda, turno} → {recinto_id, tipo_dino, valido, motivo_invalidez, pts_obtenidos, creado_en}; id → todo el registro

Notas:

- Los catálogos (recintos, dinosaurios_catalogo) tienen clave natural única además de la llave técnica.
- En colocaciones permitimos pts_obtenidos como atributo derivado almacenado por performance; su DF queda igualmente determinada por la PK (o por la clave natural operativa).

2) Verificación por formas normales

1FN (Primera Forma Normal)

- Todos los atributos son atómicos (sin listas/repeticiones en columnas).
- No hay grupos repetitivos: las repeticiones de eventos (jugadas) van en colocaciones.

  Todas las tablas cumplen 1FN.

2FN (Segunda Forma Normal)

Criterio: ningún atributo no clave depende de una parte de una clave compuesta.

- Tablas con PK simple (surrogada): usuarios, partidas, recintos, dinosaurios_catalogo → no pueden tener dependencias parciales.
- partida_jugadores: si usamos la candidata (partida_id, usuario_id) como clave lógica, los no-clave {orden_mesa, puntos_totales} dependen de ambas columnas en conjunto (no de una sola). Con PK técnica id, tampoco hay parcialidad.
- colocaciones: análogo; los no-clave dependen de la combinación (partida_id, usuario_id, ronda, turno) o de id.
  ✔️ Todas las tablas cumplen 2FN.

3FN (Tercera Forma Normal)
Criterio: no debe haber dependencias transitivas de atributos no-clave respecto de la clave.

- usuarios: nombre/usuario/rol/contrasena/creado_en dependen directamente de id_usuario; no hay atributos que determinen a otros no-clave.
- partidas: nombre/estado/ronda/turno/dado_restriccion/creado_en dependen de id_partida; creador_id es FK (no copiamos nombre del creador). Sin transitivas.
- partida_jugadores: con PK id, los no-clave dependen de id. Con clave candidata (partida_id, usuario_id), {orden_mesa, puntos_totales} dependen de la pareja completa. No guardamos datos del usuario ni de la partida aquí (evita transitivas).
- recintos y dinosaurios_catalogo: descripcion/categoria dependen de su PK; la clave/nombre_corto es única, pero no determina otro no-clave distinto de la propia PK (no hay transitivas).
- colocaciones: recinto_id, tipo_dino, valido, motivo_invalidez, pts_obtenidos, creado_en dependen de la clave (id, o la compuesta operativa). No duplicamos datos de recintos, dinos, usuarios ni partidas, solo llaves.

  Todas las tablas cumplen 3FN.

3) Observaciones y decisiones de diseño

- Llaves naturales y técnicas: conservamos llaves técnicas (AUTO_INCREMENT) para simplicidad en relaciones, pero documentamos candidatas naturales donde corresponden:
  - partida_jugadores: (partida_id, usuario_id)
  - colocaciones: (partida_id, usuario_id, ronda, turno) si se fuerza 1 jugada por turno
  - recintos.clave y dinosaurios_catalogo.nombre_corto como claves naturales únicas

- Atributos derivados: pts_obtenidos en colocaciones puede recalcularse; se almacena para auditoría y rendimiento. Se asegura consistencia en capa de negocio (service de puntuación).
- Integridad referencial: FKs hacia usuarios, partidas, recintos, dinosaurios_catalogo. Eliminación en cascada solo donde el concepto lo admite (partida → borra sus colocaciones y partida_jugadores).
- Unicidades lógicas:
  • partida_jugadores (partida_id, usuario_id) evita duplicar inscripciones
  • partida_jugadores (partida_id, orden_mesa) evita colisiones en la mesa
  • (Opcional) colocaciones (partida_id, usuario_id, ronda, turno) si queremos forzar 1 colocación por turno
- Índices de consulta:
  • colocaciones(partida_id, ronda, turno) para validar y listar por turno
  • colocaciones(partida_id, usuario_id) para tablero individual
  • partidas(estado) y (ronda, turno) para navegaciones del juego
  • índices implícitos/PK en catálogos

4) Conclusión
El esquema propuesto está en Tercera Forma Normal (3FN):

- No hay atributos multivaluados ni grupos repetitivos (1FN).
- No hay dependencias parciales respecto de claves compuestas (2FN).
- No hay dependencias transitivas entre atributos no-clave (3FN).
  Además, se documentan llaves naturales, restricciones de unicidad e índices prácticos para las operaciones del juego.

Identificar y documentar claramente claves primarias y foráneas

usuarios

PK: id_usuario

FK: ninguna

partidas

PK: id_partida

FK: creador_id → usuarios.id_usuario

Descripción FK: identifica al usuario que crea la partida. Mantiene integridad de autoría.

Reglas: ON UPDATE CASCADE, ON DELETE RESTRICT (no se permite borrar un usuario si tiene partidas creadas).

partida_jugadores
PK: id
FK: partida_id → partidas.id_partida
FK: usuario_id → usuarios.id_usuario
Descripción FK: vincula cada fila con su partida y con el usuario participante.
Reglas: partida_id ON UPDATE CASCADE, ON DELETE CASCADE (al borrar la partida, se eliminan sus participantes). usuario_id ON UPDATE CASCADE, ON DELETE RESTRICT (no se permite borrar un usuario si está inscripto en partidas).

recintos
PK: id_recinto
FK: ninguna

dinosaurios_catalogo
PK: id_dino
FK: ninguna

colocaciones
PK: id
FK: partida_id → partidas.id_partida
FK: usuario_id → usuarios.id_usuario
FK: recinto_id → recintos.id_recinto
FK: tipo_dino → dinosaurios_catalogo.id_dino
Descripción FK: cada colocación se asocia a su partida, al jugador que la realizó, al recinto donde se colocó y al tipo de dinosaurio utilizado.
Reglas: partida_id ON UPDATE CASCADE, ON DELETE CASCADE (al borrar la partida, se borran sus colocaciones). usuario_id ON UPDATE CASCADE, ON DELETE RESTRICT (no se permite borrar un usuario con jugadas registradas). recinto_id y tipo_dino ON UPDATE CASCADE, ON DELETE RESTRICT (no se permite borrar catálogos en uso).

Notas de unicidad relacionadas (para completar el contexto):
recintos.clave es única.
dinosaurios_catalogo.nombre_corto es único.
partida_jugadores tiene unicidad en (partida_id, usuario_id) para evitar

inscripciones duplicadas y en (partida_id, orden_mesa) para evitar posiciones repetidas en la misma partida.

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 en el dump (campos, nombres y tipos), mapeando contrasena como password en el modelo de Laravel.
   Justificación: evita cambiar datos ya en producción/pruebas; reduce riesgo de incompatibilidades; permite integrar autenticación Laravel con mínima fricción.
2. Modelo orientado a eventos de juego
   Decisión: separar en entidades partidas, partida_jugadores, recintos, dinosaurios_catalogo y colocaciones.
   Justificación: cada entidad representa un concepto único; colocaciones registra cada jugada sin duplicar datos maestros; favorece auditoría y consultas por partida/jugador.
3. Normalización a 3FN
   Decisión: catálogos (recintos, dinosaurios_catalogo) propios; relación N–N resuelta con partida_jugadores; sin atributos multivaluados ni dependencias transitivas.
   Justificación: elimina redundancias y anomalías de actualización; mejora integridad y claridad del modelo.
4. Claves técnicas + unicidades naturales
   Decisión: usar claves primarias AUTO_INCREMENT en todas las tablas y unicidades en claves naturales (recintos.clave, dinosaurios_catalogo.nombre_corto) y lógicas (partida_jugadores.partida_id+usuario_id).
   Justificación: claves técnicas simplifican relaciones y FKs; las únicas naturales preservan consistencia de catálogos y reglas de negocio.

5. pk candidata operativa en colocaciones
   Decisión: permitir clave técnica id pero documentar (partida_id, usuario_id, ronda, turno) como candidata si se restringe 1 jugada por turno.
   Justificación: la clave técnica simplifica; la candidata refleja la regla del dominio y puede activarse como UNIQUE cuando se implemente “una colocación por turno”.
6. Campo derivado pts_obtenidos
   Decisión: almacenar pts_obtenidos en colocaciones aun siendo calculable.
   Justificación: mejora rendimiento para listados y resultados; se garantiza consistencia recalculando en cierres o validándolo en services.
7. Enums controlados en partidas.estado
   Decisión: usar estado ENUM('config','en_curso','cerrada') y pequeños tipos numéricos para ronda/turno (TINYINT).
   Justificación: ENUM hace explícitos los estados válidos y evita valores inválidos; tipos pequeños reducen espacio y mejoran cache de índices.
8. Charset/collation
   Decisión: usar utf8mb4_unicode_ci en toda la base.
   Justificación: compatibilidad total con Unicode y emojis, coherencia entre tablas y búsquedas.

Integridad y reglas de borrado/actualización

9. Integridad referencial estricta

Decisión: FKs con ON UPDATE CASCADE en todas, ON DELETE CASCADE donde el concepto lo admite (partidas → colocaciones/partida_jugadores) y ON DELETE RESTRICT en referencias a usuarios y catálogos.
Justificación: evitar huérfanos; proteger catálogos/usuarios en uso; limpieza automática al eliminar partidas completas.

10. Unicidades lógicas en partida_jugadores
    Decisión: UNIQUE(partida_id, usuario_id) y UNIQUE(partida_id, orden_mesa).
    Justificación: impide inscripciones duplicadas y colisiones de posición en la misma partida.

11. Índices de consulta

    Decisión: índices por estado, ronda+turno en partidas; partida+usuario y partida+ronda+turno en colocaciones; índices implícitos en FKs.

    Justificación: consultas frecuentes del flujo (tablero por jugador, validación por turno, resultados) quedan cubiertas y escalan mejor.

Arquitectura y backend

12) Laravel como framework

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

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

13. Capa de negocio en services

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

    Justificación: controladores livianos, testeo aislado de reglas, facilidad de cambio cuando evolucione el juego.

14. Autenticación con tabla personalizada

    Decisión: Auth::attempt con usuario y contrasena; mutators en el modelo para mapear password ⇄ contrasena.

    Justificación: integra el login de Laravel sin alterar la tabla; mantiene hashing seguro (bcrypt) y evita migraciones innecesarias.

15. Middleware por rol

    Decisión: middleware EsAdmin por clase en rutas para proteger /admin.

    Justificación: control explícito y simple; independencia del alias del Kernel; seguridad clara por rol.

Frontend y UX

16) Bootstrap 5.3 como base

Decisión: usar Bootstrap 5.3 para layout, grid y componentes; landing pública y vistas admin con estilos consistentes.

Justificación: responsivo rápido, coherencia visual y menor costo de CSS personalizado.

17. División por modos (Seguimiento primero)
    Decisión: entregar primero Modo Seguimiento (registro y validación de colocaciones y puntaje) y dejar Modo Digitalizado como extensión.
    Justificación: cumple el alcance base, reduce complejidad inicial y habilita recorrer el flujo completo de puntuación antes de integrar manos, paso y dado.

Seguridad y buenas prácticas

18) Consultas preparadas y sanitización

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

Justificación: evita SQL injection y reduce superficie de errores de entrada.

19. Sesiones en archivo en desarrollo
    Decisión: SESSION_DRIVER=file para evitar tabla de sesiones en dev; opción a DB/Redis en producción si se requiere.
    Justificación: simplicidad en entorno local; camino claro a mayor robustez si el tráfico lo exige.
20. Autorización por pertenencia
    Decisión: validar que el usuario pertenezca a la partida para acceder a sus datos.
    Justificación: privacidad entre jugadores y contención de alcance por recurso.

Extensión prevista (modo digitalizado) y decisiones

21) Estado de mano y bolsa

Decisión: usar manos y bolsas; contenido_json y stock_json para colecciones variables de dinos por turno.

Justificación: la mano/bolsa es naturalmente una lista; JSON en MySQL 8 simplifica

el estado por turno sin inflar el modelo con múltiples tablas auxiliares; se valida desde la capa de negocio.

22. Historial de restricciones (dado)
    Decisión: tabla restricciones con una entrada por (partida, ronda, turno).
    Justificación: auditable y reproducible; desacopla el estado actual del historial.

23. Validación cruzada app + BD
    Decisión: restricciones no estructurales (dados/recintos/una jugada por turno) se aplican en services y pueden reforzarse con UNIQUE en colocaciones cuando corresponda.
    Justificación: mantiene flexibilidad de reglas del juego a nivel app, con posibilidad de endurecer en BD si se fija la mecánica.

Despliegue y entorno

24) Fedora Server + Apache 2.4 + PHP 8.3 + MySQL 8

Decisión: stack Linux estándar, soportado, con buen rendimiento y documentación amplia.

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

25. Collation y zona horaria
    Decisión: utf8mb4_unicode_ci y timezone America/Montevideo en PHP.
    Justificación: soporte completo de caracteres y coherencia de timestamps con la operación local.

Internacionalización

26) ES/EN con archivos de idioma

Decisión: recursos en resources/lang/es y resources/lang/en y helpers de traducción en vistas.

Justificación: requisito del proyecto, desacople de textos, escalable a más idiomas.

Trade-offs explícitos

27) ENUM vs tabla de estados

Decisión: usar ENUM para partidas.estado en vez de tabla estados.

Justificación: set de valores acotado y estable, consulta más simple; si los estados cambian, se migra a tabla de soporte.

28. pts_obtenidos materializado

    Decisión: almacenar puntos por jugada.

    Justificación: performance y auditoría; costo de sincronización asumido y controlado por la capa de negocio.

29. JSON en modo digitalizado

    Decisión: usar JSON para manos/bolsa en lugar de tablas “mano_items”.

    Justificación: reduce joins y complejidad; MySQL 8 soporta índices funcionales si luego se necesitan; 3FN estricta se reserva para Seguimiento.

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

1. Estados de partida (config, en_curso, cerrada)
   Base de datos: columna estado ENUM en partidas.
   Backend: PartidaService valida estado antes de cualquier acción; políticas de acceso por estado.
   Frontend: deshabilitar acciones no válidas según estado actual.

2. Transiciones de estado (config → en_curso → cerrada)
   Base de datos: no se fuerza en SQL; se registra estado final.
   Backend: PartidaService impone la máquina de estados (no permite retrocesos salvo admin).
   Frontend: botones/acciones visibles solo si la transición es válida.

3. Rondas y turnos (2 rondas, 6 turnos)
   Base de datos: columnas ronda y turno en partidas; índices para consultas.
   Backend: PartidaService controla avance; bloquea jugadas fuera del turno/ronda.
   Frontend: UI muestra ronda/turno activos y bloquea acciones fuera de fase.

4. Participantes registrados
   Base de datos: partida_jugadores con FK a partidas y usuarios, UNIQUE(partida_id, usuario_id).
   Backend: Policies verifican pertenencia antes de permitir registrar jugadas.
   Frontend: ocultar panel de jugada a quien no pertenece.

5. Orden de mesa fijo
   Base de datos: columna orden_mesa con UNIQUE(partida_id, orden_mesa).
   Backend: PartidaService asigna orden al inscribir; no permite cambios con partida en_curso.
   Frontend: render de lista de jugadores según orden_mesa.

6. Una jugada por turno
   Base de datos: UNIQUE opcional en colocaciones (partida_id, usuario_id, ronda, turno) si se decide endurecer en BD.
   Backend: ReglasService valida que no exista ya una colocación para esa clave; transacción al guardar.
   Frontend: deshabilita el botón de confirmar jugada tras una colocación.

7. Unicidad por turno (no duplicar colocación)
   Base de datos: ver punto 6 (UNIQUE compuesto).
   Backend: ver punto 6 (consulta previa + transacción).

Frontend: prevención de doble envío (disable al confirmar).

8. Restricción por dado (recintos permitidos por turno)
   Base de datos: campo dado_restriccion en partidas (o tabla restricciones si se versiona por turno).
   Backend: DadoService fija la restricción; ReglasService valida recinto permitido al intentar colocar.
   Frontend: DnD y selects filtran a recintos válidos; mensajes claros si no cumple.

9. Obligatorio cumplir la restricción si hay jugada legal
   Base de datos: no aplica.
   Backend: ReglasService calcula si existe al menos una opción válida; si existe, rechaza jugada inválida.
   Frontend: guía visual hacia opciones válidas y bloquea inválidas.

10. Capacidad de recintos
    Base de datos: no se fija capacidad en la tabla; se evalúa por reglas.
    Backend: ReglasService consulta conteo actual en colocaciones por recinto/jugador y compara contra capacidad definida para modo base.
    Frontend: feedback en tiempo real mostrando slots restantes.

11. Compatibilidad de recintos con tipo de dinosaurio
    Base de datos: catálogos recintos y dinosaurios; la compatibilidad se define en ReglasService (mapa por recinto/tipo).
    Backend: ReglasService valida compatibilidad antes de insertar colocación.
    Frontend: evita ofrecer combinaciones inválidas (filtros y DnD bloqueado).

12. Validez de datos (existencia en catálogos)
    Base de datos: FKs en colocaciones a recintos y dinosaurios_catalogo.
    Backend: validación por FormRequest + manejo de error de FK.
    Frontend: listas desplegables alimentadas de catálogos (evita valores libres).

13. Pertenencia de jugadas (partida y jugador correctos)
    Base de datos: FKs en colocaciones a partidas y usuarios.
    Backend: Policy verifica que auth()->id() coincida con usuario de la jugada y que pertenezca a la partida.
    Frontend: usa el contexto de partida y usuario logueado; no solicita IDs arbitrarios.

14. Estado para jugar (solo en_curso)
    Base de datos: estado en partidas.
    Backend: middleware/guard en rutas de jugada para verificar estado; PartidaService bloquea si no está en_curso.

Frontend: oculta/inhabilita UI de jugada si no está en_curso.

15. Puntuación automática por recinto
    Base de datos: opcionalmente se almacena pts_obtenidos en colocaciones.
    Backend: PuntajeService calcula al confirmar la colocación; guarda pts_obtenidos y actualiza totales si corresponde.
    Frontend: muestra puntaje parcial actualizado luego de cada jugada.

16. Cálculo de ganador al final
    Base de datos: totales en partida_jugadores (campo puntos_totales) para cachear; también se puede recalcular al vuelo.
    Backend: PuntajeService::final() suma por jugador y determina ganador; PartidaService marca estado cerrada.
    Frontend: pantalla de resultados con ranking y desglose por recinto.

17. Inmutabilidad tras cierre
    Base de datos: no hay trigger; se confía en estado de partidas.
    Backend: Policies + PartidaService rechazan cualquier inserción/edición si estado='cerrada'.
    Frontend: UI de lectura sola en partidas cerradas.

18. Integridad de catálogos (no borrar usados)
    Base de datos: FKs en colocaciones con ON DELETE RESTRICT a recintos y dinosaurios_catalogo.
    Backend: manejo de errores de integridad con mensajes claros.
    Frontend: impedir borrar desde UI si hay uso (pre-chequeo).

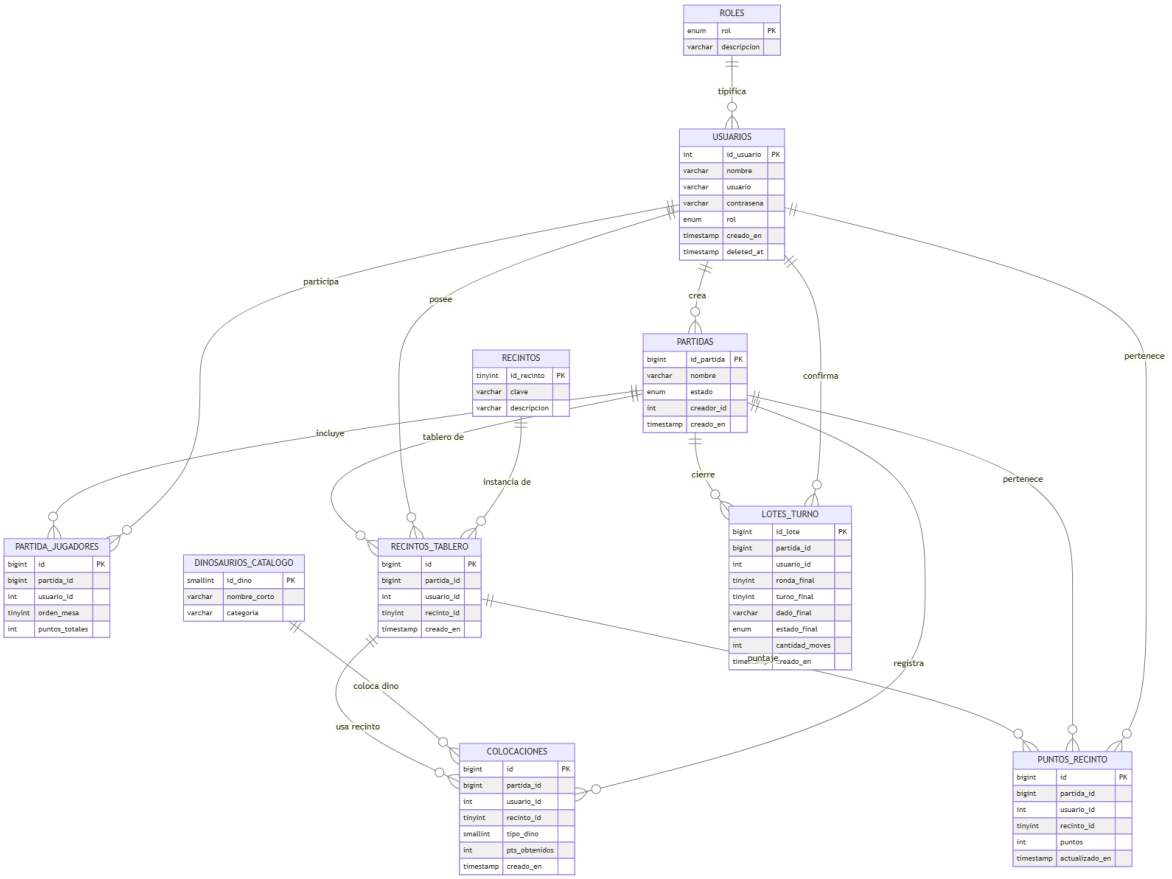
19. Acciones de administrador
    Base de datos: no aplica (rol está en usuarios).
    Backend: middleware EsAdmin en rutas de administración; Policies para operaciones sensibles (cerrar/revertir).
    Frontend: acciones de administración visibles solo para rol admin.

20. Registro de auditoría
    Base de datos: timestamps en colocaciones (creado_en) y en partidas; FKs a usuarios y partidas garantizan trazabilidad.
    Backend: logs de aplicación al revertir o cerrar; opción de ActivityLog si se desea (más adelante).
    Frontend: vista de historial de jugadas con usuario, fecha y turno (lectura de colocaciones ordenadas).

Notas de implementación transversal
a) Validación de entrada: FormRequests en Laravel para cada endpoint de

jugada/estado/avance (tipos, rangos, pertenencia).
b) Transacciones: cualquier acción que implique validar y grabar múltiples pasos (validar reglas, calcular puntos, guardar jugada) se ejecuta dentro de DB::transaction.
c) Concurrencia: se previene doble submit y se consulta unicidad antes de insertar; si se endurece, se agrega UNIQUE en (partida_id, usuario_id, ronda, turno).
d) Seguridad: Policies por pertenencia de partida; middleware auth y EsAdmin; CSRF activo en formularios.
e) Rendimiento: índices ya definidos (partidas por estado; colocaciones por (partida,ronda,turno) y (partida,usuario)); cache de catálogos si hiciera falta.
f) Internacionalización: los mensajes de validación y errores se leen de resources/lang (ES/EN) sin afectar las reglas de backend.

Implementar ajustes al DER según retroalimentación de la primera entrega



Link para visualizar mejor el DER:
https://drive.google.com/file/d/1rPdrUhz5rLFUQ1lp9f47LvBnqfQmrH46/view?usp=sharing

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

Tercera Forma Normal (3FN)

Criterio: no debe haber dependencias transitivas de atributos no-clave respecto de la clave.

Usuarios
Los atributos nombre, usuario, contrasena, rol, creado_en y deleted_at dependen directamente de id_usuario. El atributo rol es una clave foránea hacia la tabla roles, por lo tanto no hay duplicación de información. No se observan dependencias transitivas.

Roles
El atributo descripcion depende directamente de la clave rol. No se presentan dependencias transitivas.

Partidas
Los atributos nombre, estado y creado_en dependen de id_partida. El atributo creador_id es una clave foránea hacia usuarios, no se replica el nombre ni otros datos del creador. No hay dependencias transitivas.

Partida_jugadores
La clave primaria es id, pero existe una clave candidata formada por (partida_id, usuario_id). El atributo orden_mesa depende de la clave. El atributo puntos_totales se eliminó para evitar redundancia, ya que puede derivarse de puntos_recinto. No se almacenan datos del usuario ni de la partida, evitando dependencias transitivas.

Dinosaurios_catalogo
Los atributos nombre_corto y categoria dependen de id_dino. nombre_corto es único pero no determina otro atributo no-clave. No existen dependencias transitivas.

Recintos
Los atributos clave y descripcion dependen de id_recinto. Clave es única pero no determina otro atributo. No hay dependencias transitivas.

Recintos_tablero
La clave natural compuesta (partida_id, usuario_id, recinto_id) asegura que no se repitan recintos para un mismo jugador en una partida. Los atributos dependen de la clave completa. No hay dependencias transitivas.

Colocaciones
Incluye partida_id, usuario_id, recinto_id, tipo_dino, pts_obtenidos y creado_en.

Todos los atributos dependen de la clave primaria id. No se duplican datos de usuarios, recintos ni dinosaurios, evitando transitivas.

Puntos_recinto
Su clave compuesta (partida_id, usuario_id, recinto_id) determina el atributo puntos. Este atributo depende de la clave completa y no de un subconjunto. No existen transitivas.

Lotes_turno
Los atributos ronda_final, turno_final, dado_final, estado_final, cantidad_moves y creado_en dependen de id_lote. No se replica información de partidas ni usuarios. No hay transitivas.

Observaciones y decisiones de diseño

Llaves naturales y técnicas: se mantienen llaves técnicas (AUTO_INCREMENT) por simplicidad, y se documentan candidatas naturales como partida_jugadores (partida_id, usuario_id) y recintos_tablero (partida_id, usuario_id, recinto_id).

Atributos derivados: se eliminó puntos_totales de partida_jugadores. El total puede obtenerse desde puntos_recinto mediante agregación. pts_obtenidos se conserva en colocaciones para trazabilidad y cálculo de reglas.

Integridad referencial: claves foráneas hacia usuarios, roles, partidas, recintos y dinosaurios_catalogo. Eliminación en cascada solo en los casos donde tiene sentido conceptual (una partida elimina sus colocaciones, jugadores, recintos_tablero, puntos_recinto y lotes_turno).

Unicidades lógicas: partida_jugadores (partida_id, usuario_id) evita inscripciones duplicadas, partida_jugadores (partida_id, orden_mesa) evita duplicación de posición, recintos_tablero (partida_id, usuario_id, recinto_id) asegura que no se repita el mismo recinto.

Índices de consulta: colocaciones(partida_id, usuario_id, creado_en) para reconstruir jugadas, puntos_recinto(partida_id, usuario_id) para sumar rápido, partidas(estado) para listados.

Conclusión

El esquema cumple con:

- Primera Forma Normal: no existen atributos multivaluados ni grupos repetitivos.
- Segunda Forma Normal: no existen dependencias parciales respecto de claves compuestas.
- Tercera Forma Normal: no existen dependencias transitivas entre atributos no-clave, dado que se eliminó la redundancia de puntos_totales.

El modelo se encuentra en 3FN, con atributos consistentes, llaves claras y sin redundancias innecesarias.

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.

Crear scripts para inserción de registros iniciales de prueba

https://drive.google.com/file/d/1dGsN-jllkpMVi3puSyadhU6cFHMnSWkW/view?usp=sharing

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.