ChatGPT

Proyecto Geo Challenge Web – Especificación Técnica

Estructura General del Proyecto

El proyecto se estructura en una arquitectura de **cliente-servidor** tradicional: un **frontend** para la interfaz web y un **backend** que provee la lógica del juego, la comunicación en tiempo real y el acceso a datos [1]. El servidor web mantendrá el estado de múltiples partidas simultáneas, llevando registro de las partidas activas y los jugadores en cada una [2]. A continuación, se detalla la organización de carpetas y módulos:

- **Frontend/** – Aplicación web (SPA/PWA) en React u otro framework moderno. Código fuente, assets estáticos e implementaciones de UI.
- **public/** – Archivos estáticos públicos (ej. `index.html`, iconos, manifiesto PWA).
- **src/** – Código fuente de la aplicación frontend.
  - **components/** – Componentes reutilizables de interfaz (botones, temporizador, modal de resultados, etc.).
  - **pages/** (o **views/**) – Vistas o páginas principales (Inicio, Juego, Resultados, Rankings, Perfil…).
  - **services/** – Servicios/utilidades (ej. cliente API para llamadas al backend, manejo de sockets).
  - **i18n/** – Archivos de internacionalización (ej. textos en español e inglés).
  - **styles/** – Hojas de estilo (CSS/Sass o configuración de framework CSS).
- **Backend/** – Servidor de aplicación (API REST y/o WebSocket) con la lógica del juego.
- **src/** – Código fuente del servidor backend.
  - **models/** – Definición de modelos de datos (p. ej. Usuario, Pregunta, Partida, Puntaje).
  - **controllers/** (o **routes/**) – Endpoints de la API REST (p. ej. rutas para auth, juego, rankings, desafíos).
  - **services/** – Lógica de negocio agrupada (p. ej. servicio de juego que genera preguntas, servicio de rankings que calcula posiciones).
  - **sockets/** – Manejo de eventos WebSocket (p. ej. lógica de sala de juego en tiempo real).
  - **config/** – Configuraciones (p. ej. conexión a BD, variables de entorno).
- **tests/** – (Opcional) Pruebas unitarias/integración para la lógica del backend.
- **Docs/** – (Opcional) Documentación adicional, scripts de inicialización de base de datos, etc.

Esta separación facilita el desarrollo modular, permitiendo escalar cada parte independientemente. El frontend se comunica con el backend mediante llamadas HTTP (para operaciones estándar) y mediante **WebSockets** para funcionalidades en tiempo real (ej. partidas simultáneas). Ambos módulos pueden desplegarse por separado en infraestructura escalable (por ejemplo, frontend en un CDN/hosting estático y backend en un servidor cloud o servicios serverless).

Stack Tecnológico Sugerido

La elección del stack se orienta a tecnologías web modernas, priorizando rendimiento, escalabilidad y facilidad de desarrollo. A continuación se sugiere un stack para cada capa del proyecto:

Frontend

Se recomienda implementar el frontend como una **Aplicación Web Progresiva (PWA)** desarrollada con **React.js** [3] u otro framework SPA similar (Angular, Vue, Svelte). React es una opción madura y con amplio soporte, que permite crear una interfaz de usuario interactiva y modular. Al ser PWA, la aplicación podrá funcionar offline (cargando recursos estáticos en caché) y agregarse a la pantalla de inicio del dispositivo móvil, ofreciendo una experiencia cercana a una app nativa [4].

- **Librerías/UI:** Utilizar **HTML5** y **CSS3** con un enfoque **responsive** para adaptar el juego a pantallas móviles. Se puede emplear un framework de estilos ligero (por ejemplo, **Tailwind CSS** o **Bootstrap** en su modo más minimalista) para acelerar el desarrollo de una interfaz limpia. El diseño seguirá principios de *minimalismo*, enfatizando solo los elementos esenciales del juego y eliminando distracciones [5]. Esto implica utilizar paletas de colores suaves, tipografía clara y mínima ornamentación en pantalla, priorizando el rendimiento y la claridad visual.
- **Internacionalización:** Integrar una biblioteca de i18n (por ejemplo **i18next** para React) para gestionar textos en múltiples idiomas. Iniciar con soporte en español y estructurar el proyecto para poder añadir fácilmente traducciones al inglés en el futuro. Los textos de la interfaz se mantendrán en archivos JSON/YAML separados por idioma.

- **Optimización y PWA:** Configurar la app como PWA incluye generar un `manifest.json` con meta-información (nombre, iconos, tema) y un *service worker* para gestionar la caché de recursos estáticos. Esto permitirá que el juego cargue rápidamente y funcione parcialmente sin conexión (por ejemplo, permitiendo practicar offline con preguntas almacenadas) [4]. También habilitará la función "Añadir a pantalla de inicio" en móviles. Adicionalmente, se emplearán técnicas de optimización como *code splitting* (carga diferida de módulos) para reducir el tamaño inicial de la aplicación y mejorar la experiencia en dispositivos de gama baja.

- **Bibliotecas específicas:** Para funcionalidades geográficas, se pueden utilizar librerías especializadas. Por ejemplo, **Leaflet** (con mapas de OpenStreetMap) o la API de **Google Maps** (si se dispone de clave) para implementar el juego de ubicaciones en el mapa. También pueden usarse conjuntos de datos estáticos: banderas (imágenes SVG/PNG de cada país), listas de países y capitales (p. ej. de una API pública o dataset conocido). Estas imágenes y datos geográficos se manejarán como assets estáticos o a través de un CDN para mejorar la velocidad de carga.

Backend y Servidor

El backend estará construido sobre **Node.js**, aprovechando su modelo asíncrono y eficiente para aplicaciones en tiempo real. Se sugiere utilizar **Express.js** (framework minimalista) o **Nest.js** (framework estructurado) para organizar la API. Node.js es apropiado por su capacidad de manejar gran número de conexiones concurrentes (ideal para modo competitivo en tiempo real) y por permitir usar el mismo lenguaje (JavaScript/TypeScript) en todo el stack.

- **API REST:** El servidor expondrá endpoints RESTful para las operaciones principales: autenticación de usuarios, obtención de preguntas, envío de resultados, consulta de rankings, gestión de desafíos, etc. Estos endpoints seguirán principios REST, usando JSON como formato de intercambio. Por ejemplo: `POST /api/auth/register`, `POST /api/auth/login`, `GET /api/game/start`, `POST /api/game/submit`, `GET /api/leaderboard`, `POST /api/challenge` etc. Cada endpoint realizará validaciones de entrada (p. ej. campos requeridos, formatos) y devolverá códigos HTTP adecuados (200 OK, 201 Created, 400 Bad Request, etc.).

- **Tiempo real (WebSockets):** Para desafíos en vivo, se integrará **Socket.IO** u otra biblioteca WebSocket en el servidor Node. Esto permitirá comunicaciones bidireccionales en tiempo real entre servidor y clientes durante las partidas competitivas. Por ejemplo, al iniciar un desafío en tiempo real, el servidor creará una *sala* de Socket.IO para los jugadores y les enviará eventos: inicio de partida, nueva pregunta, respuestas recibidas, actualización de puntajes, etc. La elección de websockets es la forma más común de sincronizar el estado de un juego multijugador en tiempo real [6]. En una arquitectura escalable, podría haber un servicio central de *matchmaking* que organice las partidas (agrupando jugadores en lobbies) y uno o varios servicios de *game server* que ejecuten la lógica de cada partida [6]. Para simplificar en una primera versión, esto puede implementarse dentro de la misma aplicación Node (un módulo de emparejamiento que gestiona partidas y jugadores).

- **Alternativa backend BaaS:** Como alternativa o complemento, se puede considerar usar **Firebase** (Realtime Database o Firestore) para manejar la sincronización de juego y almacenamiento de datos en tiempo real. Por ejemplo, en vez de implementar sockets manualmente, Firebase permite que los clientes escuchen cambios en la base de datos (ej. actualización de preguntas o respuestas en una partida) sin configurar un servidor WebSocket propio [7] [8]. Bajo el capó, Firebase utiliza conexiones en tiempo real de forma similar, pero abstrae la complejidad y ofrece escalado automático en la nube. Esta vía podría acelerar el desarrollo (especialmente para el modo asíncrono), modelando datos como colecciones de *games*, *players*, *leaderboard*, etc., y delegando la distribución de eventos a la plataforma cloud [8]. No obstante, usar Firebase implica ajustar la lógica de nuestro juego a su modelo de datos y reglas de seguridad; por ello, en esta especificación asumiremos un backend propio con Node para mayor control, pero queda la puerta abierta a usar Firebase en la implementación si se prioriza rapidez de desarrollo sobre personalización.

- **Autenticación y seguridad:** Para gestionar la autenticación de usuarios de forma escalable, el backend usará **JSON Web Tokens (JWT)**. Tras el login, el servidor emitirá un token JWT firmado que el cliente almacenará (p. ej. en *localStorage* o en una *HTTP-only cookie* segura) y enviará en cada petición subsecuente. El uso de JWT permite un sistema *stateless* (sin sesión persistente en memoria del servidor), ya que el token en sí contiene la información de identidad verificada [9]. Cada petición a rutas protegidas será validada mediante un *middleware* que verifica la firma JWT y extrae la identidad del usuario. Esto elimina la necesidad de consultas a base de datos por cada validación de sesión [9], mejorando la escalabilidad horizontal (múltiples instancias del servidor pueden autenticar usuarios sin compartir estado de sesión). Para la implementación, se utilizará una librería como **jsonwebtoken** en Node. Los **passwords de usuarios se almacenarán en la base de datos de forma segura (hash + salt, p.ej. usando bcrypt)** en lugar de texto plano, y se aplicarán medidas de seguridad estándar (validación de entradas para prevenir inyección SQL/NoSQL, uso de HTTPS obligatorio en producción, restricciones CORS adecuadas para el dominio del frontend, etc.).

Base de Datos y Almacenamiento

Para el almacenamiento de datos se propone un enfoque híbrido que combine una base de datos primaria (relacional o NoSQL) con almacenamientos secundarios optimizados para casos de uso específicos:

- **Base de datos principal:** Se puede optar por una base **SQL relacional (PostgreSQL/MySQL)** o una base **NoSQL documental (MongoDB)**, dependiendo de la familiaridad del equipo y de la forma de los datos. En este juego, muchas entidades (usuarios, resultados, preguntas) tienen

estructura bien definida, por lo que una base relacional con un esquema normalizado sería adecuada. Por ejemplo:

- Tabla/colección **Usuarios**: (`id`, `nombreUsuario`, `email`, `passwordHash`, `fechaRegistro`, `idiomaPreferido`, etc.).
- Tabla/colección **Preguntas**: (`id`, `categoria` - mapa, bandera, capital..., `datoPregunta` - p.ej. nombre del país a preguntar, `opciones` - JSON con opciones múltiples, `respuestaCorrecta`, `datosAdicionales` - coordenadas para mapas, imagen para bandera, etc.).
- Tabla/colección **Partidas** (o **Resultados**): (`idPartida`, `jugadorId`, `puntuación`, `fecha`, `detalles` - JSON con desglose de aciertos, etc.), utilizada para almacenar resultados de juegos individuales o desafíos.
- Tabla **Amistades** (opcional, para lista de amigos/seguidores en la plataforma).

Si se emplea SQL, ORMs como **Sequelize** o **TypeORM** (en Node) pueden facilitar la definición de modelos y consultas. Si se usa NoSQL (MongoDB), se pueden usar ODMs como **Mongoose** o interactuar directamente con la API nativa para mayor rendimiento.

- **Datos geográficos estáticos:** Gran parte de la información de preguntas (banderas, capitales, coordenadas) es estática. Estos datos podrían precargarse en la base de datos o incluso almacenarse en forma de JSON estáticos dentro del frontend para reducir llamadas. Por ejemplo, un archivo JSON con `{ pais: "Chile", capital: "Santiago", bandera: "chile.png", coordenadas: [...] }` para cada país. Alternativamente, se puede crear un pequeño módulo de administración para cargar/actualizar este banco de preguntas en la BD.

- **Leaderboards y cache:** Para la funcionalidad de rankings globales en tiempo real, es recomendable usar un almacén en memoria como **Redis**. Redis ofrece la estructura de datos *Sorted Set*, que es muy eficiente para almacenar pares `usuario:puntuación` ordenados por puntuación, permitiendo obtener rápidamente el top N de jugadores o el puesto de un jugador dado [10] [11]. Cada vez que un jugador termine una partida competitiva, su puntuación puede actualizarse en este ranking en memoria. Consultar los **top 10** o el rango de un usuario será muy rápido gracias a Redis, y esto reduce la carga sobre la base de datos principal [11]. La base de datos principal aun así almacenaría los resultados completos y podría usarse para rankings históricos o detalles, pero Redis manejaría las consultas de ranking frecuentes. En caso de no usar Redis, se deben crear índices adecuados en la base de datos (por ejemplo, índice por puntuación en la tabla de usuarios o resultados) para poder obtener los mejores puntajes de forma eficiente, aunque a gran escala un enfoque cache basado en memoria es preferible.

- **Hosting y despliegue:** Se puede desplegar la base de datos en un servicio gestionado en la nube:

- Bases SQL: servicios como **Amazon RDS**, **Google Cloud SQL** o **Azure Database** podrían usarse.
- MongoDB: usar **MongoDB Atlas** u otro proveedor de Mongo gestionado.
- Redis: servicios como **Amazon ElastiCache** o **Redis Labs**.

El **backend Node** podría alojarse en una instancia de servidor (AWS EC2, Heroku, Railway, etc.) o empaquetarse en un contenedor Docker para despliegue flexible. También es posible usar **Functions as a Service** (AWS Lambda, Cloud Functions) para endpoints REST, y un servicio separado para WebSockets si fuese necesario (ya que los entornos serverless tradicionales no manejan conexiones persistentes fácilmente, en ese caso servicios como **Firebase** o **Ably** podrían cubrir la parte en tiempo real). El **frontend** al ser estático puede alojarse en **Netlify**, **Vercel**, **GitHub Pages** u otro CDN, sirviendo

los archivos optimizados de React. Se configurará el dominio y HTTPS. Además, se deben configurar pipelines de CI/CD para desplegar automáticamente tras pasar pruebas.

Módulos Principales del Sistema

A nivel de arquitectura de software, el sistema se divide en varios **módulos o subsistemas principales**, encargados de diferentes aspectos: usuarios, juego (preguntas y lógica), rankings y desafíos/competencias. A continuación, se describen cada módulo con sus responsabilidades:

Módulo de Usuario y Autenticación

Este módulo gestiona todo lo relacionado con los **usuarios** del juego: registro, inicio de sesión, perfil y autenticación en las interacciones con el sistema.

- **Registro de Usuarios:** Se ofrecerá la creación de cuenta mediante correo electrónico y contraseña (y/o opciones de OAuth social en el futuro, como Google o Facebook). Al registrarse, se valida que el email no esté ya en uso, se comprueba que la contraseña cumpla políticas de seguridad básicas (longitud mínima, complejidad) y se almacena el usuario en la base de datos. La contraseña se guarda hasheada (ej. usando bcrypt con salt). Se pueden guardar además un nombre de usuario visible (nickname) y el idioma preferido. En esta etapa, podría enviarse un correo de verificación (opcional, dependiendo del alcance).

- **Inicio de Sesión:** Permite a un usuario existente autenticarse con sus credenciales. Si son válidas, el servidor genera un token JWT firmado que contiene la identificación del usuario (p. ej. su ID o email) y lo devuelve. El módulo de usuario incluye un mecanismo para **refrescar** el token o controlar su expiración (por ejemplo, tokens válidos por 24h). Tras el login, todas las peticiones del cliente incluirán el token (vía encabezado `Authorization: Bearer` o cookie segura) y el backend validará dicho token para autorizar las acciones solicitadas [9].

- **Gestión de Perfil:** Los usuarios autenticados podrán acceder a una sección de perfil donde consultar y eventualmente editar su información básica: nombre visible, idioma, etc. También podría mostrar estadísticas personales (por ejemplo, partidas jugadas, puntuación máxima alcanzada, ranking actual). Este módulo provee endpoints como `GET /api/user/profile` y `PUT /api/user/profile` protegidos por auth.

- **Amigos/Redes Sociales (Opcional):** Para fomentar el aspecto competitivo, se podría implementar una funcionalidad de **amigos** o seguimiento de otros jugadores. Esto implicaría permitir que un usuario envíe solicitudes de amistad a otro (quizás buscando por nombre de usuario o importando contactos de Facebook/Google). Si se acepta, ambos se tendrían en su lista de amigos y podrían retarse más fácilmente entre sí (aparecerían en la sección de desafíos). Este es un extra no imprescindible de la primera versión, pero la arquitectura está preparada para agregarlo (p.ej. tabla de relaciones de amistad). Inicialmente, los desafíos se podrán hacer via enlace o código sin necesidad de lista de amigos.

- **Autorización y Roles:** En principio, todos los usuarios serán jugadores con privilegios equivalentes. Sin embargo, se prevé la posibilidad de un rol **admin** que pueda, por ejemplo, agregar o modificar preguntas en la base de datos o moderar el contenido. El sistema de auth JWT puede extenderse con información de roles (claims en el token) y el backend tendría middleware para verificar permisos en ciertas rutas (por ejemplo, rutas de administración de preguntas).

En resumen, el módulo de usuario asegura que solo usuarios autenticados acceden a las funciones competitivas y persistentes del juego, proporcionando la base de seguridad y personalización del sistema.

Módulo de Juego (Preguntas y Lógica de Quiz)

El módulo de juego encapsula la **lógica principal del quiz geográfico**: selección y presentación de preguntas, validación de respuestas, cálculo de puntajes y manejo de la dinámica de cada partida.

- **Banco de Preguntas:** Este subcomponente administra las distintas categorías de preguntas (mapas, banderas, capitales, etc.). Podría implementarse como un servicio que consulta la base de datos de preguntas o carga en memoria un conjunto de preguntas predefinidas. Debe garantizar aleatoriedad y no repetir preguntas recientes. Por ejemplo, al iniciar una partida de 10 preguntas mixtas, el módulo de juego seleccionará 10 preguntas aleatorias (podría ser cierto número de cada categoría para balancear). Cada pregunta incluye los datos necesarios: enunciado (o elemento visual a mostrar, como imagen de bandera o mapa), posibles respuestas (en caso de opción múltiple) y la respuesta correcta. En casos como las preguntas de mapa, en lugar de opciones puede esperar una interacción (clic en el mapa), y habrá que determinar la corrección en base a coordenadas.

- **Flujo de Partida Individual:** Este módulo controla la secuencia de las preguntas en una partida. En el modo clásico individual, por ejemplo: mostrar pregunta 1, esperar respuesta o hasta agotar tiempo, luego mostrar pregunta 2, y así sucesivamente. Lleva un contador de tiempo por pregunta (p. ej. 10 segundos máximo cada una) y un contador de puntaje acumulado. Por cada respuesta enviada por el jugador, el módulo comprueba si es correcta comparándola con la solución en la base de preguntas. De ser correcta, calcula los puntos ganados para esa pregunta (p. ej. un puntaje base + bonus por responder rápido, o en el caso de preguntas de mapa, puntos según la precisión de la ubicación indicada). Si es incorrecta, suele otorgar 0 puntos para esa pregunta (o potencialmente restar puntos si se quisiera penalizar, aunque inicialmente no se considerará puntaje negativo). El módulo avanza de pregunta en pregunta, manejando la lógica de temporizador: si el jugador no responde dentro del límite, se registra como incorrecta y se continúa.

- **Cálculo de Puntaje:** La lógica de puntuación se puede ajustar por tipo de juego:

- Para **preguntas de opción múltiple** (banderas, capitales, siluetas), un esquema común es otorgar, por ejemplo, **100 puntos** por acierto, más un **bono de tiempo**. El bono podría ser, por ejemplo, hasta 50 puntos adicionales si respondió muy rápido, calculado proporcional al tiempo restante. Así, responder correctamente en 2 segundos de 10 podría dar 100 + 40 = 140 puntos, mientras que tomar 9 segundos daría ~100 + 5 = 105 puntos.
- Para **preguntas de mapa** (ubicación), el puntaje puede basarse en la **distancia** entre el punto marcado y la ubicación correcta. Por ejemplo, 100 puntos si clavas la ubicación exacta, y menos puntos cuanto mayor sea el error (usando una fórmula que llegue a 0 puntos cuando el error > cierta distancia máxima). Se podría usar la fórmula del error inversamente proporcional, o segmentos (p. ej. 100 pts si <50 km, 50 pts si <200 km, 10 pts si <1000 km, 0 más allá). El cálculo exacto puede ajustarse para equilibrio del juego.
- El **módulo de juego realizará estos cálculos internamente en el servidor**, de modo que el cliente simplemente envía la respuesta (o coordenada) y el servidor devuelve cuánto puntuó esa respuesta y el acumulado. **Esto previene trampas**, ya que no se confía en que el cliente calcule la puntuación [9] . Todo cálculo crítico ocurre del lado servidor.

- Al final de la partida, el puntaje total es la suma de puntos de todas las preguntas. El módulo podría también compilar estadísticas como número de aciertos, porcentaje de precisión, etc., para mostrarlas en resultados.

- **Feedback y Respuestas:** El módulo de juego también determina el **feedback** que se da al jugador tras cada pregunta o al final. Por ejemplo, puede indicar inmediatamente si la respuesta fue correcta o incorrecta, mostrando la respuesta correcta si falló (esto ayuda al aprendizaje del jugador). Alternativamente, en un modo contrarreloj se puede posponer la retroalimentación detallada hasta el final para no distraer durante la ronda. La decisión de diseño aquí afectará la implementación: si se da feedback inmediato, el frontend necesita esa información (correcto/incorrecto, opción correcta) tras cada respuesta, que el backend puede enviar en la respuesta a la acción de responder.

- **Persistencia de resultados:** Al terminar una partida individual, el módulo registra el resultado. Si es un usuario registrado, creará una entrada en la base de datos con su puntuación y posiblemente detalles (p. ej. para un historial o para rankings). Si el jugador es invitado (no logueado), el resultado podría descartarse o usarse solo localmente. Se podría permitir al invitado registrarse después de jugar para guardar su puntuación, pero eso es un detalle de producto. En esta especificación asumimos que para competir en rankings es necesario estar autenticado, por lo que los invitados son solo para modo de práctica.

En síntesis, el módulo de juego maneja *qué* pregunta aparece *cuándo*, evalúa *cómo* respondió el jugador y produce *cuánto* puntaje obtiene, centralizando así la mecánica de trivia geográfica del sistema.

Módulo de Rankings (Clasificación de Jugadores)

Este módulo se encarga de la **clasificación y puntuaciones globales** entre jugadores, soportando el modo competitivo del juego. Sus responsabilidades incluyen:

- **Leaderboard Global:** Mantener un ranking global de puntuaciones (por ejemplo, la mejor puntuación de cada jugador, o puntos acumulados, dependiendo de la modalidad escogida). Probablemente se optará por tomar la puntuación más alta que cada usuario haya logrado en una partida individual como métrica principal para el ranking global (similar a como *Geo Challenge* original mostraba el mejor puntaje semanal/global de cada jugador). Alternativamente, podría ser un sistema de puntos acumulados o *ELO* de duelo; sin embargo, para iniciar, el enfoque de mejor puntaje es más sencillo. El módulo de rankings expone un endpoint `GET /api/leaderboard` que devuelve, por ejemplo, el top 50 global, incluyendo nombre de usuario y puntaje, y posiblemente el rank del usuario actual aunque no esté en el top (para que sepa en qué posición está).

- **Cálculo y Actualización:** Cada vez que un jugador termina una partida, el sistema comparará su resultado con su récord previo. Si es mayor, actualizará su puntuación en el ranking. Estas operaciones deben ser eficientes; por ello, se sugiere usar una estructura en memoria como **Redis Sorted Set** para insertar/actualizar la puntuación y mantener ordenado el ranking [10]. Redis puede directamente calcular la posición de un elemento y extraer rangos ordenados, permitiendo actualizaciones en tiempo real de manera muy rápida. En paralelo, la puntuación se guarda en la base de datos principal (por persistencia). Si no se dispone de Redis, el módulo de rankings usará consultas indexadas en la BD (p. ej. `SELECT ... ORDER BY score DESC LIMIT 10` para top 10), aunque a gran escala es menos eficiente. Usando Redis como caché, las lecturas frecuentes del leaderboard no cargarán la base de datos [11].

- **Rankings Temporales:** Podría haber rankings por período (semanal, mensual) para incentivar la competencia continua. Esto implica resetear o iniciar contadores cada periodo. El módulo puede gestionar tablas/keys separadas, e.j.: `leaderboard_global` y `leaderboard_semana_2026_04` etc. En una primera versión, se puede omitir esta complejidad y solo tener un ranking acumulado global.

- **Amigos vs Global:** Si se implementa el sistema de amigos, también se puede ofrecer un **ranking entre amigos**. Esto consiste simplemente en filtrar el leaderboard global a solo aquellos usuarios conectados contigo. El módulo de rankings puede tener una función para dado un usuario, obtener sus amigos (desde módulo de Usuario) y luego sus puntajes para ordenarlos. Esto se puede hacer en memoria (si la lista de amigos es manejable) o con consultas con JOIN/WHERE sobre la tabla de puntuaciones filtrando por amigos. Es un añadido útil para competencia amistosa, pero no requerido desde el inicio.

- **Desafíos y Torneos:** En modo desafíos directos (1 a 1), se podría llevar un conteo de victorias/derrotas aparte, o un rating estilo Elo, pero eso entra más en el módulo de Desafíos. El módulo de Rankings global es más bien el "marcador público" del juego.

En la implementación, el módulo de rankings incluirá funciones para: - Insertar/actualizar la puntuación de un usuario. - Consultar top N global. - Obtener la posición de un usuario dado (lo que implica consultar cuántos tienen puntuación mayor, fácilmente obtenible con comandos de sorted set). - (Opcional) Obtener ranking filtrado por algún criterio (país del jugador, amigos, etc., si se desea en el futuro).

Módulo de Desafíos y Modo Competitivo

Este módulo engloba las funcionalidades de **competencia entre jugadores**, ya sea en **tiempo real** o de forma **asíncrona**. Es el módulo más dinámico y que integra a los anteriores (usuario, juego, rankings) para crear enfrentamientos.

- **Desafíos en Tiempo Real (Multijugador simultáneo):** En esta modalidad, dos o más jugadores compiten respondiendo las mismas preguntas **simultáneamente**, comparando sus resultados en vivo. La implementación sigue el esquema cliente-servidor con WebSockets:
- Un jugador (o el sistema) puede **crear una partida multijugador**. Esto puede hacerse mediante un endpoint o socket event (ej. `socket.emit('createMatch', configuración)`), especificando quizá el tipo de juego (categoría o mixto) y número de jugadores si es pública. El servidor crea una nueva partida en estado "esperando jugadores" y le asigna un ID/sala.
- **Matchmaking:** Otros jugadores pueden unirse a esa partida. Si es un desafío privado a un amigo, el servidor podría enviar una notificación/invitación al amigo (vía WebSocket o push notification) y este usaría el ID de la sala para unirse. Si es un emparejamiento automático (p. ej. "jugar contra un rival aleatorio"), el servidor de matchmaking buscará si hay una sala libre o creará una nueva. Este concepto de *match-making server* y *game server* puede ser lógico dentro de la misma aplicación [6] : el matchmaking agrupa jugadores y luego la lógica de juego se lleva en la instancia de la partida.
- **Inicio sincronizado:** Una vez que los jugadores requeridos estén listos (por ejemplo, 2/2 en un duelo, o 4/4 si fueran más), el servidor emite un evento a todos los sockets de esa sala: `startGame`. Desde ese momento, el módulo de juego se encarga de distribuir las preguntas: podría enviar la Pregunta 1 a todos (`socket.to(sala).emit('newQuestion', datosPregunta)`), iniciar un temporizador común, recoger las respuestas entrantes de cada jugador (`socket.on('answer', respuesta)`), validar y actualizar puntuaciones parciales,

posiblemente emitiendo actualizaciones de marcador en vivo. Después de cada pregunta, se envía la siguiente hasta completar la ronda.

- **Sincronización y latencia:** El módulo debe manejar cuestiones de sincronización: por ejemplo, asegurarse de que todos los jugadores reciban la pregunta al mismo tiempo. Socket.IO se encarga de la distribución pero pequeñas latencias pueden ocurrir; para equidad, se podría introducir una cuenta regresiva de 3 segundos antes de mostrar la primera pregunta, de forma que todos estén preparados. Además, si algún jugador se desconecta a mitad de desafío, el servidor puede decidir pausarlo o terminarlo (eso puede ser diseño: quizás si un jugador se cae, se le considera abandonado y pierde). La arquitectura de Socket.IO maneja conexiones de forma asíncrona y tolerante a desconexiones breves [12] [13].
- **Cálculo de resultados multi:** Al finalizar, el servidor compila los resultados de todos los jugadores: puntaje de cada uno, aciertos, etc., determina un **ganador** (quien tenga mayor puntuación, o empate si igual). Emite un evento `gameOver` con esta información a la sala. Los clientes muestran la tabla final de posiciones. El módulo de desafíos luego registra el resultado de ese enfrentamiento: podría aumentar un contador de victorias/derrotas en el perfil de cada jugador, o generar una entrada de historial. También podría actualizar rankings globales si la actuación individual lo justifica (p. ej. si en este duelo alguien logró su máximo puntaje personal, actualizar en ranking global).

- **Escalabilidad:** Esta lógica en tiempo real puede escalarse horizontalmente mediante múltiples procesos/nodos de Socket.IO con Redis (pub/sub) para sincronizar eventos entre instancias, en caso de altísima concurrencia. Dado que el proyecto debe ser escalable, se tiene en cuenta que más adelante podría implementarse un sistema de salas distribuidas. Sin embargo, inicialmente una sola instancia Node puede manejar cientos de salas activas gracias a la eficiencia de sockets y la asincronía de Node.

- **Desafíos Asíncronos (Turnos diferidos):** Este modo permite competir sin que ambos jugadores estén conectados al mismo tiempo. Por ejemplo, un jugador juega una ronda y luego envía el **reto** a un amigo para que lo juegue después:

- Un jugador inicia un desafío asíncrono seleccionando categoría y quizá especificando a quién reta (o genera un link de desafío que puede compartir).
- Juega la partida normalmente; al terminar, el sistema guarda el conjunto de preguntas utilizadas y su desempeño.
- Se crea un registro de **Desafío pendiente** en la base de datos: contiene referencia al jugador retador, el oponente retado, el set de preguntas (identificadores o semillas para regenerarlas idénticas) y la puntuación del retador.
- El amigo retado, al entrar al juego, recibe una notificación o en una pantalla de “Desafíos” ve que tiene un reto pendiente. Puede aceptarlo y jugar. El módulo de juego le presentará **las mismas preguntas** que tuvo el retador (para garantizar equidad) en el mismo orden. Al finalizar, se compara su puntuación con la del retador:
    - Si la supera, gana el desafío (y quizás su resultado se convierte en el nuevo puntaje a batir, permitiendo revancha).
    - Se registra el resultado: quién ganó, y se actualiza el estado del desafío a “completado”.
- Ambos jugadores pueden consultar el resultado. Incluso si el segundo jugador nunca acepta el desafío, el sistema podría expirar el reto tras cierto tiempo.
- Este modo no requiere sincronización en tiempo real, pero sí guardar el estado del desafío y reproducir consistentemente las preguntas. Se debe asegurar que el jugador retado **no pueda conocer las respuestas del retador** antes de jugar; para esto, se podría ocultar el puntaje del retador hasta que el segundo termine, o simplemente no revelar las respuestas correctas al

retador en el momento (aunque podría haber comunicación externa entre ellos, lo cual escapa al control técnico).

- Implementar esto es más sencillo que el tiempo real: el backend necesita endpoints como `POST /api/challenge` (crear desafío) y `GET /api/challenge` (obtener desafíos pendientes para el usuario), y puede reutilizar el módulo de juego para cuando el retado juega, pasándole la semilla/ID de preguntas a usar.

- **Emparejamiento y variedad:** En competitivo también cabe la posibilidad de **torneos** o partidas con más de dos jugadores (ej. todos contra todos respondiendo preguntas y comparando puntajes). Esto sería una extensión donde el "matchmaking" une a varios jugadores en una misma sala (por ejemplo, 10 jugadores en una partida pública asincrónica diaria, donde todos juegan el mismo cuestionario durante el día y al final del día se ve quién ganó). La arquitectura de desafíos podría acomodar diferentes tamaños de partida, aunque para empezar nos enfocamos en duelos 1vs1 o pequeños grupos.

- **Integración con Rankings:** El módulo de desafíos comunicará sus resultados al módulo de Rankings y Usuario. Por ejemplo, tras un duelo, puede incrementar un contador de *partidas ganadas* en el perfil de quien ganó, y notificar al módulo de rankings si la puntuación obtenida por un jugador fue notable para el leaderboard. También puede tener un ranking específico de duelos ganados, pero eso sería complementario.

- **Notificaciones en tiempo real:** Para mejorar la experiencia, cuando un amigo te envía un desafío, si estás en línea, podrías recibir al instante un aviso en la UI (ej. "¡Juan te ha retado a un duelo de Banderas!"). Esto se logra fácilmente con el canal WebSocket del usuario (un evento personal). Si el usuario no está conectado, se pueden enviar notificaciones push al dispositivo (usando service workers) o un email, pero eso es opcional avanzado.

En conclusión, el módulo de desafíos habilita el **modo competitivo** del juego en sus dos variantes. Inicialmente se puede implementar el desafío asíncrono por simplicidad (no requiere mantener conexiones simultáneas), y luego añadir el modo en vivo con websockets para una experiencia más emocionante. Ambas estrategias no son excluyentes y comparten bastante lógica (preguntas, validación, puntuación), diferenciándose sobre todo en la sincronización. Cabe destacar que sincronizar juego multijugador añade complejidad, pero es factible con la tecnología propuesta (Node + Socket.IO) siguiendo patrones conocidos de matchmaking y game server [6].

Ideas para Tipos de Juegos y Preguntas Geográficas

El juego **Geo Challenge Web** incluirá diversos tipos de preguntas geográficas para hacerlo entretenido y educativo. A continuación, se proponen las categorías principales y cómo implementarlas:

Ubicación en el Mapa (Ciudades/Países)

Descripción: Se presenta al jugador el nombre de una ciudad o país, y debe señalar su ubicación en un mapa mundial. Por ejemplo: "¿Dónde está **París**?" – el jugador toca en el mapa el punto donde cree que está París.

Implementación: En el frontend, mostrar un mapa mundial interactivo (se puede usar un componente tipo Leaflet con un mapa simplificado sin etiquetas). El mapa debe permitir hacer *tap/clic* para marcar una ubicación. Una vez marcada la ubicación, el jugador confirma su respuesta. El backend compara las

coordenadas seleccionadas con las coordenadas reales del objetivo: - Para ciudades, usar una base de datos de coordenadas (latitud/longitud) de las principales ciudades o capitales. - Para países, se puede definir una coordenada representativa (por ejemplo la capital) o verificar si el punto cae dentro de las fronteras del país (esto último requeriría tener polígonos geográficos de cada país, lo cual es más complejo; la solución sencilla es evaluar distancia a la capital). - Calcular la **distancia** entre la respuesta y el destino real (utilizando fórmula del haversine para distancias geográficas). - Asignar puntaje según la distancia (distancia 0 = puntuación máxima, distancia mayor = menor puntuación, con un umbral a partir del cual es 0). En la respuesta al jugador, después de contestar, mostrar un marcador en la ubicación correcta y tal vez la distancia en km de su respuesta para aprendizaje.

Ejemplo de flujo: *Pregunta:* "¿Dónde se encuentra **Buenos Aires**?". El jugador toca en Sudamérica, quizá erra unos 300 km. El sistema responde "La ubicación correcta estaba 300 km al este" y otorga puntos en proporción (por ejemplo 70/100 puntos). Luego pasa a la siguiente.

Consideraciones: Debe haber un límite de tiempo (p. ej. 10-15 segundos) para evitar que el jugador busque externamente. El mapa puede tener zoom deshabilitado o limitado para mantener la dificultad (o permitir zoom para no penalizar demasiado en móvil). El estilo debe ser minimalista, sin nombres que den pistas, quizá solo contornos de países y líneas principales.

Banderas de Países

Descripción: Se muestra la imagen de una **bandera nacional**, y el jugador debe identificar a qué país pertenece.

Implementación: Preparar un conjunto de imágenes de banderas de todos los países (preferiblemente en formato SVG o PNG optimizado). En cada pregunta de este tipo, seleccionar aleatoriamente una bandera y generar opciones múltiples de respuesta: - Mostrar la bandera grande en pantalla. - Ofrecer 4 nombres de países como opciones (1 correcta y 3 distractores). Los distractores pueden elegirse de regiones similares para aumentar la dificultad (por ejemplo, si la bandera correcta es de un país africano, poner otros países africanos como opciones incorrectas). - El jugador pulsa la opción que crea correcta.

En backend, validar si la opción elegida corresponde al país correcto. Devolver si acertó o no. El frontend puede marcar la respuesta correcta (p.ej. resaltar en verde la opción del país correcto, y en rojo la elegida si fue distinta).

Ejemplo: Aparece la bandera con franjas horizontales rojo-blanco-rojo (Perú). Opciones: *México*, *Perú*, *Paraguay*, *Austria*. El jugador selecciona "Perú". Respuesta correcta, se otorgan 100 puntos (si fue rápido, más bonus).

Consideraciones: Debe confirmarse que las banderas estén bien dimensionadas para móvil. Evitar banderas muy parecidas juntas como distractores a menos que se busque mucha dificultad (por ejemplo, Luxemburgo vs Países Bajos). Este tipo de pregunta es visualmente atractiva y no requiere mucha escritura del jugador (solo tocar opción).

Capitales del Mundo

Descripción: Preguntas clásicas de capitales de país. Por ejemplo: "¿Cuál es la capital de **España**?" con opciones múltiples, o inversa "**Ottawa** es la capital de... ?".

Implementación: Usar la base de datos de países y sus capitales. La pregunta puede formularse de dos maneras para variedad: - Dado un país, elegir entre 4 ciudades cuál es su capital. - Dado el nombre de una ciudad capital, elegir entre 4 países cuál es el correcto.

Estas preguntas son puramente textuales. En el frontend se muestran la pregunta y 4 botones de opción con nombres. El jugador selecciona uno: - Backend comprueba si coincide con la capital correcta del país dado (o el país de la capital dada). - Retroalimentación similar: indicar correcto/incorrecto.

Ejemplo: "¿Cuál es la capital de **Australia**?" Opciones: *Sídney*, *Melbourne*, *Canberra*, *Perth*. (Mucha gente podría pensar Sídney, pero la correcta es Canberra). Si el jugador acierta Canberra, obtiene puntos completos; si no, 0 puntos para esa pregunta.

Consideraciones: Pueden incluirse preguntas de capitales de dependencias o estados no tan conocidos para mayor reto, pero inicialmente enfocar en países reconocidos globalmente. También se puede limitar a capitales nacionales para la versión inicial. La dificultad aquí radica en las opciones; es importante escoger opciones plausibles (en el ejemplo, ofrecer ciudades australianas incorrectas hace la pregunta más desafiante que ofrecer opciones de países distintos).

Siluetas de Países

Descripción: Se muestra la **silueta (contorno) de un país** sin ningún contexto ni color distintivo, como una sombra negra, y se pregunta de qué país se trata.

Implementación: Requiere tener las imágenes de siluetas de cada país. Se pueden obtener de mapas vectoriales simplificados. Para cada pregunta, seleccionar un país al azar, mostrar su contorno escalado a un tamaño estándar. Proveer 4 nombres de países de opción múltiple (incluyendo el correcto). El jugador elige la respuesta: - Backend verifica si el país seleccionado es el correcto para esa silueta. - Retroalimentación: mostrar el contorno etiquetado correctamente tras responder, tal vez sobre un mapa pequeño para referencia.

Ejemplo: Se muestra la silueta con forma de bota (Italia). Opciones: *Italia*, *Grecia*, *Estados Unidos*, *India*. La forma de Italia es icónica, el jugador elige "Italia", acierta.

Consideraciones: Algunas siluetas son muy características (Italia, Chipre) mientras otras son difíciles (muchos países africanos tienen forma irregular poco conocida). Para balancear, se podría en preguntas difíciles dar una pista sutil, p.ej. el continente. Pero inicialmente puede presentarse sin pistas para expertos. Es vital que las opciones incluyan países de silueta parecida para no trivializar la pregunta. Por ejemplo, si la silueta es de **Noruega**, poner opciones como Suecia o Finlandia de distractor.

Otros Tipos (Expansiones Futuras)

Además de las categorías anteriores, el juego puede expandirse con **nuevas modalidades** para mantener el interés: - **Monumentos Famosos:** Mostrar una fotografía o ilustración de un monumento (Torres Eiffel, Taj Mahal, etc.) y preguntar en qué país o ciudad se encuentra. - **Divisas o Idiomas:** Preguntar "¿Cuál es la moneda oficial de X país?" o "¿Qué idioma es oficial en X país?" – útil para aprendizaje general, aunque menos visual que las anteriores. - **Fronteras o Vecinos:** Preguntas del tipo "¿Con cuántos países limita Alemania?" o "¿Cuál de estos países NO tiene frontera con Argentina?". Esto aumenta la variedad de conocimiento geográfico puesto a prueba. - **Regiones/Subnacionales:** Para jugadores avanzados, identificar estados de un país (ej. estados de EEUU por forma, provincias, etc.) – aunque esto podría ser un nivel extra.

Estas ideas comparten la misma estructura de juego: pregunta + posibles respuestas y se pueden incorporar reutilizando gran parte de la lógica existente (cambiando el banco de datos y la forma de presentar). Se recomienda comenzar con **mapas, banderas, capitales y siluetas**, que fueron el núcleo del Geo Challenge original, y luego iterar añadiendo categorías adicionales en actualizaciones.

Lógica de Juego y Flujo de Usuario

A continuación se detalla cómo es la experiencia de un usuario típico y la secuencia de pasos internos del sistema, tanto en partidas individuales como en desafíos competitivos. También se describen las reglas de juego más importantes y cómo se gestionan en la aplicación.

Flujo de Juego Individual (Single Player)

1. **Inicio/Splash:** El usuario abre la aplicación web. Se le presenta una pantalla de bienvenida con el logo/título del juego. Desde aquí puede iniciar sesión/registro o continuar como invitado. La UI de inicio ofrece esas opciones de autenticación claramente.
2. **Menú Principal:** Una vez autenticado (o como invitado), el usuario ve el menú principal. Este menú contiene botones para las principales acciones:
3. *Jugar* (modo individual/práctica),
4. *Competir* (modo desafÍo contra otros),
5. *Rankings* (ver leaderboard global),
6. *Perfil* (si está registrado, para ver/editar perfil),
7. posiblemente *Configuración* (cambiar idioma, sonido, etc.).
8. **Selección de Modo Individual:** Al elegir *Jugar* (modo individual), el sistema puede ofrecer sub-opciones: jugar un **quiz mixto** (que combina preguntas de todos los tipos) o **especializado** en una categoría (solo banderas, solo capitales, etc.). En la primera versión, un modo mixto general es suficiente. El usuario selecciona y pulsa *Comenzar*.
9. **Desarrollo de la Partida:** Inicia la ronda de preguntas:
10. Se muestra la **Pregunta 1** con su formato correspondiente (mapa, bandera, etc.). En pantalla hay un indicador de tiempo regresivo (ej. 10 segundos). El usuario responde (sea tocando opción, clicando el mapa, etc.).
11. Inmediatamente o tras acabar el tiempo, se procesa la respuesta: se indica si fue correcta y se actualiza el puntaje parcial. Puede mostrarse brevemente una indicación ("¡Correcto! +100", "Incorrecto, era Canberra").
12. Luego avanza a **Pregunta 2**, repitiendo el proceso hasta completar, digamos, **10 preguntas**.
13. Si el tiempo se agota en una pregunta antes de respuesta, cuenta como incorrecta y continúa.
14. **Pantalla de Resultados:** Al terminar la última pregunta, se presenta la pantalla final de resultados de esa partida:
15. Puntaje total obtenido.
16. Número de respuestas correctas de X preguntas.
17. Detalle opcional: listar cada pregunta con la respuesta correcta y la del jugador (esto ayuda a aprendizaje; se puede mostrar un botón "Revisar respuestas" que expanda esta lista).
18. Mensajes de felicitación si lo hizo muy bien, o de ánimo para mejorar.
19. Botones: *Volver al Menú*, *Jugar de nuevo*, y si está logueado, quizás *Desafiar a un amigo* a superar ese puntaje.
20. **Actualización de Ranking:** Si el jugador estaba registrado, en este momento el sistema envía su puntaje al backend para actualizar sus récords. El backend del módulo de rankings compara y si es su mejor puntaje, lo actualiza en el leaderboard. Si el puntaje califica para el top (ej. top 10), la respuesta podría incluir su nueva posición. Esta actualización ocurre tras mostrar resultados, de forma transparente para el usuario (podría mostrarse una notificación como "¡Nuevo récord personal!" si corresponde).

21. **Continuar Jugando:** El usuario decide si volver a menú o iniciar otra ronda. La app permanece disponible para jugar cuantas veces quiera. Si cierra sesión o la app, sus puntajes quedaron guardados hasta ese momento.

Durante todo este flujo, el *frontend* se encarga de transiciones suaves entre pantallas (puede usar animaciones al pasar de pregunta a pregunta, barra de progreso de preguntas 1/10, etc., para mejorar la UX). El *backend* maneja las validaciones y el cronómetro de cada pregunta puede ser manejado en frontend (visualmente) pero controlado en backend para no ser manipulable; por ejemplo, el cliente manda la respuesta con un timestamp y el servidor verifica que esté dentro del tiempo límite.

Flujo en Modo Competitivo (Multijugador)

Si el usuario desde el menú elige el modo *Competir*, el flujo diverge según sea en **tiempo real** o **asíncrono**:

- **En tiempo real (duelo directo):**
- El usuario entra a la sección Competir. Ahí puede tener opciones como *"Jugar duelo rápido"* (contra jugador aleatorio) o *"Retar a un amigo"*.
- Si elige duelo rápido, se envía una solicitud al servidor de matchmaking. El sistema busca otro jugador que también esté buscando duelo. Cuando encuentra pareja, crea una sala y notifica a ambos de que se iniciará la partida. Si no hay, podría mostrar "esperando oponente..." con un botón para cancelar.
- Si elige retar a amigo, aparece su lista de amigos (o un campo para ingresar un código/ invitación). Selecciona al amigo deseado. El servidor envía una invitación al amigo (si está online por WebSocket, o lo almacenará para que lo vea luego, similar al asíncrono). Si el amigo acepta de inmediato, se crea la sala de juego. Si el amigo no está disponible, tal vez se propone convertirlo en un desafío asíncrono.
- Una vez en sala ambos jugadores, la partida inicia con una cuenta regresiva "Comenzando en 3...2...1". Luego se presentan las preguntas de la misma forma que en individual, *pero sincronizadas*. Ambos jugadores ven la **misma pregunta** al mismo tiempo. Cada uno responde en su dispositivo.
- El servidor recibe las respuestas de ambos. Puede enviar en tiempo real alguna indicación, por ejemplo "Jugador A respondió correcto en 5s, Jugador B incorrecto", dependiendo de si se quiere exponer información durante la partida. A veces es más emocionante *no* saber hasta el final cómo va el otro, manteniendo el suspenso.
- Se procede pregunta por pregunta. Al final, el servidor determina el ganador. Criterio: mayor puntuación total. (Si empate en puntos, podría declararse empate o desempatar por quién respondió más rápido en promedio).
- Pantalla final comparativa: ambos jugadores ven una pantalla con las puntuaciones de cada uno, quién ganó, y posiblemente desglose por pregunta (ej. "Pregunta 1: A acertó (+100), B falló; Pregunta 2: ambos acertaron...").
- Estos resultados se envían al backend para:
    - Actualizar estadísticas de cada jugador (victorias/derrotas).
    - Actualizar ranking global si alguno logró puntaje notable.
    - Registrar la partida en un historial si se lleva.

- Los jugadores pueden optar por revancha (botón "Revancha" que si ambos pulsan inicia otro duelo con los mismos) o salir al menú.

- **Asíncrono (retos diferidos):**

- El usuario va a Competir -> *Desafíos Pendientes*. Aquí puede ver una lista de desafíos enviados por amigos o desafíos que él haya enviado y estén en espera.
- Para enviar un nuevo desafío: selecciona un amigo de la lista o genera un link de reto. También elige categoría o mixto para el reto, luego juega la partida inmediatamente (como en modo individual pero etiquetada para desafío).
- Al terminar, su resultado se guarda y el amigo es notificado (si online) o al próximo inicio verá un desafío disponible: "¡Juan te desafía! Puntaje a vencer: 800 en Quiz Mixto." (Podemos mostrar el puntaje o mantenerlo oculto; probablemente es más motivador mostrarlo como meta a superar).
- El amigo acepta y juega la misma secuencia de preguntas. Su puntaje final se compara con el guardado. Se declara quién ganó el desafío. Ambos ven el resultado en sus pantallas (el retador podría recibir una notificación cuando el otro termina: "María jugó tu desafío: ¡Te ha superado con 900 pts!").
- Se actualizan registros de ambos (ej. +1 desafío jugado, quizás un conteo de victorias en perfil).
- Pueden hacer *revanchas* fácilmente: invertir roles y jugar otro set, etc.

Si un jugador recibe múltiples desafíos, puede jugarlos en cualquier orden. Los desafíos expirados o completados se retiran de la lista.

Comunicación cliente-servidor: En tiempo real se usa principalmente WebSocket para sincronización. En asíncrono, las acciones (crear desafío, aceptar desafío) pueden hacerse via llamadas REST, ya que no requieren inmediatez estricta, aunque usando WebSockets también se puede notificar eventos (por ejemplo, para refrescar la lista de desafíos en tiempo real cuando llegue uno nuevo).

Manejo de errores/ausencias: Si en un duelo en vivo un jugador se desconecta, el servidor puede dar la partida por terminada. Podría adjudicar victoria al que permanece o simplemente cancelarla. Para no penalizar desconexiones involuntarias, una idea es: si alguien se desconecta brevemente y reconecta, permitirle reengancharse a la sala mientras la ronda sigue (quizá perdiendo una pregunta). Estos detalles se pueden pulir; inicialmente, una desconexión implicará perder ese juego.

Sistema de Puntuación y Reglas de Juego

Además de las mecánicas ya mencionadas, resumimos las reglas globales y cómo se aplican técnicamente:

- **Número de preguntas:** Por defecto, cada partida (ya sea individual o desafío) tendrá un número fijo de preguntas (por ejemplo, 10). Este número se puede ajustar según pruebas de usabilidad (suficientemente largo para ser desafiante pero no tan largo que aburra). Para duelos, todos reciben la misma cantidad de preguntas.
- **Tiempo por pregunta:** Un límite de tiempo estricto (p. ej. 10 segundos por pregunta en banderas/capitales, tal vez un poco más para mapas) para mantener la presión y evitar búsquedas externas. El temporizador se muestra al usuario y al agotarse se bloquea la respuesta. En implementación, el frontend inicia un cronómetro local para la barra de tiempo, pero la validación real la hace el servidor: si llega una respuesta con timestamp fuera de tiempo, se considera nula. Para simplificar, se puede hacer que el frontend no permita enviar después de 0 seg.
- **Puntuación:** Como detallado antes, puntos por acierto con posibles bonus. No se otorgan puntos por respuestas incorrectas o omitidas. No hay penalización negativa, el puntaje mínimo de una pregunta es 0.

- **Secuencia aleatoria:** El orden de las preguntas en cada partida debe ser aleatorio. También el orden de las opciones múltiples debe randomizarse cada vez para que no haya patrón (la API puede proveer las opciones ya en orden mezclado).
- **Unicidad de preguntas:** Dentro de una misma partida no se debe repetir la misma pregunta. El banco de preguntas debe ser suficientemente grande para variedad. Entre partidas distintas, puede haber repetición, aunque idealmente el sistema podría evitar darle al mismo usuario la misma pregunta muy frecuentemente (se podría llevar un registro de preguntas ya vistas recientemente por usuario, pero es un extra).
- **Dificultad progresiva:** Podría implementarse que las preguntas se tornen más difíciles hacia el final de la ronda (por ejemplo, primeras son fáciles, últimas más difíciles). Esto no es imprescindible en la primera versión, pero es un nice-to-have. Requeriría marcar preguntas con un nivel de dificultad en la base de datos y seleccionarlas acorde.
- **Integridad del juego:** Para asegurar fair play:
- Toda lógica crítica (validar respuestas, calcular puntos) corre del lado servidor, el cliente no puede manipular sus puntos.
- Usar HTTPS evita *sniffing* de respuestas correctas en tránsito.
- En desafíos en vivo, los jugadores no pueden interferir uno con el otro excepto compitiendo limpiamente en rapidez/conocimiento.
- Se podría implementar algún sistema de detección de trampa (por ejemplo, si alguien siempre responde en 0.1 segundos quizás esté usando un bot) y tomar acciones, pero esto es muy avanzado. En principio confiamos en la comunidad.
- **Multi-idioma preguntas:** Al expandir a inglés u otros idiomas, las preguntas de texto (capitales, por ejemplo) deben traducirse o adaptarse. Por ejemplo, "¿What is the capital of Germany?" vs "¿Cuál es la capital de Alemania?". El sistema de i18n debe contemplar textos dinámicos. Un enfoque es almacenar el *dato* en un idioma neutro (p. ej. nombres propios de países, capitales usualmente no se traducen, salvo algunos casos - *Germany/Alemania*) y las plantillas de pregunta se traducen. Para banderas y mapas no cambia nada por idioma. Este detalle se resolverá al añadir idiomas.
- **Audio/FX:** Opcionalmente, se pueden agregar efectos de sonido (un sonido correcto/incorrecto). Técnicamente, incluir archivos de sonido y reproducirlos en eventos de acierto/error; esto no afecta la lógica pero mejora la experiencia.

En resumen, las reglas del juego se han diseñado para ser claras y fáciles de implementar, inspiradas en el juego original: tiempo limitado, puntuación por conocimiento y velocidad, y distintas categorías de preguntas que ponen a prueba diferentes habilidades geográficas del jugador.

Especificación Técnica para Desarrollo (por IA)

A continuación se desglosan las tareas técnicas necesarias para construir el proyecto, de forma que podrían ser ejecutadas por desarrolladores o incluso delegadas a una IA programadora por módulos. Las tareas se separan en **Backend** (servidor, lógica) y **Frontend** (cliente, interfaz):

Backend: Tareas de Implementación

1. **Inicialización del Proyecto Backend:** Crear un nuevo proyecto Node.js (inicializar `package.json`). Configurar el entorno básico: incluir TypeScript (si se usará) o definir estándar ES6+, configurar un sistema de build si aplica, e instalar frameworks principales (Express o Nest). Organizar la estructura de carpetas según *src/models, src/controllers,* etc. y preparar un archivo de entrada (por ej. `src/index.js/ts`) que levante el servidor Express en un puerto configurado.

2. **Configuración de Base de Datos:** Elegir y conectar la base de datos. Instalar el cliente/ORM (por ej. `mongoose` para MongoDB o `pg` /Sequelize para PostgreSQL). Implementar la conexión en un módulo de configuración, leyendo credenciales/URL de la BD desde variables de entorno para facilitar despliegue. Probar la conexión inicial.
3. **Definición de Modelos de Datos:** Implementar los modelos/esquemas:
4. **Usuario:** con campos `nombreUsuario`, `email`, `passwordHash`, etc. Si SQL, escribir migración/DDL para la tabla; si NoSQL, definir esquema Mongoose. Incluir índices necesarios (por ejemplo, unique index en email).
5. **Pregunta:** esquema que abarque todos los tipos (campos para categoría, enunciado, opciones[], respuestaCorrecta, etc.). Posiblemente varios sub-esquemas o una jerarquía por tipo.
6. **Resultado/Partida:** para registrar partidas jugadas (id de usuario, puntaje, fecha, detalles opcionales).
7. **Desafío:** (si se maneja en BD) con campos como `jugadorRetador`, `jugadorDesafiado`, `estado (pendiente/completado)`, `preguntas` (lista de IDs o semilla), `puntajeRetador`, `puntajeDesafiado`.
8. Si se utiliza Redis, no requiere modelo explícito, pero sí planificar las keys (por ej. `leaderboard_global` sorted set).
9. **Implementación de Autenticación (Auth Controller):**
10. Crear rutas POST `/api/auth/register` y `/api/auth/login`.
11. En register: validar inputs, hash de contraseña (bcrypt), guardar nuevo usuario en BD, manejar duplicados. Devolver éxito o errores formateados (ej. 400 si email ya existe).
12. En login: verificar credenciales, si correctas generar JWT (con secret seguro almacenado en env) que incluya, p.ej., `userId` y quizá `username`. Devolver el token y algunos datos básicos del usuario (ej. nombre para mostrar).
13. Implementar middleware `authenticateJWT` para Express: leer token de cabecera, verificar firma y extraer payload. Si inválido o ausente, responder 401; si válido, adjuntar info de usuario (req.user = payload).
14. Proteger con este middleware todas las rutas posteriores que requieran usuario logueado (juego, envio de puntaje, crear desafío, etc.).
15. **Endpoints de Juego (Game Controller):**
16. `GET /api/game/start` (o similar): Iniciar una nueva partida individual. Puede recibir parámetro de categoría o modo. La lógica: seleccionar un conjunto de preguntas aleatorias según lo pedido. Opcionalmente, se podrían generar todas las preguntas del tirón y enviarlas al cliente, **o** generar de una en una. Dos enfoques:
    - *Una a una:* el cliente pide `GET /api/game/question` repetidamente. Más sencillo es enviar todas juntas en un array con la respuesta correcta oculta para evitar muchas idas y vueltas. No obstante, enviar todas juntas podría exponer respuestas correctas si el cliente inspecciona (a menos que se envíen cifradas). Alternativamente, enviar preguntas con un token por pregunta.
    - Para empezar más simple: el endpoint start devuelve la lista de preguntas (cada una con opciones, etc., pero sin marcar cuál es correcta; el cliente aún así podría deducir si no se cuida, pero confiaremos en que no hagan trampa inspeccionando objetos JS).
17. `POST /api/game/answer` (opcional si vamos pregunta por pregunta): Recibe la respuesta del jugador a la pregunta actual. En el backend, verifica y calcula puntaje de esa pregunta. Puede guardar progreso en una sesión server-side si fuéramos con estado (pero mejor no). Si stateless, el cliente tendría que mandar identificador de pregunta y su respuesta, el servidor la evalúa y devuelve correct/incorrect y puntos obtenidos. También podría devolver la siguiente pregunta, combinando pasos.
18. `POST /api/game/finish`: Si se optó por mandar todas preguntas de golpe, este endpoint recibiría el resumen de respuestas del jugador al final. El backend recalcula el puntaje total

(asegurándose de no fiar del todo en lo que envió cliente) y guarda el resultado. Devuelve puntaje final y feedback final.

19. En cualquiera de las estrategias, garantizar que el cálculo de puntos se haga en servidor. Usar la lógica de puntaje definida (posiblemente refactorizada en un módulo de servicio utilizable tanto en individual como en duelos).
20. Guardar en BD el resultado si usuario logueado.
21. **Endpoints de Rankings (Leaderboard Controller):**
22. `GET /api/leaderboard`: Retorna el top N (por ejemplo 10 o 50) de jugadores ordenados por puntuación. Incluye rank, nombre y puntuación. Si el usuario solicitante no está en ese top, incluir también su posición y puntuación aparte (requiere búsqueda de su rank, que con Redis sorted set se logra con `ZREVRANK` fácilmente).
23. Posiblemente `GET /api/leaderboard/friends`: similar pero filtrando a amigos (si hay sistema de amigos implementado).
24. Internamente, estas rutas leerán de Redis (o de BD con ORDER if no Redis). Se debe asegurar manejo de cache: por ejemplo, actualizar Redis en tiempo real con cada nueva puntuación, y quizás tener un fallback para recalcular ranking completo de BD periódicamente.
25. También puede haber un endpoint `GET /api/user/:id/score` para consultar el mejor puntaje de un usuario específico (útil para perfil), que igual puede venir de Redis.
26. **Endpoints de Desafíos (Challenge Controller):**
27. `POST /api/challenge`: Crear un nuevo desafío. Body incluirá `opponentId` (usuario retado) y quizás `mode` / `category`. Lógica: generar set de preguntas (como en game/start), guardar un registro de desafío en estado pendiente con las respuestas correctas guardadas en servidor (para luego poder evaluar o comparar). Devolver un ID de desafío. Notificar al oponente si está online (por WebSocket, ver sección sockets).
28. `GET /api/challenges`: Listar desafíos pendientes para el usuario actual (donde es oponente y no ha jugado, o los que él lanzó y esperan). El cliente usará esto para mostrar la lista de retos.
29. `POST /api/challenge/:id/accept`: Cuando el retado decide jugar. El servidor recupera el desafío, verifica que sigue pendiente y que el solicitante es el retado correcto. Luego le devuelve las preguntas del desafío para jugar (similar a start, pero usando las ya determinadas). Alternativamente, podemos integrar este flujo con el mismo endpoint de game start/finish pero pasando referencia al desafío.
30. `POST /api/challenge/:id/submit`: El retado envía sus respuestas/puntaje al terminar. El servidor calcula su puntaje, compara con el guardado del retador, marca ganador, actualiza el desafío a completado con los datos del segundo jugador. Posiblemente devuelve resultado (quién ganó, puntajes).
31. Seguridad: asegurar que un jugador no pueda acceder a desafíos ajenos.
32. **Socket.IO Event Handlers:** Configurar Socket.IO en el servidor (integrado con Express server).
33. Al conectar un cliente, autenticar el socket (por ejemplo, enviando el JWT en el handshake). Rechazar/no permitir no autenticados en canales de juego.
34. **Matchmaking events:** e.g. `socket.on('duel:queue')` cuando un jugador busca partida rápida. Añadir el socket a una cola o emparejar si hay otro esperando. Si empareja, crear sala (room) y emitir a ambos `duel:found` con salaID.
35. **Challenge invite events:** e.g. `socket.on('challenge:invite', opponentId)` para enviar invitación en tiempo real. Buscar el socket del oponente (mantener un mapa de userId->socketId conectados) y emitirle `challenge:invited` con info (desafíoID, nombre retador, categoría). El oponente puede emitir `challenge:accept` o `challenge:decline`. Si acepta, notificar al retador y comenzar partida en sala.
36. **Game play events:** Dentro de una sala de duelo, manejar:
    - `socket.on('ready')` de cada jugador para confirmar que cargó la partida. Cuando todos listos, servidor envía `game:start`.
    - En cada pregunta: servidor emite `game:question` con datos.

    - Jugadores emiten `game:answer` con su elección. Servidor registra y cuando todos respondieron (o tiempo agotado) emite `game:result` parcial (p.ej. correcto/incorrecto de cada uno, puntuaciones acumuladas hasta ahora).
    - Repetir hasta fin, luego emitir `game:over` con resultado final.
37. Manejar `socket.on('disconnect')` para capturar abandonos: retirar de colas, informar a la sala si estaba en medio de juego (ej. emitir a oponente que "jugador X se desconectó").
38. Estas funciones de socket se implementarán posiblemente en un módulo aparte (src/sockets/handlers.js) pero interactúan con los servicios de juego ya existentes (para obtener preguntas, validar respuestas). Se debe tener cuidado de no duplicar lógica: idealmente el **servicio de juego** debe exponer métodos reutilizables por REST y por sockets (por ejemplo, `GameService.getQuestionsForGame()` o `GameService.checkAnswer()`).
39. **Servicios Internos y Utilidades:** Implementar clases o módulos para lógica reusable:
40. `GameService` con métodos: generación de preguntas (con parámetros de categoría, cantidad), cálculo de puntaje (dado pregunta y respuesta, retorna puntos), etc.
41. `RankingService`: métodos para actualizar y consultar Redis/DB (e.g. `updateScore(userId, score)`, `getTopN(n)`).
42. `NotificationService` (opcional): para enviar notificaciones push/email en desafíos asíncronos.
43. Utilidades varias: cálculo de distancia geográfica (función haversine), funciones de barajar arrays (para opciones aleatorias), etc.
44. **Pruebas y Verificación:** Escribir pruebas unitarias básicas para funciones puras (cálculo de puntaje, selecciones aleatorias). Realizar pruebas manuales o automatizadas con herramientas tipo Postman para los endpoints REST (registro, login, flujo de juego, etc.). Simular con dos clientes el flujo de socket duelos en un entorno de desarrollo para garantizar sincronización.
45. **Documentación de API:** (Recomendado) Usar Swagger/OpenAPI o al menos un README para documentar los endpoints, request/response esperados, para que los desarrolladores frontend (o IA) sepan cómo interactuar correctamente con el backend.

Frontend: Tareas de Implementación

1. **Configuración del Proyecto Frontend:** Inicializar proyecto React (usando Create React App, Vite, Next.js – cualquiera que se acomode, aquí supongamos CRA por simplicidad). Estructurar carpetas conforme a /src/components, /src/pages, etc. Configurar el sistema de rutas (React Router si SPA tradicional, o páginas si Next). Incluir librerías iniciales: e.g. axios o fetch polyfill para llamadas API, Socket.IO client, i18next, library de UI (opcional).
2. **Diseño de Navegación y Estados Globales:** Implementar el router de la aplicación con rutas: `/` (inicio), `/login`, `/register`, `/menu`, `/play`, `/duel`, `/leaderboard`, `/profile`, etc. Configurar un contexto global o store (Redux, Context API) para manejar estado de autenticación (usuario logueado, token) y posiblemente estado de juego actual (preguntas en curso, puntaje actual). Por ejemplo, crear un `AuthContext` que provea `user` y `login/logout` methods a los componentes, almacenando el token JWT.
3. **Pantallas de Autenticación (Login/Registro):** Crear componentes **Login** y **Registro** con formularios. Controlar campos con estado local o form library. Al hacer submit, llamar a la API REST correspondiente (`/auth/login` o `/auth/register`). Manejar respuestas de error (mostrar mensajes en formulario si email ya existe, credenciales inválidas, etc.). Si login es exitoso, guardar el token (p.ej. en localStorage o en memoria) y marcar usuario como autenticado, luego redirigir al menú principal. Implementar también opción de *"Continuar como Invitado"* que básicamente omite login (podría simplemente setear un estado `guest=true` y seguir).

4. **Menú Principal:** Componente que muestra las opciones de juego. Debe saludar al usuario (ej. "Hola, {nombre}"). Botones: *Jugar Solo*, *Competir*, *Rankings*, *Perfil*, *Cerrar Sesión*. Cada botón lleva a la ruta o funcionalidad correspondiente.
5. Si el usuario es invitado, quizás *Perfil* está deshabilitado, y *Competir* podría pedirle que se registre (porque para rankings/desafíos idealmente debería tener cuenta). Se puede implementar que invitados solo accedan a juego rápido sin ranking.
6. **Interfaz de Juego (Quiz Single Player):**
7. Componente principal **GamePlay** que muestra la pregunta actual y maneja la lógica de temporizador y respuestas. Este componente recibiría via props o contexto la lista de preguntas de la partida actual (si las obtuvo todas al inicio) o podría ir pidiendo pregunta a pregunta.
8. Mostrar el número de pregunta (1/10), el enunciado, la representación (imagen de bandera, mapa interactivo, texto según el tipo). Mostrar opciones si las hay (como botones). Mostrar un temporizador (una barra o círculo que se vacía en 10 segundos).
9. Al montarse, iniciar el temporizador de la primera pregunta. Cuando el jugador responde (clic en opción o confirmación en mapa), detener temporizador, bloquear interacción. Enviar la respuesta al backend:
    - Si usando modo *todas preguntas pre-cargadas*: posiblemente no hace falta petición por pregunta, solo almacena local si fue correcta para cálculo final. Pero mejor feedback inmediato: se podría llamar a un endpoint `/game/answer` con `questionId` y respuesta para saber si es correcta y puntos.
    - Alternativamente, toda la corrección puede hacerse en frontend para inmediatez, *pero* esto expone la respuesta correcta en el código. Es más seguro consultar al backend. Quizá un compromiso: backend devuelve con la lista de preguntas también un *hash/ID* que permite validar. O cada pregunta lleva un `correctIndex` encriptado.
10. Para no complicar demasiado: El frontend decide si fue correcto comparando con datos que tiene (podemos enviar la respuesta correcta oculta y revelarla tras respuesta). Dado que confiamos en el jugador en single player no competitiva, esto podría pasar. En multi no, pero multi se maneja distinto.
11. Mostrar feedback visual: pintar en verde la respuesta correcta, en rojo la opción elegida si era incorrecta. Mostrar "+X pts" obtenido. Esperar 2 segundos.
12. Pasar a la siguiente pregunta (reset UI, timer). Repetir.
13. Tras la última pregunta, navegar a pantalla de resultados.
14. Durante este flujo, si el usuario intenta salir, preguntar confirmación (para no perder progreso accidentalmente).
15. **Pantalla de Resultados:** Componente que muestra el resumen final. Recibe via estado o props el puntaje total, y posiblemente info de desempeño. Muestra mensajes según puntaje (puede haber mensajes predefinidos para rangos de score). Incluye botón para volver al menú y quizás para compartir resultado (ej. "¡Obtuve 8/10 en GeoChallenge!").
16. Si el usuario está logueado, la lógica de actualización de ranking ocurrirá ya en backend. El frontend podría opcionalmente mostrar algo como "Consultando ranking..." y luego mostrar "Estás en la posición 15" si la API proporciona esa info. Un endpoint como `/leaderboard/me` podría ser útil que devuelva rank actual del usuario.
17. **Pantallas de Rankings:**
18. **Leaderboard Global:** Un componente que al montarse hace `GET /api/leaderboard` y muestra en forma de tabla o lista los top jugadores: posición, nombre, puntuación. Destacar quizás el usuario actual en la lista (si está presente o fuera de top, mostrar último elemento "Tú: puesto X con Y puntos"). Este componente debe actualizarse periódicamente o tener pull-to-refresh, pero no es muy dinámico que cambie segundo a segundo, quizás recargar cada vez que se visita.
19. (Si se implementa ranking amigos: otra pestaña en esta pantalla para amigos).
20. Asegurar formato responsivo, lista scrollable en móvil.

21. **Pantallas de Desafíos/Competición:**
22. **Lista de Desafíos:** Muestra los desafíos pendientes. Para cada entrada: nombre del retador/oponente, categoría, puntaje a vencer (si el jugador es el retado), y botones "Jugar ahora" o "Rechazar". Esta lista se obtiene de `GET /api/challenges`. Se podría usar WebSocket para actualizar en vivo: suscribiendo a eventos de nuevas invitaciones.
23. **Enviar Desafío:** Al elegir un amigo para retar (o ingresar código), llamar a `POST /api/challenge` y luego iniciar el juego inmediatamente. Posiblemente integrarlo con la misma pantalla de juego single player pero en modo "challenge". Tras terminar, notificar "Desafío enviado" y volver a menú o a lista de desafíos.
24. **Juego en vivo (Duelos):** La interfaz en un duelo en tiempo real es similar a la de single player, pero con algunas diferencias:
    - Debe mostrar el estado del oponente también. Por ejemplo, podría tener un pequeño panel con el nombre del oponente y su puntaje actual, actualizado cada pregunta o en tiempo real. O al menos indicar "Oponente respondió X de Y preguntas" si se desea.
    - El temporizador es común, así que una sola visualización sirve para ambos.
    - No se puede pausar entre preguntas esperando input del usuario manualmente; aquí las preguntas van cadenciadas automáticamente. El cliente recibirá eventos `game:question` y debe inmediatamente mostrarla. Si un jugador no responde en el tiempo, simplemente 0 puntos esa pregunta.
    - Al finalizar, la pantalla de resultados de un duelo comparará ambos puntajes y destaca ganador. Posiblemente con confeti o así para el ganador.
25. Implementar la conexión Socket.IO: En la sección Competir, cuando se inicia un duelo rápido o se acepta invitación, conectar el socket (si no conectado ya), unirse a la sala adecuada. Manejar los eventos:
    - on `startGame` -> mostrar cuenta regresiva.
    - on `newQuestion` -> renderizar la pregunta.
    - on `result` -> actualizar marcadores (podría guardarse en estado el score del jugador y del oponente).
    - on `gameOver` -> navegar a la pantalla final de resultados del duelo con los datos proporcionados.
26. Proporcionar feedback de conexión: si el oponente se desconecta, podría mostrar "El oponente se ha desconectado, partida cancelada".
27. Asegurar que la interfaz de duelo es altamente responsive ya que la simultaneidad exige fluidez (utilizar adecuadamente `useEffect` hooks para inscribir/desinscribir listeners, etc.).
28. **Perfil de Usuario:** En esta pantalla, mostrar info del usuario (nombre, desde cuándo juega, su mejor puntaje, estadísticas como % aciertos global, número de duelos ganados, etc. según qué registremos). Permitir cambiar configuración simple, p.ej. idioma del juego (lo que actualiza i18n instantáneamente y guarda en backend preferencia). Si avatar o otras personalizaciones son soportadas, permitir cambiarlos. Esto es opcional; inicialmente, con mostrar el nombre y mejor score puede ser suficiente.
29. **Internacionalización:** Configurar i18next o similar:
    - Definir archivos de traducción en `/src/i18n/es.json` y `/src/i18n/en.json`.
    - Añadir en la UI toggles o selección de idioma (quizás banderitas) en la pantalla de configuración o perfil.
    - Enlazar los textos visibles a keys de i18n. Asegurar que incluso los nombres de países/capitales se traduzcan si es necesario (p.ej. "Germany" vs "Alemania" – se puede resolver mediante lista de países en cada idioma).
    - Probar cambiando idioma que todo el texto de interfaz (no necesariamente los datos de preguntas) cambia adecuadamente.

30. **Estilos y Responsive Design:** Escribir CSS o utilizar utilidades (Tailwind) de forma que la interfaz se adapte a pantallas pequeñas:
    - Usar diseño mobile-first: botones grandes y fáciles de tocar, texto legible en móvil.
    - Vista de mapa: asegurarse que en pantallas móviles se pueda hacer scroll o que el mapa quepa visible (posiblemente usar todo el ancho y permitir al jugador desplazarse dentro de un contenedor).
    - Minimizar distracciones: un esquema de colores agradable (fondo claro u oscuro con buen contraste para preguntas).
    - Iconografía: usar íconos simples (quizá un icono para marcador de mapa, trofeo para ranking, etc., utilizando alguna librería de iconos).
    - Probar en varios tamaños (emular dispositivos en devtools).
31. **Integración con Backend:**
    - Configurar las URLs base de API (diferentes para desarrollo y producción probablemente).
    - Utilizar axios/fetch en un módulo de servicio (e.g. `api.js`) que tenga métodos: `api.login()`, `api.register()`, `api.getQuestions()`, etc., para centralizar llamadas. Incluir `Authorization` header con el token si está disponible automáticamente (axios interceptors o adjuntando manual).
    - Manejar errores globalmente: si el backend devuelve 401 en alguna llamada (token expirado), interceptar y forzar logout del usuario en frontend.
    - Socket.IO: configurar la conexión al servidor de sockets en un módulo `socket.js`. Posiblemente iniciar la conexión solo cuando necesario (por ejemplo, al entrar a modo duelo) para ahorrar recursos. Asegurar de incluir el token JWT en la query de conexión para autenticar.
    - Probar todas las integraciones con el backend real o simulado (puede usarse un mock server primero con datos ficticios para ajustar el frontend).

32. **Testing y QA en Frontend:** Realizar pruebas manuales cubriendo:

    - Registro/login flows (incluyendo manejo de errores).
    - Juego individual completo (varias veces, verificar puntajes calculados correctos, timeouts).
    - Enfrentamiento con dos navegadores en modo duelo (en entorno dev, abrir dos ventanas, dos cuentas diferentes, hacer que se emparejen y jugar para ver sincronía).
    - Ver que la app PWA se pueda instalar en móvil (aparezca prompt de "Add to Home Screen"), y que offline muestre al menos la pantalla de inicio.
    - A/B test con idiomas cambiando la configuración.
    - Revisar que ninguna llamada inapropiada se hace cuando no debe (p.ej. invitado intentando acceder ranking -> quizás redirigir a login).

33. **Optimización y Deployment del Frontend:** Ejecutar build de producción (minificación, etc.). Subir el build estático a un hosting (ej. vercel/netlify). Configurar dominio/URL base correcta para llamadas API (posiblemente usar variables de entorno en frontend para API URL). Verificar tras despliegue que todo funciona en la URL pública (especialmente las comunicaciones con backend, CORS configurado correctamente en backend para permitir el dominio del frontend). Registrar el service worker para PWA y comprobar que se cachean archivos.

Cada una de estas tareas puede ser tomada como un **prompt** para una IA programadora (o como subtarea para un desarrollador humano), asegurando que el desarrollo avance de manera estructurada y coherente con la especificación. Al completar todas las tareas, se obtendrá un juego web de geografía completo, escalable y listo para atraer a jugadores en dispositivos móviles y web, reproduciendo la esencia del clásico Geo Challenge adaptado a tecnologías modernas.

Referencias Utilizadas: Las decisiones arquitectónicas y de diseño en esta especificación se basaron en buenas prácticas y recomendaciones actuales en desarrollo web y juegos en tiempo real, incluyendo el uso de módulos cliente/servidor [1], la adopción de JWT para auth sin sesión [9], estrategias de sincronización multiusuario con WebSockets [6] y alternativas serverless [8], así como mecanismos de leaderboard eficientes con Redis [10] [11]. También se tuvieron en cuenta principios de diseño minimalista para la interfaz [5] y características propias de las PWA para mejorar la experiencia en móvil [4]. Todas estas fuentes respaldan la viabilidad técnica del proyecto y orientan su implementación paso a paso.

---

[1] [2] lbd.udc.es
https://lbd.udc.es/Repository/Publications/Drafts/DesdeWebint.pdf

[3] [4] How to Build Progressive Web App Games?: An Expert Guide
https://www.juegostudio.com/blog/progressive-web-app-games-how-to-create-them

[5] Minimalismo en Diseño de Juegos: Ejemplos, Consejos, e Ideas | Envato Tuts+
https://code.tutsplus.com/es/minimalismo-en-diseno-de-juegos-ejemplos-consejos-e-ideas--cms-23446a

[6] [7] [8] java - In my Online multiplayer quiz game how can i make players to join and compete at a time? - Stack Overflow
https://stackoverflow.com/questions/68607768/in-my-online-multiplayer-quiz-game-how-can-i-make-players-to-join-and-compete-at

[9] Autenticación JWT, qué es y cuándo usarla | Ciberseguridad
https://ciberseguridad.com/guias/prevencion-proteccion/autenticacion-jwt/

[10] [11] HLD: Realtime Ranking/Leaderboard for a Multiplayer Game | by Saurabh Choudhary | Medium
https://medium.com/@choudharys710/hld-realtime-ranking-leaderboard-for-a-multiplayer-game-67332a083252

[12] [13] ¿Pueden los juegos en tiempo real que usan conexiones WebSocket o TCP ser arruinados por jugadores con mala conexión a internet? : r/AskProgramming
https://www.reddit.com/r/AskProgramming/comments/p7l8f4/can_realtime_games_that_use_websockets_or_tcp/?tl=es-es