SportHub – Documento para Cliente

Sistema Web de Gestión de Centros Deportivos y Reservas
Fecha: 30/10/2025
Equipo: Javier · Rares · Pablo · Mario

1. Resumen ejecutivo

SportHub es una plataforma web para la reserva y gestión de instalaciones deportivas. Esta edición, orientada a cliente y al ámbito académico, elimina cualquier integración de pagos y correo SMTP. La autenticación y gestión de identidad se realiza con Auth0 (OIDC), mientras que los datos, tiempo real y almacenamiento se gestionan con Supabase (PostgreSQL, Realtime, Storage, Edge Functions).

El objetivo del MVP es ofrecer una experiencia fluida para jugadores y administradores de centro: descubrir centros, consultar disponibilidad sin choques de horario, reservar/cancelar dentro de política y administrar la operativa del centro — todo con seguridad por roles y actualizaciones en tiempo real in-app.

2. Alcance

Incluido (MVP):

- Gestión de usuarios y perfiles por rol (player, center_admin, coach, referee, admin).
- Centros e instalaciones: catálogo, horarios base y bloqueos puntuales.
- Disponibilidad y reservas con prevención de doble reserva a nivel de BBDD.
- Panel de administración de centro y backoffice global.
- Notificaciones in-app (Realtime)
- Soporte multilenguaje (ES/EN) y diseño responsive.

Fuera de alcance (podrá considerarse más adelante):

- Cobros online y reembolsos (ningún proveedor activado en esta versión).
- Envío de correos (SMTP/servicios de e-mail).
- Sincronización con calendarios externos y wearables.

3. Detalle funcional y técnico

Especificación Técnica – Sistema Web de Gestión de Centros Deportivos y Reservas

Introducción y Alcance del Proyecto

Este documento describe la arquitectura y el diseño técnico de una aplicación web para gestión de centros deportivos y reserva de actividades. El sistema permitirá gestionar usuarios (jugadores, administradores de centros deportivos, entrenadores, árbitros), espacios deportivos (instalaciones y canchas), horarios y reservas en tiempo real, pagos en línea, cancelaciones, historial de reservas y notificaciones. También se incluyen un panel de administración para los centros deportivos y un backoffice general para estadísticas, promociones y gestión de personal. El sistema soportará multilenguaje para ofrecer la interfaz en varios idiomas.

Fuera de alcance (próximas fases): Integración con wearables, sincronización con calendarios externos (Google Calendar, etc.) o sistemas avanzados de matchmaking entre jugadores. No se implementarán en la primera versión, pero la arquitectura se diseña pensando en facilitar estas ampliaciones en el futuro. En particular, la base de datos y los componentes de software serán modulares y escalables, permitiendo incorporar nuevas funcionalidades sin refactorizaciones mayores.

Tecnologías principales: Frontend en React (JavaScript/TypeScript) para una SPA responsiva; Backend en Node.js con Express para exponer una API REST segura; Base de datos PostgreSQL gestionada a través de Supabase, que provee autentificación, almacenamiento de archivos, funciones serverless (Edge Functions) y capacidades realtime. Se optó por Supabase por ser una plataforma abierta y escalable que incluye los servicios esenciales (BBDD relacional, auth JWT, storage, WebSockets, etc.) de forma integrada, acelerando el desarrollo. Opcionalmente se emplearán bibliotecas y frameworks complementarios como Tailwind CSS para la interfaz de usuario, y posiblemente TypeScript en frontend y backend para mejorar la calidad y mantenibilidad del código (estas elecciones se justifican más adelante).

A continuación, se detallan los requisitos técnicos del frontend, backend y base de datos, seguidos de consideraciones de arquitectura, seguridad y buenas prácticas.

Arquitectura General del Sistema

La solución seguirá una arquitectura de cliente-servidor tradicional, con separación clara de responsabilidades:

Aplicación Frontend (React): Aplicación de página única (SPA) que se ejecuta en el navegador del usuario. Gestiona la interfaz de usuario, interacciones y lógica de presentación. Consume la API REST del backend para las operaciones (reservas, pagos, etc.) y puede conectarse directamente a algunos servicios de Supabase (p. ej. suscripción realtime a cambios de la base de datos). La

app almacena un token de sesión (JWT) tras la autenticación y lo usa en las peticiones al servidor.

Base de Datos (PostgreSQL via Supabase): Almacena de forma centralizada todos los datos de la aplicación: usuarios, instalaciones, reservas, pagos, etc. Supabase proporciona un Postgres gestionado con Row Level Security (RLS) para control de acceso a nivel de fila, lo que permite garantizar que cada usuario solo acceda a los datos que le corresponden. También se utilizarán foreign keys y constraints en la base de datos para asegurar la integridad referencial (por ejemplo, borrar o actualizar en cascada reservas si se elimina un usuario o una instalación). Supabase aporta adicionalmente un servicio en tiempo real que hace broadcast de los cambios en las tablas via WebSockets, y un servicio de almacenamiento de archivos (S3-compatible) para guardar fotos y documentos de usuarios o centros.

Administración y Backoffice: La aplicación distinguirá entre diferentes roles de usuario. Los administradores de centro deportivo tendrán acceso a una sección de administración propia dentro de la app (o en una app separada si se decide) donde pueden ver y gestionar sus instalaciones, horarios, reservas y personal. Adicionalmente, habrá un rol de administrador global (backoffice) con permisos para ver estadísticas globales, crear promociones en toda la plataforma y gestionar usuarios de todos los tipos. Estas interfaces administrativas pueden implementarse como partes del mismo frontend de React, mostrando u ocultando rutas según el rol del usuario autenticado, o incluso como una aplicación separada si se requiere aislar mejor la administración.

A grandes rasgos, el flujo será: un usuario inicia sesión → el frontend almacena el JWT emitido por Auth0 → las peticiones a la API Node/Express incluyen este token → el backend valida el token (firma JWT) y lo usa para identificar al usuario → la lógica del servidor y las políticas RLS de la BBDD combinadas garantizan que el usuario solo modifica/lee lo que debe. Para ciertas operaciones simples (por ejemplo, obtener listados públicos de instalaciones disponibles, o suscribirse en tiempo real a actualizaciones de una cancha), el frontend podría comunicarse directamente con Supabase (usando la biblioteca @supabase/supabase-js) aprovechando que Supabase expone un API REST generada automáticamente (PostgREST) y el servicio de realtime. Sin embargo, para la mayoría de acciones críticas (reservas, pagos, etc.), el patrón recomendado es pasar por la API de Node para encapsular la lógica de negocio adicional (p. ej. iniciar un cobro antes de crear una reserva, o aplicar validaciones complejas).

Escalabilidad: Esta arquitectura soporta crecimiento: se pueden escalar horizontalmente los servidores Node sin estado y Supabase puede escalar la base de datos verticalmente y mediante replicas de solo lectura si aumenta la carga. La separación de servicios (auth, storage, funciones, BBDD) de Supabase sigue principios de microservicio bajo el capó, aunque para el desarrollador funciona como una solución unificada. Adicionalmente, la modularidad permitirá integrar futuros microservicios (por ejemplo, un servicio de análisis de datos de wearables o un motor de matchmaking) comunicándose vía APIs o colas de mensajes con el sistema central.

En resumen, se ha optado por una arquitectura modular, segura y preparada para el tiempo real, usando herramientas open-source modernas. A continuación, se detalla la implementación del frontend, backend y base de datos por separado.

Frontend – Aplicación React

El frontend será una Single Page Application construida con React (posiblemente usando Vite para inicializar el proyecto, por su rapidez). Se prioriza una arquitectura de componentes reutilizables, un manejo claro del estado de la aplicación y una organización que facilite escalar la base de código. A continuación, se describen los aspectos clave:

Librería UI y Estilos: Se propone utilizar Tailwind CSS como framework de estilos por su enfoque de utilidad que acelera el desarrollo y asegura consistencia en diseños responsivos. Tailwind permite definir estilos directamente en las clases CSS de los componentes, evitando escribir hojas CSS manualmente, y cuenta con un sistema de theming configurable para mantener la identidad visual. Esta elección está fundamentada en su facilidad de uso y en la calidad que aporta al desarrollo (código más conciso, menos context-switching entre HTML y CSS, y excelente soporte responsivo). Alternativamente, se valoró Material UI (MUI) por sus componentes preconstruidos de alta calidad, pero se optó por Tailwind para mayor flexibilidad de diseño y rendimiento (MUI agrega cierta carga de JS y CSS adicional). En caso de necesitar componentes complejos listos para usar, se pueden incorporar librerías compatibles con Tailwind, como Headless UI o DaisyUI, que proveen componentes accesibles estilizados con utilidades Tailwind.

Estructura de carpetas del proyecto: Se seguirá una estructura modular y limpia dentro de src/, inspirada en buenas prácticas de proyectos React modernos:

Ejemplo de estructura de carpetas en un proyecto React (organización por tipo y funcionalidad).

src/pages/: Componentes de página, cada uno representando una ruta principal de la aplicación (ej. Página de Inicio, Panel de Usuario, Pantalla de Reservas, Panel de Admin Centro, etc.). Cada página puede componerse a su vez de secciones o componentes específicos.

src/components/ (dentro de una posible carpeta shared/): Componentes reutilizables y genéricos de la interfaz, como botones, cuadros de diálogo, formularios, calendario/selector de fecha-hora, lista de reservas, etc. Siguiendo un enfoque atomic/molecular, estos componentes son independientes y pueden usarse en distintos lugares.

src/context/: Definición de contextos de React para proveer estado global. Por ejemplo, un AuthContext para el estado de autenticación (usuario actual, token, roles) accesible por todo el árbol, o contextos para la selección de idioma, tema, etc. .

src/hooks/: Hooks personalizados que abstraen lógica común. Por ejemplo: useAuth() para encapsular login/logout y obtención del usuario actual, useBooking() para lógica de reservas (consultar disponibilidad, realizar una reserva), useNotifications() para suscribirse a

notificaciones en tiempo real de Supabase, etc. Estos hooks mejoran la reutilización y mantienen los componentes más limpios.

src/utils/: Utilidades y helpers puros (formateadores de fecha, validadores, conversión de zonas horarias, etc.).

src/assets/: Recursos estáticos como imágenes, iconos SVG, etc., de ser necesarios (aunque con storage de Supabase, muchas imágenes de usuarios se cargarán dinámicamente).

src/styles/: Configuración global de estilos, por ejemplo archivos CSS globales o definiciones de temas. Con Tailwind, aquí residirá el archivo tailwind.config.js y posiblemente un CSS base que importe las directivas de Tailwind.

Adicionalmente, si usamos TypeScript, una carpeta o archivo para types/interfaces para definir los tipos de datos (por ejemplo, interfaz Reservation con campos id, fecha, usuario, etc.).

Esta estructura modular garantiza mantenibilidad en el largo plazo, agrupando archivos por funcionalidad y responsabilidad, y es adaptable según crezca el proyecto.

Rutas y Navegación: Se usará React Router para definir la SPA routing. Las rutas principales podrían ser:

Publicas (no requieren login): /login, /register (acceso y registro de usuarios), quizás páginas informativas como inicio público o ayuda.

Privadas (requieren autenticación):

Jugador estándar: Pantalla principal con listado de centros deportivos o deportes disponibles, opción para filtrar y seleccionar un centro > luego ver sus instalaciones y horarios disponibles. Ej: /centros (lista de centros) → /centros/:id (detalle de centro, con instalaciones y agenda) → proceso de reserva.

Mis Reservas: página donde el jugador ve su calendario personal de reservas pasadas y futuras, con opciones de cancelación si la política lo permite.

Admin de Centro: rutas bajo /admin-centro (o similar) que incluyan dashboard con estadísticas básicas (reservas del día, próximos eventos), listado CRUD de instalaciones (canchas) de ese centro, gestión de horarios especiales (ej. cierres por mantenimiento o eventos), lista de reservas en ese centro (con posibilidad de cancelarlas o comunicarse con jugadores), sección para gestionar entrenadores/árbitros asociados (invitarlos al sistema, asignarlos a actividades) y posiblemente configurar promociones locales (descuentos para su centro).

Backoffice Admin Global: rutas bajo /admin con panel general (métricas de toda la plataforma: número de usuarios, reservas totales, ingresos, etc.), gestión de promociones globales (p.ej. cupones de descuento para todos los centros o destacar ciertos centros), gestión de usuarios

(ver/bloquear usuarios problemáticos), y gestión de personal global si aplica (por ejemplo, lista de todos los entrenadores/árbitros registrados, posiblemente validación de perfiles).

Multilenguaje en rutas: Si se desea SEO multilenguaje o URLs específicas, podríamos incluir el código de idioma en la ruta (ej. /es/centros, /en/centers), aunque no es estrictamente necesario para funcionamiento interno. En este punto, priorizaremos la funcionalidad; la localización de rutas se puede agregar después si se requiere.

El React Router protegerá las rutas privadas mediante un middleware de ruta o componente <PrivateRoute> que verifique la autenticación (por ejemplo, chequeando AuthContext o la existencia de un token válido) antes de renderizar la página. Asimismo, para roles, se implementarán componentes de orden superior o hooks que permitan redirigir si el rol del usuario no coincide (ej: un usuario jugador intentando acceder a /admin sería redirigido fuera).

Gestión de Estado: Dada la naturaleza de la aplicación, se usará principalmente el estado local de React y Context API para estados globales sencillos (usuario autenticado, configuraciones). Muchas de las datos (reservas, instalaciones, etc.) vendrán de la base de datos, por lo que para evitar re-fetching manual y manejo de caché, es recomendable usar una librería como React Query (react-query) o SWR. Por ejemplo, React Query puede ayudarnos a obtener la lista de instalaciones de un centro y cachearla, actualizándola eficientemente cuando haya cambios (incluso puede integrarse con las notificaciones realtime de Supabase para hacer refetch automático). Para estados muy simples quizá no haga falta, pero dado que tendremos varias entidades (users, facilities, bookings, etc.), una herramienta de gestión de datos remotos es útil. No se prevé necesario usar Redux en este punto (sería añadir complejidad innecesaria) a menos que el estado global crezca mucho; de ser así, se podría evaluar Redux Toolkit o alternativas ligeras como Zustand, pero inicialmente Context + React Query cubren las necesidades con menos código boilerplate.

Multilenguaje (Internacionalización): Se implementará soporte multiidioma usando la librería react-i18next (basada en i18next). Esta librería permite definir archivos de traducción JSON para cada idioma y proporciona hooks (useTranslation) para traducir textos en los componentes de forma sencilla. Se mantendrá una estructura de carpetas como public/locales/{lang}/translation.json para los textos estáticos. El idioma predeterminado se puede detectar con i18next-browser-languagedetector (por ejemplo según la configuración del navegador), y ofrecer al usuario un selector de idioma en la interfaz. Gracias a react-i18next, es posible cambiar de idioma dinámicamente sin recargar la página, persistiendo la ruta actual. En el código se cuidará que los textos no estén codificados fijos, sino referenciados por claves de traducción. Asimismo, cualquier mensaje de error o validación que venga del backend se podrá mapear a mensajes traducibles. Dado que el cliente tiene usuarios en España, se incluirá al menos español e inglés desde el lanzamiento.

Componentes y UX notable: La aplicación incluirá componentes específicos para la domain deportiva, p. ej.:

Calendario/Picker de horarios: para mostrar los slots horarios disponibles en una instalación. Podría usarse un componente de calendario semanal donde el usuario ve qué horas están libres/ocupadas en tiempo real (actualizándose vía websockets cuando otro usuario reserva). Alternativamente, un listado de franjas horarias por día con indicador de disponibilidad.

Formulario de reserva: para confirmar la reserva elegida, seleccionar si es reserva simple o con entrenador (si esa funcionalidad existe), aplicar código promocional, etc. Incluirá resumen de precio y un botón de pago.

Notificaciones al usuario: si un usuario tiene la app abierta, podría recibir notificaciones en tiempo real (por ejemplo, "tu reserva de mañana ha sido confirmada" o "tu reserva ha sido cancelada por el administrador"). Esto se implementa ya sea con toast notifications en la UI y/o mediante la API de notificaciones del navegador (si el usuario dio permiso). Para ello, el frontend podría escuchar eventos via Supabase Realtime: Supabase puede emitir eventos cuando se inserta/actualiza una fila, por ejemplo en la tabla de reservas. El cliente se suscribiría a esos eventos filtrados por el usuario o la instalación relevante, mostrando la notificación inmediatamente.

Accesibilidad y Responsive: Se garantizará que la interfaz siga buenas prácticas de accesibilidad (atributos ARIA en botones, navegación por teclado, etc.) y que sea responsive para usarse en móviles. Muchas reservas pueden gestionarse sobre la marcha, así que la UI deberá funcionar bien en smartphones. Tailwind facilita esto con sus utilidades responsive mobile-first.

Buenas prácticas de Frontend: Se implementará formateo de código consistente (Prettier) y linting (ESLint con reglas recomendadas de React/JS). Se compondrán los componentes para evitar duplicación (DRY) y se añadirá tipado (TypeScript) para evitar errores comunes. También se considerarán pruebas unitarias (con Jest/React Testing Library) en componentes críticos (ej. el componente de calendario de reservas) y pruebas end-to-end (con Cypress) para flujos clave como “reservar y pagar” o “cancelar reserva”, para garantizar calidad.

En resumen, el frontend React se construirá con énfasis en modularidad, experiencia de usuario fluida y soporte internacional, usando herramientas modernas para estado, estilos y pruebas.

Backend – API Node.js con Express

El backend será una aplicación Node.js usando el framework Express para estructurar una API RESTful clara y mantenible. A continuación, se describe la arquitectura interna del backend, sus componentes y cómo interactúa con Supabase y otros servicios:

Estructura de proyecto y organización del código: Se optará por una estructura en capas: rutas, controladores, servicios (o modelos de datos) y middlewares, inspirada en guías de Express reconocidas. En la raíz del proyecto residirá el fichero principal (por ej. index.js o app.js) que inicia el servidor Express y carga la configuración. Se crearán directorios:

/routes: agrupará las definiciones de endpoints por módulo (ej: auth.js, users.js, centers.js, bookings.js, payments.js, admin.js...). Cada archivo define las rutas HTTP y las enlaza con un controlador correspondiente.

/controllers: contiene las funciones controladoras que manejan la lógica de cada endpoint (reciben req, res). El controlador se encarga de procesar la petición, llamar a los servicios o la capa de datos necesaria, y devolver la respuesta HTTP adecuada (JSON, códigos de estado, etc.). Por ejemplo, bookingsController.createBooking(req,res) validará la entrada, llamará a un servicio de reserva y enviará la respuesta con éxito o error.

/services (o /models): implementa la lógica de negocio y el acceso a datos. Aquí es donde interactuaremos con Supabase. Cada servicio puede corresponder a una entidad del negocio: p. ej. BookingService tendrá métodos como create(data), cancel(bookingId,user), listByUser(userId), etc., que internamente usarán el SDK de Supabase (u otro método) para ejecutar las consultas SQL necesarias. Separar esta capa permite cambiar la implementación de datos (por ejemplo, podríamos sustituir supabase-js por llamadas directas con pg o por otro ORM en el futuro) sin afectar a controladores.

/middlewares: código de middleware de Express reutilizable. Aquí habrá, por ejemplo, un middleware de autenticación que valide el JWT de Auth0 en cada petición protegida y adjunte los datos del usuario autenticado a req (o rechace con 401 si inválido). También middlewares de validación de datos (por ejemplo usando celebrate/joi para validar req.body en ciertas rutas), middleware de logging (morgan) para registrar las peticiones, cors para permitir peticiones del frontend, y helmet para configurar cabeceras de seguridad HTTP.

Esta estructura modular coincide con recomendaciones generales para APIs Node, facilitando escalabilidad y orden.

Integración con Supabase (Base de Datos y Auth): Dentro del backend, utilizaremos la biblioteca oficial @supabase/supabase-js (en entorno Node) para interactuar con la base de datos y servicios Supabase. En el arranque de la aplicación, se creará un cliente supabase usando la URL y la API Key de servicio (la clave secreta con todos los permisos) desde variables de entorno. Este cliente nos permite hacer consultas como supabase.from('bookings').select(...).eq(...), inserciones, actualizaciones, etc., de forma similar a como se haría desde el frontend, pero aquí podemos usar la Service Role Key para omitir las restricciones de RLS cuando sea necesario (por ejemplo, ciertas operaciones administrativas).

Sin embargo, se respetarán las reglas de seguridad: para operaciones de usuario normal, preferiremos aplicar las políticas RLS y/o validaciones manuales para garantizar que, incluso con la clave de servicio, no se alteren datos indebidos. Por ejemplo, al obtener reservas, podríamos usar el token JWT del usuario con supabase-js para que aplique RLS automáticamente, o filtrar en la consulta por el userId. Una estrategia es: cuando el middleware de auth valide el JWT, obtendremos de él el user.id y user.role, y luego en los servicios usaremos esos datos para filtrar. Alternativamente, podemos inicializar un supabase client con el JWT de usuario

(supabase.auth.setAuth(token)) por petición, de modo que las llamadas .from() a la DB respeten RLS como si fuera el cliente directamente. Esto añade seguridad en profundidad. Por simplicidad, podría empezar filtrando manualmente con el userId en consultas (ya que tendremos dicho id siempre en backend).

La integración de Auth0 implica que no gestionaremos contraseñas manualmente en la base de datos (Auth0/Gotrue se encarga de almacenar hashes, verificación de email, etc.). Los flujos de registro/login podrán hacerse desde el frontend utilizando supabase (por ejemplo, supabase.auth.signUp(email,password)), y el backend solo recibirá peticiones autenticadas con el token. También se puede exponer en el backend rutas para registro/login que internamente llamen a la API de Auth0 (usando el Admin API), pero dado que el frontend ya puede hacerlo directamente, no es obligatorio duplicar eso. Lo que sí haremos en backend es al momento de registro crear la entrada correspondiente en nuestro tabla de perfiles (profile/usuario) para guardar datos adicionales como nombre, rol, etc., que Auth0 no guarda por defecto. Esto se puede hacer aprovechando los triggers de Supabase (una función que se ejecute tras crear un usuario nuevo en auth.users) o manejándolo en la lógica de registro del backend.

Endpoints principales (Routing y Controladores): Se definirán rutas RESTful claras para cada recurso:

Auth: (si se decide manejar en backend) endpoints como POST /auth/register y POST /auth/login que usen Auth0. Aunque podría delegarse al cliente, podríamos implementarlos para, por ejemplo, permitir registro de distintos tipos de usuario con un solo endpoint que luego llame a supabase y establezca el rol.

Usuarios: GET /usuarios/me para obtener mi perfil, PUT /usuarios/me para editar datos (nombre, foto, etc.), endpoints de admins para listar usuarios o cambiar roles.

Centros deportivos: GET /centros (lista todos los centros con info pública), POST /centros (crear nuevo centro – solo admins globales o mediante flujo especial), GET /centros/:id (detalle de un centro, incluyendo instalaciones, horario base, etc.), PUT /centros/:id (editar info – solo admins del centro o global).

Instalaciones (espacios deportivos): GET /centros/:id/instalaciones (lista las canchas/espacios de un centro), POST /centros/:id/instalaciones (admin centro crea nueva cancha), PUT /instalaciones/:id (editar características, p.ej. nombre, tipo, precio base por hora, etc.), DELETE /instalaciones/:id.

Horarios/Disponibilidad: Esto podría ser parte de instalaciones. Quizá un endpoint GET /instalaciones/:id/disponibilidad?fecha=YYYY-MM-DD que devuelva los slots horarios disponibles ese día, calculado a partir de horario de apertura del centro y reservas existentes. Alternativamente, si se almacenan slots en la BD, un GET /instalaciones/:id/slots directo. Para bloquear horarios (ej. mantenimiento), el admin centro podría crear "reservas especiales"

marcadas como bloqueos o tener un endpoint POST /instalaciones/:id/bloqueos con fecha/hora a bloquear.

Reservas:

POST /reservas: crear una nueva reserva. El cuerpo incluirá instalación, fecha y hora deseada, y opcionalmente otros detalles (ej. si reserva con entrenador tal vez). La lógica aquí será crítica: verificar que el slot sigue libre (condición que también se asegura con constraint único en BD), posiblemente realizar el cobro antes de confirmar (flujo de pago), crear la reserva en BD, enviar notificación. Se hará dentro de una transacción lógica: si el pago falla, no se crea reserva; si reserva no disponible, se aborta antes del pago.

GET /reservas?centroId=X&fecha=...: para admin centro, obtener reservas de su centro en cierto rango (o todas futuras). Para usuario normal, quizás GET /reservas?usuarioId=yo para sus reservas (o usar /usuarios/me/reservas).

Administración Centro: GET /admin-centro/resumen (datos de reservas recientes, etc.), GET /admin-centro/reservas (todas reservas de mi centro), POST /admin-centro/entrenadores (invitar o agregar un entrenador a mi centro), etc. Estas rutas internamente usan las mismas entidades pero filtrando por el centro del usuario (usando req.user.centerId quizás).

Backoffice Admin: GET /admin/estadisticas (consolidar métricas como total de reservas plataforma, ingresos totales, etc.), POST /admin/promociones (crear un código de promoción global), GET /admin/usuarios?rol=coach (listar entrenadores registrados, etc.), PUT /admin/usuarios/:id (cambiar rol o estado de un usuario).

Todas estas rutas estarán documentadas y seguirán convenciones REST en la medida de lo posible (usando verbos HTTP adecuados, códigos 200/201 en operaciones exitosas, 400/401/403 para errores de validación o permisos, etc.).

Lógica de negocio y Reglas clave: En los controladores/servicios se implementarán reglas como:

Evitar dobles reservas: antes de crear una reserva, verificar en BD que no existe ya una reserva para esa instalación en ese intervalo. Además de la verificación en código, se define un índice único en la tabla de reservas (facility_id, fecha, hora debe ser único) para prevenir condiciones de carrera a nivel de base de datos. Si dos peticiones concurrentes intentan la misma hora, una fallará por la constrain única si el código no la atrapó.

Política de cancelación: no permitir cancelaciones el mismo día de la reserva (por ejemplo). Esto se chequeará en DELETE /reservas/:id mirando la fecha actual vs la fecha de la reserva. Si viola la política, devolver error 400 con mensaje adecuado.

Gestión de roles y permisos: A nivel de backend, además de las políticas RLS, implementaremos checks de autorización en los endpoints. Es decir, aunque el JWT indica user id y tal, no confiamos solo en RLS de supabase, sino que en la capa de servicios verificamos: si un jugador

intenta acceder a reservas de otro, lanzamos 403; si un coach intenta crear un centro, 403; etc. Esto se hará comparando req.user.role y los parámetros. Por ejemplo, en getReservasDeCentro(centroId), comprobamos que el req.user es admin de ese centroId (mediante un campo o mediante consulta). Estas validaciones aseguran Defense in Depth junto con RLS.

Row Level Security en Supabase: Se habilitará RLS en las tablas sensibles y se escribirán policies para reflejar las reglas de permisos:

En tabla de reservas: policy para SELECT que permita a usuarios autenticados ver reservas donde user_id == auth.uid() (sus propias) o si tienen rol admin centro y la reserva pertenece a su centro (esto quizás usando una función definida que chequea centro). Para admins globales, se puede decidir que pasen por el backend con la service key (ya que ellos pueden ver todo). Similarmente, policy de INSERT en reservas que permita a cualquier usuario autenticado crear reserva (ya que la lógica de negocio adicional la controlamos en backend, podemos permitirlo a nivel BD con constraints) – o incluso se puede restringir a que user_id del insert sea auth.uid().

En profiles/usuarios: permitir a cada usuario ver/editar solo su perfil; y admins ver/editar todos. Esto se suele lograr con un campo role o is_admin en el perfil y políticas que chequean eso.

En instalaciones: permitir SELECT libre (todos pueden ver instalaciones disponibles), pero INSERT/UPDATE solo al admin de ese centro. Esto se puede lograr almacenando en la tabla un campo center_id y en la policy comprobar que auth.uid() es admin de ese center (quizá comparando con un campo en profiles o una tabla relacional).

En centros: SELECT libre (lista centros para todos), UPDATE solo si el usuario es admin global o corresponde al centro (similar lógica).

En pagos: quizás solo admin global puede ver todos, un usuario podría ver sus propios pagos (vinculados por user_id).

En promociones: posiblemente solo admins globales o admins de centro (si promociones locales).

Cabe destacar que Supabase proporciona funciones helper para comprobar roles JWT, o podemos codificar IDs de roles en JWT de Auth0 (mediante JWT custom claims si quisiéramos). Pero dado que manejamos roles en nuestra tabla profiles, con RLS podemos usar subconsultas en USING para verificar permisos.

La implementación cuidadosa de RLS añadirá una capa extra de seguridad en caso de que accidentalmente se exponga la API supabase directa. Sin embargo, en la arquitectura propuesta, la mayoría de accesos pasan por el backend (que ya valida), por lo que RLS es refuerzo. Se aprovechará la integración nativa de RLS con Auth0 para lograr seguridad extremo a extremo desde el navegador hasta la base de datos.

En caso de fallo de pago, se cancela la reserva (liberando el slot).

Notificaciones y Comunicaciones: El backend gestionará el envío de notificaciones importantes:

SMS/WhatsApp: Opcional, para notificaciones de última hora o verificaciones, podríamos integrar Twilio si se desea. No es prioritario para MVP.

Notificaciones push/web: Si hacemos una PWA o similar, podríamos usar la Notification API del navegador para notificaciones in-app. Para push a móviles nativos, habría que integrar FCM/APNs, lo cual queda fuera de la versión web inicial.

Panel de Administración para Centros: En el backend, muchas de las rutas ya mencionadas servirán para el panel de centro. Por ejemplo, el administrador de centro autenticado podrá llamar:

GET /admin-centro/reservas?desde=...&hasta=... que internamente filtra reservas de su centro (usando center_id asociado a su cuenta). Esto permite poblar el calendario de reservas en su panel, mostrando ocupación de sus canchas en tiempo real.

POST /centros/:id/instalaciones para añadir una nueva cancha o editar horarios.

GET /admin-centro/personal para listar sus entrenadores/árbitros asociados, y POST /admin-centro/personal para invitarlos (quizás enviándoles un email de registro). Al crear un entrenador, podríamos crear un usuario con rol entrenador ligado a ese centro.

Vale destacar que muchas de estas operaciones son similares a las de la aplicación normal pero filtradas, y con más permisos de escritura en su ámbito. Por eso, el middleware de autenticación incluirá también la verificación de rol en muchas rutas. Por ejemplo, la ruta POST /centros/:id/instalaciones verificará que req.user.role === 'centro_admin' y que req.user.center_id == :id antes de proceder.

Backoffice para Estadísticas y Gestión Global: De forma análoga, habrá endpoints reservados a administradores globales:

GET /admin/dashboard – estadísticas globales (usuarios totales, reservas totales, top 5 centros con más actividad, etc.). Estos datos se pueden obtener con queries agregadas en Postgres, quizás exponiendo vistas materializadas para eficiencia. Por ejemplo, llevar un conteo de reservas por día, etc., para gráficas.

POST /admin/promociones – crear un nuevo código promocional global (por ej. 10% de descuento en primera reserva), con campos para código, tipo de descuento, fecha vigencia. El backend guardará esto en tabla promotions, y la lógica de uso de promociones al pagar se implementa chequeando esta tabla.

GET /admin/centros – listar todos los centros registrados, posiblemente con opciones de aprobar si hay un flujo de aprobación.

POST /admin/usuarios/:id/rol – cambiar el rol de un usuario (ascender a admin centro, etc.) o asignar un usuario como admin de un centro específico.

Gestión de personal (entrenadores/árbitros): Podría implicar aprobar solicitudes de entrenadores (si se registran solos quizás), o mantener un registro de árbitros disponibles para asignar a torneos. Dado que en MVP los árbitros/entrenadores los gestionan por centro, el backoffice global podría no necesitar intervenir mucho, salvo listar globalmente y eventualmente borrar o modificar datos.

Contenido y promociones: Si la plataforma quiere destacar ciertos eventos o mandar comunicaciones a todos, se podría tener endpoints para eso (por ejemplo, un admin crea una promoción "Black Friday" aplicable a todos los centros, o envía un mensaje global a todos los jugadores; esto requiere mecanismos de broadcast de notificaciones).

Al igual que con admin de centro, los endpoints globales estarán protegidos por middleware que verifica el rol admin. Incluso se puede separar en el routing: prefijar todas las rutas de admin con /admin y aplicar un middleware específico que compruebe req.user.role == 'admin'.

Supabase Functions & Triggers: Cabe reiterar el posible uso de Supabase Edge Functions para ciertas tareas server-side que podríamos ubicar fuera del servidor Node. Ejemplos:

Otra función podría generar reportes o realizar limpiezas (e.g., diario a medianoche marcar reservas pasadas como 'completed' o borrar notificaciones viejas). Estas funciones se pueden programar con un CRON (Supabase permite programar llamadas a edge functions).

Integración con calendarios externos (futuro): una edge function podría, dado un token OAuth de Google Calendar almacenado para un usuario, crear eventos en su calendario personal cada vez que reserva. Esto se planearía en una fase posterior, pero la arquitectura lo contempla (guardando tokens seguros en Supabase, y usando functions para no exponer credenciales en el cliente).

Seguridad y Buenas Prácticas del Backend: Además de RLS y validaciones mencionadas:

Se usarán HTTPS obligatoriamente en producción para todas las peticiones, protegiendo los tokens JWT en tránsito.

JWT: Las solicitudes deben incluir el token de Auth0 en la cabecera Authorization: Bearer <token>. El middleware de auth en Node validará la firma JWT usando la JWKS de Auth0 (disponible vía JWKS o descargable del proyecto). Esto asegura que el token no fue modificado. También se checará que no esté expirado.

CORS: Configurar CORS para permitir solo el dominio del frontend oficial en producción, evitando que scripts maliciosos en otros orígenes puedan usar nuestra API con credenciales del usuario.

Rate limiting: Implementar un middleware de limitación de solicitudes (por IP/usuario) en endpoints sensibles para prevenir abuso (por ejemplo, limitar intentos de login para evitar fuerza bruta, o spam de creación de reservas).

Logging y monitoreo: Se incluirá logging estructurado (p. ej. usando morgan para peticiones HTTP, y un logger como winston para eventos de la aplicación) para poder depurar. En producción, estos logs podrían enviarse a un servicio de monitoreo. Asimismo, se considerará instrumentar con métricas (Prometheus, etc.) para medir rendimiento si la escala lo requiere.

Testing: Se planificarán pruebas unitarias para la lógica de negocio (por ejemplo, funciones de BookingService que calculan disponibilidad) y pruebas de integración para los endpoints (usando supertest o similar para llamar a la API Express con distintos roles y ver que responde adecuadamente). Esto garantizará que las reglas de permisos y negocio funcionan como esperado.

Deployment: El backend Node se puede desplegar en un servicio de hosting Node (Heroku, Fly.io, Vercel serverless functions, etc.). Si se desplegara en Vercel como Serverless Functions, habrían que modularizar endpoints, pero quizás es más sencillo usar Heroku/Fly para mantener un proceso Node corriendo que maneje websockets también si usamos socket.io en el futuro. De momento, un contenedor Docker con la app Node podría levantarse en servicios cloud sin complicaciones.

En síntesis, el backend Express actuará como el cerebro de la aplicación, garantizando que las reglas de negocio se cumplan, que los datos fluyan correctamente entre frontend, base de datos y servicios externos, todo ello siguiendo altos estándares de seguridad y limpieza de código.

Base de Datos – Modelo en Supabase (PostgreSQL)

El modelo de datos se implementará en PostgreSQL utilizando las herramientas de Supabase (migraciones SQL o el editor de tablas). A continuación se describen las entidades principales, sus campos y relaciones en el esquema público de la base de datos:

Usuarios / Perfiles: Se manejará con dos niveles:

Auth0 crea un registro en auth.users para cada usuario registrado (contiene UUID, email, hash de password, etc.).

En paralelo, tendremos una tabla pública profiles (o usuarios) para información adicional: por ejemplo:

id (UUID, Primary Key, coincide con el ID de auth.users mediante trigger de Supabase).

nombre (texto),

rol (tipo de usuario: enum player, center_admin, coach, referee, admin),

center_id (FK opcional a Centros, si este usuario administra o pertenece a un centro),

otros datos de perfil: teléfono, foto_url (ruta en Supabase Storage para su foto de perfil), etc.

timestamps de creación/actualización.

Cuando un usuario se registra via Auth0, mediante un trigger se insertará automáticamente un row en profiles copiando su ID. La columna rol por defecto será player (jugador) a menos que el registro lo haga un admin creando otro rol. Los roles center_admin, coach, referee podrían asignarse posteriormente mediante acciones admin (o un flujo de registro distinto). Tener todos en una tabla facilita manejar un solo sistema de usuarios con distintas atribuciones.

Centros Deportivos: Tabla centers con campos:

id (UUID o BIGSERIAL PK),

nombre (nombre del complejo deportivo o club),

direccion, ciudad, etc. (ubicación para búsquedas geográficas en el futuro),

admin_user_id (FK hacia profiles.id del usuario que es administrador principal del centro),

horario_apertura y horario_cierre (por ejemplo "08:00" y "22:00" si todos los días igual, o se podría tener una tabla separada de horarios por día de semana),

quizá dias_activo (ej: Lunes-Domingo, si cierra algún día),

created_at.

Relaciones: Un centro tiene muchas instalaciones. Si admin_user_id no es null, apunta al perfil del admin; en caso de centros sin admin asignado (ej: inicialmente creado por admin global), ese campo ayuda a saber a quién transferir control.

Instalaciones Deportivas (Espacios/Cancha): Tabla facilities (o instalaciones) con:

id (UUID/serial PK),

center_id (FK references centers.id, ON DELETE CASCADE idealmente para eliminar cascada instalaciones si se borra centro),

nombre (ej: "Cancha 1", "Piscina", etc.),

tipo (ej: enum: tenis, fútbol, gimnasio, etc. para tipo de deporte/espacio),

capacidad (opcional, si relevante, ej. 4 jugadores en cancha de pádel),

precio_hora (DECIMAL, precio base por hora, podría ser null si gratuito),

facilitator_id (FK a profiles.id de un entrenador/monitor asignado permanente a esta instalación, opcional),

activo (bool, para marcar si está disponible para reservas en general),

created_at.

Indices: se indexará center_id para consultas por centro. La relación es Centro 1 - N Instalaciones. Un campo facilitator_id permite asociar un entrenador responsable (por ej. una cancha de tenis siempre tiene un entrenador disponible), pero no es esencial; se podría omitir en MVP.

Reservas: Tabla bookings para las reservas de espacios:

id (UUID o BIGSERIAL PK),

facility_id (FK a instalaciones.id),

user_id (FK a profiles.id del jugador que reserva),

fecha (DATE de la reserva),

hora_inicio (TIME o TIMESTAMP de inicio) – si el modelo es siempre slots de 1 hora exacta, con fecha+hora es suficiente; si permitimos distintas duraciones, podríamos añadir hora_fin o una duración en minutos,

estado (enum: e.g. PENDING_PAYMENT, CONFIRMED, CANCELLED, COMPLETED),

created_at (timestamp de reserva hecha),

cancelled_at (timestamp si cancelada, para registro),

price_paid (monto pagado, podría almacenarse aquí para histórico, o en tabla de pagos),

payment_id (FK opcional a tabla pagos si queremos relacionar).

Constraints y reglas: - Unicidad: Debe haber una restricción única en (facility_id, fecha, hora_inicio) para prevenir doble reservas. Si consideramos duraciones variables y almacenáramos hora_fin, entonces sería más complejo, pero asumiendo slots fijos, esta unique es ideal. - Integridad referencial: FKs con ON DELETE CASCADE en facility y user, para que si un usuario se elimina, sus reservas se borren (o ON DELETE SET NULL dependiendo de políticas; quizás mejor CASCADE o RESTRICT ya que eliminar usuarios no sería común; se puede también usar soft-delete de usuarios para no perder historial). - Indexes: índice por user_id para consultar reservas de un usuario rápidamente; índice compuesto por facility_id+fecha para buscar reservas de una instalación en una fecha (aunque unique ya sirve también como índice). - RLS policies: como mencionado, usuario solo ve sus filas; admin centro ve filas donde facility pertenece a su centro.

Pagos: Tabla payments (o integrar pagos en reservas, pero mejor separada):

id (UUID or serial),

booking_id (FK a reserva.id, quizá unique si un pago corresponde a una reserva específica),

monto (DECIMAL(6,2) por ej),

moneda (e.g. "EUR"),

status (enum: PAID, REFUNDED, FAILED),

metodo (texto: "card", "transfer", etc. o detalles si se quiere),

created_at, updated_at.

Promociones/Descuentos: Tabla promotions:

id (serial),

codigo (varchar único, el código que introduce el usuario, p. ej. "VERANO2025"),

descripcion (texto, opcionalmente multilenguaje en futuro),

tipo_descuento (enum: PERCENTAGE o FIXED),

valor_descuento (porcentaje o cantidad según tipo),

centro_id (FK si la promo aplica solo a un centro en particular, null si global),

fecha_inicio y fecha_fin (ventana de validez),

uso_maximo (opcional, si solo X usos totales),

usos_realizados (contador de usos hasta ahora, para controlar uso_maximo),

activo (bool).

Esta tabla la usarían los servicios de reserva/pago para verificar si un código es válido y calcular el descuento en el monto.

Personal (Entrenadores y Árbitros): Ya que los entrenadores y árbitros son también usuarios en profiles con rol específico, podríamos no tener tabla separada para ellos. Sin embargo, podríamos necesitar asociar entrenadores con centros (muchos a muchos si un entrenador trabaja en varios clubes). Dos enfoques:

Añadir campo center_id en profiles que se llena si el entrenador está fijo en un centro (limita 1 centro por entrenador).

Crear tabla intermedia center_staff con columnas center_id y user_id para poder asociar múltiples. Contiene quizá también rol_centro (entrenador o árbitro, aunque el rol global ya lo dice).

Para MVP, asumamos cada entrenador/ref está asociado a un único centro (simplifica, y suele ser el caso si son empleados fijos). Entonces podemos utilizar profiles.center_id para admin, coaches, refs. Nota: Ya usamos center_id para admin centro. Podríamos usarlo también para coaches/refs indicando su centro principal. Si alguno trabaja en varios, habría que extender con la tabla pivot en el futuro.

Disponibilidad y Horarios Específicos: No se definirá inicialmente una tabla de horarios por instalación, se deduce de horario de centro + reservas. Si se quisiese permitir disponibilidad por entrenador o cancha (ej: entrenador X solo está lu/mi/vi), se podrían añadir tablas:

coach_availability con coach_id, dia_semana, hora_inicio, hora_fin.

facility_closed_slots para excepciones (fechas y horas donde la instalación no está disponible). Esto para que admin centro bloquee horarios.

Inicialmente, estos detalles se pueden manejar lógicamente sin nuevas tablas: un admin que quiere bloquear un horario podría simplemente crear una "reserva fantasma" por el sistema o marcar la instalación inactiva temporalmente. Sin embargo, diseñaremos con miras a agregar estas tablas luego para un control más fino de agenda.

Historial y Auditoría: PostgreSQL ofrece extensión pgcrypto para generar UUIDs, etc., y podríamos habilitar registros de auditoría con triggers (por ejemplo, logs de quién canceló una reserva y cuándo). Por simplicidad, en este diseño se considera suficiente con campos de timestamp y algunos logs en texto. Más adelante, se podría implementar tablas de historial (por ej, booking_history que registra cada cambio en reservas).

A continuación, se presenta una visión general del modelo de datos relacional, ilustrando las tablas principales y relaciones:

Diagrama entidad-relación simplificado del sistema de reservas deportivas (inspirado en un caso de Sports Complex Booking). Se muestran las tablas de Usuarios (membership/profiles), Instalaciones (facilities), Reservas (bookings), Personal (facilitator) y Pagos, junto con sus claves y relaciones.

Como se aprecia, un usuario puede tener muchas reservas (relación 1:N de profiles a bookings), una instalación puede tener muchas reservas (facilities 1:N bookings), y cada reserva vincula un usuario con una instalación en un horario dado. Los facilitadores (entrenadores) también se almacenan como usuarios, pero en este esquema se conceptualizan en una tabla separada facilitator solo para visualizar la relación con instalaciones. En la implementación real un entrenador sería un profile con rol específico y referenciado en la instalación. La tabla de pagos está relacionada con reservas y usuarios (un pago corresponde a una reserva pagada por un

miembro). Las políticas de acceso (RLS) garantizarán, por ejemplo, que un jugador (miembro) solo vea sus propias filas en bookings, mientras que un administrador de centro podrá consultar los bookings de su centro (porque sus instalaciones le pertenecen).

Además del modelo principal, Supabase nos proporciona: - Storage (almacenamiento de archivos): Se creará al menos un bucket de almacenamiento, por ejemplo avatars para fotos de perfil de usuario, y otro documents o center_images para fotos de centros o documentos (si se requiere subir, por ejemplo, un acta de un partido para árbitros, etc., en futuras expansiones). El frontend podrá subir archivos directamente a Supabase Storage usando los permisos de usuario (configuraremos reglas de acceso: típicamente, solo usuarios autenticados pueden subir y solo a su propio espacio/nombre). Por ejemplo, una convención: avatar de user uid se guarda en avatars/{uid}.png. Supabase Storage puede restringir lectura a usuarios autenticados, pero para fotos de perfil es habitual hacerlas públicas o servirlas vía URLs firmadas. Decidiremos caso a caso: fotos de perfil quizá públicas (no son muy sensibles), mientras que documentos privados se protegerán. - Funciones y Triggers en DB: - Trigger on user signup: Supabase instala uno por defecto si se usa plantilla de profile (crea profile row). Lo ajustaremos para setear el rol default. - Trigger on booking insert: Podría llamar una función que envía notificaciones (si no lo hacemos via edge functions). - Functión check_availability(facility_id, timestamp) RETURNS boolean: se podría crear para reusar lógica de ver si un slot está libre (buscando conflicto en bookings). - Vista reservas_completas: podríamos crear una vista o materialized view que junte info de reserva con usuario e instalación (SELECT con joins) para facilitar consultas de listado en una sola llamada. Esto es útil para Backoffice o reportes.

También usaremos constraints para mantener integridad: - Cascada en deletes/updates como se mencionó (FOREIGN KEY ... ON DELETE CASCADE para bookings->user y bookings->facility). - Uniqueness en reservas para evitar duplicados. - Checks: por ejemplo, podríamos añadir CHECK que hora_inicio sea en intervalos de 30 min o 60 min válidos dentro del horario del centro – aunque eso puede ser complejo en SQL y se controla mejor en lógica de app. Otra check: en pagos, que monto >= 0.

Ejemplo de Reglas de Acceso (RLS) concretas:

Tabla bookings:

```
-- Permitir a usuario autenticado ver su reserva
CREATE POLICY "Players can view own bookings"
ON bookings FOR SELECT
TO authenticated
USING ( user_id = auth.uid() );
-- Permitir a usuario autenticado insertar reserva solo para sí mismo
CREATE POLICY "Players can create bookings for self"
ON bookings FOR INSERT
```

TO authenticated
WITH CHECK (user_id = auth.uid());

Para admin de centro, dado que no tenemos su id directamente en bookings, podríamos hacer:

-- Admin centro puede ver reservas de sus instalaciones
CREATE POLICY "Center admins can view their center's bookings"
ON bookings FOR SELECT
TO authenticated
USING (-- exists one facility for this booking that belongs to a center administered by current user facility_id IN (SELECT f.id FROM facilities f JOIN centers c ON f.center_id = c.id WHERE c.admin_user_id = auth.uid())
);

O usar un join con profiles si un admin pudiera no ser único. Similar para UPDATE/DELETE (cancelar reservas), permitirlo si user es dueño o admin del centro.

Tabla profiles:

CREATE POLICY "Each user can view own profile"
ON profiles FOR SELECT
USING (id = auth.uid());
CREATE POLICY "Each user can edit own profile"
ON profiles FOR UPDATE
USING (id = auth.uid());

Para admin global, podríamos skip RLS usando service role en backend. O definir un campo rol='admin' y:

CREATE POLICY "Admin can view all profiles"
ON profiles FOR SELECT
USING (profiles.rol = 'admin' OR id = auth.uid());

(O un security definer function approach, pero por simplicidad podemos manejar admin via backend service key).

En cuanto a volúmenes esperados de datos, el diseño con Postgres es adecuado: podemos manejar muchos usuarios y reservas. Cada reserva es un registro pequeño; se indexarán por fechas para rendimiento. Si se prevé un crecimiento muy grande (ej. miles de reservas por minuto), se podría más adelante hacer particionamiento por fecha o por centro, pero inicialmente no es necesario.

Mantenimiento de la base de datos: Supabase ofrece backup automáticos y la posibilidad de escribir migraciones en SQL para versión controlada del esquema. Usaremos migraciones para

crear estas tablas y constraints reproduciblemente. También definiremos datos iniciales (seeding) como quizás roles por defecto o un usuario admin global.

Para concluir, la base de datos relacional propuesta cubre los elementos requeridos: usuarios (y sus roles), centros e instalaciones, reservas con horarios, pagos asociados, y soporte para notificaciones y promociones. Se han incorporado restricciones para garantizar consistencia (e.j., una regla impide dos reservas en la misma cancha y horario) y se ha pensado en la escalabilidad vertical del modelo. PostgreSQL nos da además la posibilidad de expandir funcionalidades con procedimientos almacenados, vistas materializadas para reportes, etc., conforme el sistema crezca.

Consideraciones Finales y Buenas Prácticas de Desarrollo

Además de lo expuesto, se seguirán estas buenas prácticas generales durante el desarrollo:

Control de versiones y despliegue: Se utilizará Git para el control de versiones. Se pueden configurar workflows de CI/CD (por ejemplo, GitHub Actions) para ejecutar tests en cada commit y desplegar automáticamente en entornos de staging/producción. El despliegue del frontend podría ser en servicios como Vercel, Netlify o GitHub Pages (ya que es una SPA estática una vez construida). El backend podría desplegarse en un servicio de Node (Heroku, Fly, AWS) o contenedores Docker en un pequeño VPS según escala. Dado que Supabase es un servicio gestionado, no hay que ocuparse de la base de datos aparte de la configuración inicial; sí es importante mantener las claves de API seguras en los entornos.

Documentación: Se mantendrá documentación tanto del código (comentarios donde sea no obvio) como para el usuario/admin. Podría proveerse un pequeño manual de API (una documentación Swagger u OpenAPI) para que futuros desarrolladores o integraciones externas sepan cómo interactuar con el sistema. Asimismo, documentar cómo configurar un entorno nuevo (instalar dependencias, variables de entorno necesarias, comandos para correr migraciones, etc.).

Escalabilidad futura: La arquitectura propuesta es suficientemente flexible para añadir los componentes planificados a futuro:

Un servicio de matchmaking podría implementarse como un módulo separado que lee datos de usuarios (nivel, preferencias) y genera emparejamientos, insertando reservas o sugerencias de partidos. Gracias a la base de datos centralizada, ese módulo podría ser un script externo o microservicio leyendo la DB (con una cuenta restringida o usando APIs) sin trastocar lo existente.

La integración con wearables (por ejemplo, importar datos de Fitbit/Strava) podría ser responsabilidad de un nuevo microservicio que almacena esos datos en tablas nuevas relacionadas con usuarios, sin afectar tablas de reserva. La presencia de un campo user_id común permite correlacionar fácilmente. El front podría entonces incorporar una página de estadísticas personales alimentada por esos datos.

Calendarios externos: se podría guardar en el perfil del usuario tokens OAuth de Google Calendar. Un Supabase Function o el propio backend podrían subscribirse a eventos de nuevas reservas y crear eventos en el calendario del usuario utilizando la API de Google. Igualmente, si se quisiese sincronizar disponibilidades (por ej. bloquear horarios en nuestro sistema si el usuario tiene algo en su calendario personal), se podría consultar periódicamente. Esto implica trabajo pero nada impide añadirlo ya que la arquitectura modular lo soporta.

Multipaís/Multimoneda: Si la plataforma expande a distintos países, la base de datos puede almacenar moneda por centro o reserva. El frontend puede internacionalizar no solo lenguaje sino formatos de fecha y divisa. Son consideraciones compatibles con lo diseñado (solo habría que incluir campo de moneda en precios, etc.).

Fuentes utilizadas: Las decisiones de diseño y buenas prácticas expuestas se fundamentan en documentación oficial y experiencias previas en sistemas similares, por ejemplo en software de reserva deportiva con disponibilidad en tiempo real, integraciones de pago en línea y control de acceso a datos, así como en guías de estructura de proyectos React y Node exitosos. La arquitectura propuesta aprovecha las fortalezas de Supabase (Postgres + Auth + Realtime + Storage) para replicar capacidades de sistemas robustos de backend, manteniendo al mismo tiempo la flexibilidad de un backend propio en Node para la lógica específica del dominio. Con todo ello, se garantiza que el sistema cumplirá con los requisitos actuales y podrá escalar funcionalmente en el futuro.

4. Identidad y seguridad (Auth0 + Supabase)

La autenticación se realiza con Auth0 (OIDC). El frontend obtiene un JWT de Auth0 y lo envía en cada petición al backend. El backend valida la firma mediante JWKS de Auth0, comprueba expiración y aplica autorización por rol. Los datos residirán en Supabase (PostgreSQL) con políticas RLS selectivas para tablas sensibles (p. ej., bookings, profiles), añadiendo defensa en profundidad.

Las actualizaciones en tiempo real se implementan con Supabase Realtime. Las notificaciones al usuario se muestran in-app (por ejemplo, avisos de reserva confirmada o cancelada dentro de la interfaz), sin uso de correo.

5. Endpoints indicativos (sin /pagos ni emails)

- Auth: gestionado en frontend con Auth0; el backend solo valida JWT.
- Usuarios: GET /usuarios/me, PUT /usuarios/me, (admin) GET/PUT /admin/usuarios/:id.
- Centros: GET /centros, GET /centros/:id, POST/PUT /centros/:id.
- Instalaciones: GET /centros/:id/instalaciones, POST /centros/:id/instalaciones, PUT/DELETE /instalaciones/:id.
- Disponibilidad: GET /instalaciones/:id/disponibilidad?fecha=YYYY-MM-DD.

- Reservas: POST /reservas, GET /reservas?usuarioId=..., GET /reservas?centroId=..., DELETE /reservas/:id.
- Admin-Centro: GET /admin-centro/resumen, GET /admin-centro/reservas, POST /admin-centro/personal.
- Backoffice: GET /admin/estadisticas, POST /admin/promociones, GET /admin/centros.

6. Requisitos no funcionales

- Seguridad: HTTPS, validación de JWT (JWKS Auth0), CORS restringido, Helmet, rate limiting y RLS selectiva en BBDD.
- Calidad: TypeScript, ESLint + Prettier, tests (Jest/RTL, Supertest, Cypress).
- Rendimiento: API stateless, caché en cliente (React Query/SWR), índices y EXPLAIN en Postgres.
- Accesibilidad y UX: ARIA, navegación por teclado, responsive mobile-first.
- Observabilidad: logs estructurados y trazas básicas; métricas según necesidad.

7. Supuestos y riesgos

- Integraciones de correo/pagos descartadas en el MVP: pueden afectar a flujos futuros si se reactivan.
- Dependencia de Auth0 para OIDC: se mitigará con documentación de fallback y pruebas de renovación de tokens.
- Uso de RLS: requiere políticas bien testeadas para evitar bloqueos inesperados.

8. Equipo

Javier · Rares · Pablo · Mario