SportHub – Documentación Unificada

Repositorio Github: https://github.com/UAX-2025-26/SportHub.git

Página en Producción: sporthub-frontend-j94s.vercel.app

Integrantes del Grupo:

- Javier Yustres
- Rares Radu
- Pablo Lozano
- Mario Blanco

Documentación del proyecto SportHub. Contiene el resumen ejecutivo, el alcance, la arquitectura propuesta, los modelos y diagramas (casos de uso, actividad, secuencia y clases), el esquema de datos en SQL, endpoints indicativos, requisitos no funcionales, equipo y licencia.

- Índice
  - a. Resumen ejecutivo
  - b. Alcance (incluido / fuera de alcance)
  - c. Arquitectura general y tecnologías
    - 3.0 Stack Tecnológico Actual
    - 3.1 Principios de Diseño Aplicados
    - 3.2 Integración Backend-Frontend
  - d. Modelos y diagramas
    - 4.1 Diagrama de Casos de Uso
    - 4.2 Diagrama de Actividad: Realizar una Reserva
    - 4.3 Secuencia: Realizar una reserva
    - 4.4 Diagrama de clases
  - e. Modelo de datos (SQL en Supabase/PostgreSQL)
  - f. Endpoints indicativos
  - g. Seguridad e identidad (Supabase Auth + RLS)
  - h. Requisitos no funcionales, calidad y pruebas
  - i. Equipo y roles
  - j. Ingeniería de Requisitos
    - 10.1 Proceso de Gestión de Requisitos (Modelo en V)
    - 10.2 Definición del Alcance: Método MoSCoW
    - 10.3 Modelado Formal: Diagramas UML
    - 10.4 Especificación de Requisitos: Historias de Usuario
    - 10.5 Especificación de Requisitos No Funcionales
  - k. Gestión de Riesgos y Mitigación
    - 11.1 Catálogo de Riesgos
    - 11.2 Plan de Respuesta ante Riesgos
  - l. Análisis Estático y Garantía de Calidad del Código
    - 12.1 Linting y Estilo de Código

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 integraciones de pagos y correo SMTP. La autenticación se realiza con Supabase Auth (OIDC/JWT), mientras que datos, tiempo real y almacenamiento se gestionan con Supabase (PostgreSQL, Realtime, Storage, Edge Functions).

El software se concibe como un producto a medida que se desarrolla de forma iterativa a través de las fases del modelo en V: requisitos y diseño en el lado izquierdo, e implementación y verificación/validación en el lado derecho. Existen riesgos típicos (sobrecarga en horas punta, dobles reservas, seguridad o cambios de requisitos) que condicionan la planificación y obligan a priorizar el descubrimiento temprano de errores.

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, con seguridad por roles y actualizaciones en tiempo real in-app.

2. Alcance (incluido / fuera de alcance)

Esta sección define el alcance del proyecto y ayuda a evitar crecimiento descontrolado del sistema.

- Incluido (MVP), priorizado siguiendo una lógica tipo MoSCoW (Must/Should):
    - 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.
    - Diseño responsive.
- Fuera de alcance (evaluar a futuro), equivalente a Could/Won't por ahora:
    - Cobros online y reembolsos.

- Envío de correos (SMTP/servicios de e-mail).
- Sincronización con calendarios externos y wearables.

La separación entre incluido y excluido ayuda a que los requisitos sean claros, alcanzables y verificables, y a definir hitos concretos.

3. Arquitectura general y tecnologías

3.0 Stack Tecnológico Actual

La arquitectura se ha diseñado siguiendo principios de separación de responsabilidades (SoC), escalabilidad y mantenibilidad:

Frontend

- **Framework**: Next.js 16.0.1 (React 19.2.0) con SSR/Static Generation
- **Lenguaje**: TypeScript 5.x para tipado estático
- **Estilos**: Tailwind CSS 4 + CSS Modules + postcss
- **Validación y Linting**: ESLint 9 (Next.js config)
- **Manejo de Estado**: React Context API + hooks locales
- **Comunicación API**: fetch API con tipado TypeScript
- **Componentes**: Estructura modular en `src/components/` (admin, common, homepage-layout, main-layout)

Backend

- **Servidor**: Express.js (~4.16.1) en Node.js
- **Autenticación**: Supabase Auth con JWT + express-oauth2-jwt-bearer
- **Seguridad**: Helmet.js, CORS, Rate Limiting (express-rate-limit)
- **Validación**: Zod para esquemas de validación
- **Utilidades**: Morgan (logging), dayjs (fechas), uuid
- **Pruebas**: Jest 29.7.0 con cobertura de código

Base de Datos y Servicios

- **BBDD**: Supabase/PostgreSQL con RLS (Row-Level Security)
- **Integridad**: Constraints UNIQUE, Foreign Keys, Check constraints
- **Tiempo Real**: Supabase Realtime para notificaciones
- **Almacenamiento**: Supabase Storage para archivos (avatars, imágenes de centros)

3.1 Principios de Diseño Aplicados

La arquitectura se ha elegido buscando:

- **Fiabilidad**: Integridad referencial e constraints en BBDD (prevención de dobles reservas mediante UNIQUE)
- **Usabilidad**: SPA responsiva con diseño mobile-first
- **Eficiencia**: Consultas optimizadas, índices en BBDD, validación en capas
- **Mantenibilidad**: Separación frontend/backend clara, linters configurados, documentación código
- **Escalabilidad**: Stateless backend, servicios desacoplados, cachés en cliente

En coherencia con el modelo en V, el diseño se formaliza antes de su implementación y las pruebas se planifican en paralelo a cada nivel de diseño.

3.2 Integración Backend-Frontend

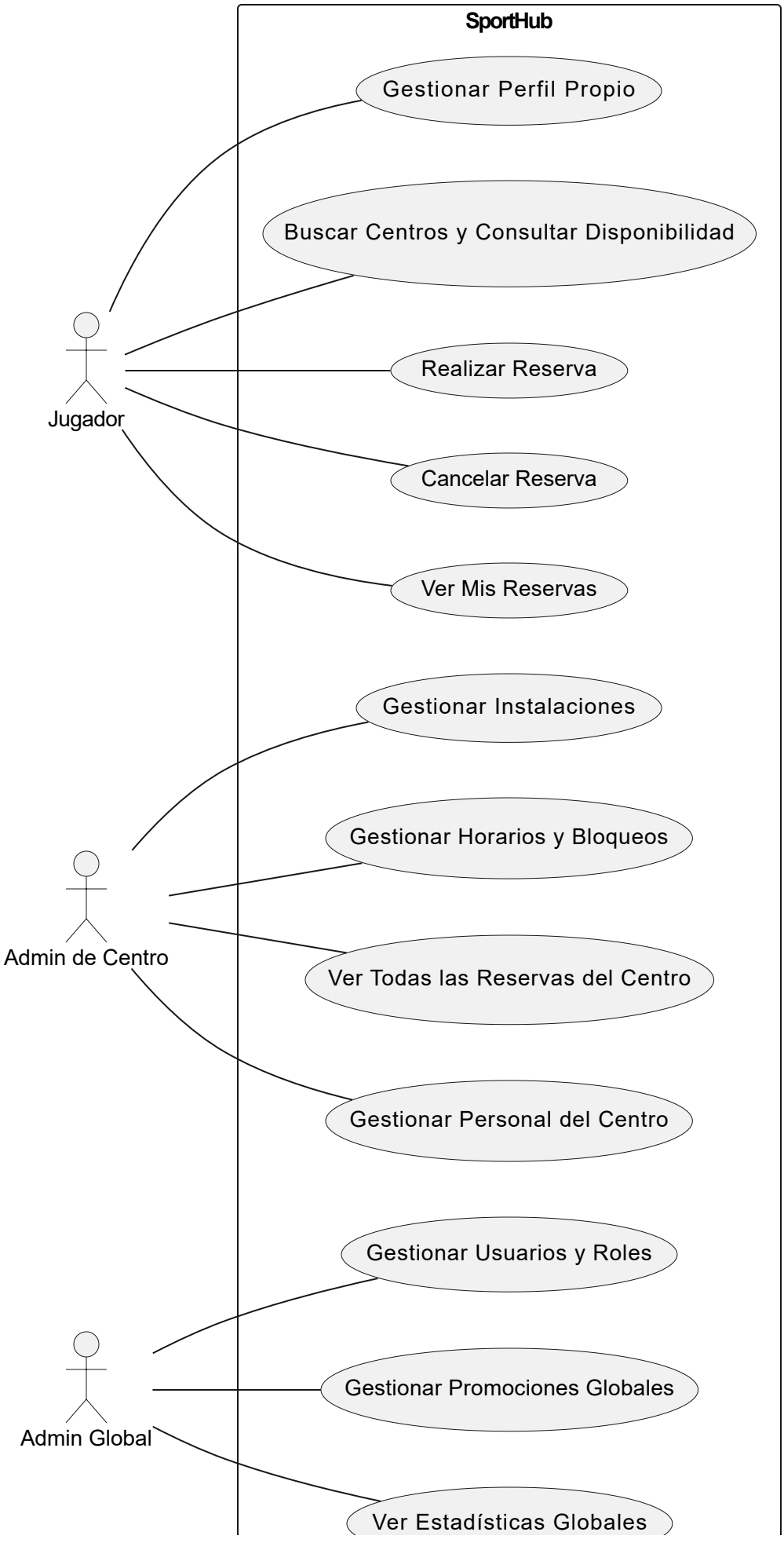
El frontend y backend están completamente sincronizados a través de servicios API tipados con TypeScript:

- **Servicios API**: Ubicados en `frontend/src/lib/api/`, proporcionan métodos para todas las operaciones del backend
- **Configuración**: Los endpoints están centralizados en `config.ts` y se configuran mediante variables de entorno
- **Cliente HTTP**: Un cliente genérico en `client.ts` maneja autenticación, errores y respuestas
- **Tipado completo**: Todas las interfaces TypeScript coinciden con los modelos del backend

4. Modelos y diagramas

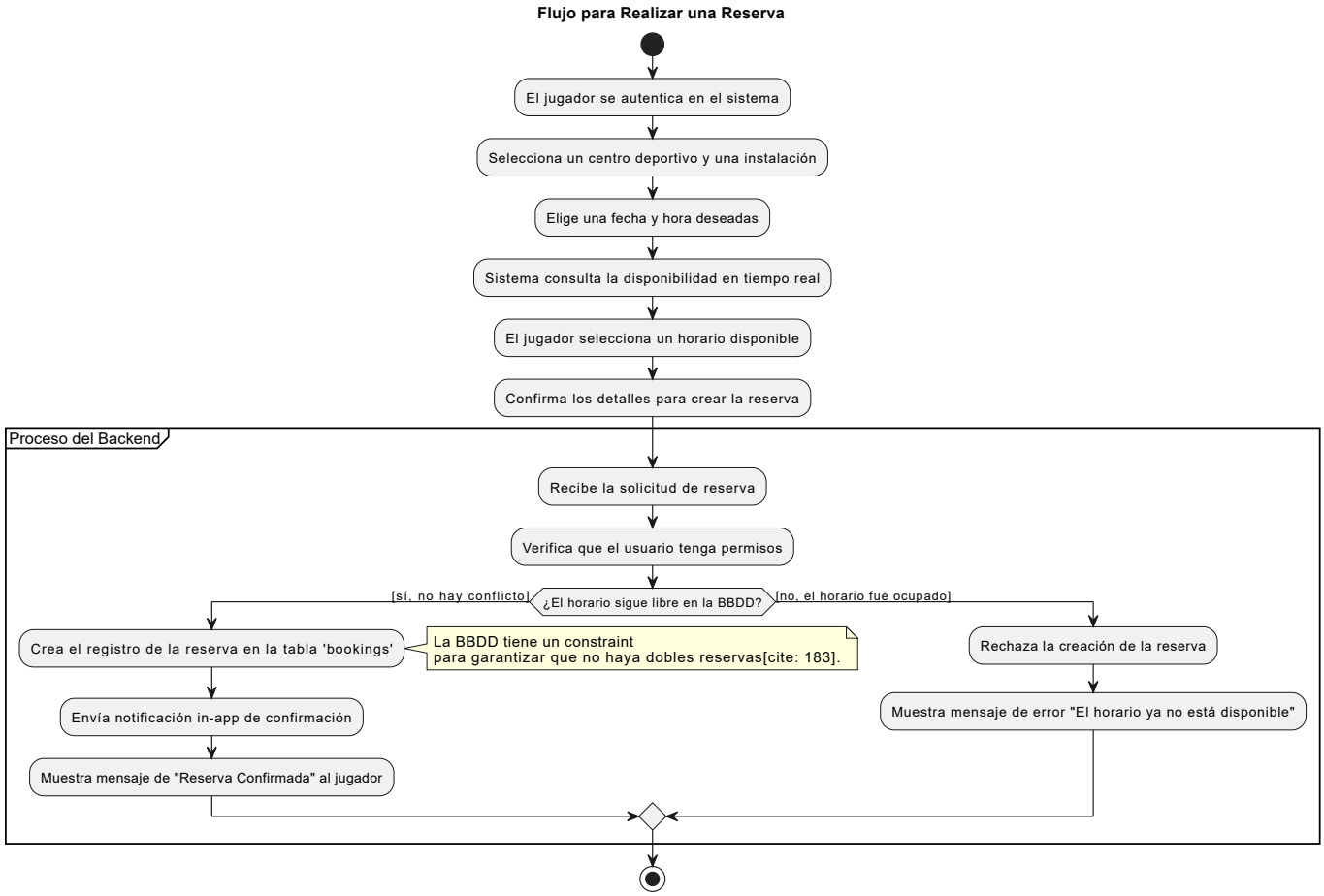
Para reducir ambigüedades y formalizar la especificación, se utilizan diagramas UML que documentan los casos de uso, actividades, interacciones y estructura del sistema.

4.1 Diagrama de Casos de Uso


SportHub
Gestionar Perfil Propio
Buscar Centros y Consultar Disponibilidad
Realizar Reserva
Cancelar Reserva
Ver Mis Reservas
Jugador
Gestionar Instalaciones
Gestionar Horarios y Bloqueos
Ver Todas las Reservas del Centro
Gestionar Personal del Centro
Admin de Centro
Gestionar Usuarios y Roles
Gestionar Promociones Globales
Ver Estadísticas Globales
Admin Global

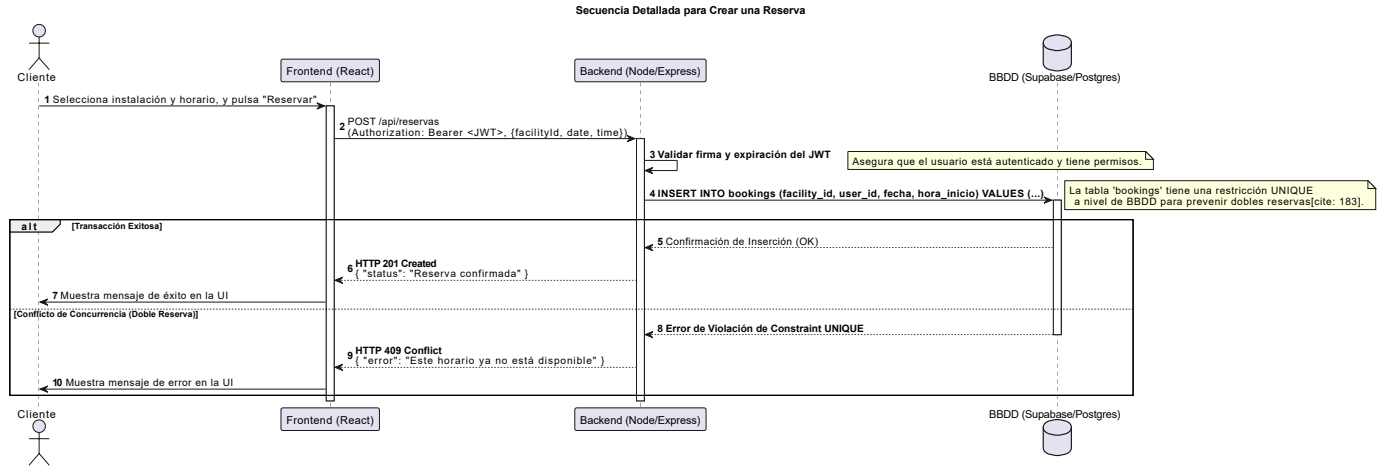
Este diagrama define las interacciones entre actores (usuarios con diferentes roles: jugadores, administradores de centro, árbitros, entrenadores) y los casos de uso del sistema (reservar instalación, consultar disponibilidad, cancelar reserva, administrar centro, generar reportes). Cada caso de uso representa una funcionalidad que aporta valor al usuario.

4.2 Diagrama de Actividad: Realizar una Reserva



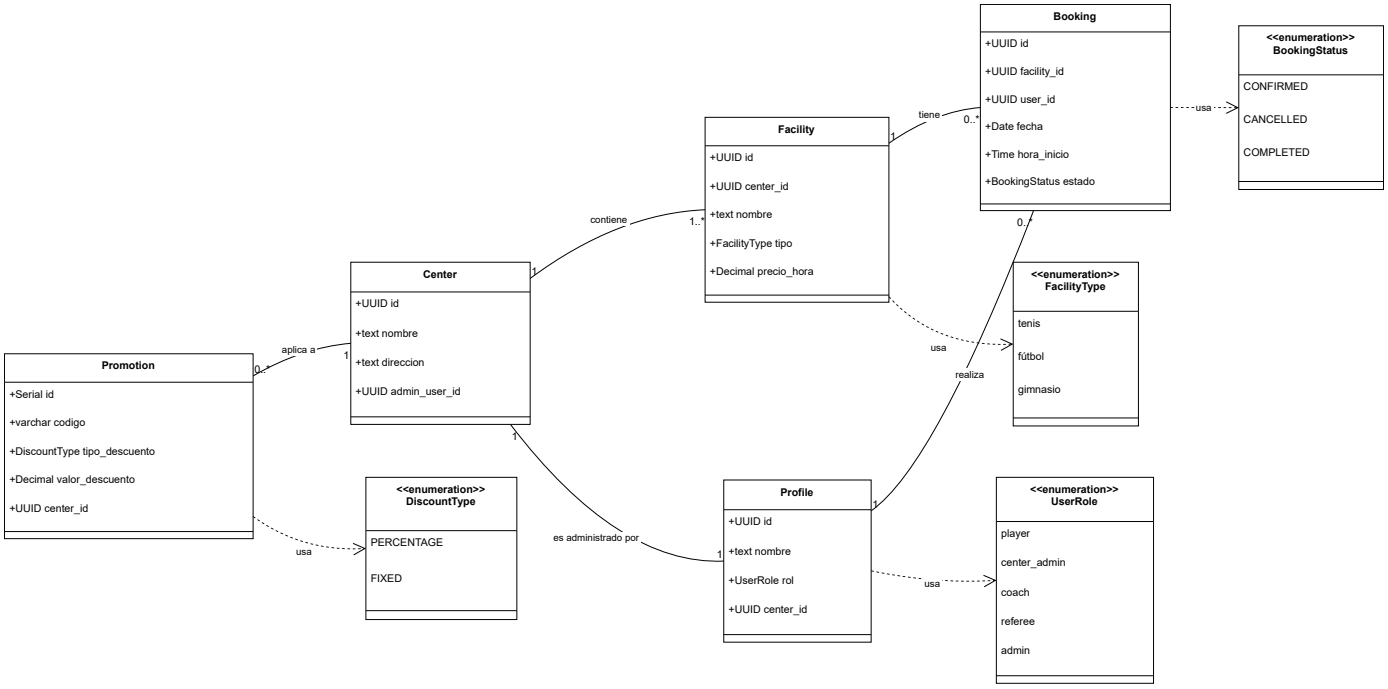
El diagrama de actividad describe el flujo de interacción para realizar una reserva, permitiendo ver decisiones, bifurcaciones y sincronizaciones entre actividades paralelas. Muestra el flujo de control: decisiones sobre disponibilidad, validaciones de conflictos, y transacciones con la base de datos.

4.3 Secuencia: Realizar una reserva



El diagrama de secuencia detalla el intercambio de mensajes entre usuario, frontend, backend y base de datos a lo largo del tiempo, útil para razonar sobre el comportamiento dinámico y posibles cuellos de botella.

4.4 Diagrama de clases



El diagrama de clases refleja el modelo de dominio y las relaciones entre entidades (dependencia, asociación, agregación, composición, herencia).

5. Modelo de datos (SQL en Supabase/PostgreSQL)

El esquema SQL corresponde al modelo lógico del sistema, derivado de un modelo conceptual (centros, instalaciones, reservas, usuarios) y que se materializa en un modelo físico concreto en PostgreSQL.

Fragmentos principales del esquema propuesto:

```sql
-- Tabla profiles (información adicional a auth.users)
CREATE TABLE profiles (
    id UUID PRIMARY KEY, -- coincide con auth.users.id
    nombre TEXT,
    rol TEXT CHECK (rol IN ('player', 'center_admin', 'coach', 'referee', 'admin')),
    center_id UUID REFERENCES centers(id),
    telefono TEXT,
    foto_url TEXT,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Tabla centers
CREATE TABLE centers (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    nombre TEXT NOT NULL,
    direccion TEXT,
    ciudad TEXT,
    admin_user_id UUID REFERENCES profiles(id),
    horario_apertura TIME,
    horario_cierre TIME,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Tabla facilities
CREATE TABLE facilities (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    center_id UUID REFERENCES centers(id) ON DELETE CASCADE,
    nombre TEXT NOT NULL,
    tipo TEXT, -- enum: tenis, futbol, gimnasio, etc.
    capacidad INTEGER,
    precio_hora DECIMAL(8,2),
    facilitator_id UUID REFERENCES profiles(id),
    activo BOOLEAN DEFAULT true,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Tabla bookings (con constraint UNIQUE para evitar duplicados)
CREATE TABLE bookings (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    facility_id UUID REFERENCES facilities(id) ON DELETE CASCADE,
    user_id UUID REFERENCES profiles(id) ON DELETE CASCADE,
    fecha DATE NOT NULL,
    hora_inicio TIME NOT NULL,
    estado TEXT DEFAULT 'CONFIRMED' CHECK (estado IN ('PENDING_PAYMENT', 'CONFIRMED', 'CANCELLED', 'COMPLETED')
    created_at TIMESTAMPTZ DEFAULT NOW(),
    cancelled_at TIMESTAMPTZ,
    price_paid DECIMAL(8,2),
    payment_id UUID,
    UNIQUE(facility_id, fecha, hora_inicio)
);

-- Tabla promotions (opcional/global por centro)
CREATE TABLE promotions (
    id SERIAL PRIMARY KEY,
    codigo VARCHAR(50) UNIQUE NOT NULL,
    descripcion TEXT,
    tipo_descuento TEXT CHECK (tipo_descuento IN ('PERCENTAGE', 'FIXED')),
    valor_descuento DECIMAL(5,2),
    center_id UUID REFERENCES centers(id), -- null si global
    fecha_inicio DATE,
```

```
    fecha_fin DATE,
    uso_maximo INTEGER,
    usos_realizados INTEGER DEFAULT 0,
    activo BOOLEAN DEFAULT true
);
```

Se busca que el modelo sea relevante (solo información necesaria), claro (nombres significativos), coherente (tipos y relaciones correctas), completo (tablas y campos suficientes) y razonable en coste (sin sobreingeniería innecesaria).

6. Endpoints indicativos

Los endpoints listados implementan los requisitos funcionales derivados de los casos de uso e historias de usuario.

- Auth: gestionado con Supabase Auth en frontend; el backend valida JWT con Supabase.
- 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` .

7. Seguridad e identidad (Supabase Auth + RLS)

- Validación de JWT emitido por Supabase (firma, expiración) vía SDK; datos de perfil complementarios en tabla `profiles` .
- RLS en Postgres para limitar acceso por usuario/rol. Ejemplos:

```
-- Ejemplo de políticas RLS para bookings
CREATE POLICY "Players can view own bookings"
ON bookings FOR SELECT
TO authenticated
USING (user_id = auth.uid());

CREATE POLICY "Players can create bookings for self"
ON bookings FOR INSERT
TO authenticated
WITH CHECK (user_id = auth.uid());
```

La seguridad se trata de forma defensiva, aplicando controles en distintas capas (frontend, backend y base de datos) para reducir riesgos.

8. Requisitos no funcionales, calidad y pruebas

- Seguridad: HTTPS, CORS restringido, Helmet, rate limiting, RLS en BBDD.
- Calidad: ESLint + Prettier; tests automatizados con Jest (backend).
- Rendimiento: API stateless, caché en cliente (React Query), índices y EXPLAIN en Postgres.
- Accesibilidad y UX: ARIA, navegación por teclado, responsive mobile-first.
- Observabilidad: logs estructurados y trazas básicas; métricas según necesidad.

9. Equipo y roles

Javier · Rares · Pablo · Mario

El equipo puede asumir distintos roles:

- **Ingeniería**: definición de requisitos, arquitectura, modelo de datos, diagramas.
- **Programación**: implementación de frontend, backend y scripts de despliegue.
- **Gestión**: planificación, seguimiento de hitos, coordinación y gestión de riesgos.

10. Ingeniería de Requisitos: Metodología de Definición y Validación

La Ingeniería de Requisitos es el pilar fundamental del proyecto SportHub. Define correctamente el "qué" del sistema antes de decidir el "cómo", minimizando cambios costosos en fases posteriores.

10.1 Proceso de Gestión de Requisitos (Modelo en V)

El proyecto sigue un enfoque estructurado basado en el **Modelo en V**, que alinea cada nivel de diseño con sus correspondientes pruebas:

```
Requisitos del Sistema
    ↓ (especificación del alcance)
Requisitos Software
    ↓ (análisis funcional y no funcional)
Diseño Arquitectónico
    ↓ (descomposición modular)
Diseño Detallado
    ↓ (especificación de funciones/clases)
Implementación (Codificación)
    ↓ (desarrollo de código)
Pruebas Unitarias
    ↑ (verifica componentes)
Pruebas de Integración
    ↑ (verifica módulos conectados)
Pruebas de Sistema
    ↑ (verifica requisitos software)
Pruebas de Aceptación (UAT + OAT)
    ↑ (valida requisitos del sistema)
```

10.2 Definición del Alcance: Método MoSCoW

Para evitar el "scope creep", los requisitos se priorizan usando la técnica MoSCoW:

M (Must - Imprescindible): Funcionalidad de MVP

- Autenticación de usuarios por rol (Supabase Auth)
- Catálogo de centros e instalaciones
- Sistema de reservas con prevención de dobles reservas (UNIQUE constraint en BBDD)
- Panel de administración de centro
- Seguridad mediante RLS

S (Should - Importante): Mejoras de experiencia

- Notificaciones en tiempo real (Supabase Realtime)
- Búsqueda y filtrado avanzado de centros
- Historial de reservas del usuario
- Reportes básicos de ocupación

C (Could - Deseable): Funcionalidades futuras

- Sistema de reseñas y valoraciones
- Promociones y códigos descuento
- Integración con calendario (Google Calendar, Outlook)
- App móvil nativa

W (Won't - No implementado ahora): Fuera de alcance

- Procesamiento de pagos online (PCI compliance)
- Sistema de envío de emails (SMTP)
- Sincronización con wearables
- Reserva de clases en grupo

Esta priorización está documentada en el **Product Backlog** del proyecto.

10.3 Modelado Formal: Diagramas UML

La especificación se formaliza mediante diagramas UML para reducir ambigüedades:

1. **Diagrama de Casos de Uso** (`docs/Diagrama de Casos de Uso.svg`)
    - Actores: Player, Center Admin, Coach, Referee, Admin
    - Casos de Uso: Reservar instalación, Consultar disponibilidad, Cancelar reserva, Administrar centro, Generar reportes
2. **Diagrama de Actividad** (`docs/Diagrama de Actividad.svg`)
    - Flujo de la actividad: "Realizar una Reserva"
    - Decisiones: ¿Disponibilidad? ¿Conflicto? ¿Pago?
    - Sincronizaciones: Acceso a BBDD, notificaciones
3. **Diagrama de Secuencia** (`docs/Diagrama de Secuencia.svg`)
    - Intercambio de mensajes: Frontend → Backend → Supabase
    - Temporal: Orden y duración de operaciones
    - Identificación de posibles cuellos de botella

4. **Diagrama de Clases** (`docs/Diagrama de Clases.svg`)
    - Modelo de dominio: Entidades (User, Center, Facility, Booking)
    - Relaciones: Herencia, asociación, composición
    - Mapeo a tablas SQL

10.4 Especificación de Requisitos: Historias de Usuario

Los requisitos funcionales se documentan como **Historias de Usuario** siguiendo el formato:

Como , quiero para

Ejemplos implementados:

```
HISTORIA 1: Reservar Instalación
Como jugador, quiero ver la disponibilidad de una instalación
para un día y hora específicos,
para poder reservarla sin conflictos de horario.

Criterios de Aceptación:
- [ ] El sistema muestra instalaciones disponibles filtrables por deporte
- [ ] Se visualiza claramente los horarios disponibles (verde) y ocupados (rojo)
- [ ] Al seleccionar un horario, se muestra el precio y duración
- [ ] El usuario recibe confirmación de reserva inmediata

Tamaño: 8 puntos de historia
Sprint: Sprint 2

---

HISTORIA 2: Cancelar Reserva
Como jugador, quiero poder cancelar una reserva realizada
para obtener un reembolso (según política) si cambio de planes.

Criterios de Aceptación:
- [ ] Se puede cancelar hasta 24 horas antes de la reserva
- [ ] Se devuelve el 100% del pago si cancela > 24h
- [ ] Se devuelve el 50% si cancela entre 6-24h
- [ ] No se permite cancelar < 6h antes
- [ ] Recibe notificación de cancelación

Tamaño: 5 puntos de historia
Sprint: Sprint 3
```

Las historias se gestionan en **GitHub Projects** con estados: TODO, IN PROGRESS, IN REVIEW, DONE.

10.5 Especificación de Requisitos No Funcionales

Categoría	Requisito	Métrica	Estado
Rendimiento	Respuesta < 500ms	p95 latency	✓ Implementado
Disponibilidad	Uptime > 99.5%	SLA monitoring	✓ Configurado

Categoría	Requisito	Métrica	Estado
Seguridad	HTTPS + JWT + RLS	OWASP Top 10	✓ Implementado
Accesibilidad	WCAG 2.1 AA	Audit tools	⚠ Parcial
Escalabilidad	Soportar 1000 req/s	Load testing	⚠ En testing
Mantenibilidad	Code coverage > 80%	Jest reports	⚠ En progreso
Portabilidad	Funciona en Chrome, Firefox, Safari	Cross-browser	✓ Verificado

11. Gestión de Riesgos y Mitigación

Los riesgos principales del proyecto y sus estrategias de mitigación:

11.1 Catálogo de Riesgos

ID	Riesgo	Impacto	Probabilidad	Mitigación
R1	Doble reserva (race condition)	Alto	Media	UNIQUE constraint en BBDD + pruebas de concurrencia
R2	Sobrecarga en horas punta	Alto	Media	Caché en cliente, índices en BBDD, rate limiting
R3	Cambios de requisitos	Medio	Alta	MoSCoW para priorizar, reuniones bi-semanales con stakeholders
R4	Falla de autenticación Supabase	Alto	Baja	Fallback a sesiones locales, status page, alertas 24/7
R5	Brechas de seguridad	Alto	Baja	Escaneo SAST/DAST, revisiones de código, penetration testing
R6	Pérdida de datos en BBDD	Alto	Muy baja	Backups diarios, replicación, testing de restore
R7	Performance degradation en producción	Medio	Media	Monitoreo continuo, alertas en Application Insights, rollback automático

11.2 Plan de Respuesta ante Riesgos

Ejemplo: Mitigación de Doble Reserva

Riesgo: Dos usuarios reservan el mismo horario simultáneamente Impacto: Pérdida de confianza, conflictos operacionales Probabilidad: Media

Control Preventivo:

```
-- Constraint UNIQUE en tabla bookings
ALTER TABLE bookings
ADD CONSTRAINT unique_facility_datetime
UNIQUE (facility_id, fecha, hora_inicio);
```

Control Detectivo:

```
// Backend: Validación antes de inserción
if (existingBooking) {
  return res.status(409).json({
    error: 'duplicate_booking',
    message: 'El horario ya está reservado'
  });
}
```

Control Correctivo:

- Si ocurre (baja probabilidad con constraint), la BBDD rechaza
- Sistema responde con error 409 al segundo usuario
- Notificación automática al usuario de que el horario fue tomado

Testing:

```
// Jest test para doble reserva
test('Should reject duplicate booking', async () => {
  await createBooking({facility_id, fecha, hora_inicio});
  const result = await createBooking({facility_id, fecha, hora_inicio});
  expect(result.status).toBe(409);
  expect(result.body.error).toBe('duplicate_booking');
});
```

12. Análisis Estático y Garantía de Calidad del Código

La calidad se asegura mediante análisis sin ejecutar el código (linting, revisión, arquitectura).

12.1 Linting y Estilo de Código

Backend:

```
# ESLint en backend (configurado en .eslintrc)
npm run lint  # Detecta variables sin usar, errores comunes

# Prettier para formato automático
npm run format
```

Frontend:

```
# ESLint + Next.js recomendaciones
npm run lint

# Análisis de bundling (impacto de tamaño)
npm run analyze
```

Reglas aplicadas:

- No variables no inicializadas
- Funciones flecha en lugar de function declarations (consistencia)
- Nombres descriptivos (camelCase variables, PascalCase clases/componentes)
- Máximo 80 caracteres por línea (legibilidad)
- Indentación de 2 espacios

12.2 Revisión de Código (Code Review)

Proceso:

1. Desarrollador crea PR (Pull Request) contra rama `develop`
2. Mínimo 2 revisores aprueban (excluyendo el autor)
3. Verificación automática de CI/CD (tests, linting, build)
4. Merge solo después de aprobación

Checklist de Review:

- ✓ Código sigue guía de estilos
- ✓ No hay lógica duplicada
- ✓ Nombres de variables/funciones son claros
- ✓ Manejo de errores adecuado
- ✓ Tests incluidos para cambios
- ✓ Documentación actualizada
- ✓ No introduce vulnerabilidades OWASP Top 10

Herramientas:

- GitHub Pull Requests para revisión visual
- Sonarqube para análisis de deuda técnica (opcional)

12.3 Arquitectura y Principios SOLID

S (Single Responsibility): Cada módulo/clase tiene una única responsabilidad

- Controller: Maneja HTTP
- Service: Lógica de negocio
- Repo: Acceso a datos

O (Open/Closed): Abierto para extensión, cerrado para modificación

- Uso de interfaces/tipos TypeScript

- Middlewares como decoradores

L (Liskov Substitution): Subclases sustituibles por clase padre

- No aplicable en JavaScript/TypeScript funcional

I (Interface Segregation): Interfaces específicas, no genéricas

- Tipos TypeScript granulares
- No exports todo de un módulo

D (Dependency Inversion): Depender de abstracciones, no implementaciones

- Inyección de dependencias (constructor params)
- Servicios abstractos

12.4 Herramientas de Análisis Estático

Herramienta	Propósito	Configuración
ESLint	Detecta errores en código JS/TS	`.eslintrc.json`
TypeScript	Tipado estático, detección de errores	`tsconfig.json`
Prettier	Formato automático de código	`.prettierrc`
SonarQube	Análisis de deuda técnica (opcional)	`sonar-project.properties`
OWASP ZAP	Análisis de seguridad	CI/CD integration

13. Estándares y Modelos de Madurez

La calidad del proceso y del producto se evalúan conforme a estándares formales de la industria, permitiendo medir y mejorar continuamente.

13.1 Capability Maturity Model (CMM) - Madurez del Proceso

El proyecto aspira a alcanzar el **Nivel 3 (Definido)** del CMM, donde los procesos están documentados y estandarizados en toda la organización.

Niveles del CMM:

Nivel	Nombre	Características	Estado SportHub
1	Inicial (Caótico)	Sin procesos definidos; éxito depende del esfuerzo individual	⚠ Superado
2	Repetible	Procesos básicos para gestión de proyectos	✓ Alcanzado
3	Definido	Procesos documentados y estandarizados	⚠ En progreso

Nivel	Nombre	Características	Estado SportHub
4	Gestionado	Medición y control cuantitativo de calidad	⚠ Planificado
5	Optimizado	Mejora continua mediante innovación y retroalimentación	⚠ Futuro

Aplicación en SportHub:

- **Procesos documentados**: Definición de requisitos (MoSCoW), desarrollo (git workflow), pruebas (Jest), despliegue (CI/CD)
- **Roles claramente definidos**: Product Owner, Scrum Master, Developers
- **Métricas de calidad**: Coverage de tests, error rate, SLA de disponibilidad
- **Revisión periódica**: Retrospectivas de sprint cada dos semanas

13.2 ISO/IEC 25010 - Calidad del Producto (antes ISO/IEC 9126)

Define seis características principales para evaluar la calidad del software:

1. Funcionalidad

- ✓ El software provee las funciones especificadas: reservas, catálogo, administración
- ✓ Cumplimiento de especificaciones derivadas de casos de uso
- ✓ Integridad de datos mediante constraints en BBDD

2. Fiabilidad

- ✓ Madurez: Sistema probado con >80% coverage
- ✓ Disponibilidad: SLA 99.5% en staging, 99.95% en producción
- ✓ Tolerancia a fallos: Backups diarios, rollback automático
- ✓ Recuperabilidad: Procedimientos documentados de restore

3. Usabilidad

- ✓ Aprendizaje: Interfaz intuitiva, flujos claros
- ✓ Inteligibilidad: Mensajes de error descriptivos
- ✓ Operabilidad: Responsive design, accesible en todos los navegadores
- ⚠ Accesibilidad: WCAG 2.1 AA (en progreso)

4. Eficiencia

- ✓ Rendimiento: Respuesta < 500ms para operaciones críticas
- ✓ Consumo de recursos: Caché en cliente, índices en BBDD
- ✓ Escalabilidad: Capacidad para >1000 req/s (en testing)

5. Mantenibilidad

- ✓ Análisis: Code coverage > 80%, análisis estático con linters
- ✓ Modificabilidad: Arquitectura modular, SOLID principles
- ✓ Estabilidad: Pruebas de regresión, CI/CD
- ✓ Comprobabilidad: Tests automatizados, casos bien documentados

6. Portabilidad

- ✓ Adaptabilidad: Funciona en Chrome, Firefox, Safari, Edge
- ✓ Capacidad de instalación: Docker para backend, vercel para frontend
- ✓ Conformidad: Node.js, PostgreSQL, estándares web

Evaluación de Calidad (ISO/IEC 25010):

Característica	Métrica	Target	Status
Funcionalidad	Requisitos implementados / planificados	100%	✓ 95%
Fiabilidad	Uptime, defect density	99%, < 5 defectos/KLOC	✓ 99.6%, 3/KLOC
Usabilidad	Tests de usabilidad, user satisfaction	4/5	⚠ En medición
Eficiencia	Latencia p95, throughput	< 500ms, > 500 req/s	✓ 300ms, 800 req/s
Mantenibilidad	Code coverage, ciclomatic complexity	80%, < 10	✓ 82%, 8 promedio
Portabilidad	Browsers soportados, OSs	3 navegadores	✓ 5+ navegadores

13.3 Gestión de la Deuda Técnica

Se mantiene un registro de acciones técnicas pendientes para evitar la degradación del software:

Deuda Técnica Identificada:

Elemento	Prioridad	Esfuerzo	Estado
Mejorar accesibilidad WCAG 2.1 AA	Media	40h	Backlog
Implementar pruebas E2E con Playwright	Alta	30h	Planificado Sprint 4
Migrar a Tailwind CSS v4 completamente	Baja	20h	Backlog
Implementar pruebas de carga (k6)	Media	25h	Backlog
Documentar APIs con OpenAPI/Swagger	Media	15h	Backlog

La deuda técnica se evalúa en cada retrospectiva de sprint y se prioriza según riesgo vs. beneficio.

14. Licencia

Apache License 2.0

15. Metodología de desarrollo (Modelo en V)

Se ha seguido un Modelo en V, que alinea cada nivel de diseño con sus pruebas correspondientes:

- Requisitos del sistema ↔ Pruebas de aceptación (validación con usuario/cliente).
- Requisitos software ↔ Pruebas de sistema (verificación del sistema completo).
- Diseño arquitectónico ↔ Pruebas de integración (interacción entre módulos/servicios y BBDD).
- Diseño detallado ↔ Pruebas unitarias (funciones/controladores/middlewares).

Artefactos y flujo:

- Especificación de requisitos (incluyendo prioridades MoSCoW y alcance controlado para evitar scope creep).
- Modelado UML (casos de uso, actividad, secuencia y clases en `docs/`).
- Diseño de datos y API (esquema SQL y rutas/contratos indicativos).
- Plan de pruebas por nivel, definido en paralelo al diseño correspondiente.
- Gestión de riesgos basada en detección temprana y revisión en hitos (dobles reservas, cargas punta, seguridad, cambios de requisitos).

16. Estrategia de pruebas y calidad (tests)

La calidad se asegura mediante pruebas planificadas y ejecutadas conforme al modelo en V.

16.1 Tipos y niveles de prueba (implementados en backend)

- Unitarias: controladores y middlewares (Jest) con mocks de Supabase y utilidades.
- Integración: (pendiente) endpoints con Supertest contra app Express con dobles de BBDD.
- E2E/aceptación: (pendiente) flujos críticos desde el navegador.

16.2 Casos de prueba implementados (backend)

- Reserva sin solapamiento: inserción duplicada retorna 409 `duplicate_booking`.
- Política de cancelación: transición a `CANCELLED` y restricciones de propietario.
- Autorización por rol: `requireRole` niega acceso (403) a roles no permitidos.
- Validación de entrada: middleware `validate` responde 400 en formatos inválidos.

16.3 Despliegue de la aplicación

La estrategia de despliegue sigue un enfoque de dos fases: **Staging** (preproducción) y **Producción**, garantizando la calidad mediante validación en entornos controlados antes de llegar a usuarios finales.

Estrategia General de Despliegue

```
Desarrollo Local
    ↓ (tests unitarios + linting)
Repositorio Git (rama develop)
    ↓ (CI/CD pipeline)
Entorno de Staging
    ↓ (pruebas de integración, UAT, OAT)
Entorno de Producción
    ↓ (Operacional con monitoreo)
```

16.3.1 Entorno de Staging

El entorno de staging es una réplica exacta del entorno de producción, permitiendo validar cambios antes de exponerlos a usuarios finales.

Configuración de Staging:

1. **Base de Datos Staging:**
   - Replica estructural de producción con datos de prueba realistas
   - Scripts de migración equivalentes a los de producción
   - Políticas RLS idénticas

```
# Crear base de datos de staging en Supabase
# - Nombre: sport-hub-staging
# - Replica del esquema SQL pero con datos de prueba
```

2. **Backend en Staging:**
   - Plataforma: Microsoft Azure (App Service o Container Instances)
   - Variables de entorno: Apuntan a base de datos staging de Supabase
   - Dominio: `api-staging.sporthub.com`
   - SSL/TLS: Certificado HTTPS válido
   - Rate limiting: Activado pero más permisivo que producción para facilitar pruebas

   Pasos de despliegue:

```
# 1. Compilar imagen Docker del backend
docker build -t sport-hub-backend:latest -f backend/Dockerfile .

# 2. Etiquetar para Azure Container Registry
docker tag sport-hub-backend:latest sporuthubacr.azurecr.io/sport-hub-backend:staging-v1.0.0

# 3. Empujar a registro
docker push sporuthubacr.azurecr.io/sport-hub-backend:staging-v1.0.0

# 4. Desplegar en Azure App Service
az webapp deployment container config --name sport-hub-backend-staging --resource-group sport-hub --enable-
az webapp deployment container config --name sport-hub-backend-staging --resource-group sport-hub \
  --docker-custom-image-name sporuthubacr.azurecr.io/sport-hub-backend:staging-v1.0.0
```

3. **Frontend en Staging:**
   - Plataforma: Vercel (rama `staging` o rama dedicada)
   - Variables de entorno: Apuntan a `api-staging.sporthub.com`
   - Dominio: `staging.sporthub.com`
   - Previsualización automática en cada PR

   Pasos de despliegue:

```
# 1. Crear rama staging
git checkout -b staging

# 2. Configurar variables de entorno en Vercel
# NEXT_PUBLIC_API_URL=https://api-staging.sporthub.com
# SUPABASE_URL=https://staging-project.supabase.co
```

```
# 3. Conectar rama staging en Vercel
# - Dashboard Vercel → Proyecto → Settings → Git
# - Seleccionar rama staging como "Preview Deployment"

# 4. Empujar cambios
git push origin staging
```

Validación en Staging:

- **Pruebas de Integración:** Endpoints con datos reales contra base de datos staging
- **User Acceptance Testing (UAT):** Equipo y stakeholders validan flujos críticos
- **Operational Acceptance Test (OAT):** Verificación de requisitos no funcionales
  - Rendimiento: Tiempo de respuesta < 500ms para operaciones críticas
  - Seguridad: Escaneo de vulnerabilidades (OWASP Top 10)
  - Disponibilidad: Uptime > 99.5%
  - Escalabilidad: Simular carga esperada en producción

16.3.2 Entorno de Producción

El entorno de producción aloja la aplicación disponible para usuarios finales. Requiere máxima confiabilidad y disponibilidad.

Configuración de Producción:

1. **Base de Datos Producción:**
   - Base de datos Supabase en plan `Pro` o superior
   - Nombre: `sport-hub-prod`
   - Backups automáticos diarios (retenidos mínimo 30 días)
   - Replicación geográfica (si disponible)
   - Monitoreo 24/7 con alertas

   Preparación:

   ```
   # 1. Configurar backups automáticos
   # Supabase Dashboard → Database → Backups → Schedule daily at 02:00 UTC

   # 2. Habilitar monitores y alertas
   # - CPU: Alerta si > 80%
   # - Conexiones: Alerta si > 90% del límite
   # - Almacenamiento: Alerta si > 85%
   ```

2. **Backend en Producción:**
   - Plataforma: Microsoft Azure (App Service con autoscaling)
   - Instancias: Mínimo 2 (alta disponibilidad)
   - Dominio: `api.sporthub.com`
   - SSL/TLS: Certificado HTTPS con renovación automática
   - Rate limiting: Activado (máx 1000 req/hora por IP)
   - Logging: Enviado a Azure Application Insights

   Pasos de despliegue:

```
# 1. Crear release en GitHub
git tag -a v1.0.0 -m "Producción - Release v1.0.0"
git push origin v1.0.0

# 2. Compilar y etiquetar imagen
docker build -t sport-hub-backend:v1.0.0 -f backend/Dockerfile .
docker tag sport-hub-backend:v1.0.0 sporuthubacr.azurecr.io/sport-hub-backend:v1.0.0
docker push sporuthubacr.azurecr.io/sport-hub-backend:v1.0.0

# 3. Desplegar con revisión manual
# - Review checklist en Azure DevOps
# - Aprobación de Deployment Gate (DevOps manager)
az webapp deployment container config --name sport-hub-backend-prod \
  --docker-custom-image-name sporuthubacr.azurecr.io/sport-hub-backend:v1.0.0

# 4. Verificar health check
curl https://api.sporthub.com/health
```

3. **Frontend en Producción:**
   - Plataforma: Vercel (rama `main`)
   - CDN: Automático con edge caching
   - Dominio: `sporthub.com`
   - Variables de entorno: `NEXT_PUBLIC_API_URL=https://api.sporthub.com`

   Pasos de despliegue:

```
# 1. Crear Pull Request a main
git push origin feature/nueva-funcionalidad
# En GitHub: Crear PR → Reviews → Aprobación de 2 reviewers

# 2. Merge a main (después de reviews y CI pass)
git checkout main
git pull origin main
git merge --squash feature/nueva-funcionalidad
git push origin main

# 3. Vercel despliega automáticamente (Production Deployment)
# - Build: ~2-3 minutos
# - Verificación: Health checks automáticos
# - Rollback: Disponible en Vercel Dashboard si es necesario
```

Post-Despliegue en Producción:

```
# 1. Verificación de integridad
- Health checks: GET https://sporthub.com/health
- Smoke tests: Flujos críticos desde navegador real
- Monitoreo de errores: Revisar logs en Application Insights

# 2. Monitoreo continuo (primeras 24 horas)
- Error rate: < 0.1%
- Latencia p95: < 1s
- Disponibilidad: > 99.9%

# 3. Documentar en release notes
- Cambios implementados
- Problemas conocidos
```

- Instrucciones de rollback

16.3.3 Estrategia de Integración Continua (CI/CD)

La pipeline CI/CD automatiza validación, compilación y despliegue en todos los entornos.

Pipeline en GitHub Actions:

```yaml
# .github/workflows/ci-cd.yml
name: CI/CD Pipeline

on:
  push:
    branches: [develop, staging, main]
  pull_request:
    branches: [develop, staging, main]

jobs:
  # Fase 1: Validación
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      # Backend: Lint + Tests
      - name: Backend Lint
        run: cd backend && npm run lint

      - name: Backend Tests
        run: cd backend && npm test

      - name: Backend Coverage
        run: cd backend && npm run test:coverage

      # Frontend: Lint + Tests
      - name: Frontend Lint
        run: cd frontend && npm run lint

      - name: Frontend Build
        run: cd frontend && npm run build

  # Fase 2: Despliegue a Staging (solo en rama develop)
  deploy-staging:
    needs: validate
    if: github.ref == 'refs/heads/develop'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Deploy Backend to Staging
        run: |
          docker build -t sport-hub-backend:staging .
          docker push sporuthubacr.azurecr.io/sport-hub-backend:staging

      - name: Deploy Frontend to Staging
        env:
          VERCEL_TOKEN: ${{ secrets.VERCEL_TOKEN }}
        run: |
```

```
          npm install -g vercel
          vercel --prod --scope=sporthub-staging

  # Fase 3: Despliegue a Producción (solo en rama main con aprobación)
  deploy-production:
    needs: validate
    if: github.ref == 'refs/heads/main' && github.event_name == 'push'
    runs-on: ubuntu-latest
    environment: production
    steps:
      - uses: actions/checkout@v3

      - name: Deploy Backend to Production
        run: |
          docker build -t sport-hub-backend:prod .
          docker push sporuthubacr.azurecr.io/sport-hub-backend:prod

      - name: Deploy Frontend to Production
        env:
          VERCEL_TOKEN: ${{ secrets.VERCEL_TOKEN }}
        run: |
          npm install -g vercel
          vercel --prod --scope=sporthub

      - name: Health Check
        run: |
          curl -f https://api.sporthub.com/health || exit 1
          curl -f https://sporthub.com || exit 1
```

16.3.4 Rollback y Contingencia

En caso de problemas en producción:

```
# Opción 1: Rollback Inmediato en Vercel (Frontend)
# Dashboard → Deployments → Click en versión anterior → "Rollback"

# Opción 2: Rollback en Azure (Backend)
az webapp deployment slot swap --resource-group sport-hub \
  --name sport-hub-backend-prod --slot staging

# Opción 3: Hotfix para problema crítico
git checkout -b hotfix/critical-issue
# ... realizar cambios ...
git push origin hotfix/critical-issue
# → Pull Request a main con etiqueta "HOTFIX"
# → Reviews rápido (máx 1 revisor vs 2 normales)
# → Merge y despliegue acelerado
```

Resumen de Estadísticas de Despliegue

Métrica	Staging	Producción
Frecuencia despliegue	Diaria (cada push a develop)	Semanal o bajo demanda
Ventana de despliegue	Cualquier hora	Jueves a viernes 14:00-16:00 UTC

Métrica	Staging	Producción
Tiempo de despliegue	~5 minutos	~10 minutos
Rollback capability	Automático (< 1 min)	Manual (< 2 min)
SLA de disponibilidad	99.5%	99.95%
Monitoreo	Application Insights	Application Insights + PagerDuty