Práctica: API REST con Node.js, Express, Redis y Supabase

Descripción General

En esta práctica desarrollaréis una aplicación web completa basada en una **API REST** utilizando **Node.js**, **Express**, **Redis** y **Supabase (PostgreSQL)**. La aplicación deberá implementar un sistema de autenticación mediante API Keys, gestión de datos con arquitectura en capas, y sistema de caché para optimizar el rendimiento.

La temática de la aplicación es **libre** y debe ser acordada por el grupo. Algunos ejemplos de proyectos válidos: - Sistema de gestión de biblioteca (libros, préstamos, usuarios) - Plataforma de gestión de eventos (eventos, asistentes, ubicaciones) - Sistema de gestión de restaurante (platos, pedidos, mesas, clientes) - Aplicación de gestión de tareas (proyectos, tareas, usuarios, equipos) - Sistema de gestión de cursos online (cursos, estudiantes, módulos, evaluaciones)

Objetivos de Aprendizaje

Al completar esta práctica, seréis capaces de: - Diseñar e implementar una API RESTful completa siguiendo principios de arquitectura en capas - Implementar autenticación y autorización mediante API Keys con control de roles - Gestionar bases de datos relacionales con Supabase (PostgreSQL) - Optimizar rendimiento mediante caché con Redis - Aplicar buenas prácticas de desarrollo (separación de responsabilidades, validación, gestión de errores) - Trabajar en equipo utilizando control de versiones (Git)

Estructura del Proyecto (Dada)

Este proyecto utiliza una **arquitectura en capas** que separa responsabilidades y facilita el mantenimiento.

Descripción de Responsabilidades

Capa	Responsabilidad	Ejemplo
Routes	Definir endpoints y vincular con controladores	`GET /api/libros` -> `controller.getLibros`
Controllers	Manejar peticiones HTTP, validar entrada, enviar respuesta	Recibir `req.body`, llamar servicio, devolver `res.json()`
Services	Implementar lógica de negocio	Validaciones complejas, cálculos, orquestación
Repositories	Acceso a base de datos (CRUD)	`findAll()`, `create()`, `update()`, `delete()`

Capa	Responsabilidad	Ejemplo
Models	Definir estructura de entidades	Clase `Libro` con propiedades y métodos
Middlewares	Interceptar peticiones (auth, validación)	Verificar API Key antes de procesar ruta

Requisitos Técnicos

1. Base de Datos (Supabase - PostgreSQL)

- **Mínimo 3 tablas** con relaciones entre ellas
- Debe existir al menos una relación **1:N** (uno a muchos)
- Debe existir al menos una relación **N:M** (muchos a muchos) mediante tabla intermedia
- Cada tabla debe tener:
  - Clave primaria (`id SERIAL PRIMARY KEY`)
  - Timestamps (`created_at`, `updated_at`)
  - Campos relevantes al dominio

Ejemplo de relaciones:

```
-- Relación 1:N: Un autor tiene muchos libros
autores (1) ----< (N) libros

-- Relación N:M: Muchos libros tienen muchos géneros
libros (N) >----< libros_generos >----< (M) generos
```

2. Arquitectura y Código

2.1 Separación de Responsabilidades (Obligatorio)

- **Rutas → Controladores → Servicios → Repositorios**
- Cada capa debe estar en su carpeta correspondiente
- **NO** mezclar lógica de negocio en controladores
- **NO** acceder directamente a la base de datos desde controladores

2.2 Modelos (Obligatorio)

- Un modelo (clase) por cada tabla de la base de datos
- Los modelos deben incluir:
  - Constructor que acepte objeto de datos
  - Método `toJSON()` para serialización completa
  - Método `toPublic()` para datos seguros (sin campos sensibles)

Ejemplo:

```
export class Libro {
  constructor(data) {
```

```
    this.id = data.id;
    this.titulo = data.titulo;
    this.isbn = data.isbn;
    this.autor_id = data.autor_id;
    this.created_at = data.created_at;
  }

  toJSON() { /* ... */ }
  toPublic() { /* ... */ }
}
```

2.3 Validación (Obligatorio)

- **Todas** las entradas del usuario deben validarse en el servidor
- Validar tipos de datos, formatos, rangos, etc.
- Devolver errores descriptivos (código 400) si la validación falla

Ejemplo de validaciones: - Campos requeridos no vacíos - Formato de email válido - Longitud mínima/máxima de strings - Valores numéricos en rangos válidos - Fechas válidas y lógicas (ej: fecha_fin > fecha_inicio)

3. Sistema CRUD Completo

Para **cada entidad principal** (al menos 2), implementar:

Método HTTP	Endpoint	Acción	Autenticación
`POST`	`/api/entidad`	Crear nuevo registro	API Key (user)
`GET`	`/api/entidad`	Listar todos los registros	API Key (user)
`GET`	`/api/entidad/:id`	Obtener un registro por ID	API Key (user)
`PUT`	`/api/entidad/:id`	Actualizar registro completo	API Key (user)
`DELETE`	`/api/entidad/:id`	Eliminar registro	API Key (admin)

Nota: Además del CRUD dado por el boilerplate para API Keys, debéis implementar CRUD para vuestras entidades de negocio.

4. Consultas Avanzadas (Obligatorio - Mínimo 2)

Implementar **al menos 2 endpoints** que aporten valor mediante consultas avanzadas. Estas consultas deben usar: - Agregaciones (`COUNT`, `SUM`, `AVG`, `MAX`,

`MIN`) - Joins entre tablas - Filtros complejos - Agrupaciones (`GROUP BY`)

Ejemplos de consultas avanzadas:

```
// Estadísticas
GET /api/estadisticas/libros-por-genero
// -> { genero: "Ficción", total: 45 }

// Top 5 autores con más libros
GET /api/estadisticas/top-autores

// Libros prestados actualmente con información del usuario
GET /api/prestamos/activos

// Ingresos totales del último mes
GET /api/estadisticas/ingresos?mes=2026-01

// Productos más vendidos por categoría
GET /api/estadisticas/productos-top
```

5. Autenticación y Autorización

5.1 Sistema de API Keys (Ya implementado en boilerplate)

- Registro público: `POST /api/register`
- Todas las rutas protegidas requieren header `X-API-Key`
- Rate limiting: 10 peticiones por minuto por API Key (Redis)

5.2 Sistema de Roles

- **user**: Puede hacer CRUD en sus propios datos y consultas generales
- **admin**: Puede gestionar API Keys y eliminar cualquier registro

Aplicar middlewares según necesidad:

```
// Solo requiere API Key válida
router.get('/api/libros', apiKeyMiddleware, controller.getLibros);

// Requiere rol de administrador
router.delete('/api/libros/:id', adminMiddleware, controller.deleteLibro);
```

6. Redis - Sistema de Caché (Obligatorio)

Implementar **caché** para al menos 2 tipos de consultas frecuentes:

6.1 Rate Limiting (Ya implementado)

- Limitar peticiones por API Key usando Redis

6.2 Caché de Datos (A implementar) Cachear resultados de consultas costosas:

Ejemplo:

```
// Buscar en caché primero
const cacheKey = `libros:genero:${generoId}`;
const cached = await redis.get(cacheKey);

if (cached) {
  return JSON.parse(cached);
}

// Si no está en caché, consultar DB
const libros = await repository.findByGenero(generoId);

// Guardar en caché (TTL: 5 minutos)
await redis.setex(cacheKey, 300, JSON.stringify(libros));

return libros;
```

Estrategias de invalidación: - TTL (Time To Live): expiración automática - Invalidación manual al crear/actualizar/eliminar

7. Variables de Entorno

El archivo `.env` **NO** se sube a Git (incluido en `.gitignore`).

Proporcionar `.env.example` con estructura:

```
# Supabase
SUPABASE_URL=https://tu-proyecto.supabase.co
SUPABASE_KEY=tu-clave-anonima

# Redis
REDIS_URL=redis://localhost:6379

# Servidor
PORT=3000
NODE_ENV=development
```

8. Formato de Respuestas JSON

Todas las respuestas deben seguir un formato consistente:

Éxito:

```
{
  "data": { /* ... */ },
```

```
  "message": "Operación exitosa"
}
```

Error:

```
{
  "error": "Nombre del error",
  "message": "Descripción del error",
  "details": { /* info adicional opcional */ }
}
```

Códigos HTTP correctos: - `200 OK`: Consulta exitosa - `201 Created`: Recurso creado - `400 Bad Request`: Error de validación - `401 Unauthorized`: Sin API Key o inválida - `403 Forbidden`: Sin permisos - `404 Not Found`: Recurso no encontrado - `429 Too Many Requests`: Rate limit excedido - `500 Internal Server Error`: Error del servidor

Requisitos de Documentación

1. README.md (Obligatorio)

Debe incluir: - **Descripción** del proyecto - **Tecnologías** utilizadas - **Instalación** paso a paso - **Configuración** (.env) - **Ejecución** (comandos npm) - **Endpoints** disponibles con ejemplos - **Estructura** del proyecto - **Autores** del grupo

2. Comentarios en Código (Obligatorio)

- Comentar funciones complejas explicando qué hacen
- Documentar endpoints con descripción, parámetros y respuestas
- Explicar lógica de negocio no obvia

Requisitos de Trabajo en Grupo

1. Distribución de Trabajo

- Utilizar **Git** con múltiples commits descriptivos
- Cada miembro debe contribuir equitativamente
- **Mínimo 5 commits** por miembro con mensajes significativos
- Usar **branches** y **pull requests** si es posible

Commits NO válidos:

```
"cambios"
"fix"
"asdf"
```

Commits válidos:

```
"feat: Implementar CRUD de libros"
"fix: Corregir validación de email en registro"
"docs: Añadir documentación de endpoints"
```

```
"refactor: Mejorar estructura de repositorios"
```

2. Conocimiento Compartido

- **Todos** los miembros deben conocer **todas** las partes del código
- Durante la presentación se preguntará a **cualquier** miembro sobre **cualquier** parte
- Respuestas insatisfactorias restarán puntuación **individualmente**

3. Reparto Sugerido (Ejemplo para 3 personas)

Miembro	Responsabilidades
Miembro 1	Diseño de BD, setup inicial, modelos, documentación
Miembro 2	Repositorios, servicios, CRUD básico
Miembro 3	Consultas avanzadas, caché Redis, testing

Importante: Todos deben revisar y entender el código de los demás.

Criterios de Evaluación

Evaluación Individual (70%)

Criterio	Puntos	Descripción
Conocimiento del proyecto	50	Respuestas durante presentación
Contribución al equipo	20	Commits, participación, colaboración

Evaluación Grupal (30%)

Criterio	Puntos	Descripción
Funcionalidad completa	20	Todos los requisitos cumplidos
Presentación	10	Claridad, estructura, demostración

Penalizaciones

- **0 puntos** individuales si hay evidencia de no haber trabajado
- **-2 puntos** por cada requisito técnico no cumplido
- **-1 punto** por commit no descriptivo o muy pocos commits
- **-3 puntos** si se sube `.env` a Git
- **-5 puntos** si no se sigue arquitectura en capas

Formato de Entrega

Archivo PDF: `aad-psp_grupo_X.pdf`

Contenido del PDF: 1. **Portada:** - Nombre del proyecto - Integrantes del grupo (nombre completo) - Fecha de entrega

2. **Enlace al repositorio público GitHub**
3. **Descripción breve** del proyecto (máximo 200 palabras)
4. **Instrucciones de instalación** (resumidas)
5. **Endpoints implementados** (tabla con método, ruta, descripción)
6. **Diagrama de base de datos** (esquema de tablas y relaciones)
7. **Reparto de tareas** (tabla con miembro y responsabilidades)

Repositorio Git

- Debe ser **público**
- Incluir **README.md** completo
- Incluir `.gitignore` (sin `node_modules`, `.env`)
- Histórico de commits visible de todos los miembros

Presentación

Formato

- **Duración:** 8 minutos por grupo
- **Participación:** Equilibrada entre todos los miembros
- **Estructura sugerida:**
  1. Introducción del proyecto (1 min)
  2. Demostración en vivo (4 min)
  3. Explicación técnica (2 min)
  4. Conclusiones (1 min)

Demostración en Vivo

Mostrar funcionamiento de: - Operaciones CRUD básicas - Uso de Redis como caché - Al menos 1 consulta avanzada - Sistema de roles (intentar acceso admin sin permisos) - Rate limiting en acción

Herramientas recomendadas: Postman, Thunder Client, curl, cliente HTTP personalizado

Preguntas Frecuentes Durante Presentación

- ¿Por qué utilizaste esta arquitectura?
- ¿Cómo funciona el middleware de autenticación?

- ¿Qué hace este repositorio/servicio específico?
- ¿Cómo implementaste la caché con Redis?
- ¿Qué validaciones aplicas en este endpoint?
- ¿Cómo manejas los errores en esta capa?

Recursos Proporcionados

Boilerplate Inicial

- [OK] Estructura de carpetas completa
- [OK] Sistema de autenticación con API Keys
- [OK] Rate limiting con Redis
- [OK] Conexión a Supabase configurada
- [OK] Middlewares de autenticación y autorización
- [OK] Scripts de base de datos (setup/drop)
- [OK] Documentación de arquitectura

Debes Implementar

- [PENDIENTE] Diseño de base de datos (3+ tablas con relaciones)
- [PENDIENTE] Modelos de tu dominio de negocio
- [PENDIENTE] Repositorios para tus entidades
- [PENDIENTE] Servicios con lógica de negocio
- [PENDIENTE] Controladores para tus endpoints
- [PENDIENTE] Rutas de tu API
- [PENDIENTE] CRUD completo (mínimo 2 entidades)
- [PENDIENTE] Consultas avanzadas (mínimo 2)
- [PENDIENTE] Sistema de caché para consultas
- [PENDIENTE] Validaciones completas
- [PENDIENTE] Documentación (README, comentarios)
- [PENDIENTE] Tests (opcional pero recomendado)

Consejos Prácticos

Errores Comunes a Evitar

- [EVITAR] Mezclar lógica de negocio en controladores
- [EVITAR] Acceder directamente a BD desde controladores
- [EVITAR] No validar entradas del usuario
- [EVITAR] Subir `.env` a Git
- [EVITAR] Commits poco descriptivos
- [EVITAR] No manejar errores apropiadamente
- [EVITAR] Devolver códigos HTTP incorrectos
- [EVITAR] No documentar endpoints