Diseño de Arquitectura del Sistema - TaskManager

**Fecha:**

**Autor:**

**Versión:** 1.0

# 1. Introducción

Este documento describe de manera detallada la arquitectura del sistema para la aplicación web TaskManager, cuyo propósito es permitir a los usuarios gestionar tareas personales de forma intuitiva y eficiente. Se define la estructura general del sistema, los componentes que lo conforman, sus interacciones y las tecnologías empleadas.

# 2. Visón General de la Arquitectura

## 2.1. Estilo Arquitectónico

El sistema adopta una arquitectura en capas, basada en el patrón cliente-servidor, y se organiza en tres capas principales:

* Capa de Presentación (Frontend): Interfaz con la que interactúa el usuario.
* Capa de Lógica de Negocio (Backend): Contiene la lógica que gobierna las operaciones del sistema.
* Capa de Persistencia de Datos (Base de Datos): Encargada del almacenamiento estructurado y recuperación de información.

## 2.2. Diagrama General de Arquitectura:

Diagrama

El contenido generado por IA puede ser incorrecto.

# 3. Descripción de Componentes Principales

## 3.1. Frontend (Presentación)

* Framework: React.js
* Estilo: Tailwind CSS
* Responsabilidades:
  + Renderizar interfaz gráfica amigable
  + Capturar entradas del usuario
  + Comunicarse con el backend vía fetch/axios
  + Mostrar mensajes, errores y validaciones

## 3.2. Backend (Lógica de negocio)

* Lenguaje y Framework: Node.js con Express
  + Java 17 con Spring Boot
  + Spring Data JPA para acceso a base de datos
  + Spring Security para autenticación y autorización
  + MySQL como motor de base de datos relacional
  + Maven/Gradle como gestor de dependencias
* Responsabilidades:

Exponer API RESTful

* Controladores (@RestController) para exponer los servicios.
* Endpoints organizados por recursos (e.g., /usuarios, /tareas, /chatbot, /reportes).

Gestión de la lógica de negocio

* + Servicios (@Service) que aplican reglas de negocio:
  + Validación de fechas de tareas (vencidas, próximas a vencer).
  + Reglas para asignación de tareas a alumnos.
  + Generación de recomendaciones pedagógicas en base a la inteligencia predominante.
  + Control de reportes individuales y grupales.

**Persistencia y manejo de datos**

* Repositorios (@Repository) con **Spring Data JPA** para acceso a MySQL.
* ORM con **Hibernate**.
* Relaciones: Usuarios ↔ Tareas ↔ Resultados de Test.
* Migraciones de base de datos con **Flyway o Liquibase**.

**Integración con Chatbot (DeepSeek API)**

* Cliente HTTP (con **Spring WebClient o RestTemplate**) para enviar prompts y recibir respuestas.
* Adaptador en el servicio de negocio para formatear respuestas del chatbot.
* Gestión de roles (ej: docente puede consultar progreso, alumno recibe orientación personalizada).

**Seguridad**

* **JWT (JSON Web Tokens)** para autenticación y autorización.
* Roles: ALUMNO, DOCENTE, ADMIN.
* Protección de endpoints sensibles.
* Encriptación de contraseñas con **BCrypt**.

**Gestión de errores y validaciones**

* Manejo centralizado de excepciones con @ControllerAdvice.
* Validaciones con javax.validation (Bean Validation).
* Mensajes de error claros y consistentes.

### 3.2.1. Módulos del Backend

* authController.js: Manejo de login y registro
* taskController.js: Operaciones CRUD de tareas
* listController.js: Operaciones sobre listas de tareas
* middleware/auth.js: Validación de tokens JWT
* services/emailService.js: Envío de notificaciones

## 3.3. Base de Datos (Persistencia)

* Sistema Gestor: PostgreSQL
* ORM: Sequelize o Prisma
* Esquema de Datos:
  + Tabla ***users***: id, nombre, correo, contraseña (encriptada)
  + Tabla ***tasks***: id, título, descripción, fecha\_vencimiento, estado, user\_id, list\_id
  + Tabla ***lists***: id, nombre, user\_id

# 4. Integraciones Externas (Opcionales)

* Servicios de Notificación: EmailJS, SendGrid o Nodemailer para notificar tareas vencidas
* Autenticación Social: OAuth2 con Google (opcional en versiones futuras)
* Monitorización: Herramientas como LogRocket o Sentry

# 5. Seguridad

* Encriptación de contraseñas (bcrypt)
* Tokens JWT con expiración
* Validación y sanitización de entradas
* Uso de HTTPS.

# 6. Escalabilidad y Despliegue

Posibles problemas que pueden surgir durante el desarrollo.

* **Frontend**: Desplegado en Vercel o Netlify
* **Backend**: Deploy en Render, Railway o Heroku
* **Base de Dato**s: PostgreSQL en Supabase o ElephantSQL
* La arquitectura permite escalar horizontalmente el backend y separar la base de datos en instancias dedicadas

# 7. Conclusiones

La arquitectura propuesta busca lograr un balance entre simplicidad, robustez y escalabilidad. Se alinea con buenas prácticas de desarrollo web moderno y permite la extensión progresiva del sistema.