Principales decisiones técnicas tomadas durante el desarrollo

Este documento resume las decisiones más importantes del proyecto, por qué se tomaron y qué impacto tienen.

---

1) Stack tecnológico

Decisión:

- Backend con Node.js + TypeScript + Express + PostgreSQL.
- Frontend con React + TypeScript + Vite.

Justificación:

- TypeScript mejora robustez y mantenibilidad en ambos lados.
- Express permite una API clara y extensible.
- PostgreSQL aporta consistencia transaccional para el flujo de aprobación.
- Vite acelera desarrollo y build del frontend.

Trade-off:

- Mayor configuración inicial comparado con stacks más simples.

---

2) Arquitectura por capas en backend

Decisión:

- Separar en Routes -> Controller -> Service -> Repository.

Justificación:

- Claridad de responsabilidades.
- Lógica de negocio aislada del transporte HTTP.
- SQL centralizado en repositorios.

Impacto:

- Menor acoplamiento y cambios más seguros.

Trade-off:

- Más archivos/boilerplate al inicio.

---

3) Patrones complementarios

Decisión:

- Incorporar patrones adicionales para resolver problemas específicos del dominio:
  - RBAC por permisos
  - Strategy para normalización de rol
  - State/Flow control para el proceso de autenticación

Justificación:

- RBAC desacopla autorización de la lógica de endpoints.
- Strategy evita condicionales dispersos para normalizar entradas de rol.
- State/Flow hace explícitas las transiciones del proceso de registro/login y facilita validaciones por etapa.

Impacto:

- Seguridad más consistente.
- Menor duplicación de lógica.
- Flujo de autenticación más predecible y mantenible.

Trade-off:

- Aumenta ligeramente la complejidad conceptual para nuevos colaboradores.

---

4) Control de acceso RBAC por permisos

Decisión:

- Implementar autorización por permisos (TASK_VIEW_ALL, TASK_APPROVE_CHANGES, etc.) en lugar de validaciones por rol hardcodeadas en cada endpoint.

Justificación:

- Escala mejor cuando crecen roles o capacidades.
- Evita duplicar lógica de autorización.

Impacto:

- Reglas más explícitas y auditables.

Trade-off:

- Requiere mantener catálogos roles/permisos y su asignación.

---

5) Flujo de aprobación basado en solicitudes de cambio

Decisión:

- Modelar cambios de tareas mediante task_change_requests con estados PENDING/APPROVED/REJECTED.

Justificación:

- El problema exige separación entre quien solicita (STANDARD) y quien aprueba (SUPERVISOR).
- Permite historial y trazabilidad de cada solicitud.

Impacto:

- Flujo controlado y auditable para CREATE/UPDATE/COMPLETE/DELETE.

Trade-off:

- Mayor complejidad respecto a actualizar tareas directamente.

---

6) Lógica crítica en base de datos para aplicar cambios

Decisión:

- Centralizar aprobación/rechazo y aplicación de cambios en la función SQL apply_task_change_request.

Justificación:

- Garantiza atomicidad e integridad en una sola transacción.
- Reduce inconsistencias entre varios pasos de aplicación.

Impacto:

- Mayor seguridad de datos en escenarios concurrentes.

Trade-off:

- Parte de la lógica queda en SQL (requiere conocimiento de PL/pgSQL para evolucionar).

---

7) Modelo de datos con enums y JSONB

Decisión:

- Enums para estados/tipos (task_status, task_priority, change_type, approval_status).
- JSONB payload para cambios propuestos en solicitudes.

Justificación:

- Enums: consistencia fuerte de dominio.
- JSONB: flexibilidad para distintos tipos de cambios sin multiplicar tablas.

Impacto:

- Validación robusta + adaptabilidad del esquema.

Trade-off:

- JSONB requiere cuidado para validación de estructura en capa de servicio.

8) Soft delete en tareas

Decisión:

- Eliminar tareas lógicamente con is_active en lugar de borrado físico.

Justificación:

- Preserva histórico y referencias.
- Favorece auditoría y reportes.

Impacto:

- Mejor trazabilidad del ciclo de vida.

Trade-off:

- Todas las consultas deben filtrar correctamente registros activos.

---

9) Frontend organizado por features + Atomic Design

Decisión:

- Estructura de módulos por dominio funcional (auth, supervisor, standard).
- Componentes UI en niveles atoms/molecules/organisms.

Justificación:

- Escala mejor que organizar solo por tipo técnico.
- Incrementa reutilización y consistencia visual.

Impacto:

- Navegación de código más clara y mantenible.

Trade-off:

- Requiere disciplina de estructura para evitar duplicación.

---

10) Integración de pruebas con Postman

Decisión:

- Definir flujo de pruebas manuales por colecciones y variables de entorno.

Justificación:

- Permite validar rápidamente todos los casos funcionales y de permisos.
- Facilita demostración del sistema en evaluación técnica.

Impacto:

- Menor fricción para QA manual y handoff.

Trade-off:

- No reemplaza pruebas automatizadas de integración/E2E.

---

11) Contenerización de entorno backend + BD

Decisión:

- Uso de Docker Compose para PostgreSQL y backend.

Justificación:

- Entorno reproducible entre máquinas.
- Menor fricción de setup local.

Impacto:

- Onboarding más rápido.

Trade-off:

- Dependencia de Docker Desktop en desarrollo.

---

12) Seguridad y robustez básica

Decisión:

- JWT para autenticación.
- Hash de contraseñas con bcrypt.
- Middlewares de autenticación/autorización.
- Validaciones de negocio en capa de servicio.

Justificación:

- Estándar adecuado para un sistema de gestión con roles.

Impacto:

- Protección de endpoints y coherencia de reglas.

Trade-off:

- Requiere disciplina en manejo de tokens y variables sensibles.

---

13) Conclusión técnica

La combinación de arquitectura por capas, RBAC, flujo de aprobación y modelo relacional con trazabilidad responde directamente al objetivo del producto: controlar cambios de tareas por unidad organizacional con supervisión formal.

Estas decisiones priorizan mantenibilidad, integridad de datos y capacidad de crecimiento, aceptando como costo una complejidad inicial mayor respecto a una implementación monolítica y sin flujo de aprobación.