ARQUITECTURA

Task Management System

31 DE DICIEMBRE DE 2024

EDGARDO BOLIVAR

Tabla de contenido

Propósito de este documento

Este documento técnico detalla el diseño, arquitectura e implementación del Sistema de Gestión de Tareas, desarrollado usando NestJS y React. Su objetivo es proporcionar una guía técnica completa para el entendimiento, mantenimiento y evolución del sistema.

Alcance

Este sistema abarca la gestión de usuarios y tareas a través de una API RESTful con su respectivo frontend. Incluye operaciones CRUD completas, paginación, gestión de estados de tareas y validaciones tanto en backend como frontend. La documentación cubre la arquitectura del sistema, patrones de diseño implementados y guías de configuración, excluyendo procesos de DevOps y configuraciones de producción.

Partes interesadas y preocupaciones

La identificación de stakeholders y sus preocupaciones es crucial para asegurar que el sistema cumpla con las expectativas de todos los involucrados. Cada grupo tiene necesidades específicas que impactan directamente en el diseño y desarrollo del sistema.

- Desarrolladores: Necesitan documentación clara sobre la arquitectura y patrones implementados.
- Equipo de QA: Requieren documentación de endpoints y casos de prueba.
- Gerentes de Proyecto: Necesitan visibilidad del alcance y funcionalidades implementadas.
- Mantenedores: Requieren guías de configuración y documentación de dependencias.

Definición de los requisitos de alto nivel

Estos requisitos de alto nivel abarcan los aspectos funcionales y no funcionales clave de la aplicación. Proporcionan una visión general de las principales capacidades que debe tener la aplicación, así como las consideraciones de rendimiento, seguridad, usabilidad y compatibilidad.

Requisitos funcionales:

- Gestión completa de usuarios (CRUD)
- Gestión de tareas asignables a usuarios
- Sistema de estados para tareas
- Paginación de listados
- Validaciones en ambas capas

Requisitos no funcionales:

- Tiempo de respuesta < 2 segundos
- Interfaz responsiva
- Mensajes de error claros

Restricciones:

- Stack tecnológico definido: NestJS/React
- Base de datos PostgreSQL/MongoDB
- TypeScript obligatorio

Selección del patrón de arquitectura

Después de analizar los requisitos funcionales y no funcionales de la aplicación, se han seleccionado los siguientes patrones arquitectónicos para el diseño del backend y frontend:

Se utiliza el patrón de **Arquitectura de Microservicios**. Aunque es una versión simplificada, el frontend y backend son servicios independientes que:

1. Se comunican vía API REST
2. Pueden desarrollarse/desplegarse independientemente
3. Tienen responsabilidades claramente separadas
4. Permiten escalamiento independiente

Backend (NestJS):

1. **Arquitectura Modular:** El sistema se divide en módulos independientes (usuarios, tareas) facilitando el mantenimiento y permitiendo que diferentes equipos trabajen en paralelo sin conflictos.
2. **Patrón Repository:** Abstrae la lógica de acceso a datos, permitiendo cambiar la base de datos sin afectar la lógica de negocio. Útil para pruebas y mantenimiento.
3. **Inyección de Dependencias:** Facilita el testing y permite intercambiar implementaciones sin modificar el código cliente. Los servicios son más fáciles de mantener y testear.
4. **Service Layer:** Separa la lógica de negocio de los controladores, mejorando la reusabilidad y mantenibilidad del código.

Frontend (React):

1. **Arquitectura de Componentes:** Permite reutilizar elementos UI y mantener un código consistente. Facilita el testing y la colaboración entre equipos.
2. **Container/Presenter Pattern:** Separa la lógica de negocio de la presentación, mejorando la mantenibilidad y permitiendo cambios en la UI sin afectar la lógica.
3. **Custom Hooks:** Encapsula lógica compartida, reduciendo la duplicación de código y facilitando las actualizaciones globales.

La arquitectura implementada combina microservicios con patrones específicos para cada capa. En backend, los patrones modulares y de repositorio garantizan un código mantenible y testeable, mientras que en frontend la arquitectura de componentes y hooks optimiza la reutilización y mantenimiento del código. Esta estructura permite que el sistema sea escalable, mantenible y fácil de evolucionar, cumpliendo con los requisitos establecidos y facilitando el trabajo colaborativo entre equipos.

Definición de la arquitectura inicial

En esta sección, se presenta la arquitectura inicial de la aplciacion utilizando el modelo C4 (Context, Containers, Components, Code). El modelo C4 nos permite visualizar y comunicar la arquitectura del sistema en diferentes niveles de abstracción.

El modelo completo puede verlo en https://s.icepanel.io/g9JlWUd1ByVtBZ/zEbE

Diagrama de Contexto (Nivel 1):

Propósito: Mostrar el sistema en su contexto, identificando los sistemas externos y los usuarios que interactúan con él.

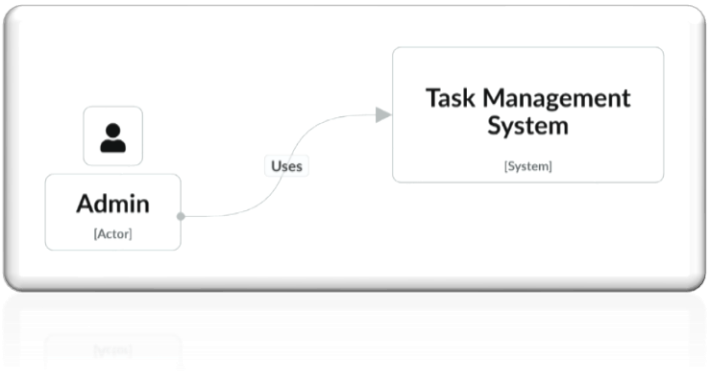


Diagrama de Contenedores (Nivel 2):

Propósito: Descomponer el sistema en contenedores (aplicaciones o servicios) y mostrar las tecnologías utilizadas y las interacciones entre ellos.

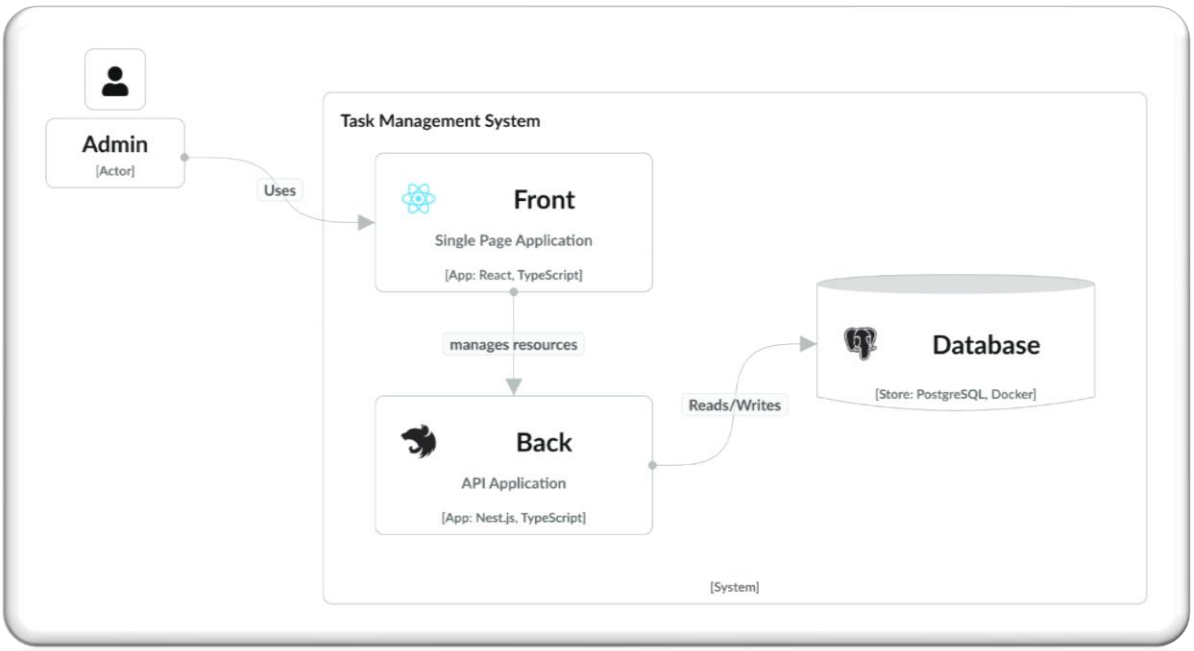
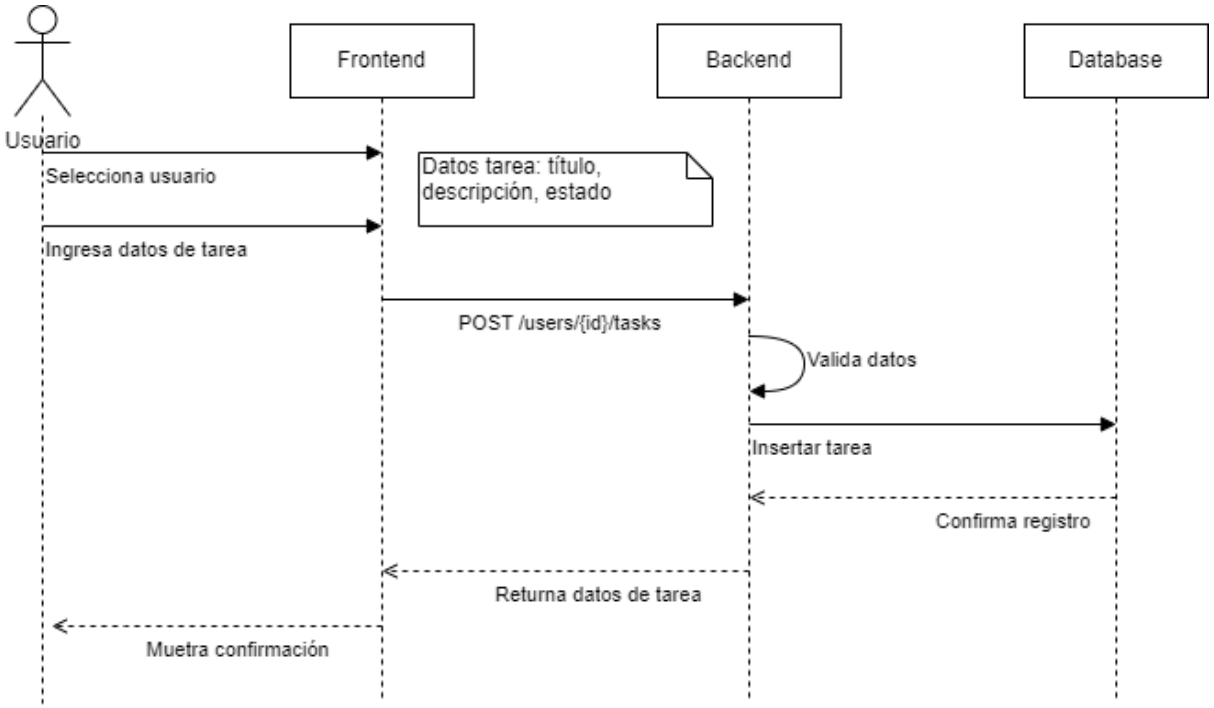


Diagrama de Secuencia:



El diagrama representa el flujo secuencial para asignar una tarea a un usuario en el sistema. Inicia cuando el usuario selecciona un destinatario y completa el formulario de tarea. La solicitud viaja desde el frontend al backend a través de una llamada POST, donde se validan los datos antes de almacenarlos en la base de datos. El proceso finaliza con la confirmación visual al usuario.

Estructura del Proyecto

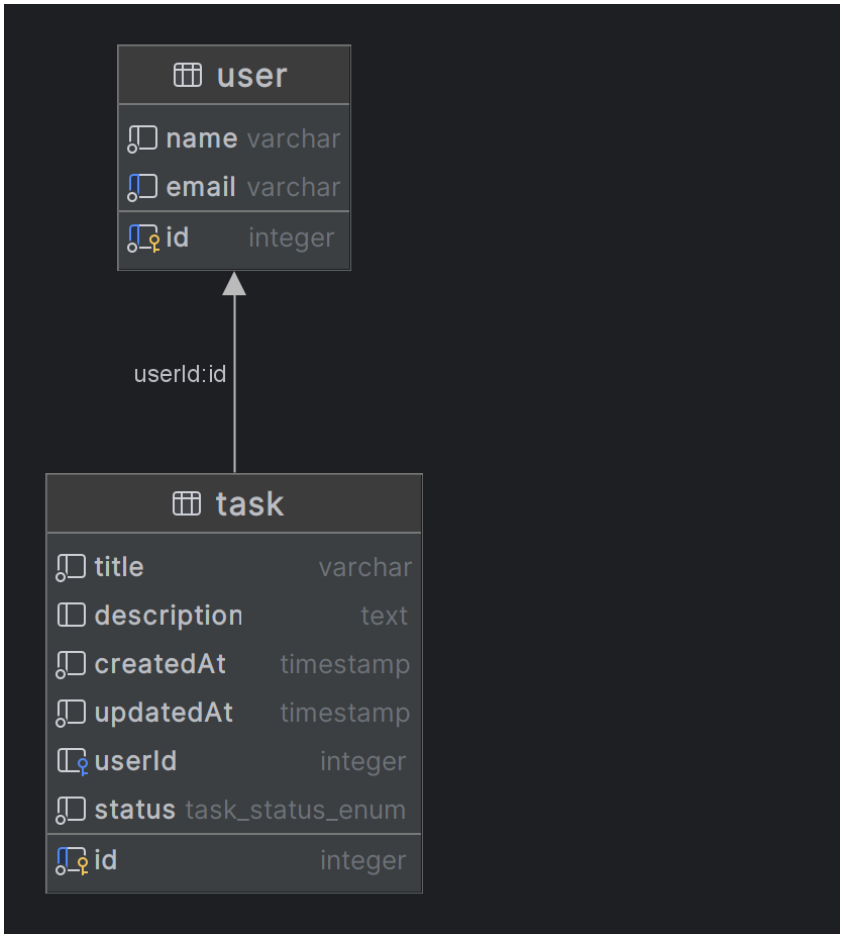
Para mantener el código modular, mantenible y claramente segmentado por módulos de aplicación, se define la siguiente estructura de directorios:

```
task-management-system/
├── backend/                          # NestJS API
│   ├── src/
│   │   ├── app.controller.ts
│   │   ├── app.module.ts
│   │   ├── app.service.ts
│   │   ├── config/
│   │   │   └── database.config.ts
│   │   ├── tasks/
│   │   │   ├── dto/
│   │   │   ├── entities/
│   │   │   ├── enums/
│   │   │   ├── tasks.controller.ts
│   │   │   ├── tasks.module.ts
│   │   │   └── tasks.service.ts
│   │   └── users/
│   │       ├── dto/
│   │       ├── entities/
│   │       ├── users.controller.ts
│   │       ├── users.module.ts
│   │       └── users.service.ts
│   ├── test/
│   ├── .env
│   └── README.md
│
├── frontend/                         # React Application
│   ├── src/
│   │   ├── api/
│   │   │   ├── config.ts
│   │   │   ├── task.api.ts
│   │   │   └── user.api.ts
│   │   ├── components/
│   │   │   ├── shared/
│   │   │   ├── tasks/
│   │   │   └── users/
│   │   ├── interfaces/
│   │   ├── models/
│   │   ├── pages/
│   │   ├── utils/
│   │   ├── App.tsx
│   │   └── main.tsx
│   ├── public/
│   ├── index.html
│   └── README.md
│
├── postman/                          # API Documentation
│   └── task-management-api.postman_collection.json
│
├── docker-compose.yml                # Docker configuration
└── README.md                         # Project documentation
```

Esta estructura mantiene la separación de responsabilidades del código base mientras segmenta claramente las funcionalidades específicas de cada módulo del sistema.

Definición del modelo de datos

El sistema utiliza dos entidades principales: User y Task. User almacena información básica del usuario con email único. Task mantiene el registro de tareas con estados (PENDING, IN_PROGRESS, COMPLETED) y timestamps de auditoría. La relación uno-a-muchos entre User y Task permite asignar múltiples tareas a un usuario.



Decisiones Arquitectónicas

Backend: NestJS

- Framework modular basado en decoradores que aprovecha TypeScript
- Arquitectura basada en SOLID y DDD
- Soporte nativo para inyección de dependencias
- CLI integrada para generación de código
- Excelente documentación y comunidad activa

Frontend: React

- Virtual DOM para renderizado eficiente
- Ecosistema robusto de componentes y librerías
- Soporte nativo para TypeScript
- Hooks para manejo de estado y efectos

Base de Datos: PostgreSQL

- ACID compliant para integridad de datos
- Soporte robusto para relaciones y constraints
- Buen rendimiento para operaciones CRUD
- Amplio soporte en ORMs y herramientas

TypeScript

- Tipado estático que reduce errores en desarrollo
- Interfaces y tipos para contratos claros

Patrón Repository

- Abstracción de la capa de datos
- Facilita cambios en la fuente de datos
- Centraliza lógica de acceso a datos

Container/Presenter Pattern

- Separación clara de lógica y UI
- Componentes más mantenibles
- Facilita testing unitario
- Mejor reutilización de código
- Reduce acoplamiento