Sistema de Reservas para Espacios Comunitarios

Maestrante: Wilver Vargas Anagua

explicacion del proyecto

Este proyecto implementa un sistema de gestión de reservas para espacios comunitarios, evolucionando desde una arquitectura modular tradicional hacia una arquitectura CQRS (Command Query Responsibility Segregation). La migración se realizó para abordar desafíos específicos de escalabilidad, rendimiento y mantenibilidad.

Arquitectura del Sistema

Evolución Arquitectónica

Fase 1: Arquitectura Modular

La implementación inicial seguía un patrón modular tradicional:

```
app-modular/
├── src/
│   ├── modules/
│   │   ├── users/
│   │   │   ├── controllers/
│   │   │   ├── services/
│   │   │   ├── dto/
│   │   │   └── entities/
│   │   ├── spaces/
│   │   ├── reservations/
│   │   └── notifications/
│   └── shared/
```

Problemas Identificados:

- Acoplamiento fuerte entre capas
- Dificultad para escalar operaciones de lectura/escritura

- Complejidad en el manejo de transacciones
- Limitaciones en la implementación de caché

Fase 2: Arquitectura CQRS

La nueva arquitectura implementa CQRS con los siguientes componentes:

```
app-cqrs/
├── src/
│   ├── modules/
│   │   ├── users/
│   │   │   ├── commands/
│   │   │   │   ├── create-user/
│   │   │   │   └── update-user/
│   │   │   ├── queries/
│   │   │   │   ├── get-user/
│   │   │   │   └── list-users/
│   │   │   └── events/
│   │   │       ├── user-created/
│   │   │       └── user-updated/
│   │   ├── spaces/
│   │   ├── reservations/
│   │   └── notifications/
│   └── shared/
```

Patrones y Principios Aplicados

1. CQRS (Command Query Responsibility Segregation)

- **Comandos**: Operaciones de escritura que modifican el estado

```
export class CreateReservationCommand {
  constructor(
    public readonly spaceId: string,
    public readonly userId: string,
    public readonly date: Date
  ) {}
}
```

- **Queries**: Operaciones de lectura que no modifican el estado

```
export class GetSpaceAvailabilityQuery {
  constructor(
    public readonly spaceId: string,
    public readonly date: Date
  ) {}
}
```

2. Event Sourcing

- Eventos de dominio para tracking de cambios
- Reconstrucción del estado a partir de eventos
- Mejor trazabilidad y debugging

3. Domain-Driven Design (DDD)

- Agregados claramente definidos
- Bounded Contexts
- Value Objects y Entities

Atributos de Calidad

1. Escalabilidad

- **Horizontal**: Separación de comandos y queries permite escalar independientemente
- **Vertical**: Optimización de recursos por tipo de operación
- **Caché**: Implementación de caché en el lado de lectura

2. Mantenibilidad

- **Modularidad**: Componentes independientes y cohesivos
- **Testabilidad**: Facilidad para escribir pruebas unitarias
- **Documentación**: Código autoexplicativo y documentado

3. Rendimiento

- **Optimización de Queries**: Modelos de lectura optimizados
- **Concurrencia**: Mejor manejo de operaciones concurrentes
- **Latencia**: Reducción de tiempos de respuesta

Decisiones Arquitectónicas (ADR)

ADR-001: Migración a CQRS

Contexto:

- Necesidad de mejorar el rendimiento en operaciones de lectura
- Escalabilidad horizontal requerida
- Complejidad creciente en el manejo de transacciones

Decisión:
Implementar CQRS separando operaciones de lectura y escritura

Consecuencias:

- Mejor rendimiento en operaciones de lectura
- Escalabilidad horizontal
- Mayor complejidad inicial
- Duplicación de modelos

ADR-002: Implementación de Event Sourcing

Contexto:

- Necesidad de mejor trazabilidad
- Requisitos de auditoría
- Desacoplamiento de componentes

Decisión:
Implementar Event Sourcing para tracking de cambios

Consecuencias:

- Mejor trazabilidad
- Facilidad para debugging

Estrategia de Pruebas

1. Pruebas Unitarias

- Comandos y Queries
- Eventos

- Value Objects
- Servicios de dominio

Principales Cambios en la Migración

1. **Separación de Comandos y Consultas**
    - Los comandos (mutaciones) y consultas (lecturas) ahora están separados
    - Mejor manejo de la concurrencia
    - Optimización de rendimiento para operaciones de lectura
2. **Eventos de Dominio**
    - Implementación de eventos para mejor desacoplamiento
    - Mejor trazabilidad de cambios
    - Facilita la implementación de nuevas características
3. **Optimización de Base de Datos**
    - Modelos de lectura y escritura separados
    - Mejor rendimiento en operaciones de consulta
    - Escalabilidad horizontal

Stack Tecnológico

- **Framework**: NestJS
- **Lenguaje**: TypeScript
- **Base de Datos**: SQLite
- **ORM**: Prisma
- **Testing**: Jest

Conclusión

La migración a CQRS ha permitido mejorar significativamente la escalabilidad y mantenibilidad del sistema, aunque con un costo inicial en términos de complejidad. Los beneficios obtenidos justifican la decisión arquitectónica tomada, especialmente considerando los requisitos futuros del sistema.