INFORME FINAL DEL SISTEMA DE RESERVAS DE ESPACIOS COMUNES

Contexto del Proyecto

Este proyecto consiste en el desarrollo de un sistema de reservas de espacios comunes, la solución implementa una arquitectura moderna escalable y mantenible, distribuida en frontend, backend y desarrollada con Angular y Node.js (con typeScript) respectivamente.

Propuesta de Proyecto

Sistema de Reservas para Espacios Comunitarios:

Una aplicación web donde los usuarios pueden:

- Registrarse en el sistema con validación de credenciales
- Ver disponibilidad de espacios (salones, auditorios, canchas) en tiempo real
- Reservar un espacio por día y hora específica
- Recibir confirmación por email (implementado como notificación local en el backend)
- Consultar sus reservas activas y historial personal

Los administradores pueden:

- Ver todas las reservas por fecha y tipo de espacio
- Gestionar espacios disponibles (solo ver y crear)
- Monitorear ocupación

Funcionalidades adicionales implementadas:

- Validación automática de conflictos de horarios
- Sistema de estados de reserva (pendiente, confirmada, cancelada)
- Búsqueda y filtrado de espacios por tipo y capacidad
- Autenticación segura con JWT

Arquitectura General

Estilo Arquitectónico Aplicado: Onion Architecture

Se ha implementado el patrón Onion Architecture debido a sus ventajas para separar responsabilidades, garantizar testabilidad y mantener independencia de frameworks. Esta arquitectura se refleja tanto en el backend como en el frontend, organizando el código en capas concéntricas donde las dependencias apuntan hacia el centro.

Modularidad

El sistema está dividido en módulos funcionales claramente definidos:

- **Usuarios**: Autenticación, registro y gestión de perfiles
- **Espacios**: Creación, consulta y administración de espacios físicos
- **Reservas**: Lógica de reservas, validaciones y confirmaciones
- **Notificaciones**: Sistema de confirmaciones y alertas(mock)

Cada módulo contiene su propio conjunto de controladores, casos de uso, entidades y repositorios, siguiendo el principio de separación de responsabilidades.

Tecnologías Utilizadas

Frontend:

Angular 17+ con TypeScript, standalone components para mejor modularidad, angular Signals para manejo de estado reactivo , tailwind CSS para estilos utilitarios y lazy loading para optimización de carga.

Backend:

Node.js con TypeScript para tipado fuerte, express.js como framework web, sequelize ORM para abstracción de base de datos, postgreSQL como base de datos relacional, JWT + bcrypt para autenticación seguro y Jest para testing unitario.

Casos de Uso Críticos Probados

Los siguientes casos de uso fueron identificados como críticos para el funcionamiento del sistema y cuentan con pruebas unitarias exhaustivas:

1. VerificarDisponibilidadEspacio

- **Funcionalidad**: Validacion de disponibilidad de espacios en horarios específicos
- **Casos probados**:
    - Validación de existencia del espacio
    - Detección de conflictos de horarios
    - Filtrado de reservas canceladas
    - Manejo de espacios no disponibles
- **Cobertura**: 100% statements, 63.63% branches, 100% functions

2. CrearReserva

- **Funcionalidad**: Creacion de nuevas reservas con validaciones
- **Casos probados**:

    Validación de datos de entrada

    Verificación de disponibilidad previa

    Prevención de reservas duplicadas

    Manejo de errores de negocio
- **Cobertura**: 96.15% statements, 80% branches, 100% functions

3. Sistema de Excepciones

- **Funcionalidad**: Manejo centralizado de errores de dominio
- **Casos probados**:

    AuthenticationException para errores de autenticacion

    AuthorizationException para errores de permisos

    BusinessRuleException para violaciones de reglas de negocio

    ValidationException para errores de validación
- **Cobertura**: 100% statements, 100% branches, 100% functions

Justificación de criticidad: Estos casos de uso forman el núcleo funcional del sistema. Su correcto funcionamiento es esencial para garantizar la integridad de las reservas, evitar conflictos de horarios y mantener la seguridad del sistema.

Atributos de Calidad

1. Mantenibilidad

- **Implementación**: Código modular y reutilizable con separación clara de responsabilidades
- **Evidencia**: Arquitectura en capas, tipado fuerte en ambos entornos
- **Beneficio**: Facilita modificaciones futuras y reduce tiempo de desarrollo

2. Testabilidad

- **Implementación**: Inyección de dependencias, casos de uso independientes, interfaces abstractas
- **Evidencia**: Cobertura de testing del 98.71% en statements y 100% en funciones
- **Beneficio**: Detección temprana de errores y confiabilidad del sistema

3. Seguridad

- **Implementación**: Autenticación con JWT, contraseñas hasheadas con bcrypt, middleware de autorización
- **Evidencia**: Tokens con expiración, validación de permisos en cada endpoint
- **Beneficio**: Protección de datos de usuarios y prevención de accesos no autorizados

4. Escalabilidad

- **Implementación**: Arquitectura modular, lazy loading, separación frontend/backend
- **Evidencia**: Posibilidad de desacoplamiento con eventos, componentes independientes
- **Beneficio**: Capacidad de crecimiento sin reestructuración mayor

Decisiones Arquitectónicas (ADR)

ADR-001: Adopción de Onion Architecture para el Backend

Contexto: El sistema de reservas requiere alta mantenibilidad y testabilidad debido a su naturaleza académica y potencial crecimiento futuro. Se necesitaba una arquitectura que permitiera cambios en tecnologías sin afectar la lógica de negocio.

Opciones Consideradas:

1. **MVC Tradicional**: Implementación rápida pero alto acoplamiento entre capas
2. **Clean Architecture**: Separación completa pero complejidad excesiva para el alcance
3. **Onion Architecture**: Balance óptimo entre separación y simplicidad

Decisión: Implementar Onion Architecture con las siguientes capas:

- Core: Entidades y reglas de negocio puras
- Application: Casos de uso e interfaces de repositorios
- Infrastructure: Implementación de repositorios y servicios externos
- Presentation: Controladores HTTP y validaciones de entrada

Justificación:

- Permite testing independiente de frameworks y base de datos
- Facilita cambios en la capa de persistencia sin afectar lógica de negocio
- Mejora la mantenibilidad mediante inversión de dependencias

- Cumple con principios SOLID, especialmente inversión de dependencias

Consecuencias:

- Ventajas: Alta testabilidad, mantenibilidad mejorada, independencia de frameworks
- Desventajas: Mayor complejidad inicial, curva de aprendizaje para nuevos desarrolladores

ADR-002: JWT vs Sessions para Autenticación

Contexto: La aplicación requiere autenticación segura que soporte tanto aplicaciones web como potenciales clientes móviles futuros. Se necesitaba una solución stateless que facilite la escalabilidad horizontal.

Opciones Consideradas:

1. **Sessions con Cookies**: Manejo de estado en servidor revocación fácil
2. **JWT**: Tokens stateless, compatible con múltiples plataformas
3. **OAuth 2.0**: Delegación a terceros, complejidad adicional

Decisión: Implementar JWT con expiración de 24 horas, almacenado en localStorage del cliente.

Justificación:

Stateless: facilita escalabilidad horizontal sin compartir estado

Compatible con SPA y futuras aplicaciones móviles

Reduce carga en servidor al no almacenar sesiones

Estándar de la industria para APIs REST modernas

Consecuencias:

- Ventajas: Escalabilidad mejorada, soporte multi-plataforma, menor carga de servidor
- Desventajas: Dificultad para revocación inmediata, tamaño de payload mayor

ADR-003: Angular Signals vs RxJS para Estado Global

Contexto: La aplicación requiere manejo de estado reactivo para reservas, espacios disponibles y usuario autenticado. Se necesitaba una solución que simplifique el código sin sacrificar funcionalidad.

Opciones Consideradas:

1. **RxJS + Services**: Patrón tradicional de Angular, ampliamente documentado
2. **NgRx**: Store management completo, ideal para aplicaciones grandes
3. **Angular Signals**: Nueva API reactiva, más simple que RxJS

Decisión: Utilizar Angular Signals para estado local y componentes, manteniendo RxJS únicamente para operaciones HTTP asíncronas.

Justificación:

- API más intuitiva que RxJS para casos de uso simples
- Mejor performance con el nuevo sistema de detección de cambios
- Menos boilerplate comparado con NgRx para el alcance del proyecto
- Alineado con el roadmap oficial de Angular

Consecuencias:

- Ventajas: Código más legible, mejor performance, menos complejidad
- Desventajas: Requiere Angular 16+, documentación limitada comparado con RxJS

Trade-offs del Diseño

1. Complejidad vs Mantenibilidad

Decisión tomada: Mayor complejidad arquitectónica inicial **Sacrificio**: Simplicidad en la implementación inicial **Justificación**: Facilitar mantenimiento y evolución a largo plazo **Ejemplo concreto**: Implementación de interfaces para repositorios requiere más codigo pero permite testing independiente y cambio de tecnologías de persistencia

2. Performance vs Flexibilidad

Decisión tomada: Flexibilidad mediante inyección de dependencias **Sacrificio**: Overhead mínimo en tiempo de ejecución **Justificación**: Facilitar testing y intercambio de implementaciones **Ejemplo concreto**: Uso de interfaces abstractas vs acceso directo a base de datos añade una capa pero permite mocking en tests

3. Consistencia vs Inmediatez

Decisión tomada: Validaciones exhaustivas antes de confirmar reservas **Sacrificio**: Tiempo de respuesta ligeramente mayor **Justificación**: Evitar conflictos de reservas y garantizar integridad de datos **Ejemplo concreto**: Verificación de disponibilidad en tiempo real requiere múltiples consultas pero previene reservas duplicadas

4. Tamaño de Bundle vs Funcionalidad

Decisión tomada: Standalone components y lazy loading **Sacrificio**: Complejidad en la configuración de rutas **Justificación**: Mejorar tiempo de carga inicial de la aplicación **Ejemplo concreto**: Carga diferida de módulos vs bundle monolítico reduce tiempo inicial pero requiere gestión de estados de carga

Consideraciones Finales

El sistema de reservas de espacios comunes implementa patrones arquitectónicos modernos y buenas prácticas tanto en frontend como backend. El uso de Onion Architecture, modularidad funcional, autenticación segura con JWT, y pruebas unitarias de los principales casos de uso lo cual lo hace un sistema robusto, mantenible, testeable y escalable.

Las métricas de cobertura de testing (98.71% statements, 100% functions) demuestran la calidad y confiabilidad del código implementado. Las decisiones arquitectónicas documentadas en los ADRs proporcionan trazabilidad y justificación para futuras modificaciones.

Este informe documenta las decisiones técnicas clave, atributos de calidad implementados, tecnologías utilizadas y los trade-offs asumidos durante el desarrollo.