UNIVERSIDAD ESCUELA COLOMBIANA DE NGENIERÍA JULIO GARAVITO

DOCUMENTO DE ARQUITECTURA

ECI-Bienestar

Equipo Diamante

AUTORES

Vicente Garzón Ríos

Daniel Alejandro Diaz Camelo

MODULO

Gestión de Turnos para Servicios de Bienestar Universitario

PROGRAMA

Ingeniería de Sistemas

ASIGNATURA

Ciclos de Vida del Desarrollo de Software (CVDS)

PROFESORES

Ing. Rodrigo Humberto Gualtero Martínez

Ing. Andrés Martín Cantor Urrego

Descripción del Modulo

Este módulo permite a los miembros de la comunidad universitaria (estudiantes, docentes, administrativos y personal de servicios generales) gestionar y visualizar turnos para atención en los servicios de bienestar institucional: medicina general, odontología y psicología.

El sistema contempla la asignación de turnos desde tablets de autoservicio, control administrativo por parte del personal autorizado y seguimiento por parte de los profesionales de la salud.

Perfiles de usuario

Paciente:

- Estudiante.
- Docente.
- Administrativo.
- Servicios generale.

Doctor.

Administrador:

- Secretaria medica

Requisitos funcionales

1. Registro de turnos por parte de los usuarios

- El sistema debe permitir a los usuarios registrarse al llegar a las oficinas de bienestar mediante una interfaz en una tableta.
- El sistema debe permitir al usuario ingresar su nombre completo y número de documento de identidad.
- El sistema debe permitir seleccionar su rol dentro de la institución: Estudiante, Docente, Administrativo o Servicios Generales.

- El sistema debe permitir elegir una especialidad disponible: Medicina General, Odontología o Psicología.
- El sistema debe permitir marcar prioridad en caso de condiciones especiales como embarazo o discapacidad.
- El sistema debe generar y mostrar una confirmación visual del turno asignado, en formato "letra-número" (ej. “O-15” para Odontología).
- El sistema debe permitir deshabilitar temporalmente la asignación de nuevos turnos en caso de emergencia.

2. Gestión de disponibilidad por el administrador

- El sistema debe permitir al administrador habilitar o deshabilitar la asignación de turnos para cada especialidad según la disponibilidad del personal.
- El sistema debe permitir habilitar o deshabilitar todos los turnos globalmente en caso necesario.

3. Visualización de turnos y contenido multimedia

- El sistema debe mostrar en pantalla el turno actualmente llamado, incluyendo nombre del usuario.
- El sistema debe mostrar en la misma pantalla contenido multimedia informativo (GIF o MP4) sobre los servicios de bienestar universitario.
- El sistema debe permitir que solo personal autorizado pueda subir, modificar o eliminar el contenido multimedia.

4. Interfaz para profesionales de la salud

- El sistema debe permitir a los profesionales de salud ver la lista de turnos pendientes por atender.
- El sistema debe permitir al profesional llamar al siguiente turno disponible.

- El sistema debe permitir al profesional seleccionar un turno específico si lo considera necesario.

5. Actualización dinámica de turnos

- El sistema debe actualizar automáticamente la pantalla de visualización al momento de que un usuario sea atendido, eliminando su turno de la lista.
- El sistema debe mostrar la lista actualizada de los siguientes turnos programados.

6. Generación de informes

- El sistema debe permitir al administrador generar reportes detallados sobre los turnos atendidos.
- El sistema debe permitir filtrar los reportes por rango de fechas y por rol del usuario (Estudiante, Docente, etc.).
- El sistema debe mostrar estadísticas como el número de turnos por especialidad y el nivel promedio de atención brindado.

Tecnologías a usar

Categoría	Tecnología	Justificación
Lenguaje	Java 17	Estabilidad, soporte LTS (Long Term Support).
Framework principal	Spring Boot	Estandarizado para microservicios RESTful.
Dase de datos	PostgreSQL	Robusto, relacional y gratuito. Ideal para consistencia transaccional.
Mensajería / Bus de datos	Apache Kafka + Spring Cloud Bus	Comunicación asíncrona y eventos de emergencia / actualización de estados.
Seguridad	JWT (JSON Web Token)	Autenticación segura, stateless.
ORM / Utilidades	Lombok	Para reducir código boilerplate (constructores, getters/setters

		automáticos).
Gestor de proyecto	Maven	Construcción de proyectos Java estandarizada.
Actualización	WebSocket	Para actualizar la pantalla de turnos dinámicamente sin recargar.
Herramientas adicionales	JaCoCo, SonarQube	Control de calidad de código y cobertura de pruebas.
Documentación de API	Swagger / Springdoc OpenAPI	Generación automática y visualización interactiva de la documentación de APIs.

Comparativa de Tecnologías Alternativas Evaluadas:

Node.js vs Java

Se optó por Java en lugar de Node.js debido a varias razones clave asociadas al contexto del proyecto:

- **Robustez y madurez:** Java es un lenguaje consolidado con décadas de evolución y es ampliamente usado en sistemas críticos, especialmente en entornos empresariales, donde se valora la estabilidad a largo plazo.
- **Soporte empresarial:** El ecosistema Java (especialmente con frameworks como Spring Boot) está diseñado para aplicaciones empresariales escalables, seguras y mantenibles. Cuenta con herramientas integradas para autenticación, manejo de errores, pruebas automatizadas y más.
- **Tipado estático y control estricto:** A diferencia de Node.js (JavaScript/TypeScript), Java permite detectar más errores en tiempo de compilación, lo que reduce problemas en producción.
- **Multihilo y concurrencia:** Java ofrece un manejo avanzado de concurrencia, útil en escenarios donde múltiples procesos pueden estar interactuando con el sistema (como la atención simultánea de turnos).

Por estas razones, Java fue la opción más adecuada para una solución sólida, con requerimientos estructurados y alta expectativa de mantenibilidad.

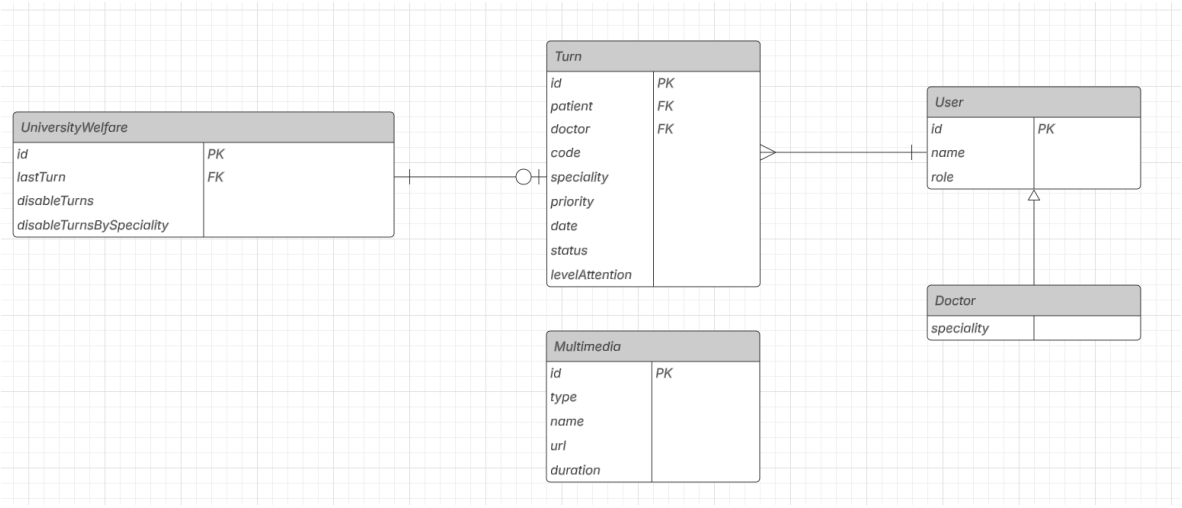
MongoDB vs PostgreSQL

Aunque el sistema actual gestiona un conjunto relativamente pequeño de entidades (usuarios, turnos y multimedia), se optó por PostgreSQL debido a que:

- Modelo relacional estructurado: A pesar del número limitado de tablas, las entidades tienen estructuras definidas y relaciones directas entre sí (por ejemplo, un turno asociado a un usuario). PostgreSQL permite modelar estas relaciones de manera clara, utilizando claves foráneas y restricciones que aseguran la consistencia de los datos.
- Integridad de datos garantizada: PostgreSQL ofrece mecanismos nativos para validar la integridad (como unicidad de usuarios, o restricciones de formato en los turnos), lo cual es fundamental en sistemas donde no puede haber ambigüedad ni duplicidad, especialmente en procesos secuenciales como la gestión de turnos.
- Escalabilidad estructurada: Aunque actualmente son pocas tablas, el modelo relacional facilita una evolución controlada del sistema en el futuro (por ejemplo, agregando especialidades, reportes o historiales de atención) sin perder coherencia.
- Consultas analíticas y filtros: PostgreSQL brinda capacidades avanzadas de consultas SQL, útiles para reportes y análisis (fechas, conteos por usuario, estados de turnos), que pueden ser más complejas de optimizar en un modelo NoSQL como MongoDB.

En resumen, se eligió PostgreSQL no por la cantidad de entidades, sino por la necesidad de mantener consistencia, validación y claridad en la lógica de datos, fundamentales incluso en sistemas con pocos modelos.

Diagrama de datos



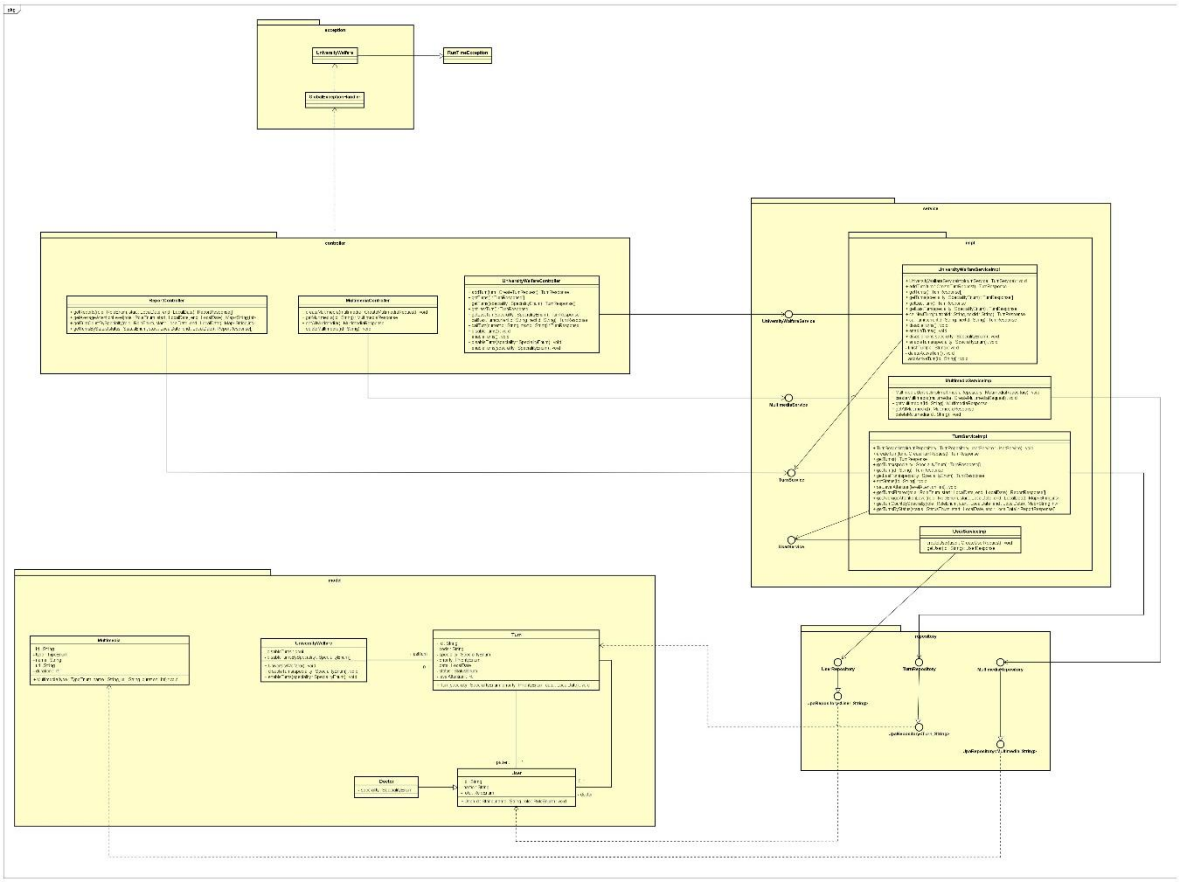
El sistema utiliza una base de datos relacional (PostgreSQL) con tablas que representan entidades clave como usuarios, turnos médicos, contenidos multimedia y bienestar universitario.

Aunque existe un módulo de usuarios en la arquitectura general del sistema, en este microservicio decidimos almacenar localmente una tabla Users para evitar dependencia directa con dicho módulo.

Esta decisión sigue el principio de independencia y autonomía de los microservicios, asegurando que el servicio de turnos pueda operar de forma aislada y resiliente ante fallos externos.

Cada tabla incluye los atributos necesarios para reflejar las relaciones, estados y condiciones del dominio, manteniendo consistencia e integridad con JPA.

Diagrama de Clases



El proyecto sigue una arquitectura en capas típica de aplicaciones Spring Boot, organizada en:

1. Model (Modelo / Entidades)

En esta capa se encuentran las clases que representan las tablas de la base de datos. Cada clase corresponde a una entidad del dominio.

2. Repository (Repositorio)

Aquí se encuentran las interfaces que extienden JpaRepository o CrudRepository, permitiendo acceder a los datos de las entidades sin necesidad de escribir SQL directamente.

3. Service (Lógica de Negocio)

Esta capa contiene la lógica central del sistema. Aquí se procesan las reglas de negocio antes de interactuar con la base de datos o responder al cliente.

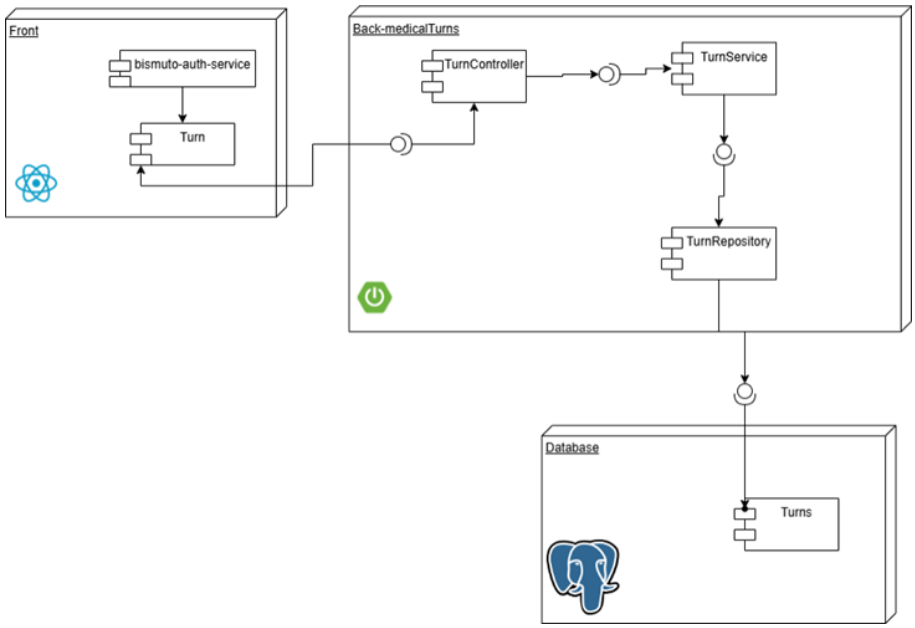
4. Controller (Controladores / API REST)

Los controladores exponen las funcionalidades como servicios web RESTful. Están anotados con @RestController y definen los endpoints accesibles desde el frontend u otros sistemas. Estos reciben las solicitudes HTTP, delegan el procesamiento a los servicios y devuelven las respuestas al cliente.

Diagrama de componentes

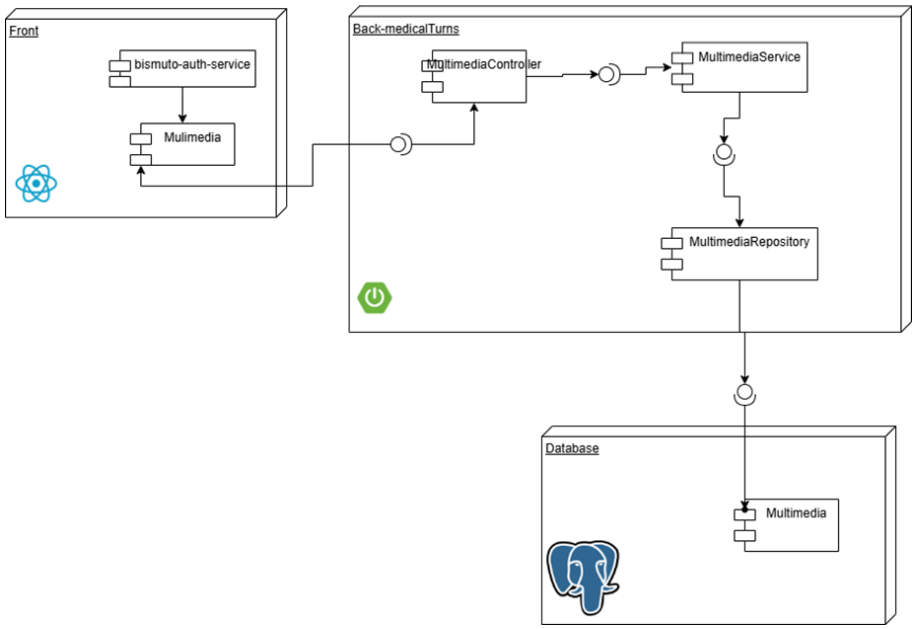
1.Turn Management Service

Este microservicio se encarga de gestionar los diferentes turnos, ya sea por parte de un administrador o de un usuario común. Se comunica con la clase controladora, la cual, dependiendo de la solicitud, ejecuta una u otra acción. Esta clase se conecta con el servicio de turnos, que mantiene la lógica separada para cada tipo de solicitud. Finalmente, se comunica con el repositorio de turnos para guardar o modificar los datos en la base de datos.



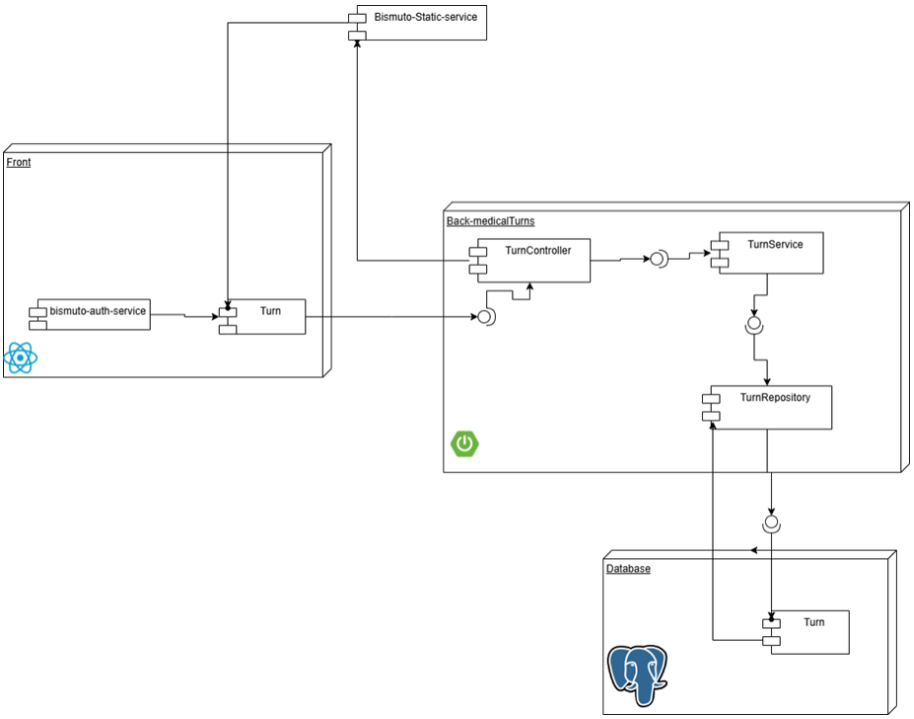
2. Multimedia Management Service

Este microservicio funciona de manera similar al de turnos, pero está orientado a la gestión de los elementos multimedia utilizados con fines informativos. Solo los administradores tienen acceso a este servicio.



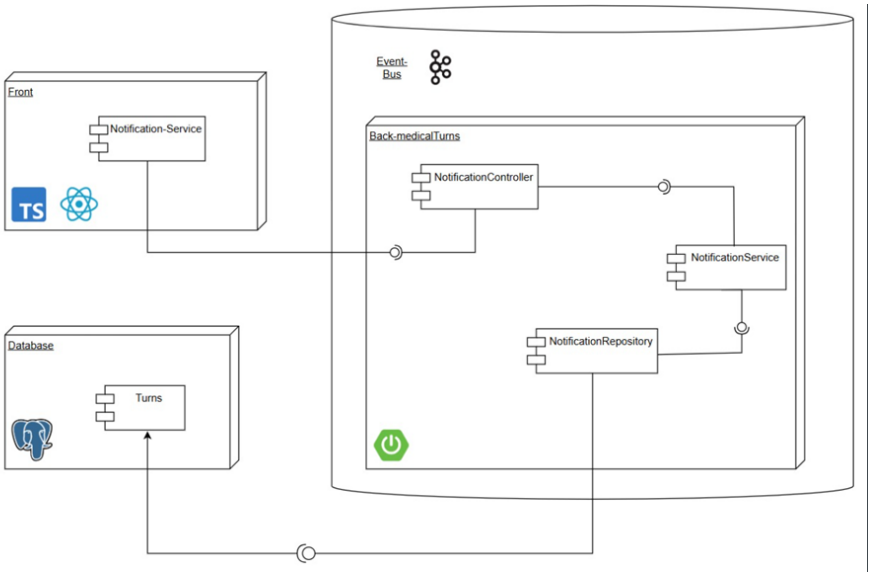
3. Report Service

Este microservicio muestra, de manera general, cómo se generan los reportes dentro de nuestro módulo. La solicitud se realiza desde el frontend, y nuestro backend se encarga de enviar los datos al módulo de Static Service del equipo Bismuto, el cual devuelve la información al frontend.



4. Notification Service

Este microservicio gestiona el seguimiento del funcionamiento del bus de eventos, permitiendo la conexión con el módulo de notificaciones según sea necesario.



Funcionalidades expuestas

UniversityWelfareController

1. Crear nuevo turno

- **Método:** POST
- **Endpoint:** /api/turns
- **Descripción:** Crea un nuevo turno.
- **Entrada:** Objeto JSON del turno (CreateTurnRequest)
- **Salida:** Turno creado (ApiResponse<TurnResponse>)

2. Obtener turnos del día

- **Método:** GET
- **Endpoint:** /api/turns
- **Descripción:** Devuelve todos los turnos del día actual.
- **Entrada:** N/A
- **Salida:** Lista de turnos del día (ApiResponse<List<TurnResponse>>)

3. Obtener turnos del día de una especialidad.

- **Método:** GET
- **Endpoint:** /api/turns/{speciality}
- **Descripción:** Devuelve todos los turnos de una especialidad del día actual.
- **Entrada:** Path param → speciality (valor del enum SpecialityEnum)
- **Salida:** Lista de turnos de una especialidad del día (ApiResponse<List<TurnResponse>>)

4. Obtener el último turno llamado

- **Método:** GET
- **Endpoint:** /api/turns/last-turn
- **Descripción:** Devuelve el ultimo turno que fue llamado.

- Entrada: N/A
- Salida: Último turno llamado (ApiResponse<TurnResponse>)

5. Obtener el último turno llamado de una especialidad

- Método: GET
- Endpoint: /api/turns/last-turn/{speciality}
- Descripción: Devuelve el ultimo turno de una especialidad que fue llamado.
- Entrada: Path param → speciality (valor del enum SpecialityEnum)
- Salida: Último turno llamado (ApiResponse<TurnResponse>)

6. Llamar al siguiente turno en secuencia

- Método: POST
- Endpoint: /api/turns/call-next
- Descripción: Llama al turno siguiente.
- Entrada: Objeto JSON con currentTurnId y nextTurnId (CallTurnRequest)
- Salida: Turno llamado (ApiResponse<TurnResponse>)

7. Llamar a un turno específico

- Método: POST
- Endpoint: /api/turns/call
- Descripción: Llama a un turno en específico.
- Entrada: Objeto JSON con currentTurnId y nextTurnId (CallTurnRequest)
- Salida: Turno llamado (ApiResponse<TurnResponse>)

8. Habilitar todos los turnos

- Método: POST
- Endpoint: /api/turns/enable
- Descripción: Habilita todos los turnos de todas las especialidades.

- Entrada: N/A
- Salida: N/A

9. Deshabilitar todos los turnos

- Método: POST
- Endpoint: /api/turns/disable
- Descripción: Deshabilita todos los turnos de todas las especialidades.
- Entrada: N/A
- Salida: N/A

10. Habilitar turnos por especialidad

- Método: POST
- Endpoint: /api/turns/enable/{speciality}
- Descripción: Habilita todos los turnos de una especialidad.
- Entrada: Path param → speciality (valor del enum SpecialityEnum)
- Salida: N/A

11. Deshabilitar turnos por especialidad

- Método: POST
- Endpoint: /api/turns/disable/{speciality}
- Descripción: Deshabilita todos los turnos de una especialidad.
- Entrada: Path param → speciality (valor del enum SpecialityEnum)
- Salida: N/A

MultimediaController

1. Subir nuevo contenido multimedia

- Método: POST
- Endpoint: /api/multimedia
- Descripción: Sube un nuevo contenido informativo al sistema.

- **Entrada:** Objeto JSON del contenido (CreateMultimediaRequest)
- **Salida:** N/A

2. Obtener contenido multimedia por ID

- **Método:** GET
- **Endpoint:** /api/multimedia/{id}
- **Descripción:** Devuelve un contenido informativo específico.
- **Entrada:** Path param → id del contenido informativo
- **Salida:** Contenido informativo (ApiResponse<MultimediaResponse>)

3. Obtener todos los contenidos multimedia

- **Método:** GET
- **Endpoint:** /api/multimedia
- **Descripción:** Devuelve la lista de todos los contenidos informativos.
- **Entrada:** N/A
- **Salida:** Lista de contenidos informativos (ApiResponse<List<MultimediaResponse>>)

4. Eliminar contenido multimedia

- **Método:** DELETE
- **Endpoint:** /api/multimedia/{id}
- **Descripción:** Elimina un contenido informativo específico.
- **Entrada:** Path param → id del contenido informativo
- **Salida:** N/A

ReportController

1. Obtener detalle de atención

- **Método:** GET
- **Endpoint:** /api/reports

- Descripción: Devuelve los reportes de atención según el rol del paciente y un rango de fechas.
- Entrada (query params):
    - role: valor del enum RoleEnum
    - start: fecha de inicio
    - end: fecha de fin
- Salida: Lista de reportes (ApiResponse<List<ReportResponse>>)

2. Obtener promedio de calidad de atención por especialidad

- Método: GET
- Endpoint: /api/reports/attention-level/average
- Descripción: Devuelve los promedios de nivel de atención por especialidad y tipo de usuario.
- Entrada (query params):
    - role: valor del enum RoleEnum
    - start: fecha de inicio.
    - end: fecha de fin.
- Salida: Objeto JSON con especialidad como clave y promedio de atención como valor (ApiResponse<Map<String, Integer>>)

3. Contar turnos por especialidad

- Método: GET
- Endpoint: /api/reports/turns/count-by-specialty
- Descripción: Devuelve la distribución de turnos atendidos por especialidad y tipo de usuario.
- Entrada (query params):
    - role: valor del enum RoleEnum
    - start: fecha de inicio
    - end: fecha de fin

- **Salida:** Objeto JSON con especialidad como clave y número de turnos como valor (ApiResponse<Map<String, Integer>>)

4. Obtener turnos por estado

- **Método:** GET
- **Endpoint:** /api/reports/turns/status
- **Descripción:** Devuelve los reportes de atención filtrados por estado del turno y rango de fechas.
- **Entrada (query params):**
    - status: valor del enum StatusEnum
    - start: fecha de inicio.
    - end: fecha de fin.
- **Salida:** Lista de reportes (ApiResponse<List<ReportResponse>>

Manejo de errores

Código HTTP	Mensaje de error	Causa probable
400	"Datos de entrada inválidos"	Validaciones fallidas en el formulario
401	"Usuario no autenticado"	Token inválido o ausente
404	"Turnos no disponibles"	Los turnos están deshabilitados
404	“Especialidad no disponible”	Especialidad deshabilitada
500	"Error interno del servidor"	Fallo inesperado