MANUAL TÉCNICO

CheTurno

Índice General

4.2. Módulos y enrutamiento

4.3. Gestión de seguridad en el cliente

4.4. Servicios compartidos y manejo de estado

5. Módulos funcionales (detalle técnico)

5.1. Gestión de instituciones y recursos físicos

5.2. Gestión de usuarios y perfiles

5.3. Gestión médica

5.4. Core de turnos

5.5. Auditoría y reportes

1. Introducción y arquitectura general

1.1. Propósito y Alcance

El sistema **CheTurno** es una plataforma de gestión de turnos Full Stack diseñada bajo un modelo **Software as a Service (SaaS)** para ser adoptada por múltiples Centros de Atención Médica. El propósito fundamental se articula en tres pilares:

1. **Mejora de la experiencia del paciente:**
    - Proveer una interfaz web intuitiva (Angular) que permite a los pacientes gestionar sus citas médicas (solicitar, reprogramar, cancelar y consultar historial) de manera autónoma, mejorando la satisfacción del usuario y reduciendo la carga operativa del personal.
2. **Eficiencia operacional y control de gestión (local):**
    - Ofrecer a los administradores de clínica y operadores herramientas avanzadas para la gestión interna de sus recursos (médicos, consultorios, horarios) y para el control de la calidad del servicio. Esto se logra mediante dashboards, analytics, y un sistema de encuestas de satisfacción post-atención, permitiendo la toma de decisiones basada en métricas específicas de su centro.
3. **Habilitación multi-tenant (SaaS):**
    - Soportar la operación de múltiples clínicas de manera segura y escalable desde una única instancia de la aplicación. Este requisito arquitectónico garantiza el **aislamiento total de los datos** y la personalización de la gestión para cada cliente (tenant).

El sistema **CheTurno** está diseñado para gestionar todas las fases del ciclo de vida del turno médico y la administración de los recursos asociados.

A. Alcance funcional

Área Funcional	Descripción del Módulo
Gestión de turnos (core)	Reserva, reagendamiento, cancelación, gestión de estados y lógica de asignación automática de consultorios.
Infraestructura médica	Mantenimiento de centros de atención (tenants), consultorios, especialidades y configuración horaria.
Gestión de personal	CRUD de médicos, operadores, y gestión de la disponibilidad horaria del staff médico.
Monitoreo y calidad	Módulo de auditoría (AuditLog) para trazabilidad inmutable y generación de reportes de métricas (KPIs) y resultados de encuestas.
Seguridad y usuarios	Autenticación JWT, registro de pacientes, y servicios de recuperación/activación de cuenta.

B. Alcance arquitectónico (multi-tenant):

La plataforma implementa una estrategia de **multi-tenant por columna discriminadora**, utilizando el campo `centro_atencion_id` en las entidades de negocio críticas para segregar los datos. El rol de usuario determina el alcance de la visibilidad de los datos:

Rol de usuario	Alcance de datos	Descripción del permiso
ROLE_SUPERADMIN	Global (Plataforma)	Visibilidad y gestión de todos los centros de atención (tenants). Acceso irrestricto a los datos para administración y soporte.
ROLE_ADMIN	Local (clínica)	Acceso y gestión restringidos exclusivamente a los datos asociados a su `centro_atencion_id`. Administra recursos (médicos, consultorios) de su clínica.
ROLE_MEDICO	Local (clínica)	Acceso a su agenda y a la información de los pacientes de su centro de atención.
ROLE_OPERADOR	Local (clínica)	Gestión de turnos y pacientes dentro del contexto de su centro de atención.
ROLE_PACIENTE	Personal	Acceso a sus propios turnos e información dentro de la plataforma.

1.2. Arquitectura de alto nivel

El sistema **CheTurno** está diseñado bajo una arquitectura de **Cliente-Servidor** que se estructura en un modelo tradicional de **tres capas**. Esta elección promueve la modularidad, la escalabilidad horizontal de cada componente y la clara separación de responsabilidades, lo cual es fundamental para el modelo SaaS.

La comunicación es estrictamente asíncrona y basada en **API RESTful** utilizando el formato JSON.

Capa 1: Capa de presentación

- **Responsabilidad primaria:** Gestionar la Interfaz de Usuario (UI) y la Experiencia de Usuario (UX). Esta capa se encarga de la visualización de datos, la interacción del usuario y el manejo de la navegación.
- Tecnología**:** Desarrollada con **Angular 19**.
- Características clave:
    - Single Page Application (SPA): Proporciona una experiencia fluida y reactiva.
    - Consumo de API: Se comunica exclusivamente con el backend a través de servicios HTTP (utilizando `HttpClient`).
    - Contexto de tenencia: Implementa lógica para adaptar la interfaz y los menús de navegación basándose en el rol del usuario (ROLE_SUPERADMIN vs. ROLE_ADMIN) y el contexto del `centro_atencion_id` local.
    - Seguridad: Almacena de forma segura el token JWT para ser enviado en cada solicitud autenticada a través de Interceptors.

Capa 2: Capa de lógica y negocio

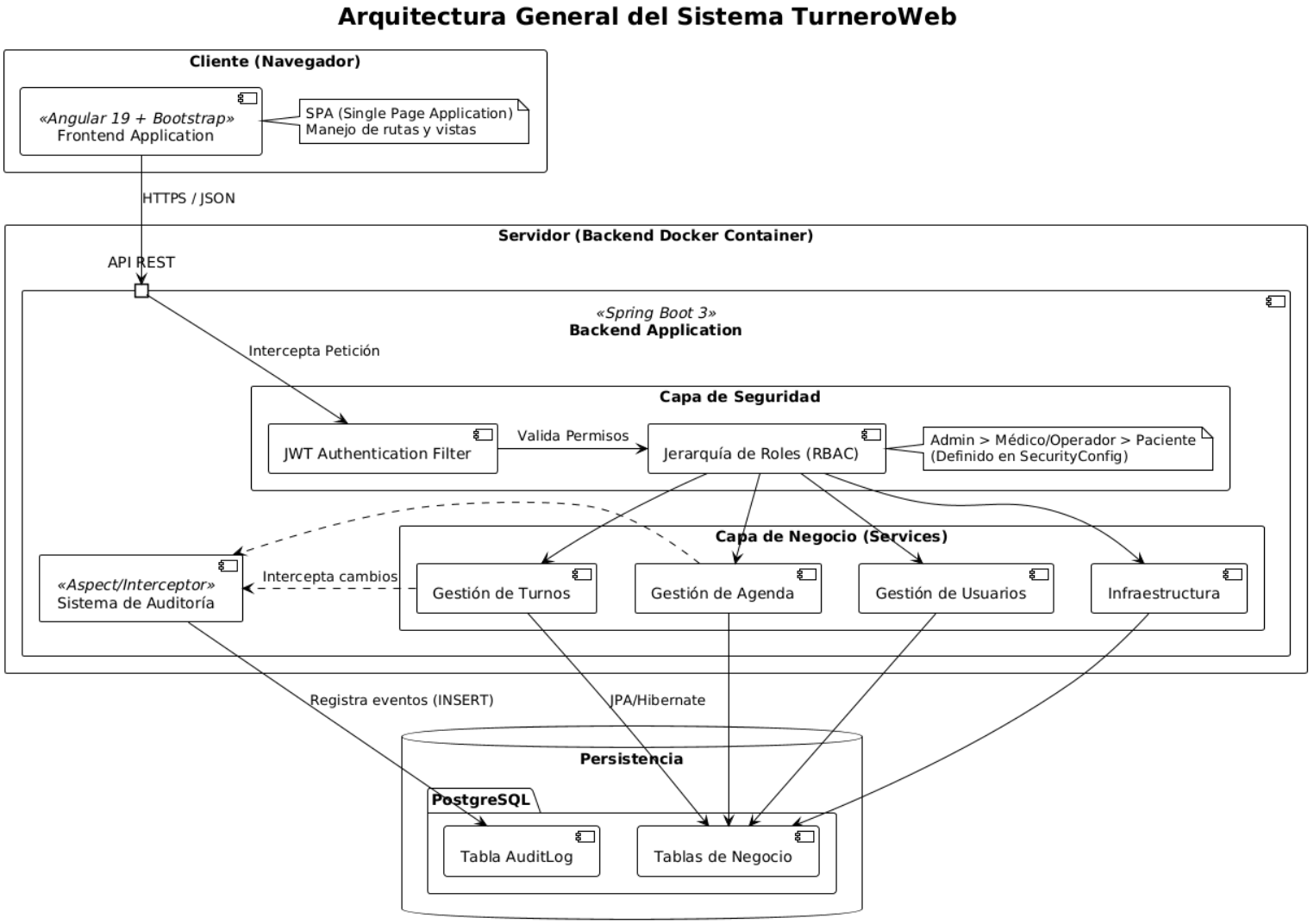
- **Responsabilidad primaria:** Contener toda la lógica de negocio, procesar las peticiones del frontend, aplicar las reglas de validación (por ejemplo,

validar disponibilidad de turnos), gestionar la seguridad y coordinar las transacciones con la base de datos.

- Tecnología: Desarrollada con **Spring Boot 3.0.2** (Java 17).
- Características clave:
    - API RESTful: Expone los *endpoints* para la manipulación de recursos.
    - Seguridad (Spring Security): Centraliza la autenticación (JWT y Google Auth) y la autorización, aplicando la jerarquía de roles (ROLE_SUPERADMIN, ROLE_ADMIN, etc.).
    - Multi-tenant por columna: Esta capa es la encargada de inyectar el filtro `WHERE centro_atencion_id = [ID del Tenant]` en casi todas las consultas y escrituras, asegurando el aislamiento de datos entre clínicas.
    - Servicios transversales: Incluye lógica para auditoría (`AuditLog`), envío de correos electrónicos y generación de reportes (PDF/CSV).

Capa 3: Base de datos

- **Responsabilidad primaria:** Almacenamiento, persistencia y recuperación de todos los datos del sistema, manteniendo la integridad referencial y la consistencia transaccional.
- Tecnología: **PostgreSQL**.
- Características clave:
    - Modelo relacional único: Se utiliza una única base de datos física para todos los tenants, optimizando costos de infraestructura.
    - Segregación lógica: La separación de los datos se realiza a nivel lógico mediante la columna clave `centro_atencion_id` en las tablas de negocio, lo que permite el aislamiento de los datos sin requerir bases de datos separadas.
    - Persistencia: Gestionada por Spring Data JPA y Hibernate, que mapea las entidades de Java a las tablas de PostgreSQL.



1.3. Stack tecnológico

El proyecto **CheTurno** está construido utilizando un *stack* tecnológico moderno y robusto, garantizando rendimiento, mantenibilidad y escalabilidad. A continuación, se detallan los componentes principales y sus versiones:

Backend y lógica de negocio

Componente	Tecnología
Framework principal	Spring Boot 3.0.2

Lenguaje	Java 17 (LTS)
Persistencia	Spring data JPA (Hibernate)
Base de datos	PostgreSQL
Seguridad	Spring Security

Frontend y presentación

Componente	Tecnología
Framework principal	Angular 19.2.0
Lenguaje	TypeScript
Estilos / UI	Bootstrap
Gráficos	ng2-charts y charts.js

Infraestructura y DevOps

Componente	Tecnologia
Orquestación	Docker Compose
Contenedores	Docker
Testing	Cucumber.js

2. Configuración del entorno de desarrollo

Esta sección detalla los pasos para desplegar el entorno de desarrollo de **CheTurno**. El proyecto utiliza una arquitectura contenerizada basada en **Docker**.

Toda la gestión del ciclo de vida de los contenedores se facilita mediante el script de utilidad `lpl` incluido en la raíz del proyecto.

2.1. Prerrequisitos

Requisito	Propósito	Instalación
Git	Control de versiones	`sudo apt install git`
Docker Engine	Motor de ejecución de contenedores	Guía Oficial Docker
Docker Compose	Orquestación de servicios multicontenedor	Guia Oficial Docker Compose

Para que el script `lpl` funcione correctamente sin pedir contraseña de superusuario constantemente, su usuario debe pertenecer al grupo `docker`:

```
# 1. Crear el grupo docker si no existe

sudo groupadd docker

# 2. Agregar su usuario actual al grupo

sudo usermod -aG docker $USER

# 3. Refrescar la sesión de grupo (o reiniciar la sesión)

newgrp docker
```

2.2. El script de control

El proyecto incluye un script bash llamado `lpl` (Laboratorio de Programación y Lenguajes) que actúa como *wrapper* o interfaz simplificada para los comandos de Docker Compose. Este es el método estándar para interactuar con el sistema.

Permisos de ejecución: Antes de empezar, asegúrese de que el script tenga permisos de ejecución:

```
chmod +x lpl
```

Comandos principales:

Comando	Acción técnica	Descripción
`./lpl build`	`docker compose build`	Descarga las imágenes base y construye las imágenes locales del proyecto.
`./lpl up`	`docker compose up -d`	Levanta todos los servicios (Backend, Frontend, DB) en segundo plano (modo *detached*).
`./lpl down`	`docker compose down`	Detiene y elimina los contenedores y redes creados.
`./lpl logs`	`docker compose logs -f`	Muestra la salida unificada de logs de todos los servicios en tiempo real.
`./lpl sh <servicio>`	`docker exec -it <servicio> sh`	Abre una terminal interactiva dentro del contenedor especificado (ej. `backend`, `frontend`).
`./lpl mvn <comando>`	`docker exec -t backend mvn <comando>`	Ejecuta comandos Maven directamente en el contenedor `backend`
`./lpl staging <file>`	Inyecta el archivo SQL en el contenedor de PostgreSQL para cargar datos	Carga scripts de inicialización o migración

2.3. Inicialización del proyecto paso a paso

Paso 1: Construcción de Imágenes

Este paso preparará los contenedores. Mediante los archivos `Dockerfile`, las dependencias como **Maven** (para el backend) y **Angular CLI / Node modules** (para el frontend) se descargarán e instalarán automáticamente **dentro** de las imágenes Docker durante este proceso.

```
./lpl build
```

Paso 2: Arranque de servicios

Una vez construidas las imágenes, inicie el entorno.

```
./lpl up
```

El sistema iniciará con los siguientes contenedores

1. **Database:** PostgreSQL
2. **Backend:** Servidor Spring Boot
3. **Frontend:** Servidor de desarrollo Angular

Paso 3: Verificación

Puede verificar que el sistema está corriendo accediendo a:

- **Frontend:** http://localhost:4200
- **Backend API:** http://localhost:8080

Si desea ver los logs en tiempo real para detectar errores de arranque:

```
./lpl logs
```

2.4. Configuración de componentes

Aunque la ejecución es contenerizada, la configuración se realiza mediante archivos en su directorio de trabajo que son montados como volúmenes en los contenedores. Esto permite editar la configuración y ver los cambios reflejados (gracias al *Hot Reload* configurado en los Dockerfiles).

Backend

- **Ubicación:** *backend/src/main/resources/application.properties*
- **Configuración típica:** Credenciales de base de datos, clave JWT, configuración de correo SMTP

Frontend

- **Ubicación:** *frontend/cli/src/environments/*
- **Proxy:** El archivo *proxy.conf.json* ya está configurado para redirigir las peticiones */rest* al contenedor backend, solucionando problemas de CORS en desarrollo
- **Dependencias:** Se instalan ingresando al contenedor frontend `./lpl sh frontend` mediante *npm install*

Base de datos:

La base de datos se inicializa automáticamente. Los scripts SQL ubicados en la carpeta `/staging` se pueden utilizar para poblar la base de datos con información de prueba o estructuras iniciales.

Para cargar un script de staging específico (ej. `userinit.sql` para cargar usuarios de prueba):

```
./lpl staging userinit
```

3. Arquitectura del Backend (Spring Boot)

El backend de **CheTurno** está construido sobre el framework **Spring Boot 3.0.2** utilizando **Java 17**. Sigue una arquitectura en capas clásica, enriquecida con patrones de diseño modernos para manejar la seguridad, la auditoría y la lógica de negocio compleja.

3.1. Estructura de paquetes

El código fuente se organiza bajo el paquete raíz `unpsjb.labprog.backend`, siguiendo una estructura semántica que separa claramente las responsabilidades.

- Paquete ***business*** **(lógica de negocio)**

Este es el núcleo de la aplicación, donde residen las reglas de negocio y el acceso a datos. Se subdivide en:

model ***(****`unpsjb.labprog.backend.model`****):***

- Contiene las entidades JPA que mapean directamente a las tablas de la base de datos PostgreSQL.
- **Entidades clave:** `Turno`, `Paciente`, `Medico`, `CentroAtencion`, `Consultorio`, `Especialidad`.
- **Seguridad:** `User`, `Role` (Enum), `Administrador`, `Operador`.
- **Auditoría:** `AuditLog`.
- **Configuración:** `Configuracion`, `ConfiguracionExcepcional`.

repository ***(****`unpsjb.labprog.backend.business.repository`****):***

- Interfaces que extienden de `JpaRepository` o `CrudRepository`.
- Proporcionan métodos CRUD automáticos y consultas personalizadas (JPQL/SQL nativo).
- Ejemplos: `TurnoRepository`, `MedicoRepository`, `UserRepository`.

service ***(****`unpsjb.labprog.backend.business.service`****):***

- Clases marcadas con `@Service` que implementan la lógica de negocio pura.
- Actúan como intermediarios entre los controladores (presenters) y los repositorios.

- Gestionan transacciones con `@Transactional`.
- **Servicios Complejos:**
    - `TurnoService`: Orquesta el ciclo de vida del turno.
    - `AgendaService`: Maneja la generación de slots y validación de horarios.
    - `ConsultorioDistribucionService`: Contiene los algoritmos para asignar consultorios automáticamente.
    - `TurnoAutomationService`: Clase que ejecuta la cancelación automática de turnos usando la anotación `@Scheduled` de Spring.

- Paquete ***presenter*** **(capa de controladores)**

En esta arquitectura, los controladores REST se denominan "Presenters" (ej. `TurnoPresenter`, `MedicoPresenter`).

- Están anotados con `@RestController` y `@RequestMapping`.
- Su única responsabilidad es recibir peticiones HTTP, validar la entrada (DTOs), invocar al servicio correspondiente y retornar una respuesta `ResponseEntity` con el objeto `Response` estandarizado o los datos solicitados.
- Manejan endpoints específicos para funcionalidades como `AuthController` (login), `DashboardPresenter` (métricas) y `ExportPresenter` (reportes).

- Paquete ***dto (Data Transfer Objects)***

Contiene objetos planos (POJOs) utilizados para transferir datos entre el frontend y el backend sin exponer directamente las entidades JPA.

- Se utilizan para:
    - **Requests:** `LoginRequest`, `RegisterRequest`, `TurnoFilterDTO`.

    - **Responses:** `LoginResponse`, `TurnoDTO`, `MedicoDTO`, `MetricasDashboardDTO`.
    - Ayudan a desacoplar la vista del modelo de datos interno.

- Paquete ***config*** **(configuración transversal)**

Configuraciones globales de Spring Boot:

- `SecurityConfig`: Configuración de Spring Security (Filtros, CORS, Rutas públicas/privadas).
- `JacksonConfig`: Configuración de serialización JSON (Manejo de fechas).
- `JwtAuthenticationFilter` y `JwtTokenProvider`: Lógica de seguridad JWT.
- `AuditInterceptor` y `AuditContext`: Infraestructura para la auditoría automática.

3.2. Configuración de seguridad

La seguridad es un pilar fundamental, implementada mediante **Spring Security** y **JSON Web Tokens (JWT)**.

Flujo de autenticación (Stateless)

1. **Login:** El usuario envía credenciales (`LoginRequest`) al `AuthController`.
2. **Validación:** El `AuthenticationManager` verifica las credenciales contra la base de datos (`UserDetailsService`).
3. **Generación de Token:** Si es válido, `JwtTokenProvider` genera un token JWT firmado que contiene el `username`, `rol` y `centro_atencion_id` (claims).
4. **Uso:** El frontend debe incluir este token en el header `Authorization`: `Bearer <token>` en cada petición subsiguiente.

Filtro de autenticación (`JwtAuthenticationFilter`)

Este filtro intercepta cada petición HTTP antes de llegar a los controladores:

1. Extrae el token del header.
2. Valida la firma y expiración del token usando `JwtTokenProvider`.
3. Si es válido, extrae los detalles del usuario y establece el contexto de seguridad (`SecurityContextHolder`) para el hilo actual.

Jerarquía de roles (`RoleHierarchy`)

Se implementa una jerarquía personalizada para simplificar las reglas de acceso:

- PACIENTE (base)

  ├── MÉDICO (hereda de PACIENTE)

  ├── OPERADOR (hereda de PACIENTE)

  └── ADMINISTRADOR (hereda de PACIENTE, MÉDICO, OPERADOR)

- Esto significa que un ADMIN tiene implícitamente los permisos de los roles inferiores en la cadena.
- Se utiliza `RoleHierarchy` (clase de servicio) para verificar permisos programáticamente si es necesario.

Integración con Google Auth

El servicio `GoogleAuthService` permite la autenticación mediante tokens de Google, validando la identidad del usuario y vinculándola a una cuenta existente o creando una nueva (para pacientes) si corresponde.

3.3. Manejo de excepciones y respuestas

El backend no utiliza un manejo de excepciones global con `@ControllerAdvice` visible en la estructura raíz, pero sigue un patrón de respuestas estandarizado.

Clase Response

Todas las respuestas de la API siguen una estructura consistente (visible en `unpsjb.labprog.backend.Response`). Esto permite al frontend manejar errores y éxitos de forma uniforme.

Excepciones de negocio

Los servicios lanzan excepciones específicas cuando ocurren reglas de negocio inválidas (ej. "Turno no disponible", "Conflicto de horarios"). Estas son capturadas por los controladores para devolver el código HTTP adecuado (400 Bad Request, 404 Not Found, etc.).

3.4. Servicios transversales

Estos componentes dan soporte a múltiples módulos funcionales.

Sistema de auditoría (*`AuditLog`*)

- **Propósito:** Registrar quién hizo qué y cuándo. Es crítico para la trazabilidad en un sistema médico.
- **Implementación:**
    - **Interceptor:** `AuditInterceptor` intercepta las operaciones de escritura.
    - **Contexto:** `AuditContext` almacena temporalmente información del usuario actual para ser usada en el registro.
    - **Persistencia:** Se guardan registros en la tabla `audit_log` a través de `AuditLogService` y `AuditLogRepository`.
    - **Datos:** Guarda la entidad afectada, la acción (CREATE, UPDATE, DELETE), el usuario responsable y el timestamp.

Servicio de email (*`EmailService`*)

- Utiliza `JavaMailSender` (configurado en `pom.xml` con `spring-boot-starter-mail`).
- **Usos:**
    - Envío de tokens para activación de cuenta (`AccountActivationService`).
    - Envío de enlaces para restablecimiento de contraseña (`PasswordResetService`).
    - Notificaciones de turnos (confirmación, cancelación).
    - Envío de invitación a realizar la encuesta post-atención.
- Maneja el procesamiento asíncrono para no bloquear la respuesta HTTP al usuario.

Exportación de datos (*`ExportPresenter/ExportService`*)

- Permite generar reportes en formatos portables.
- Tecnologías:
    - CSV: Utiliza `opencsv` para exportar listados (ej. listado de pacientes, turnos).
    - PDF: Utiliza `itextpdf` (kernel, layout, io) para generar documentos formales, como comprobantes de turno o historias clínicas.

DeepLinking (*`DeepLinkService`*)

- Genera y valida tokens especiales (`DeepLinkToken`) que permiten acceso directo a recursos específicos desde fuera de la aplicación (ej. un enlace en un email para confirmar un turno o completar una encuesta) sin necesidad de un login tradicional previo, mejorando la UX.

4. Arquitectura del Frontend

El Frontend de **CheTurno** es una **Single Page Application (SPA)** desarrollada con **Angular 19**. Esta capa se encarga de la interacción con el usuario final, la presentación de datos y la gestión del estado de la sesión en el cliente.

El diseño sigue una arquitectura basada en **Componentes Standalone**, aprovechando las últimas características del framework para reducir la complejidad estructural (eliminación de `AppModule`) y mejorar el rendimiento mediante la carga diferida (*lazy loading*) a nivel de rutas.

4.1. Estructura del proyecto

El código fuente reside en el directorio `frontend/cli/src` y sigue una organización modular por funcionalidad ("Feature-First"), donde cada entidad del dominio tiene su propio directorio.

Organización de directorios (`src/app`)

- ***`app.config.ts`***: Punto de configuración global de la aplicación (reemplaza al `app.module.ts`). Aquí se configuran los proveedores globales como el `HttpClient`, el enrutador (`provideRouter`) y los interceptores.
- ***`app.routes.ts`***: Definición centralizada de las rutas de navegación.
- **Módulos funcionales (carpetas de dominio):**
    - `turnos/`, `pacientes/`, `medicos/`, `agenda/`, `centrosAtencion/`: Cada carpeta contiene los componentes (listado, detalle, formularios), modelos y servicios específicos de esa entidad.
- ***`components/`* (Shared)**: Componentes reutilizables en toda la aplicación, como la barra lateral (`SidebarComponent`), tarjetas de KPI (`KpiCardComponent`) y gráficos (`GraficoTortaComponent`)
- ***`services/`***: Servicios transversales singleton (ej. `NotificacionService`, `ConfigService`).
- ***`guards/`***: Lógica de protección de rutas.

- ***interceptors/***: Middleware para peticiones HTTP.
- ***modal/***: Componentes de ventanas emergentes para acciones rápidas (ej. `MapModal`, `ComentariosModal`)

4.2. Módulos y enrutamiento

El enrutamiento es el núcleo de la navegación en la SPA. Se define en `app.routes.ts` y utiliza una estrategia de protección estricta basada en roles.

Configuración de rutas

El router mapea URLs a componentes específicos. Se pueden identificar dos grandes grupos de rutas:

1. **Rutas públicas:** Accesibles sin autenticación.
    - `/inicio-sesion`: Pantalla de login.
    - `/registro-paciente`: Formulario de auto-registro.
    - `/activar-cuenta`: Para tokens de email.
    - `/public/turnos`: Consulta pública de disponibilidad.
2. **Rutas privadas (dashboard):** Protegidas por `AuthGuard`.
    - `/home`: Dashboard principal.
    - `/turnos`, `/medicos`, `/pacientes`: Gestión de entidades.
    - `/admin/*`: Rutas exclusivas para `ROLE_ADMIN` o `ROLE_SUPERADMIN`.

Navegación dinámica

El sistema no utiliza un menú estático. El `MenuService` (`src/app/services/menu.service.ts`) y `src/app/config/menu.config.ts` generan la estructura del menú lateral (`SidebarComponent`) dinámicamente basándose en el rol del usuario autenticado. Esto asegura que un médico no vea opciones de administración de centros, y un paciente solo vea sus propias gestiones.

4.3. Gestión de seguridad en el cliente

Aunque la seguridad real reside en el Backend, el Frontend implementa mecanismos para asegurar la experiencia de usuario y gestionar la sesión.

Manejo de tokens JWT

- **Almacenamiento:** El token JWT recibido tras el login se almacena en el `localStorage` del navegador.
- **Inyección Automática (`AuthInterceptor`):**
    - Ubicación: `src/app/services/auth.interceptor.ts`.
    - Función: Intercepta todas las peticiones salientes `HttpClient`. Si existe un token en `localStorage`, clona la petición y añade el encabezado: `Authorization: Bearer <TOKEN>`
- **Manejo de Errores (`AuthErrorInterceptor`):**
    - Ubicación: `src/app/interceptors/auth-error.interceptor.ts`.
    - Función: Escucha las respuestas HTTP. Si detecta un error **401 (Unauthorized)** o **403 (Forbidden)**, asume que la sesión expiró o es inválida, cierra la sesión del usuario en el cliente y lo redirige al login.

Guards (protección de rutas)

Los guards implementan la interfaz `CanActivate` para prevenir la carga de componentes no autorizados.

Guard	Archivo	Propósito
`AuthGuard`	*`auth.guard.ts`*	Verifica que exista un usuario autenticado (token válido).
`AdminGuard`	*`admin.guard.ts`*	Permite acceso solo a

		`ROLE_ADMIN` o `ROLE_SUPERADMIN`.
MedicoGuard	*medico.guard.ts*	Permite acceso a Médicos y Administradores.
OperadorGuard	*operador.guard.ts*	Permite acceso a operadores y administradores.
PatientGuard	*patient.guard.ts*	Exclusivo para vistas de pacientes.
AdminOperadorGuard	*admin-operador.guard.ts*	Guard híbrido para funcionalidades compartidas por gestión y operación.

4.4. Servicios compartidos y manejo de estado

La lógica de comunicación y el estado global se gestionan a través de servicios inyectables (`@Injectable({providedIn: 'root'})`).

Contexto de usuario (`UserContextService`)

Este es el servicio más crítico para el manejo de estado.

- **Responsabilidad:** Mantiene en memoria la información del usuario actual (`UserDTO`), su rol y, crucialmente para la arquitectura SaaS, el `centro_atencion_id` actual.
- **Reactividad:** Expone `Observables` para que los componentes (como el Sidebar o los Dashboards) reaccionen automáticamente a cambios de usuario o de centro de atención.

Servicios de infraestructura

- **`NotificacionService`**: Se encarga de la lógica de notificaciones relacionadas a los turnos (cancelación, recordatorio, confirmación), así como también el listado y eliminación de notificaciones.
- **`GeolocationService`**: Utiliza la API del navegador para obtener la ubicación del usuario, útil para funcionalidades como "Centros de Atención cercanos".
- **`DeepLinkService`**: Gestiona la lógica de entrada a la aplicación mediante enlaces profundos (ej. confirmación de turnos desde email), comunicándose con el endpoint de validación de tokens del backend.

Servicios de negocio

Cada módulo funcional tiene su propio servicio (ej. `TurnoService`, `MedicoService`, `PacienteService`).

- Estos servicios extienden la funcionalidad base de `HttpClient`.
- Manejan la serialización/deserialización de DTOs.
- Son los encargados de comunicarse con el *controller* correspondiente en el backend según el módulo.

5. Módulos funcionales (detalle técnico)

5.1. Gestión de instituciones y recursos físicos

Este módulo constituye la base operativa del sistema, definiendo la estructura jerárquica de la organización médica. Permite la administración de los **Centros de Atención** (sedes o clínicas) y sus respectivos **Consultorios** (espacios físicos de atención).

Centros de atención (entidad tenant)

En la arquitectura SaaS de **CheTurno**, el `CentroAtencion` no es solo una ubicación física; actúa como la entidad raíz para la **Multi-Tenencia**. Todos los

datos críticos (turnos, médicos, pacientes) están vinculados directa o indirectamente a un Centro de Atención.

A. Modelo de datos

La entidad JPA `unpsjb.labprog.backend.model.CentroAtencion` mapea a la tabla `centro_atencion`.

Atributo	Tipo de dato	Descripción
`id`	`Long (PK)`	Identificador único del Tenant.
`nombre`	`String`	Nombre comercial de la clínica o sede.
`direccion`	`String`	Dirección física para visualización.
`latitud/longitud`	`Double`	Coordenadas geográficas utilizadas para la integración con mapas.
`provincia/locailidad`	`String`	Lugar físico de la clínica
`telefono`	`String`	Datos de contacto institucional.

B. Lógica de backend (`CentroAtenciónService`)

- **Gestión centralizada:** El servicio permite operaciones CRUD estándar.

- **Validación de unicidad:** Se implementan reglas para evitar la duplicación de centros de atención, garantizando la integridad de la identificación del Tenant.

C. Implementación frontend

El módulo se encuentra en `src/app/centrosAtencion/`

1. **Listado (`CentrosAtencionComponent`):**
    - Muestra una grilla paginada de todos los centros disponibles.
    - **Funcionalidad de Mapa:** Integra un modal (`CentrosMapaModalComponent`) que utiliza **Leaflet** para renderizar un mapa interactivo con marcadores en las coordenadas (`lat`, `lng`) de cada centro activo.
2. **Detalle y Gestión (`CentroAtencionDetailRefactoredComponent`):**
    - Implementa un patrón de **Vistas Anidadas (Tabs)**. Al seleccionar un centro, el usuario no navega a una página nueva aislada, sino a una vista contenedora que carga dinámicamente sub-componentes mediante rutas hijas:
        - `/detalle`: Formulario de edición de datos básicos (`CentroAtencionDetalleTabComponent`).
        - `/consultorios`: Gestión de espacios físicos (`CentroAtencionConsultoriosTabComponent`).
        - `/staff-medico`: Asignación de profesionales (`CentroAtencionStaffMedicoTabComponent`).
        - `/especialidades`: Cartera de servicios del centro.
        - `/organigrama`: Visualización jerárquica.

Consultorios (recursos físicos)

Los consultorios representan los recursos finitos donde se prestan los servicios médicos. Su gestión es vital para evitar conflictos en la asignación de turnos (dos médicos no pueden usar el mismo consultorio a la misma hora).

A. Modelo de datos

La entidad un`psjb.labprog.backend.model.Consultorio` tiene una relación **Many-to-One** con `CentroAtencion`.

Atributo	Tipo de dato	Descripción
`id`	`Long (PK)`	Identificador único.
`centroAtencion`	`CentroAtencion (FK)`	Vínculo estricto con la sede propietaria
`nombre/numero`	`String`	Identificación interna (Ej. "Consultorio 1", "Sala de Rayos").
`descripcion`	`String`	Detalles de equipamiento (Ej. “Camilla ginecológica, “Pediatría”).

B. Lógica de backend (`ConsultorioService`)

- **Scope Local:** Todas las consultas a consultorios están filtradas obligatoriamente por el `centro_atencion_id`. Un administrador de la "Clínica A" nunca puede ver ni editar consultorios de la "Clínica B".
- **Validación de Disponibilidad:** Aunque la entidad es simple, el servicio está estrechamente ligado al `TurnoService` y `AgendaService`. Antes de asignar un turno, el sistema verifica que el consultorio no esté ocupado en ese *slot* de tiempo.

C. Implementación frontend

La gestión de consultorios se realiza principalmente dentro del contexto de un centro (`src/app/centrosAtencion/tabs/consultorios/`).

1. **Gestión Tabular (`CentroAtencionConsultoriosTabComponent`):**
    - Permite el alta, baja y modificación rápida de consultorios sin salir de la vista del centro.
    - Utiliza modales para la edición (`ConsultorioDetailComponent` usado como modal o vista embebida).
2. **Configuración de Horarios (`HorariosConsultorioModalComponent`):**
    - Permite definir bandas horarias de funcionamiento para cada consultorio (ej. "Lunes a Viernes de 8:00 a 20:00").
    - Esta configuración es consumida posteriormente por el algoritmo de distribución automática de turnos.
3. **Visualización de Ocupación:**
    - El sistema incluye componentes gráficos (`OcupacionConsultorio`) en el Dashboard de Gestión (`DashboardGestionComponent`) que permiten visualizar la tasa de uso de cada consultorio, ayudando a identificar cuellos de botella en la infraestructura.

5.2. Gestión de usuarios y perfiles

Este módulo abarca todos los procesos relacionados con el ciclo de vida de la identidad del usuario: desde el registro y la autenticación hasta la recuperación de credenciales y la gestión de perfiles. Implementa un modelo de seguridad robusto basado en roles jerárquicos y soporte para OAuth2 (Google).

Modelo de identidad (`User`)

La entidad `User` (`unpsjb.labprog.backend.model.User`) es el núcleo de la seguridad. Representa a cualquier actor que interactúa con el sistema.

A. Atributos críticos

Atributo	Descripción	Importancia en SaaS
`username`	Correo electrónico del usuario.	Identificador único para el login.
`password`	Hash de la contraseña (BCrypt)	Nunca se almacena en texto plano.
`enabled`	`Boolean`	Controla si la cuenta está activa (post-validación de email).
`role`	Enum `Role` (ADMIN, MEDICO, OPERADOR, PACIENTE).	Define los permisos jerárquicos.
`centroAtencion`	Relación con `CentroAtencion`.	**Clave para Multi-Tenencia**. Vincula al usuario administrativo/operativo con su clínica específica.

B. Jerarquía de roles

Se utiliza la clase `RoleHierarchy` para gestionar permisos complejos:

- **ADMIN** tiene acceso total a su Tenant.
- **MEDICO** y **OPERADOR** tienen accesos funcionales específicos.
- **PACIENTE** es el rol base.
- *Nota:* Un usuario puede tener datos extendidos en otras tablas (ej. `Medico` o `Paciente`) vinculados por su ID o email, pero `User` maneja estrictamente el acceso.

Subsistema de autenticación

El sistema soporta dos estrategias de autenticación gestionadas por el `AuthController`.

1. Autenticación tradicional (*email*/*password*)

- **Frontend:** Componente `InicioSesionComponent`. Envía `LoginRequest` (email, password).
- **Backend:**
    1. `AuthenticationManager` valida las credenciales.
    2. Si es exitoso, `JwtTokenProvider` genera un token firmado.
    3. Se devuelve un `LoginResponse` con el token y detalles del usuario (`UserDTO`).

2. Autenticación federada (*Google Sign-In*)

- **Frontend:** Utiliza `@abacritt/angularx-social-login` para obtener un `idToken` de Google.
- **Backend:**
    1. Endpoint `/api/auth/google`.
    2. `GoogleAuthService` verifica la validez del token contra los servidores de Google.
    3. **Lógica de onboarding:** Si el email no existe en la base de datos, el sistema **registra automáticamente** al usuario como `PACIENTE` activo. Si existe, lo loguea.

Registro y activación de cuenta

Para garantizar la calidad de los datos de contacto, el sistema implementa un flujo de registro con verificación obligatoria de correo electrónico.

A. Proceso de registro (`RegistrationService`)

1. El usuario completa el formulario en `RegistroUsuarioComponent`.
2. El backend crea el usuario con `enabled = false`.

3. Genera un `AccountActivationToken` (con expiración de 24hs).
4. `EmailService` envía un correo asíncrono con un enlace profundo (Deep Link).

B. Flujo de activación

1. El usuario hace clic en el enlace del correo (`/activar-cuenta?token=XYZ`).
2. **Frontend (`ActivacionCuentaComponent`):** Captura el token de la URL y lo envía al backend.
3. **Backend (`AccountActivationService`):**
    - Busca el token.
    - Verifica que no haya expirado.
    - Habilita al usuario (`user.setEnabled(true)`).
    - Elimina el token consumido.

Recuperación de contraseña

Implementa un mecanismo seguro para restablecer credenciales olvidadas.

Componentes involucrados

- **Frontend:** Módulo `RecuperacionContrasena`.
- **Backend:** `PasswordResetService` y `PasswordResetTokenRepository`.

Flujo Técnico

1. **Solicitud:** El usuario ingresa su email. Si existe, se genera un `PasswordResetToken` y se envía por email.
2. **Validación:** El usuario accede al link. El frontend valida el token mediante el endpoint `/api/auth/validate-reset-token`.

3. **Cambio:** El usuario envía la nueva contraseña junto con el token (`PasswordResetConfirmDTO`). El backend actualiza el hash en la entidad `User`.

Gestión de perfil y “completar perfil”

El sistema maneja un caso de uso especial llamado "Completar Perfil" (`CompleteProfileDTO`), esencial para usuarios creados por Google Auth que carecen de datos críticos como DNI o teléfono.

- **Frontend:** `CompleteProfileModalComponent`. Se activa si detecta el ingreso de un usuario con `profileCompleted = false`
- **Backend:** Endpoint `/api/user/complete-profile`. Actualiza la entidad `Paciente` asociada al usuario, asegurando la integridad de los datos médicos requeridos.

DeepLinking para mantenimiento de sesión (`DeepLinkService`)

El módulo de Deep Linking aborda un desafío de usabilidad y seguridad clave en sistemas web que envían notificaciones transaccionales por correo electrónico (como recordatorios de turnos, invitaciones a encuestas o enlaces de reseteo).

- **Problema resuelto:** Los enlaces contenidos en correos electrónicos asumen que el usuario ya tiene una sesión activa (token JWT válido). Si el usuario abre el enlace sin estar logueado, es forzado a iniciar sesión manualmente, duplicando esfuerzos o perdiendo el contexto de la acción deseada.
- **Solución arquitectónica:** El sistema utiliza un mecanismo de intercambio de tokens a través de la entidad `DeepLinkToken`.
    1. **Generación:** Servicios como `EmailService` o `EncuestaInvitacionService` generan un `DeepLinkToken` temporal y de **un solo uso** para un recurso o acción específicos

(ej., un turno). Este token se incluye en el enlace enviado por email.

2. **Punto de entrada el frontend:** El usuario ingresa a la ruta genérica `/deep-link-bridge?token=XYZ`. El componente `DeepLinkBridgeComponent` intercepta el token.
3. **Token swap (backend):** El frontend llama al endpoint `/api/deeplink/validate` con el `DeepLinkToken`. El **`DeepLinkService`** en el Backend:
    - Valida el token de un solo uso.
    - Identifica al `User` asociado.
    - Si es válido, **destruye el `DeepLinkToken`** (evitando su reutilización) y **genera un Token JWT de sesión estándar** para ese usuario.
4. **Mantenimiento de sesión:** El Backend devuelve el Token JWT estándar. El frontend lo almacena en `localStorage` (como en un login normal), iniciando una sesión válida sin necesidad de que el usuario ingrese credenciales.
5. **Redirección:** El frontend redirige internamente al usuario a la vista de destino (ej. la página de la encuesta o el detalle del turno).

Este flujo garantiza que el usuario pueda acceder al contenido relevante directamente desde el correo electrónico, manteniendo la seguridad (token de un solo uso) y proporcionando una experiencia de usuario fluida sin un login redundante.

5.3. Gestión médica

Este módulo administra el capital humano del sistema y la lógica requerida para transformar "horarios de contrato" en "turnos disponibles". Aborda la relación entre médicos, especialidades y la generación dinámica de la agenda.

Arquitectura del staff y especialidades

El sistema desacopla la identidad del profesional de su función en una clínica específica, permitiendo que un mismo médico trabaje en múltiples centros con distintas reglas.

Entidades del dominio

1. **`Medico` (global):** Representa a la persona física profesional. Contiene datos inmutables como DNI, Matrícula y Nombre. Es una entidad global visible por el `ROLE_SUPERADMIN`.
2. **`Especialidad` (catálogo):** Define las ramas de la medicina (cardiología, pediatría). Actúa como clasificador para la búsqueda de turnos.
3. **`StaffMedico` (vínculo contextual):**
    - **Función crítica:** Es la entidad asociativa que materializa la relación **muchos-a-muchos** entre `Medico` y `CentroAtencion`.
    - **Lógica de negocio:** Un médico *no existe* operativamente en una clínica hasta que no tiene un registro `StaffMedico` activo asociada a ese `centro_atencion_id`.
    - **Multi-especialidad:** Un registro de Staff permite asociar al médico con una o más especialidades que ejerce **específicamente en ese centro**.

Gestión desde el frontend

- **Módulo:** `src/app/staffMedicos/`.
- **Funcionalidad:** Los administradores gestionan el alta/baja de médicos en su centro.
- **Visualización:** El componente `StaffMedicoDetailComponent` permite configurar qué especialidades atiende el profesional en esa sede particular

Arquitectura del staff y especialidades

A diferencia de sistemas simples que guardan "turnos vacíos" en la base de datos, **CheTurno** utiliza un **motor de cálculo en tiempo real** para proyectar la

disponibilidad. Esto ahorra millones de registros en base de datos y facilita cambios masivos de horarios.

Definición de disponibilidad

Esta entidad configura el "contrato horario" del médico.

- **Estructura:** Define días de la semana (`DiaDeLaSemana`), hora de inicio, hora de fin, consultorio asignado (opcional) y duración del turno (slot).
- **Ejemplo:** *Dr. Pérez, Lunes, 09:00 a 13:00, turnos de 20 min.*

Componentes del `AgendaService`

El servicio un`psjb.labprog.backend.business.service.AgendaService` actúa como el cerebro de la planificación. Se compone de varias clases internas especializadas:

1. **`SlotGenerator` (generador de ranuras):**
    - Toma la configuración cruda (`DisponibilidadMedico` o `EsquemaTurno`).
    - Itera sobre el rango de fechas solicitado.
    - Divide el tiempo en bloques (`slots`) basados en la duración configurada.
    - **Resultado:** Una lista teórica de todos los turnos posibles si no hubiera ocupación.
2. **`ConflictResolver`:**
    - Cruza los *slots* teóricos generados con la tabla `Turno` (citas reales ya reservadas).
    - **Lógica:** Si existe un turno en estado `CONFIRMADO` o `PENDIENTE` que solapa con un slot teórico, marca ese slot como **OCUPADO**.
3. **`ExceptionalConfigurationHandler` (gestor de excepciones):**
    - Consulta la entidad `ConfiguracionExcepcional` para detectar feriados, licencias médicas o cambios de horario temporales.

    - **Precedencia:** Una configuración excepcional siempre sobrescribe la disponibilidad base. Si es un "bloqueo", elimina los slots; si es un "horario extra", agrega nuevos.
4. **`DisponibilidadValidator`:**
    - Utilizado al momento de crear un turno manual. Verifica atómicamente que el espacio de tiempo solicitado sea válido según las reglas vigentes, mostrando una advertencia en caso de un posible sobreturno.

Visualización de agenda

El frontend consume esta lógica procesada para mostrar calendarios intuitivos.

- **Componentes:** `MedicoHorariosComponent` y `AgendaAdminComponent`.
- **Tecnología:** Utiliza `angular-calendar` para renderizar visualmente los bloques de tiempo.
- **Flujo de datos:**
    1. Frontend solicita: `GET /rest/eventos?esquemaTurnoId=...&semanas=...`.
    2. Backend (`AgendaService`) calcula slots en memoria.
    3. Frontend recibe un JSON con objetos `TurnoDTO` y los pinta en la grilla con colores diferenciados (verde = libre, rojo = ocupado).

Esquema de turno

El esquema de turno representa la **configuración de atención específica de un médico** en un centro y consultorio determinado.

- **Propósito:** Define cuándo y dónde atiende un médico específico, incluyendo
    - Los días y horarios de atención (ej. Lunes 08:00-12:00, Miércoles 14:00-18:00)
    - El intervalo entre turnos (ej. 20 minutos)

    - El consultorio asignado
    - El centro de atención
- **Relación con Disponibilidad:** Cada esquema se basa en una `DisponibilidadMedico` previamente declarada, validando que los horarios del esquema estén dentro de la disponibilidad del profesional.
- **Generación dinámica:** El servicio `AgendaService` utiliza estos esquemas para **generar slots de turnos concretos** para fechas específicas, verificando en tiempo real feriados, mantenimientos y turnos ya ocupados.
- **Automatización:** El sistema puede:
    - Crear esquemas automáticamente desde las disponibilidades médicas que aún no tienen esquemas
    - Redistribuir consultorios según porcentajes configurados
    - Resolver conflictos de horarios cuando múltiples médicos necesitan el mismo consultorio

Gestión de dias excepcionales

Módulo para gestionar situaciones que alteran la disponibilidad normal de turnos (feriados, mantenimientos, horarios especiales).

Tipos de excepción:

- **Feriado:** Día no laborable que aplica a todo el sistema. No se generan turnos para ningún médico ni consultorio.
- **Mantenimiento:** Bloqueo temporal de un consultorio específico en un horario determinado (ej. limpieza, reparaciones).
- **Atención Especial:** Modificación puntual del horario de atención de un médico para una fecha específica (ej. guardia extendida, horario reducido).

Impacto: Al generar la agenda, `AgendaService` consulta estas configuraciones para cada fecha y slot. Los feriados bloquean el día completo, los mantenimientos afectan solo los horarios superpuestos del consultorio

indicado, y las atenciones especiales modifican dinámicamente los slots del esquema de turno asociado. Las configuraciones se desactivan lógicamente (`activo = false`) en lugar de eliminarse.

5.4. Core de turnos

El módulo de gestión de turnos es el corazón transaccional de **CheTurno**. Orquesta la interacción entre la disponibilidad teórica (agenda) y la ocupación real, garantizando la integridad de las citas médicas mediante una máquina de estados estricta y un motor de validaciones complejo.

Entidad turno y máquina de estados

La entidad un`psjb.labprog.backend.model.Turno` representa una reserva de tiempo confirmada o potencial. A nivel de persistencia, mantiene referencias redundantes (`medico`, `centroAtencion`, `especialidad`) además de la relación con `StaffMedico`, garantizando la integridad histórica de los datos para auditoría incluso si un médico es desvinculado de un centro.

Modelo de datos

Atributo	Tipo	Descripción y regla de negocio
paciente	*Paciente*	**Unicidad:** Un paciente no puede tener turnos superpuestos (duplicados) en la misma fecha y franja horaria.
medico	*Medico*	**Referencia global:** Permite validar que el profesional no tenga turnos superpuestos en **ningún** centro de atención de la plataforma.

staffMedico	*StaffMedico*	Vínculo contractual específico con el centro actual (puede ser nulo si el médico fue eliminado, pero `medico` persiste).
fecha/horaInicio/horaFin	*LocalDate/LocalTime*	Definen el slot temporal ocupado.
estado	*EstadoTurno*	Enum que controla el flujo de vida (`PROGRAMADO`, `CONFIRMADO`, `CANCELADO`, `COMPLETADO`).
asistio	*Boolean*	Flag de cierre del turno (Presente/Ausente).
observaciones	*String*	Campo de texto libre para notas médicas o administrativas.

Ciclo de vida

El sistema gestiona las transiciones de estado en `TurnoService` y `TurnoAutomationService`:

1. **PROGRAMADO (inicial):** El turno ha sido reservado.
    - *Regla de Caducidad:* Si no se confirma antes del tiempo límite configurado (default 48hs antes), el `TurnoAutomationService` lo cancela automáticamente.
2. **CONFIRMADO:** El paciente ha ratificado su asistencia.
    - *Ventana de Confirmación:* Configurable por centro de atención (entre 24hs y 29 días antes del turno).
3. **COMPLETADO:** El turno concluyó exitosamente. Se alcanza cuando el médico marca la asistencia (`asistio = true`).

4. **CANCELADO:** El turno se libera y el slot vuelve a estar disponible en la agenda. Puede ser disparado por el usuario, el médico o el proceso automático de vencimiento.
5. **AUSENTE:** Estado implícito cuando se marca `asistio = false` el día del turno.

Flujo de creación y validaciones

La creación de un turno es una operación transaccional crítica que ejecuta la siguiente cadena de validaciones de negocio:

1. **Validaciones de seguridad y roles**
    - **Auto-asignación:** Se verifica el contexto de seguridad. Si el usuario actual tiene rol `ROLE_MEDICO`, se impide que cree un turno donde `turno.medico.id == currentUser.persona.id`. *"Un médico no puede autoasignarse turnos"*.
    - **Sobreturnos:** La creación de turnos fuera de la grilla estándar (`sobreturno = true`) está estrictamente restringida a usuarios con rol **MEDICO** u **OPERADOR**.
2. **Validaciones de disponibilidad**
    - **Disponibilidad cruzada (Cross-Center):** Dado que un `Medico` es una entidad global en la plataforma SaaS, el sistema valida que el profesional no tenga turnos ocupados en **ningún otro centro de atención** en el horario solicitado.
    - **Duplicidad de paciente:** Se consulta `turnoRepository.existsByPacienteAndFecha...`. Si el paciente ya tiene un turno activo que solape con el horario solicitado, se lanza una excepción, impidiendo el acaparamiento de turnos.

Automatización y mantenimiento (`TurnoAutomationService`)

Este servicio ejecuta tareas programadas (`@Scheduled`) para mantener la higiene de la agenda y aplicar las reglas de negocio temporales.

Cancelación automática

- **Frecuencia:** Se ejecuta periódicamente (cron configurado en `application.properties`).
- **Lógica:**
    1. Calcula la fecha/hora límite de confirmación basada en la configuración del tenant (`turnos.auto-cancel.hours-before`, default 48h).
    2. Busca todos los turnos en estado `PROGRAMADO` cuya fecha de inicio sea inminente y haya superado el límite de confirmación.
    3. **Transición:** Cambia el estado a `CANCELADO` masivamente.
    4. **Notificación:** Dispara eventos al `EmailService` para avisar a los pacientes que su turno ha caducado.
    5. **Auditoría:** Registra la acción en `AuditLog` bajo el usuario sistema.

Gestión de asistencia

La marca de asistencia es el paso final del flujo operativo y tiene restricciones temporales estrictas.

- **Método:** `TurnoService.marcarAsistencia(id, asistio)`.
- **Regla Temporal:** El sistema valida contra `LocalDate.now()`:

```
if (!turno.getFecha().isEqual(LocalDate.now())) {
    throw new BusinessException("La asistencia solo se puede
marcar el día del turno.");
}
```

- **Impacto:** Actualiza el campo `asistio` y cambia el estado del turno para reflejar la conclusión del servicio, alimentando las estadísticas de ausentismo del Dashboard.

Lista de espera

Mecanismo para manejar la demanda insatisfecha cuando no hay turnos disponibles.

- **Modelo:** Entidad `ListaEspera` vinculada a `Especialidad` o `Medico`, y `Paciente`.
- **Restricción de seguridad:**
    - *Regla:* "Un paciente no puede designar a otro paciente".
    - *Implementación:* Al inscribirse, si el usuario es `ROLE_PACIENTE`, el sistema fuerza que el `paciente_id` del registro coincida con el usuario autenticado. Solo un `OPERADOR` o `ADMIN` puede inscribir a terceros.
- **Notificación de vacantes:** Cuando el `TurnoAutomationService` o un usuario cancela un turno, el sistema verifica la `ListaEspera` y notifica al siguiente paciente compatible mediante un email con *Deep Link* para tomar el turno liberado.

5.5. Auditoría y reportes

Este módulo proporciona una capa transversal de seguridad y trazabilidad, permitiendo reconstruir la historia de cambios de las entidades críticas del sistema. Se basa en el principio de **Inmutabilidad**: los registros de auditoría se crean y persisten, pero nunca se modifican ni eliminan.

Lógica de negocio

La entidad `unpsjb.labprog.backend.model.AuditLog` almacena la traza de eventos. Aunque soporta múltiples tipos de entidades mediante `entityType`, mantiene una relación física directa con `Turno` para optimizar las consultas más frecuentes del núcleo del negocio.

El servicio `unpsjb.labprog.backend.business.service.AuditLogService` no utiliza

un registro genérico único. En su lugar, implementa **métodos especializados por entidad** para manejar correctamente las relaciones y la lógica específica de cada módulo.

Métodos de registro especializados

El servicio expone métodos dedicados que encapsulan la lógica de creación del `AuditLog` según el tipo de objeto:

1. **Auditoría de turnos**
    - Se encarga de setear la relación `auditLog.setTurno(turno)` explícitamente.
    - Serializa el estado del turno preservando datos críticos (médico, paciente, estado) en los campos JSON (`oldValue`, `newValue`).
    - Registra acciones específicas como `TURNO_CREATE`, `TURNO_CANCEL`, `TURNO_ASISTENCIA`.
2. **Auditoría de configuración**
    - Método dedicado para registrar cambios en `Configuracion` o `ConfiguracionExcepcional`.
    - Útil para detectar cuándo se modificaron los parámetros del sistema (ej. tiempo límite de confirmación).
3. **Auditoría de usuarios/autenticación**
    - Registra eventos de seguridad como `USER_CREATE`, `USER_UPDATE`, etc.

Consultas y recuperación de datos

- **`getTurnoAuditHistory(Integer turnoId)`**: Recupera la lista cronológica de eventos asociados a un turno específico.
- **`getConfigChangeHistory(String clave, ...)`**: Permite rastrear la evolución de una configuración específica a lo largo del tiempo.

- **`getUltimaModificacionConfig()`**: Recupera el último cambio realizado en la configuración global, útil para cachés o validaciones de UI.
- **`getEntityAuditStatistics()`**: Genera métricas agregadas (`COUNT` agrupado por `entityType`) para el dashboard.

API REST

El controlador expone endpoints de **solo lectura** bajo `/audit`.

- `GET /audit/turno/{turnoId}`: Retorna el historial completo de un turno para la vista de detalle.
- `GET /audit/search`: Endpoint flexible que soporta filtros dinámicos (fecha, usuario, entidad) para la tabla de administración.
- `GET /audit/estadisticas/entidad`: Alimenta los gráficos del dashboard con la distribución de eventos.
- `GET /audit/recientes`: Recupera los últimos movimientos del sistema para monitoreo en tiempo real.

Visualización en frontend

El módulo de auditoría en Angular (`src/app/turnos/`) consume estos servicios para proporcionar transparencia operativa:

1. **Macro (dashboard - `AuditDashboardComponent`):**
    - Muestra gráficos de torta (`GraficoTortaComponent`) con la distribución de acciones.
    - Incluye una tabla de búsqueda avanzada (`TurnoAdvancedSearchComponent`) para investigaciones específicas.
2. **Micro (timeline - `HistorialTurnoDetalleComponent`)**
    - Componente visual que se renderiza dentro del detalle de un turno.

- Transforma la lista de `AuditLog` en una línea de tiempo vertical, permitiendo al operador ver rápidamente la secuencia de eventos (Reserva → Confirmación → Asistencia) y los responsables de cada paso.