**Manual Técnico**

**Sistema de Gestión Sesión de Bienestar**

*Desarrollador: Eduardo Steve Velásquez Gómez*

**índice**

**Introducción y arquitectura 3** **Diseño de Entidades y Base de Datos 4-5** **Seguridad con Spring Security y JWT 5-6** **Patrones de API: DTOs y Manejo de Errores 6** **Características Avanzadas 7**

**1. Introducción y Arquitectura**

Este documento describe la arquitectura de un servicio RESTful construido con **Spring Boot** para la gestión de una clínica de bienestar. El sistema maneja usuarios, roles (clientes y empleados), servicios, citas y notificaciones, con un enfoque en la seguridad moderna y el diseño de software escalable.

**Tecnologías Principales:**

* **Java 17+**
* **Spring Boot 3+**: Núcleo del framework.
* **Spring Data JPA (Hibernate)**: Persistencia de datos.
* **Spring Security 6+**: Autenticación y autorización.
* **PostgreSQL**: Base de datos relacional.
* **JWT (JSON Web Tokens)**: Para manejar sesiones de API sin estado.
* **Maven**: Gestión de dependencias.

**2. Diseño de Entidades y Base de Datos**

El diseño de la base de datos es el pilar del sistema. La decisión de diseño más importante fue la de **Composición sobre Herencia** para la asignación de roles.

**2.1. El Problema de "Herencia vs. Composición"**

* **Herencia (ES UN)**: Se usó para la base de la identidad: Usuario **ES UNA** Persona. Esto funciona porque un Usuario es una especialización de Persona y se crea como tal.
* **Composición (TIENE UN)**: Se usó para los roles. Intentar que un Usuario existente "se convierta" en Empleado (usando herencia) es conceptualmente incorrecto y causa errores de sesión en Hibernate (ej. A different object with the same identifier value...).
* **La Solución**: Un Usuario **TIENE UN** PerfilCliente o **TIENE UN** PerfilEmpleado. Esto se logra con una relación **@OneToOne** que permite "asignar" un rol a un Usuario en cualquier momento después de su creación.

**2.2. Diagrama de Entidades Clave**

* **Persona** (Clase Base):
  + @Id private BigInteger dpi;
  + @Inheritance(strategy = InheritanceType.JOINED)
* **Usuario** (Hereda de Persona):
  + Contiene: correo, contrasena.
  + @OneToOne(mappedBy = "usuario") → Cliente perfilCliente;
  + @OneToOne(mappedBy = "usuario") → Empleado perfilEmpleado;
  + @OneToMany(mappedBy = "usuario") → List<Notificacion> notificaciones;
* **Cliente** (Rol - **NO** hereda):
  + @Id @GeneratedValue private Long id; (Tiene su propio ID)
  + @OneToOne @JoinColumn(name = "usuario\_dpi") → Usuario usuario;
* **Empleado** (Rol - **NO** hereda):
  + @Id @GeneratedValue private Long id; (Tiene su propio ID)
  + @OneToOne @JoinColumn(name = "usuario\_dpi") → Usuario usuario;
* **ClienteServicio** (Tabla Pivote de Contrato):
  + @ManyToOne → Cliente cliente;
  + @ManyToOne → Servicio servicio;
* **Cita** (Entidad de Negocio):
  + @ManyToOne → ClienteServicio clienteServicio;
* **Notificacion** (Entidad de Negocio):
  + @ManyToOne → Usuario usuario;

**3. Seguridad con Spring Security y JWT**

El sistema es **sin estado** (STATELESS). La seguridad de cada petición se valida usando un Bearer Token.

**3.1. Flujo de Autenticación**

1. **Endpoints Públicos**: POST /api/auth/register y POST /api/auth/login son permitidos para todos en SecurityConfig.
2. **Login**: El AuthController usa AuthenticationManager para validar las credenciales.
3. **Generación de Token**: Si el login es exitoso, JwtUtil genera un token JWT firmado con una SECRET\_KEY (guardada de forma segura en application.properties).
4. **Respuesta**: El servidor devuelve el token al cliente.

**3.2. Flujo de Petición Segura**

1. **Petición del Cliente**: El cliente envía el token en el encabezado Authorization: Bearer <token>.
2. **Filtro**: JwtRequestFilter intercepta **todas** las peticiones.
3. **Validación**: El filtro extrae el token, lo valida usando JwtUtil, y extrae el username (correo).
4. **Carga de Usuario**: CustomUserDetailsService carga los detalles del usuario desde la BD.
5. **Contexto de Seguridad**: Si el token es válido, el filtro establece la autenticación en el SecurityContextHolder. La petición puede ahora acceder al controlador protegido.

**3.3. Manejo de Intentos Fallidos (Bloqueo de Cuentas)**

Para evitar ataques de fuerza bruta, el sistema bloquea cuentas tras 3 intentos fallidos.

* **Problema Común**: La lógica de UPDATE del contador fallaba debido a rollbacks de la transacción de autenticación.
* **Solución (Desacoplada)**:
  1. **AuthenticationFailureListener**: Un "oyente" de Spring que se activa automáticamente cuando AuthenticationManager falla (evento AuthenticationFailureBadCredentialsEvent).
  2. **LoginAttemptService**: Contiene la lógica transaccional.
  3. **@Transactional(propagation = Propagation.REQUIRES\_NEW)**: Se usa en el método sesionFallida(). Esto fuerza a Spring a crear una transacción **nueva e independiente** para actualizar el contador, asegurando que se guarde (COMMIT) incluso si la transacción de login principal falla (ROLLBACK).
  4. **usuarioRepository.save(usuario)**: Se usa explícitamente para forzar el UPDATE en la BD.

**4. Patrones de API: DTOs y Manejo de Errores**

**4.1. El Patrón DTO (Data Transfer Object)**

Nunca se exponen las entidades de JPA directamente en la API. Usamos DTOs para todo.

* **DTOs de Petición (Request)**: Usados para POST y PUT. Son "planos" y esperan IDs para las relaciones.
  + *Ejemplo*: CitaRequestDTO pide un clienteServicioId de tipo Long.
* **DTOs de Respuesta (Response)**: Usados para GET. Son "compuestos" y devuelven objetos JSON anidados y formateados.
  + *Ejemplo*: CitaResponseDTO no devuelve un ID, sino objetos ClienteSummaryDTO y ServicioSummaryDTO con los nombres y datos relevantes. Esto evita bucles de serialización infinitos.
  + **Formateo**: Se usa @JsonFormat(pattern = "dd-MM-yyyy HH:mm") en los campos de fecha de los DTOs de respuesta para entregar datos listos para el frontend.

**4.2. Manejo Global de Excepciones**

No se usan bloques try-catch en los controladores. Se usa un manejador global.

* **GlobalExceptionHandler (@ControllerAdvice)**: Una clase "guardián" que intercepta excepciones lanzadas desde cualquier controlador.
* **@ExceptionHandler**: Se definen métodos para cada tipo de excepción de negocio:
  + BadCredentialsException → 401 Unauthorized
  + LockedException → 403 Forbidden
  + UsernameNotFoundException → 404 Not Found
  + IllegalStateException (ej. "Perfil no asignado") → 403 Forbidden
* **ApiErrorResponse**: Un DTO personalizado para asegurar que todas las respuestas de error tengan una estructura JSON consistente.

**5. Características Avanzadas**

**5.1. Sistema de Notificaciones**

* **NotificacionService**: Un servicio centralizado (crearNotificacion(Usuario u, String msg)) que puede ser inyectado y llamado desde cualquier otro servicio (ej. CitaService).
* **NotificacionController**: Expone endpoints (GET /api/notificaciones, POST /{id}/marcar-leida) para que el frontend pueda consultar y actualizar las notificaciones del usuario logueado.

**5.2. Auditoría Automática**

Se implementó un sistema de registro de cambios automático usando Listeners de JPA.

* **HistorialCambios**: Entidad que almacena el log (entidad, accion, usuario, fechaHora).
* **AuditListener**: Una clase que no es de Spring pero que "escucha" eventos de JPA.
  + @PrePersist: Se dispara antes de un INSERT.
  + @PreUpdate: Se dispara antes de un UPDATE.
* **@EntityListeners(AuditListener.class)**: Anotación que se coloca sobre las entidades que queremos auditar (ej. Cliente, Servicio).
* **Manejo de Usuario Anónimo**: El listener contiene lógica (obtenerUsuarioActual()) que consulta el SecurityContextHolder. Si no hay un usuario autenticado (como en un registro), guarda el usuario como "Anónimo" o "SISTEMA".
* **try-catch**: Toda la lógica del listener está envuelta en un try-catch para asegurar que un fallo en la auditoría **nunca** interrumpa la operación de negocio principal.