MANUAL TECNICO DEFINITIVO DEL SISTEMA DE RESTAURANTES PIURA

(GustitoLocal)

ÍNDICE GENERAL

I. Guía para No Técnicos: Comparativa con un restaurante 3

II. Configuración, Seguridad y Puesta en Marcha 4

1. Protocolos de Seguridad y Configuración 4

2. Despliegue y Mantenimiento 4

III. Lógica de Negocio y Ciclo de Vida del Restaurante 5

1. Flujos de Validación y Moderación 5

2. Procesos Automáticos (Scheduler) 6

IV. Detalle Técnico del Frontend 6

1. Conexiones a las API (api.ts) 6

2. Manejo de autenticación (auth.ts) 7

V. Detalle Técnico de las Capas del Backend 9

1. Capa de Modelos 9

A. Entidad de Usuario (User.java) 9

B. Entidad de Roles (Role.java) 10

C. Entidad de Restaurante (Restaurante.java) 10

D. Entidad de Estado (Estado.java) 11

E. Entidad de Horario (Horario.java) 11

F. Entidad de Notificación (Notificacion.java) 11

2. Capa de Controladores (Controllers) 13

A. AuthController.java (Autenticación) 13

B. PublicController.java (Consultas Públicas) 14

C. PropietarioController.java (Gestión de Propietarios) 17

D. RestauranteController.java (Administración) 20

3. Capa de DTOs (Data Transfer Objects) 24

- DTOs de Solicitud (Requests) 25
- DTOs de Respuesta (Responses) 27

4. Capa de Repositorios y Consultas Eficientes 29

5. Capa de Servicios (Service) 30

5.1. Servicio de Restaurantes (RestaurantService.java) 30

5.2. Servicio de Usuario (UserService.java) 35

VI. Referencia de Endpoints y Guía Práctica de Pruebas 37

A. Referencia de Endpoints Clave 37

MANUAL TECNICO DEFINITIVO DEL SISTEMA DE RESTAURANTES PIURA (GustitoLocal)

GustitoLocal es una plataforma web para la gestión de restaurantes, construida con una arquitectura de Frontend (React + TypeScript) y Backend (Spring Boot + MongoDB).

I. Guía para No Técnicos: Comparativa con un restaurante

La Arquitectura en Capas del Backend se explica mejor con la analogía de un restaurante, asegurando que cada componente tenga una única responsabilidad.

Componente	Analogía	Función Central y Tecnológica
Frontend	**El Salón**	**Interfaz de usuario, desarrollado con React y Vite.**
Backend	**La Cocina**	**El motor de la aplicación donde reside toda la lógica.**
Controller	**El Camarero**	**Recibe el pedido (petición HTTP) y lo delega al servicio. No contiene lógica de negocio.**
Service	**El Chef Principal**	**Contiene las reglas de negocio y orquesta la operación.**
Repository	**El Encargado del Almacén**	**Acceso directo a la Base de Datos MongoDB.**
DTOs	**El Menú**	**Define el contrato exacto de la API, utilizado para validación y seguridad.**
Security	**El Portero**	**Verifica la reserva (Token JWT) antes de permitir el acceso a la cocina.**

II. Configuración, Seguridad y Puesta en Marcha

1. Protocolos de Seguridad y Configuración

Aspecto	Mecanismo	Detalle
Autenticación	**JWT HS256**	**Utilizado para USER y ADMIN. El token tiene una expiración corta (15 minutos).**
Contraseñas	**BCrypt**	**Las contraseñas se almacenan cifradas en la BD (passwordHash).**
Autorización	**SecurityFilterChain**	**Define permisos específicos: /api/mis-restaurantes/** requiere hasRole('USER') y /api/administracion/** requiere hasRole('ADMIN').**
Manejo de CORS	**CorsConfig.java**	**Configurado para permitir peticiones desde múltiples orígenes (configurable a la URL de producción).**
Inicialización	**DataInitializer.java**	**Crea automáticamente el usuario administrador (admin@restaurantes.local / Admin123!) al iniciar el servidor si no existe.**

2. Despliegue y Mantenimiento

- Puesta en Marcha Local:
  - Backend: ./mvnw spring-boot:run.
  - Frontend: npm run dev.
  - Acceso: Frontend (http://localhost:5173), Swagger (http://localhost:3000/swagger-ui.html).
- Backup y Recuperación: Se recomienda realizar copias de seguridad diarias de MongoDB utilizando los comandos mongodump y mongorestore.
- Variables Sensibles: Las claves secretas (jwt.secret, api.key) DEBEN ser gestionadas mediante variables de entorno y nunca subidas al repositorio de código.

III. Lógica de Negocio y Ciclo de Vida del Restaurante

1. Flujos de Validación y Moderación

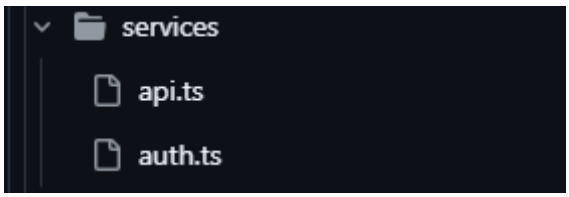
Flujo de Acción	Rol	Lógica Detallada (Service)
Registro	USER	crearSolicitud() valida que la ubicación (lat/lng) esté dentro de los límites de Piura (estaEnPiura()) y establece el estado inicial a PENDIENTE.
Apelación	USER	apelarRechazo() permite al propietario modificar la solicitud si el restaurante está RECHAZADO y no han pasado más de 3 días desde el rechazo (PLAZO_APELACION_VENCIDO).
Aprobación	ADMIN	cambiarValidacion() pasa el estado a APROBADO, activa el restaurante (activo: true) y lo hace visible públicamente. Si es RECHAZADO, se registra el motivoRechazo.
Subida de Pruebas	USER	guardarPruebas() guarda los archivos en una carpeta dedicada (uploads/restaurantes/{id}/) y actualiza pruebasCount.

2. **Procesos Automáticos (Scheduler)**

El sistema utiliza tareas programadas para el mantenimiento, con el HorariosAutomaticosScheduler como orquestador.

Tarea	**Frecuencia**	**Propósito de la Tarea**
Actualizar Horarios	Cada minuto	Ejecuta actualizarEstadosPorHorarios(), que usa consultas personalizadas para abrir o cerrar restaurantes, manejando correctamente los horarios que cruzan la medianoche.
Limpieza (24h)	Cada hora	Ejecuta eliminarRestaurantesCerradosPermanentemente(): elimina físicamente archivos, horarios y el documento de BD para restaurantes con Cierre Permanente de más de 24 horas.
Cerrar Apelaciones	Cada hora	Desactiva la posibilidad de apelar para solicitudes rechazadas que han superado el plazo de 3 días.

IV. Detalle Técnico del Frontend



1. **Conexiones a las API (api.ts)**

Aquí ocurre la conexión principal de las API con el proyecto

```
const API_BASE_URL = 'http://localhost:3000';
```

Este fragmento contiene la dirección base (Host y Puerto) donde debe estar corriendo la aplicación de Spring Boot.

```
const api = axios.create({
  baseURL: API_BASE_URL,  // URL base: todas las peticiones empiezan con "http://localhost:3000"
  headers: {
    'Content-Type': 'application/json',  // Le decimos al backend que enviamos datos en formato JSON
  },
});
```

api.get(...) combina la baseURL (http://localhost:3000) con la ruta del endpoint (/api/restaurantes/obtener) para formar la URL completa de la API a la que está llamando React: 'api/restaurantes/obtener'

```
const response = await api.get<Restaurante[]>('/api/restaurantes/obtener');
```

Y este código se asegura de que cada llamada a la API incluya el Token JWT en el encabezado. De esta manera, Spring Boot recibe el token y puede usarlo para verificar si el usuario tiene el rol ADMIN o USER antes de devolver los datos.

```
api.interceptors.request.use((config) => {
  // Buscar el token guardado en el navegador (si existe)
  const token = localStorage.getItem('token');

  // Si hay un token, agregarlo al header de la petición
  if (token) {
    config.headers = config.headers ?? {};
    // Formato estándar: "Bearer" seguido del token
    (config.headers as any).Authorization = `Bearer ${token}`;
  }

  // Devolver la configuración modificada (con el token agregado)
  return config;
});
```

2. Manejo de autenticación (auth.ts)

Primero se va a importar el servicio api configurado previamente

```
// Importar el servicio de API configurado (que ya tiene la URL del backend)
import api from './api';
```

Luego se va a llamar a un POST para poder realizar el registro de una cuenta nueva

```
register: async (email: string, password: string) => {
  // POST /api/auth/register → Envía datos al backend para crear usuario
  const res = await api.post('/api/auth/register', { email, password });
  // El backend devuelve: { token: "abc123...", role: "USER" }
  return res.data as { token: string; role: string };
},
```

Y por último vamos a llamar a otro POST que se va a encargar de realizar la verificación de las credenciales.

```
  login: async (email: string, password: string) => {
    // POST /api/auth/login → Envía credenciales al backend para verificar
    const res = await api.post('/api/auth/login', { email, password });
    // El backend devuelve: { token: "abc123...", role: "USER" o "ADMIN" }
    return res.data as { token: string; role: string };
  },
```

V. Detalle Técnico de las Capas del Backend

1. Capa de Modelos

A. Entidad de Usuario (User.java)

La entidad User gestiona la información de autenticación y perfil de una persona. Sus campos principales son el id, email (usado como login), y el passwordHash (encriptado con BCrypt). Define el nivel de permisos a través del campo role (que puede ser USER o ADMIN). Almacena datos adicionales del propietario como nombreCompleto, dni, ruc, telefono y direccion, y un nivelSeguridad basado en los datos completados.

```
@Id
private String id;

/** Email del usuario, usado como nombre de usuario para el login */
private String email;

/** Contraseña hasheada usando BCrypt */
private String passwordHash;

/** Rol del usuario en el sistema (USER o ADMIN) */
private Role role = Role.USER;

/** Timestamp de creación del usuario */
private Instant createdAt = Instant.now();

/** Nombre completo del propietario */
private String nombreCompleto;

/** DNI del propietario */
private String dni;

/** RUC del negocio (si aplica) */
private String ruc;
```

```
/** Teléfono de contacto */
private String telefono;

/** Foto de perfil (nombre del archivo almacenado) */
private String fotoPerfil;

/** Dirección personal del propietario */
private String direccion;

/** Nivel de seguridad basado en datos completados (1-10) */
private int nivelSeguridad = 1;

/** Timestamp de última actualización del perfil */
private Instant actualizadoEn;
```

B. Entidad de Roles (Role.java)

El enum Role define los dos posibles roles de los usuarios en el sistema: USER (puede registrar y gestionar sus propios restaurantes) y ADMIN (tiene permisos completos para la gestión de todos los restaurantes).

```java
public enum Role {
    /**
     * Rol de usuario regular.
     * Puede registrar restaurantes y gestionar sus propios establecimientos.
     */
    USER,

    /**
     * Rol de administrador.
     * Tiene permisos completos sobre todos los restaurantes del sistema.
     */
    ADMIN
}
```

C. Entidad de Restaurante (Restaurante.java)

La entidad Restaurante representa un establecimiento de comida en el sistema. Contiene información básica como el nombre, barrio, capacidad, contacto y tipo. Para la ubicación, utiliza un `GeoJsonPoint` llamado location y gestiona el estado operativo actual mediante un objeto estado. Un restaurante está asociado a su propietario a través del ownerUserId y a una lista de horarios asociados (`Horario.java`). Además, maneja un proceso de validación administrativa con campos como estadoValidacion ("PENDIENTE", "RECHAZADO"), motivoRechazo, pruebasFolder y pruebasCount. Si el propietario decide cerrarlo, se marcan los campos cerradoPermanente y fechaCierrePermanente.

```java
  @Id
  private String id;
  private String nombre;
  private String barrio;
  private boolean activo = true;
  @GeoSpatialIndexed
  private GeoJsonPoint location;
  private Estado estado;
  private int capacidad;
  private String contacto;
  private String tipo;
  private String ownerUserId;
  private String estadoValidacion = "PENDIENTE";
  /**
   * Motivo del rechazo de la solicitud, si fue rechazada por el administrador.
   * Solo se llena cuando estadoValidacion == "RECHAZADO".
   * Puede incluir razones específicas como "Fotos insuficientes", "Ubicación fuera de Piura", etc.
   */
  private String motivoRechazo;
  /**
   * Carpeta en disco donde se almacenan las pruebas/fotos subidas
   * por el propietario para validacion administrativa.
   */
```

D. Entidad de Estado (Estado.java)

La clase Estado es un objeto incrustado que detalla la situación operativa actual de un restaurante. Indica si el restaurante está abierto o no, cuándo fue actualizadoEn, y puede incluir una nota opcional.

```
private boolean abierto;
private Instant actualizadoEn;
private String nota;
```

E. Entidad de Horario (Horario.java)

La entidad Horario describe las horas de atención específicas de un restaurante. Incluye el diaSemana (ej: "LUNES") y las cadenas de texto horaApertura y horaCierre (ej: "08:00" y "22:00"). Contiene una referencia (@DBRef) al objeto restaurante al que pertenece e incluye un booleano activo para indicar si ese horario está vigente.

```
@Document("horarios")
public class Horario {

    @Id
    private String id;
    private String diaSemana; // "LUNES", "MARTES", "MIERCOLES", "JUEVES", "VIERNES", "SABADO", "DOMINGO"
    private String horaApertura; // "08:00"
    private String horaCierre;   // "22:00"
    private boolean activo = true;

    @DBRef
    private Restaurante restaurante;
```

F. Entidad de Notificación (Notificacion.java)

La entidad notificación está diseñada para alertar al administrador sobre eventos importantes en el sistema. Se crea para eventos como cuando un propietario cierra permanentemente su restaurante. Contiene un tipo, titulo y mensaje descriptivo. Incluye referencias opcionales a las entidades relacionadas como restauranteId, restauranteNombre y userId. Gestiona si ha sido leída con el booleano leida y los campos de tiempo fechaCreacion y fechaLectura.

```java
@Document("notificaciones")
public class Notificacion {

    @Id
    private String id;

    /**
     * Tipo de notificación (ej: "RESTAURANTE_CERRADO_PERMANENTEMENTE")
     */
    private String tipo;

    /**
     * Título de la notificación
     */
    private String titulo;

    /**
     * Mensaje descriptivo de la notificación
     */
    private String mensaje;

    /**
     * ID del restaurante relacionado (si aplica)
     */
    private String restauranteId;

    /**
     * Nombre del restaurante relacionado
     */
    private String restauranteNombre;
```

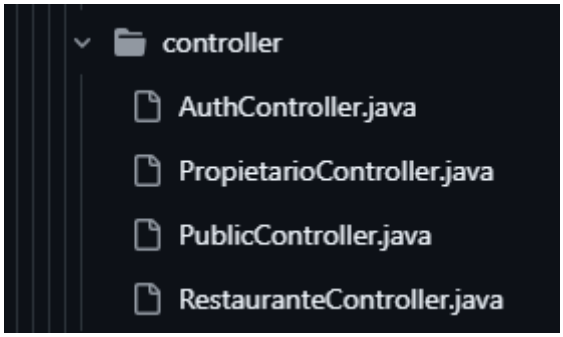
```java
/**
 * ID del usuario que generó la notificación (si aplica)
 */
private String userId;

/**
 * Indica si la notificación ha sido leída por el admin
 */
private boolean leida = false;

/**
 * Fecha y hora en que se creó la notificación
 */
private Instant fechaCreacion = Instant.now();

/**
 * Fecha y hora en que fue leída (si ya fue leída)
 */
private Instant fechaLectura;
```

2. Capa de Controladores (Controllers)



A. AuthController.java (Autenticación)

Este controlador maneja las operaciones de registro e inicio de sesión, que son endpoints públicos (no requieren JWT para acceder).

- Ruta Base: /api/auth

```
@RequestMapping("/api/auth")
```

- Funcionalidades Clave:
  - POST /api/auth/register: Registra un nuevo usuario en el sistema con el rol USER por defecto. Si el email ya está registrado, retorna un error. Devuelve un token JWT y el rol del usuario.

```
@PostMapping("/register")
public ResponseEntity<?> register(@RequestBody RegisterDto dto) {
    // Validar que los datos no sean nulos
    if (dto == null || dto.email() == null || dto.password() == null) {
        return ResponseEntity.badRequest().body(Map.of("error", "DATOS_INVALIDOS"));
    }

    // Verificar que el email no esté registrado
    if (userRepository.existsByEmail(dto.email())) {
        return ResponseEntity.badRequest().body(Map.of("error", "EMAIL_YA_REGISTRADO"));
    }

    // Crear y guardar el nuevo usuario
    User user = new User();
    user.setEmail(dto.email());
    user.setPasswordHash(passwordEncoder.encode(dto.password()));
    user.setRole(Role.USER);
    userRepository.save(user);

    // Generar y retornar el token JWT
    String token = jwtService.generate(user);
    return ResponseEntity.status(201).body(Map.of("token", token, "role", user.getRole()));
}
```

- POST /api/auth/login: Autentica a un usuario existente con email y contraseña. Si las credenciales son correctas, retorna un token JWT y el rol (USER o ADMIN).

```java
@PostMapping("/login")
public ResponseEntity<?> login(@RequestBody LoginDto dto) {
    // Validar que los datos no sean nulos
    if (dto == null || dto.email() == null || dto.password() == null) {
        return ResponseEntity.badRequest().body(Map.of("error", "DATOS_INVALIDOS"));
    }

    // Buscar usuario por email
    User user = userRepository.findByEmail(dto.email()).orElse(null);

    // Validar contraseña
    if (user == null || !passwordEncoder.matches(dto.password(), user.getPasswordHash())) {
        return ResponseEntity.status(401).body(Map.of("error", "CREDENCIALES_INVALIDAS"));
    }

    // Generar y retornar el token JWT
    String token = jwtService.generate(user);
    return ResponseEntity.ok(Map.of("token", token, "role", user.getRole()));
}
```

B. PublicController.java (Consultas Públicas)

Este controlador expone las funcionalidades de consulta de restaurantes que son públicas y no requieren autenticación (sin JWT). Solo trabaja con restaurantes que han sido APROBADOS y están activos.

- Ruta Base: /api/restaurantes

```java
@RequestMapping("/api/restaurantes")
```

- Funcionalidades Clave:
  - GET /api/restaurantes/obtener: Retorna una lista de todos los restaurantes activos y aprobados en el sistema.

```java
@GetMapping("/obtener")
public ResponseEntity<List<RestauranteResponseDto>> obtenerRestaurantes() {
    // listarPublico() ya filtra por activo=true y estadoValidacion='APROBADO'
    List<Restaurante> restaurantes = restauranteService.listarPublico();

    // Doble verificación: asegurar que solo se muestran activos y aprobados
    restaurantes = restaurantes.stream()
        .filter(r -> r.isActivo() && "APROBADO".equals(r.getEstadoValidacion()))
        .collect(Collectors.toList());

    List<RestauranteResponseDto> response = restaurantes.stream()
        .map(this::convertirAResponseDto)
        .collect(Collectors.toList());

    return ResponseEntity.ok(response);
}
```

- GET /api/restaurantes/buscar: Permite buscar restaurantes activos y aprobados usando filtros como:
  - Proximidad geográfica (lat, lng, radioKm).
  - Tipo de restaurante.
  - Estado (abierto o cerrado).
  - Capacidad mínima.

```java
@GetMapping("/buscar")
public ResponseEntity<List<RestauranteResponseDto>> buscar(
        @Parameter(description = "Latitud del centro de búsqueda", example = "-5.1945")
        @RequestParam(required = false) Double lat,
        @Parameter(description = "Longitud del centro de búsqueda", example = "-80.6328")
        @RequestParam(required = false) Double lng,
        @Parameter(description = "Radio de búsqueda en kilómetros (por defecto: 3 km)", example = "5")
        @RequestParam(required = false, defaultValue = "3") Double radioKm,
        @Parameter(description = "Tipo de restaurante", example = "Comida rápida")
        @RequestParam(required = false) String tipo,
        @Parameter(description = "Filtrar por estado (true=abierto, false=cerrado)", example = "true")
        @RequestParam(required = false) Boolean abierto,
        @Parameter(description = "Capacidad mínima de comensales", example = "20")
        @RequestParam(required = false, defaultValue = "0") Integer capacidadMin
) {
```

```
        List<Restaurante> base = restauranteService.listarPublico();

        // Proximidad simple si se proveen coordenadas
        // NOTA: buscarRestaurantesCerca ya filtra por activo=true y estadoValidacion='APROBADO'
        if (lat != null && lng != null && radioKm != null) {
            base = restauranteService.buscarRestaurantesCerca(lng, lat, radioKm);
        }

        // Filtros adicionales en memoria para mantenerlo sencillo
        // Asegurar que solo se muestran activos y aprobados (doble verificación)
        base = base.stream()
            .filter(r -> r.isActivo() && "APROBADO".equals(r.getEstadoValidacion()))
            .toList();

        if (tipo != null && !tipo.isBlank()) {
            base = base.stream().filter(r -> tipo.equalsIgnoreCase(r.getTipo())).toList();
        }
        if (abierto != null) {
            base = base.stream().filter(r -> r.getEstado() != null && r.getEstado().isAbierto() == abierto).toList();
        }
        if (capacidadMin != null && capacidadMin > 0) {
            base = base.stream().filter(r -> r.getCapacidad() >= capacidadMin).toList();
        }

        List<RestauranteResponseDto> response = base.stream()
            .map(this::convertirAResponseDto)
            .collect(Collectors.toList());
        return ResponseEntity.ok(response);
    }
```

- Conversión a DTO: Incluye un método privado para convertir la información del restaurante al DTO de respuesta pública, que incluye el nivel de seguridad del propietario como un dato público.

```
    private RestauranteResponseDto convertirAResponseDto(Restaurante restaurante) {
        // Si el estado es null, asumir que está cerrado (abierto = false)
        boolean abierto = restaurante.getEstado() != null && restaurante.getEstado().isAbierto();

        // Obtener horarios activos del restaurante
        List<Horario> horarios = horarioRepository.findByRestauranteIdAndActivoTrue(restaurante.getId());

        // Convertir horarios a DTOs
        List<HorarioResponseDto> horariosDto = horarios.stream()
            .map(h -> new HorarioResponseDto(
                h.getId(),
                h.getDiaSemana(),
                h.getHoraApertura(),
                h.getHoraCierre(),
                h.isActivo(),
                restaurante.getId()
            ))
            .collect(Collectors.toList());

        // Obtener nivel de seguridad del propietario
        Integer nivelSeguridad = null;
        if (restaurante.getOwnerUserId() != null) {
            try {
                User propietario = userRepository.findById(restaurante.getOwnerUserId()).orElse(null);
                if (propietario != null) {
                    nivelSeguridad = propietario.getNivelSeguridad();
                }
            } catch (Exception e) {
                // Si no se puede obtener el propietario, mantener nivelSeguridad como null
                System.err.println("Error al obtener nivel de seguridad del propietario: " + e.getMessage());
            }
        }
```

C. PropietarioController.java (Gestión de Propietarios)

Este controlador agrupa los endpoints protegidos con JWT y rol USER que permiten a los propietarios gestionar sus propios restaurantes y perfiles.

- Ruta Base: /api/mis-restaurantes

```
@RequestMapping("/api/mis-restaurantes")
```

- Restaurantes (Propietario):
  - GET /api/mis-restaurantes: Lista todos los restaurantes registrados por el propietario autenticado, sin importar su estado (PENDIENTE, APROBADO, RECHAZADO).

```
@GetMapping
public ResponseEntity<?> listar(Authentication auth) {
    // Extraer el ID del usuario del token JWT
    String ownerId = (String) auth.getPrincipal();

    // Retornar todos los restaurantes del propietario
    return ResponseEntity.ok(restauranteService.listarPorOwner(ownerId));
}
```

  - POST /api/mis-restaurantes: Registra una nueva solicitud de restaurante. Por defecto, queda en estado PENDIENTE para validación.

```
@PostMapping
public ResponseEntity<?> registrar(Authentication auth, @Valid @RequestBody RestauranteRequestDto dto) {
    try {
        // Validar que el usuario esté autenticado
        if (auth == null || auth.getPrincipal() == null) {
            throw new IllegalArgumentException("NO_AUTENTICADO");
        }

        // Extraer el ID del usuario del token JWT
        String ownerId = (String) auth.getPrincipal();

        // Crear la solicitud de restaurante (estado PENDIENTE por defecto)
        Restaurante restaurante = restauranteService.crearSolicitud(dto, ownerId);

        // Retornar respuesta con el ID generado por MongoDB
        return ResponseEntity.ok(Map.of("ok", true, "id", restaurante.getId()));
    } catch (IllegalArgumentException e) {
        // El GlobalExceptionHandler manejará este error
        throw e;
    }
}
```

  - PUT /api/mis-restaurantes/{id}/estado: Actualiza el estado (abierto/cerrado) de un restaurante. Solo funciona para restaurantes APROBADOS.

```java
@PutMapping("/{id}/estado")
public ResponseEntity<?> actualizarEstado(
        Authentication auth,
        @Parameter(description = "ID del restaurante", example = "507f1f77bcf86cd799439011", required = true)
        @PathVariable String id,
        @Valid @RequestBody EstadoUpdateDto dto) {

    // Extraer el ID del usuario del token JWT
    String ownerId = (String) auth.getPrincipal();

    // Actualizar estado (el servicio valida propiedad y aprobación)
    Restaurante restaurante = restauranteService.actualizarEstadoOwner(ownerId, id, dto);

    return ResponseEntity.ok(restaurante);
}
```

- PUT /api/mis-restaurantes/{id}/horarios: Actualiza o crea los horarios de atención del restaurante.

```java
@PutMapping("/{id}/horarios")
public ResponseEntity<?> actualizarHorarios(
        Authentication auth,
        @Parameter(description = "ID del restaurante", example = "507f1f77bcf86cd799439011", required = true)
        @PathVariable String id,
        @Valid @RequestBody HorarioRequestDto dto) {

    // Extraer el ID del usuario del token JWT
    String ownerId = (String) auth.getPrincipal();

    // Actualizar horarios (el servicio valida que el usuario sea propietario)
    Restaurante restaurante = restauranteService.actualizarHorariosOwner(ownerId, id, dto);

    return ResponseEntity.ok(restaurante);
}
```

- PUT /api/mis-restaurantes/{id}/apelar: Permite apelar el rechazo de un restaurante (si fue rechazado hace menos de 3 días), volviendo su estado a PENDIENTE.

```java
@PutMapping("/{id}/apelar")
public ResponseEntity<?> apelarRechazo(
        Authentication auth,
        @Parameter(description = "ID del restaurante a apelar", example = "507f1f77bcf86cd799439011", required = true)
        @PathVariable String id,
        @Valid @RequestBody RestauranteRequestDto dto) {

    // Extraer el ID del usuario del token JWT
    String ownerId = (String) auth.getPrincipal();

    // Apelar (el servicio valida propiedad, estado y plazo)
    Restaurante restaurante = restauranteService.apelarRechazo(ownerId, id, dto);

    return ResponseEntity.ok(restaurante);
}
```

- PUT /api/mis-restaurantes/{id}/cerrar-permanentemente: Cierra permanentemente un restaurante APROBADO. Será eliminado automáticamente después de 24 horas.

```java
@PutMapping("/{id}/cerrar-permanentemente")
public ResponseEntity<?> cerrarPermanentemente(
        Authentication auth,
        @Parameter(description = "ID del restaurante a cerrar permanentemente", example = "507f1f77bcf86cd799439011", required = true)
        @PathVariable String id) {

    // Extraer el ID del usuario del token JWT
    String ownerId = (String) auth.getPrincipal();

    // Cerrar permanentemente (el servicio valida propiedad y aprobación)
    Restaurante restaurante = restauranteService.cerrarPermanentemente(ownerId, id);

    return ResponseEntity.ok(Map.of(
        "mensaje", "Restaurante cerrado permanentemente. Será eliminado después de 24 horas.",
        "restaurante", restaurante
    ));
}
```

- Perfil de Usuario:
  - GET /api/mis-restaurantes/perfil: Obtiene el perfil completo del usuario (incluyendo nivel de seguridad).

```java
@GetMapping("/perfil")
public ResponseEntity<?> obtenerMiPerfil(Authentication auth) {
    String userId = (String) auth.getPrincipal();
    User user = userService.obtenerUsuario(userId);

    PerfilResponseDto dto = convertirAPerfilResponseDto(user);
    return ResponseEntity.ok(dto);
}
```

  - PUT /api/mis-restaurantes/perfil: Actualiza los datos del perfil (DNI, RUC, etc.), lo que puede aumentar el nivel de seguridad.

```java
@PutMapping("/perfil")
public ResponseEntity<?> actualizarMiPerfil(Authentication auth, @Valid @RequestBody ActualizarPerfilDto dto) {
    String userId = (String) auth.getPrincipal();
    User user = userService.actualizarPerfil(userId, dto);

    PerfilResponseDto response = convertirAPerfilResponseDto(user);
    return ResponseEntity.ok(response);
}
```

  - POST /api/mis-restaurantes/perfil/foto: Sube una foto de perfil, lo que aumenta el nivel de seguridad en +3 puntos.

```java
@PostMapping("/perfil/foto")
public ResponseEntity<?> subirFotoPerfil(
        Authentication auth,
        @Parameter(description = "Archivo de imagen de perfil", required = true)
        @RequestParam("archivo") MultipartFile archivo) throws IOException {

    String userId = (String) auth.getPrincipal();

    // Validar que sea una imagen
    String nombreArchivo = archivo.getOriginalFilename();
    if (nombreArchivo == null || nombreArchivo.isEmpty()) {
        return ResponseEntity.badRequest().body(Map.of("error", "Nombre de archivo inválido"));
    }

    String extension = nombreArchivo.substring(nombreArchivo.lastIndexOf('.'));
    if (!extension.equalsIgnoreCase(".jpg") && !extension.equalsIgnoreCase(".jpeg") &&
        !extension.equalsIgnoreCase(".png") && !extension.equalsIgnoreCase(".gif") &&
        !extension.equalsIgnoreCase(".webp")) {
        return ResponseEntity.badRequest().body(Map.of("error", "Solo se permiten archivos de imagen (JPG, PNG, GIF, WEBP)"));
    }
```

- GET /api/mis-restaurantes/perfil/foto/{nombreArchivo}: Sirve la foto de perfil del usuario.

```java
public ResponseEntity<?> obtenerFotoPerfil(
        Authentication auth,
        @Parameter(description = "Nombre del archivo de la foto", required = true)
        @PathVariable String nombreArchivo) throws IOException {

    String userId = (String) auth.getPrincipal();

    // Validar que la foto pertenezca al usuario
    if (!nombreArchivo.startsWith(userId + "_")) {
        return ResponseEntity.notFound().build();
    }
```

D. RestauranteController.java (Administración)

Este controlador está dedicado exclusivamente a los administradores y requiere autenticación JWT con rol ADMIN. Maneja la gestión total de los restaurantes y las notificaciones del sistema.

- Ruta Base: /api/administracion/restaurantes

```java
@RequestMapping("/api/administracion/restaurantes")
```

- Gestión de Restaurantes:
  - POST /api/administracion/restaurantes: Crea un nuevo restaurante directamente con estado APROBADO (saltándose el proceso de validación).

```java
@PostMapping
public ResponseEntity<?> crearRestaurante(@Valid @RequestBody RestauranteRequestDto request) {
    try {
        Restaurante restaurante = restauranteService.crear(request);

        Map<String, Object> response = new HashMap<>();
        response.put("ok", true);
        response.put("id", restaurante.getId());

        return ResponseEntity.ok(response);

    } catch (IllegalArgumentException e) {
        if ("FUERA_DE_PIURA".equals(e.getMessage())) {
            Map<String, Object> error = new HashMap<>();
            error.put("error", Map.of(
                "code", "FUERA_DE_PIURA",
                "message", "Las coordenadas están fuera de los límites de Piura"
            ));
            return ResponseEntity.badRequest().body(error);
        }
```

- GET /api/administracion/restaurantes: Lista todos los restaurantes del sistema con información administrativa completa.

```java
@GetMapping
public ResponseEntity<List<RestauranteAdminResponseDto>> listarRestaurantes() {
    List<Restaurante> restaurantes = restauranteService.listarTodos();

    List<RestauranteAdminResponseDto> response = restaurantes.stream()
        .map(this::convertirAAdminResponseDto)
        .collect(Collectors.toList());

    return ResponseEntity.ok(response);
}
```

- DELETE /api/administracion/restaurantes/{id}: Elimina un restaurante por completo (incluyendo sus archivos y horarios).

```java
@DeleteMapping("/{id}")
public ResponseEntity<?> eliminarRestaurante(
        @Parameter(description = "ID del restaurante a eliminar", example = "507f1f77bcf86cd799439011", required = true)
        @PathVariable String id) {
    try {
        // El método eliminarCompletamenteConArchivos() elimina físicamente el restaurante,
        // sus imágenes y horarios. Lanza IllegalArgumentException si no existe.
        restauranteService.eliminarCompletamenteConArchivos(id);

        Map<String, Object> response = new HashMap<>();
        response.put("ok", true);
        response.put("message", "Restaurante eliminado completamente, incluyendo imágenes y horarios");
        response.put("id", id);

        return ResponseEntity.ok(response);

    } catch (IllegalArgumentException e) {
        // El GlobalExceptionHandler manejará este error automáticamente
        throw e;
    } catch (Exception e) {
        Map<String, Object> error = new HashMap<>();
        error.put("error", Map.of(
            "code", "ERROR_INTERNO",
            "message", "No se pudo eliminar el restaurante. Error interno del servidor."
        ));
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(error);
    }
}
```

- Validación de Solicitudes:
  - GET /api/administracion/restaurantes/solicitudes: Lista solo los restaurantes en estado PENDIENTE que esperan aprobación.

```java
@GetMapping("/solicitudes")
public ResponseEntity<List<RestauranteAdminResponseDto>> solicitudes() {
    List<Restaurante> restaurantes = restauranteService.listarPorValidacion("PENDIENTE");
    List<RestauranteAdminResponseDto> response = restaurantes.stream()
        .map(this::convertirAAdminResponseDto)
        .collect(Collectors.toList());
    return ResponseEntity.ok(response);
}
```

  - POST /api/administracion/restaurantes/{id}/aprobar: Aprueba una solicitud, cambiando su estado a APROBADO y activándolo.

```java
@PostMapping("/{id}/aprobar")
public ResponseEntity<?> aprobar(
        @Parameter(description = "ID del restaurante a aprobar", example = "507f1f77bcf86cd799439011", required = true)
        @PathVariable String id) {
    try {
        restauranteService.cambiarValidacion(id, "APROBADO", true);
        return ResponseEntity.ok(Map.of("ok", true));
    } catch (IllegalArgumentException e) {
        // El GlobalExceptionHandler manejará este error
        throw e;
    }
}
```

- POST /api/administracion/restaurantes/{id}/rechazar: Rechaza una solicitud, cambiando su estado a RECHAZADO y permitiendo especificar un motivo.

```
@PostMapping("/{id}/rechazar")
public ResponseEntity<?> rechazar(
        @Parameter(description = "ID del restaurante a rechazar", example = "507f1f77bcf86cd799439011", required = true)
        @PathVariable String id,
        @io.swagger.v3.oas.annotations.parameters.RequestBody(
                description = "Motivo del rechazo",
                required = true,
                content = @io.swagger.v3.oas.annotations.media.Content(
                        mediaType = "application/json",
                        schema = @io.swagger.v3.oas.annotations.media.Schema(implementation = utp.restaurantesHorario.dto.RechazoRequestDto.class),
                        examples = @io.swagger.v3.oas.annotations.media.ExampleObject(
                                value = "{\"motivoRechazo\": \"Fotos insuficientes o de baja calidad\"}"
                        )
                )
        )
        @org.springframework.web.bind.annotation.RequestBody(required = false) utp.restaurantesHorario.dto.RechazoRequestDto request) {
    try {
        String motivo = (request != null && request.getMotivoRechazo() != null && !request.getMotivoRechazo().trim().isEmpty())
            ? request.getMotivoRechazo().trim()
            : "Solicitud rechazada por el administrador";

        // Obtener las fotos rechazadas del request (puede ser null o lista vacía)
        List<String> fotosRechazadas = (request != null && request.getFotosRechazadas() != null && !request.getFotosRechazadas().isEmpty())
            ? request.getFotosRechazadas()
            : null;

        restauranteService.cambiarValidacion(id, "RECHAZADO", false, motivo, fotosRechazadas);
        return ResponseEntity.ok(Map.of("ok", true));
    } catch (IllegalArgumentException e) {
        // El GlobalExceptionHandler manejará este error
        throw e;
    }
```

- Notificaciones (Admin):
  - GET /api/administracion/restaurantes/notificaciones: Obtiene todas las notificaciones.

```
@GetMapping("/notificaciones")
public ResponseEntity<List<utp.restaurantesHorario.model.Notificacion>> obtenerNotificaciones() {
    List<utp.restaurantesHorario.model.Notificacion> notificaciones = notificacionService.obtenerTodas();
    return ResponseEntity.ok(notificaciones);
}
```

  - GET /api/administracion/restaurantes/notificaciones/no-leidas: Obtiene solo las no leídas.

```
@GetMapping("/notificaciones/no-leidas")
public ResponseEntity<List<utp.restaurantesHorario.model.Notificacion>> obtenerNotificacionesNoLeidas() {
    List<utp.restaurantesHorario.model.Notificacion> notificaciones = notificacionService.obtenerNoLeidas();
    return ResponseEntity.ok(notificaciones);
}
```

  - GET /api/administracion/restaurantes/notificaciones/count: Cuenta las notificaciones no leídas.

```java
@GetMapping("/notificaciones/count")
public ResponseEntity<Map<String, Long>> contarNotificacionesNoLeidas() {
    long count = notificacionService.contarNoLeidas();
    return ResponseEntity.ok(Map.of("noLeidas", count));
}
```

- PUT /api/administracion/restaurantes/notificaciones/{id}/leer: Marca una notificación como leída.

```java
@PutMapping("/notificaciones/{id}/leer")
public ResponseEntity<utp.restaurantesHorario.model.Notificacion> marcarNotificacionComoLeida(
        @Parameter(description = "ID de la notificación", required = true)
        @PathVariable String id) {
    utp.restaurantesHorario.model.Notificacion notificacion = notificacionService.marcarComoLeida(id);
    return ResponseEntity.ok(notificacion);
}
```

- PUT /api/administracion/restaurantes/notificaciones/marcar-todas-leidas: Marca todas como leídas.

```java
@PutMapping("/notificaciones/marcar-todas-leidas")
public ResponseEntity<Map<String, String>> marcarTodasComoLeidas() {
    notificacionService.marcarTodasComoLeidas();
    return ResponseEntity.ok(Map.of("mensaje", "Todas las notificaciones han sido marcadas como leídas"));
}
```

3. Capa de DTOs (Data Transfer Objects)

El sistema se compone de varios Data Transfer Objects (DTOs) para gestionar la información de restaurantes, horarios y perfiles de usuario.

- DTOs de Solicitud (Requests)

  Para la creación de un restaurante, se utiliza el RestauranteRequestDto, que requiere el nombre, barrio, lat y lng (con validaciones de rango y obligatoriedad), e incluye campos opcionales como capacidad (con un mínimo de 0), contacto, y tipo.

```
@NotBlank(message = "El nombre es obligatorio")
private String nombre;

@NotBlank(message = "El barrio es obligatorio")
private String barrio;

@NotNull(message = "La latitud es obligatoria")
@DecimalMin(value = "-90.0", message = "Latitud inválida")
@DecimalMax(value = "90.0", message = "Latitud inválida")
private Double lat;

@NotNull(message = "La longitud es obligatoria")
@DecimalMin(value = "-180.0", message = "Longitud inválida")
@DecimalMax(value = "180.0", message = "Longitud inválida")
private Double lng;

@Min(value = 0, message = "Capacidad inválida")
private Integer capacidad;

private String contacto;

private String tipo;
```

  Para gestionar los horarios de atención, se usa el HorarioRequestDto, que requiere el diaSemana, horaApertura y horaCierre. Las horas deben cumplir con el formato obligatorio HH:MM.

```
@NotBlank(message = "El día de la semana es obligatorio")
private String diaSemana;

@NotBlank(message = "La hora de apertura es obligatoria")
@Pattern(regexp = "^([0-1]?[0-9]|2[0-3]):[0-5][0-9]$", message = "Formato de hora inválido (HH:MM)")
private String horaApertura;

@NotBlank(message = "La hora de cierre es obligatoria")
@Pattern(regexp = "^([0-1]?[0-9]|2[0-3]):[0-5][0-9]$", message = "Formato de hora inválido (HH:MM)")
private String horaCierre;
```

Para que el propietario actualice el estado del restaurante, se emplea el EstadoUpdateDto, que debe contener obligatoriamente el Boolean abierto e incluye un campo opcional nota.

```
@NotNull(message = "El estado abierto es obligatorio")
private Boolean abierto;

private String nota;
```

Para la actualización del perfil de usuario y el aumento de su nivel de seguridad, el ActualizarPerfilDto permite al propietario proporcionar datos como nombreCompleto, dni, ruc, telefono y direccion.

```
@Schema(description = "Nombre completo del propietario", example = "Juan Pérez García", required = false)
private String nombreCompleto;

@Schema(description = "DNI del propietario", example = "12345678", required = false)
private String dni;

@Schema(description = "RUC del negocio", example = "20612345678", required = false)
private String ruc;

@Schema(description = "Teléfono de contacto", example = "999888777", required = false)
private String telefono;

@Schema(description = "Dirección personal", example = "Av. Panamericana 123", required = false)
private String direccion;
```

Finalmente, el administrador utiliza el RechazoRequestDto para rechazar una solicitud de restaurante, especificando el motivoRechazo (obligatorio) y opcionalmente una lista de fotosRechazadas.

```
@Schema(
    description = "Motivo del rechazo de la solicitud",
    example = "Fotos insuficientes o de baja calidad",
    required = true
)
private String motivoRechazo;

/**
 * Lista de nombres de fotos/archivos que fueron rechazados específicamente.
 * El administrador puede seleccionar qué fotos no cumplen con los requisitos.
 * Si está vacío, se rechaza todo el restaurante en general.
 */
@Schema(
    description = "Lista de nombres de fotos específicas que fueron rechazadas",
    example = "[\"foto1.jpg\", \"foto2.jpg\"]",
    required = false
)
private List<String> fotosRechazadas;
```

- DTOs de Respuesta (Responses)

  Para las consultas públicas, se utiliza el RestauranteResponseDto, que expone el id, nombre, barrio, lat, lng, y el estado actual abierto. Este DTO también incluye una lista de horarios (HorarioResponseDto) y el nivelSeguridadPropietario (un entero del 1 al 10).

```
private String id;
private String nombre;
private String barrio;
private Double lat;
private Double lng;
private boolean abierto;
/**
 * Lista de horarios de atención del restaurante.
 * Solo incluye horarios activos ordenados por día de la semana.
 */
private List<HorarioResponseDto> horarios;
/**
 * Nivel de seguridad del propietario del restaurante (1-10).
 * Representa cuántos datos personales ha completado en su perfil.
 */
private Integer nivelSeguridadPropietario;
```

  El detalle del horario se gestiona con el HorarioResponseDto, que devuelve el id, diaSemana, horas (horaApertura, horaCierre), si está activo, y el restauranteId asociado.

```
private String id;
private String diaSemana;
private String horaApertura;
private String horaCierre;
private boolean activo;
private String restauranteId;
```

  Para la gestión administrativa, el RestauranteAdminResponseDto proporciona información más detallada del restaurante. Además de los campos públicos, incluye el estado activo (si está visible o no), el número de pruebasCount (archivos subidos para validación), el estadoValidacion (PENDIENTE, APROBADO, RECHAZADO), el motivoRechazo, la lista de fotosRechazadas y el estado cerradoPermanente.

```java
private String id;
private String nombre;
private String barrio;
private Double lat;
private Double lng;
private boolean abierto;
/**
 * Indica si el restaurante está activo o inactivo en el sistema.
 * true = activo (visible), false = inactivo (eliminado/oculto)
 */
private boolean activo;
/**
 * Número de archivos de prueba (fotos, documentos) subidos por el propietario
 * para validación administrativa.
 */
private Integer pruebasCount;
/**
 * Estado de validación del restaurante: PENDIENTE, APROBADO, RECHAZADO.
 * Los restaurantes con estado APROBADO y activo=true aparecen en el mapa público.
 */
```

```java
private String estadoValidacion;
/**
 * Motivo del rechazo de la solicitud, si fue rechazada.
 * Solo tiene valor cuando estadoValidacion == "RECHAZADO".
 */
private String motivoRechazo;
/**
 * Lista de nombres de fotos específicas que fueron rechazadas por el administrador.
 * Solo tiene valor cuando estadoValidacion == "RECHAZADO" y el admin especificó fotos.
 */
private List<String> fotosRechazadas;
/**
 * Indica si el restaurante fue cerrado permanentemente por el propietario.
 * true = cerrado permanentemente, false = no cerrado permanentemente
 */
private boolean cerradoPermanente;
```

Finalmente, el perfil del usuario se devuelve en el PerfilResponseDto, que contiene el id, email, la información personal opcional (como dni, ruc, telefono, direccion), el nombre del archivo de fotoPerfil, el calculado nivelSeguridad, y las marcas de tiempo createdAt y actualizadoEn.

```java
@Schema(description = "ID del usuario", example = "507f1f77bcf86cd799439011", required = true)
private String id;

@Schema(description = "Email del usuario", example = "usuario@ejemplo.com", required = true)
private String email;

@Schema(description = "Nombre completo", example = "Juan Pérez García", required = false)
private String nombreCompleto;

@Schema(description = "DNI del propietario", example = "12345678", required = false)
private String dni;

@Schema(description = "RUC del negocio", example = "20612345678", required = false)
private String ruc;

@Schema(description = "Teléfono de contacto", example = "999888777", required = false)
private String telefono;
```

```java
@Schema(description = "Nombre del archivo de foto de perfil", example = "foto_perfil_123.jpg", required = false)
private String fotoPerfil;

@Schema(description = "Dirección personal", example = "Av. Panamericana 123", required = false)
private String direccion;

@Schema(description = "Nivel de seguridad del 1 al 10", example = "7", required = true)
private int nivelSeguridad;

@Schema(description = "Fecha de creación de la cuenta", required = true)
private Instant createdAt;

@Schema(description = "Fecha de última actualización del perfil", required = false)
private Instant actualizadoEn;
```

4. Capa de Repositorios y Consultas Eficientes

Las interfaces de repositorio contienen consultas personalizadas (@Query) para optimizar el rendimiento de MongoDB.

- Búsqueda Cercana: RestauranteRepository.findByUbicacionCercana utiliza la sintaxis $near y $maxDistance en la entidad location para realizar búsquedas geográficas rápidas.

```java
@Query("{ 'location': { $near: { $geometry: { type: 'Point', coordinates: [?0, ?1] }, $maxDistance: ?2 } }, 'activo': true, 'estadoValidacion': 'APROBADO' }")
List<Restaurante> findByUbicacionCercana(double lng, double lat, double distanciaMetros);
```

- Búsqueda Flexible: findByNombreContainingIgnoreCase utiliza $regex con la opción 'i' para búsquedas parciales e insensibles a mayúsculas/minúsculas.

```java
@Query("{ 'nombre': { $regex: ?0, $options: 'i' }, 'activo': true, 'estadoValidacion': 'APROBADO' }")
List<Restaurante> findByNombreContainingIgnoreCase(String nombre);
```

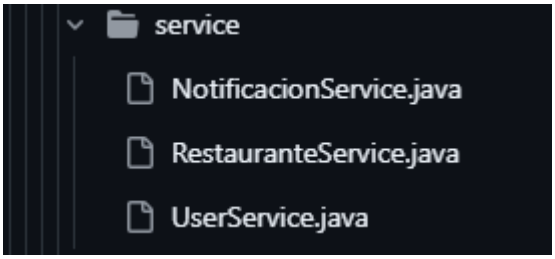
- Acceso al Propietario: UserRepository.findByEmail es utilizada por el AuthController para el login y por el JwtAuthFilter para cargar la identidad del usuario.

```java
@Repository
public interface UserRepository extends MongoRepository<User, String> {

    /**
     * Busca un usuario por su email.
     *
     * @param email Email del usuario a buscar
     * @return Optional con el usuario si existe, vacío en caso contrario
     */
    Optional<User> findByEmail(String email);
```

5. Capa de Servicios (Service)

La capa de Servicios es el corazón de nuestro sistema, donde se implementan todas las reglas de negocio y las transacciones críticas. A continuación, se destacarán las funciones más relevantes de la capa de Servicios



5.1. Servicio de Restaurantes (RestaurantService.java)

5.1.1. Validación Geográfica

```java
private boolean estaEnPiura(double lat, double lng) {
    return lat >= PIURA_LAT_MIN && lat <= PIURA_LAT_MAX &&
           lng >= PIURA_LNG_MIN && lng <= PIURA_LNG_MAX;
}
```

El siguiente método comprueba si las coordenadas caen dentro de los límites fijos. Como se pude ver a continuación.

```java
// Límites geográficos de Piura (aproximados)
private static final double PIURA_LAT_MIN = -5.3;
private static final double PIURA_LAT_MAX = -5.0;
private static final double PIURA_LNG_MIN = -80.7;
private static final double PIURA_LNG_MAX = -80.5;
```

Si la validación falla en el método de “CrearSolicitud()”, se lanza la excepción “FUERA_DE_PIURA”

```
@Transactional
public Restaurante crearSolicitud(RestauranteRequestDto request, String ownerUserId) {
    if (!estaEnPiura(request.getLat(), request.getLng())) {
        throw new IllegalArgumentException("FUERA_DE_PIURA");
    }
```

5.1.2. Apelar Rechazo:

```
@Transactional
public Restaurante apelarRechazo(String ownerUserId, String restauranteId, RestauranteRequestDto dto) {
    // Buscar el restaurante por ID
    Restaurante r = restauranteRepository.findById(restauranteId)
        .orElseThrow(() -> new IllegalArgumentException("RESTAURANTE_NO_ENCONTRADO"));

    // Validar que el usuario autenticado es el propietario
    if (!ownerUserId.equals(r.getOwnerUserId())) {
        throw new IllegalArgumentException("PROHIBIDO");
    }

    // Validar que el restaurante está rechazado
    if (!"RECHAZADO".equals(r.getEstadoValidacion())) {
        throw new IllegalArgumentException("RESTAURANTE_NO_RECHAZADO");
    }

    // Validar que no hayan pasado más de 3 días desde el rechazo
    if (r.getFechaRechazo() != null) {
        Instant ahora = Instant.now();
        long diasTranscurridos = java.time.Duration.between(r.getFechaRechazo(), ahora).toDays();
        if (diasTranscurridos > 3) {
            throw new IllegalArgumentException("PLAZO_APELACION_VENCIDO");
        }
    }
```

El método “apelarRechazo()” verifica si la duración entre “fechaRechazo” e “Instant.now()” excede los 3 días.

```
// Guardar la fecha del rechazo para poder calcular el plazo de apelación
r.setFechaRechazo(Instant.now());
```

```
    // Validar que la nueva ubicación esté dentro de Piura
    if (!estaEnPiura(dto.getLat(), dto.getLng())) {
        throw new IllegalArgumentException("FUERA_DE_PIURA");
    }

    // Actualizar los datos del restaurante
    r.setNombre(dto.getNombre());
    r.setBarrio(dto.getBarrio());
    r.setLocation(new GeoJsonPoint(dto.getLng(), dto.getLat()));
    r.setCapacidad(dto.getCapacidad() != null ? dto.getCapacidad() : r.getCapacidad());
    r.setContacto(dto.getContacto());
    r.setTipo(dto.getTipo());

    // Cambiar el estado a PENDIENTE para que vuelva a ser revisado
    r.setEstadoValidacion("PENDIENTE");
    r.setActivo(false);

    // Limpiar el motivo de rechazo y la fecha (nueva solicitud)
    r.setMotivoRechazo(null);
    r.setFechaRechazo(null);
    r.setFotosRechazadas(null);

    // Las fotos se subirán por separado usando el endpoint de subirPruebas

    return restauranteRepository.save(r);
}
```

Si excede, lanza PLAZO_APELACION_VENCIDO. Si es exitoso, actualiza el estado a PENDIENTE y limpia los campos de rechazo.

5.1.3. Horario Automaticos

```java
public void actualizarEstadosPorHorarios(String diaSemana, String horaActual) {
    // Obtener todos los horarios activos del día actual
    List<Horario> horariosHoy = horarioRepository.findByDiaSemanaAndActivoTrue(diaSemana);

    if (horariosHoy.isEmpty()) {
        return; // No hay horarios configurados para hoy
    }

    // Procesar cada horario
    for (Horario horario : horariosHoy) {
        Restaurante restaurante = horario.getRestaurante();

        // Solo actualizar si el restaurante está aprobado
        if (!"APROBADO".equals(restaurante.getEstadoValidacion())) {
            continue;
        }

        boolean deberiaEstarAbierto = estaDentroDelHorario(horario.getHoraApertura(), horario.getHoraCierre(), horaActual);
        Estado estadoActual = restaurante.getEstado();

        // Solo actualizar si el estado debe cambiar
        if (estadoActual == null || estadoActual.isAbierto() != deberiaEstarAbierto) {
            // Actualizar el estado
            if (estadoActual == null) {
                estadoActual = new Estado();
            }
            estadoActual.setAbierto(deberiaEstarAbierto);
            if (deberiaEstarAbierto) {
                estadoActual.setNota("Abierto automáticamente por horario");
            } else {
                estadoActual.setNota("Cerrado automáticamente por horario");
            }
            restaurante.setEstado(estadoActual);
            restauranteRepository.save(restaurante);
        }
    }
}
```

El método “actualizarEstadosPorHorarios()” se ejecuta cada minuto, llamando al método “estaDentroDelHorario()”.

```java
boolean deberiaEstarAbierto = estaDentroDelHorario(horario.getHoraApertura(), horario.getHoraCierre(), horaActual);
Estado estadoActual = restaurante.getEstado();
```

Este método contiene la lógica para manejar horarios que cruzan la medianoche (ej., 22:00 a 02:00) mediante la conversión de horas a minutos para una comparación simple y precisa.

5.1.4. Eliminación Total

```java
    @Transactional
    public void eliminarCompletamenteConArchivos(String id) {
        Restaurante restaurante = restauranteRepository.findById(id)
            .orElseThrow(() -> new IllegalArgumentException("RESTAURANTE_NO_ENCONTRADO"));

        // Eliminar la carpeta de imágenes
        eliminarCarpetaImagenes(restaurante);

        // Eliminar todos los horarios asociados
        horarioRepository.deleteByRestauranteId(id);

        // Eliminar físicamente de la base de datos (no soft delete)
        restauranteRepository.delete(restaurante);

        System.out.println("✅ Restaurante eliminado completamente por admin: " + id);
    }
}
```

El método eliminarCompletamenteConArchivos() es @Transactional. Primero llama al método privado eliminarCarpetaImagenes().

```java
private void eliminarCarpetaImagenes(Restaurante restaurante) {
    if (restaurante.getPruebasFolder() != null && !restaurante.getPruebasFolder().isEmpty()) {
        try {
            Path folderPath = Paths.get(restaurante.getPruebasFolder());
            if (Files.exists(folderPath) && Files.isDirectory(folderPath)) {
                // Eliminar recursivamente todos los archivos y subdirectorios
                Files.walk(folderPath)
                    .sorted((a, b) -> b.compareTo(a)) // Orden inverso: archivos primero, luego directorios
                    .forEach(path -> {
                        try {
                            Files.delete(path);
                        } catch (IOException e) {
                            System.err.println("Error al eliminar archivo/carpeta: " + path + " - " + e.getMessage());
                        }
                    });
                System.out.println("✅ Carpeta de imágenes eliminada: " + folderPath);
            }
        } catch (IOException e) {
            System.err.println("Error al eliminar carpeta de imágenes para restaurante " + restaurante.getId() + ": " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Para borrar los archivos físicos del disco y luego elimina el documento de MongoDB y los horarios asociados.

5.2. Servicio de Usuario (UserService.java)

5.2.1. Nivel de Seguridad

El método calcularNivelSeguridad() es el core de la confianza. Se inicia con una Base de 1 y se otorgan puntos: +1 por campos básicos (DNI, Teléfono), y +3 por documentos importantes (RUC y Foto de Perfil). El nivel resultante se ajusta al rango 1-10.

```
private int calcularNivelSeguridad(User user) {
    int nivel = 1; // Base

    // Campos básicos (+1 cada uno)
    if (user.getNombreCompleto() != null && !user.getNombreCompleto().trim().isEmpty()) {
        nivel++;
    }
    if (user.getDni() != null && !user.getDni().trim().isEmpty()) {
        nivel++;
    }
    if (user.getTelefono() != null && !user.getTelefono().trim().isEmpty()) {
        nivel++;
    }
    if (user.getDireccion() != null && !user.getDireccion().trim().isEmpty()) {
        nivel++;
    }

    // Documentos importantes (+3 cada uno)
    if (user.getRuc() != null && !user.getRuc().trim().isEmpty()) {
        nivel += 3;
    }
    if (user.getFotoPerfil() != null && !user.getFotoPerfil().trim().isEmpty()) {
        nivel += 3;
    }

    // Asegurar que esté entre 1 y 10
    return Math.min(Math.max(nivel, 1), 10);
}
```

5.2.2. Actualización de Perfil

```java
public User actualizarPerfil(String userId, ActualizarPerfilDto dto) {
    // Buscar el usuario
    Optional<User> userOpt = userRepository.findById(userId);
    if (userOpt.isEmpty()) {
        throw new IllegalArgumentException("USUARIO_NO_ENCONTRADO");
    }

    User user = userOpt.get();

    // Actualizar campos solo si se proporcionan valores
    if (dto.getNombreCompleto() != null && !dto.getNombreCompleto().trim().isEmpty()) {
        user.setNombreCompleto(dto.getNombreCompleto().trim());
    }
    if (dto.getDni() != null && !dto.getDni().trim().isEmpty()) {
        user.setDni(dto.getDni().trim());
    }
    if (dto.getRuc() != null && !dto.getRuc().trim().isEmpty()) {
        user.setRuc(dto.getRuc().trim());
    }
    if (dto.getTelefono() != null && !dto.getTelefono().trim().isEmpty()) {
        user.setTelefono(dto.getTelefono().trim());
    }
    if (dto.getDireccion() != null && !dto.getDireccion().trim().isEmpty()) {
        user.setDireccion(dto.getDireccion().trim());
    }

    // Calcular nivel de seguridad basado en datos completados
    int nivel = calcularNivelSeguridad(user);
    user.setNivelSeguridad(nivel);

    // Actualizar timestamp
    user.setActualizadoEn(Instant.now());

    // Guardar y retornar
    return userRepository.save(user);
}
```

Este metodo se encarga de buscar al usuario por su ID, y si no existe lanza un error, en caso de que el usuario exista, va a actualizar solo los campos enviados en el DTO (nombre, DNI, RUC, teléfono, dirección), ignorando valores null o vacíos. Va a recalcular su nivel de seguridad con la nueva informacion y a actualizar la fecha de modificacion, además de que va a guardar y deolver el usuario actualiazdo en la base de datos.

6. Propiedades de la aplicación (aplication.properties)

Se centra en la configuración. Se define la conexión a la base de datos **MongoDB** y mongodb://localhost:27017/restaurantes_piuraestablezca que la aplicación se ejecutará en el puerto 3000. Además, configura la seguridad con **JWT**, especificando la clave secreta y un tiempo de vida del token de 15 minutos, e incluye ajustes para la carga de archivos, limitando el tamaño a 5 MB por archivo, y la inicialización de un usuario administrador (admin@restaurantes.local) al arrancar el sistema.

```
spring.application.name=restaurantesHorario

# MongoDB Connection
spring.data.mongodb.uri=mongodb://localhost:27017/restaurantes_piura

# Server Configuration
server.port=3000

# Logging Configuration
logging.level.utp.restaurantesHorario=DEBUG

# JWT Configuration (modificar en despliegue)
# vscode-spring-boot: @utp.restaurantesHorario.security.JwtService
jwt.secret=CAMBIA_ESTA_CLAVE_LARGA_Y_SEGURA
jwt.ttlMillis=900000

# File Upload Configuration (pruebas de restaurantes)
# vscode-spring-boot: @utp.restaurantesHorario.controller.PropietarioController
upload.base-dir=uploads/restaurantes
spring.servlet.multipart.max-file-size=5MB
spring.servlet.multipart.max-request-size=25MB

# Admin Seed Configuration (crear al arrancar si no existe)
# vscode-spring-boot: @utp.restaurantesHorario.config.DataInitializer
admin.email=admin@restaurantes.local
admin.password=Admin123!
admin.enabled=true
```

VI. Referencia de Endpoints y Guía Práctica de Pruebas

A. Referencia de Endpoints Clave

Rol/Servicio	Función	Método HTTP	Endpoint (Ruta)	Descripción
Público (restauranteService)	obtenerRestaurantes	**GET**	/api/restaurantes/obtener	Obtiene la lista de todos los restaurantes que están **aprobados y activos** para mostrar en el mapa público.

Rol/Servicio	Función	Método HTTP	Endpoint (Ruta)	Descripción
Público (restauranteService)	actualizarEstado	**POST**	/{token}/estado	(Legacy) Permite actualizar el estado (abierto/cerrado) usando un **token único** del restaurante, sin necesidad de login JWT.
Admin (adminService)	obtenerRestaurantesAdmin	**GET**	/api/administracion/restaurantes	Obtiene **TODOS** los restaurantes del sistema (pendientes, aprobados, rechazados, etc.).
Admin (adminService)	crearRestaurante	**POST**	/api/administracion/restaurantes	Crea un nuevo restaurante directamente, asignándole el estado **APROBADO** de forma automática.
Admin (adminService)	eliminarRestaurante	**DELETE**	/api/administracion/restaurantes/{id}	Realiza un "soft delete" (borrado lógico), marcando el restaurante como inactivo.

Rol/Servicio	Función	Método HTTP	Endpoint (Ruta)	Descripción
Admin (adminService)	obtenerSolicitudes	**GET**	/api/administracion/restaurantes/solicitudes	Obtiene solo los restaurantes que están en estado **PENDIENTE** de aprobación.
Admin (adminService)	aprobar	**POST**	/api/administracion/restaurantes/{id}/aprobar	Cambia el estado de un restaurante de PENDIENTE a APROBADO.
Admin (adminService)	rechazar	**POST**	/api/administracion/restaurantes/{id}/rechazar	Cambia el estado de PENDIENTE a RECHAZADO y guarda un motivo.
Admin (adminService)	verPruebas	**GET**	/api/administracion/restaurantes/{id}/pruebas	Obtiene la lista de archivos (fotos) de prueba que subió un propietario.
Admin (adminService)	obtenerNotificaciones	**GET**	/api/administracion/restaurantes/notificaciones	Obtiene todas las notificaciones del administrador.
Admin (adminService)	obtenerNotificacionesNoLeidas	**GET**	/api/administracion/restaurantes/notificaciones/no-leidas	Obtiene solo las notificaciones no leídas.

Rol/Servicio	Función	Método HTTP	Endpoint (Ruta)	Descripción
Admin (adminService)	contarNotificacionesNoLeidas	**GET**	/api/administracion/restaurantes/notificaciones/count	Devuelve el número de notificaciones no leídas.
Admin (adminService)	marcarNotificacionComoLeida	**PUT**	/api/administracion/restaurantes/notificaciones/{id}/leer	Marca una notificación específica como leída.
Admin (adminService)	marcarTodasComoLeidas	**PUT**	/api/administracion/restaurantes/notificaciones/marcar-todas-leidas	Marca todas las notificaciones del admin como leídas.
Propietario (ownerService)	registrarRestaurante	**POST**	/api/mis-restaurantes	Crea una **nueva solicitud** de restaurante. Queda en estado PENDIENTE.
Propietario (ownerService)	misRestaurantes	**GET**	/api/mis-restaurantes	Obtiene todos los restaurantes que pertenecen al usuario autenticado (pendientes, aprobados o rechazados).

Rol/Servicio	Función	Método HTTP	Endpoint (Ruta)	Descripción
Propietario (ownerService)	actualizarEstado	**PUT**	/api/mis-restaurantes/{id}/estado	Actualiza el estado (abierto/cerrado) de un restaurante **aprobado**.
Propietario (ownerService)	actualizarHorarios	**PUT**	/api/mis-restaurantes/{id}/horarios	Actualiza los horarios de atención de un restaurante.
Propietario (ownerService)	subirPruebas	**POST**	/api/mis-restaurantes/{id}/pruebas	Sube archivos (FormData) como prueba (fotos, documentos) para la validación.
Propietario (ownerService)	apelarRestaurante	**PUT**	/api/mis-restaurantes/{id}/apelar	Permite a un propietario apelar un restaurante RECHAZADO, volviéndolo a PENDIENTE con datos actualizados.
Propietario (ownerService)	cerrarPermanentemente	**PUT**	/api/mis-restaurantes/{id}/cerrar-permanentemente	Marca un restaurante como cerrado permanentemente (para borrado automático).

Rol/Servicio	Función	Método HTTP	Endpoint (Ruta)	Descripción
Propietario (ownerService)	verPruebas	**GET**	/api/mis-restaurantes/{id}/pruebas	Obtiene la lista de archivos de prueba que el propietario subió para su restaurante.
Propietario (ownerService)	obtenerMiPerfil	**GET**	/api/mis-restaurantes/perfil	Obtiene la información del perfil del usuario autenticado.
Propietario (ownerService)	actualizarMiPerfil	**PUT**	/api/mis-restaurantes/perfil	Actualiza los datos del perfil (nombre, DNI, RUC, etc.) del usuario.
Propietario (ownerService)	subirFotoPerfil	**POST**	/api/mis-restaurantes/perfil/foto	Sube un archivo (FormData) para la foto de perfil del usuario.