UNIVERSIDAD INTERNACIONAL SAN ISIDRO LABRADOR

(ISB-32) PROGRAMACIÓN AVANZADA

TRABAJO ESCRITO

Propuesta para automatizar la gestión de reservaciones en restaurantes mediante una solución web moderna y escalable.

POR:

Javier Alonso Gallo Marchena

PROFESORA:

ESTEFANIA BOZA VILLALOBOS

FECHA:

12 de diciembre 2025

1. **Introducción**

   Este proyecto propone un sistema web para gestionar reservaciones en un restaurante. La idea es que los administradores y empleados puedan organizar clientes, mesas y menús de forma sencilla, sin depender de apuntes en papel o de la memoria.

   Se usará Spring Boot en el backend, Angular en el frontend y PostgreSQL como base de datos. Estas tecnologías permiten crear un sistema seguro, ordenado y fácil de ampliar en el futuro.

2. **Objetivo General**

   Desarrollar un sistema web que facilite la administración de reservaciones, clientes, mesas, menús y usuarios, con una interfaz clara y fácil de usar.

2.1 **Objetivos Específicos**

- Crear una base de datos relacional que guarde toda la información de forma estructurada.
- Implementar el backend en Spring Boot con operaciones CRUD para cada módulo.
- Hacer un frontend en Angular con pantallas intuitivas según el rol (administrador o empleado).
- Configurar roles para que el administrador tenga acceso completo y los empleados permisos limitados.
- Incorporar un panel de reportes con información útil como mesas más usadas o menús más pedidos.

3. **Justificación**

   Muchos restaurantes todavía usan métodos manuales para las reservaciones, lo que genera errores: mesas duplicadas, clientes olvidados o confusiones de horarios. Esto afecta tanto a empleados como a clientes.

   Este sistema centraliza toda la información en un solo lugar. Un empleado puede registrar clientes y reservaciones rápidamente, y el administrador puede ver al final del día la ocupación de mesas o los menús más

solicitados. Además, al usar tecnologías modernas, se podrá agregar más funciones en el futuro.

4. **Alcances Esperados**

    - Login con usuario y contraseña.
    - Roles diferenciados: administrador y empleado.
    - Registro, modificación y consulta de clientes, mesas, menús y reservaciones.
    - Estados de reservación: pendiente, confirmada, cancelada.
    - Dashboard con estadísticas y gráficas para el administrador.

    No incluirá por ahora:

    - Reservaciones hechas directamente por los clientes.
    - Pagos en línea.
    - Aplicación móvil nativa.

5. **Requerimientos del Proyecto**

    Funcionales

    - Login de usuarios.
    - Permisos según rol.
    - CRUD de clientes, mesas, menús y reservaciones.
    - Manejo de estados de reservaciones.
    - Reportes con tablas y gráficas.

    No Funcionales

    - Seguridad: contraseñas encriptadas y autenticación con tokens JWT.
    - Rendimiento: respuestas en menos de dos segundos.
    - Escalabilidad: arquitectura en capas para agregar módulos después.

- Usabilidad: interfaz clara y simple, optimizada para escritorio y móviles.

PARTE II: DISEÑO DEL SISTEMA (PRE-DISEÑO VISUAL)

1. Información General del Proyecto

Elemento	Descripción
Nombre del Sistema	RestaurantePro - Sistema de Reservaciones
Slogan	"Gestiona tus reservaciones de manera más fácil y segura"
Descripción General	Sistema web de gestión integral desarrollado con **Angular** y **Spring Boot** para el *backend*. La persistencia de datos utiliza **PostgreSQL**, garantizando un manejo robusto de reservaciones, clientes, mesas y menús.

---

2. Elementos de Diseño Visual

2.1 Paleta de Colores

El diseño utiliza una paleta base de **azules y grises claros**, enfocada en la modernidad y la claridad.

Color	Código HEX	Uso
Azul Principal	#3B82F6	Botones primarios, enlaces activos, elementos interactivos.
Verde Éxito	#22C55E	Estados **Confirmados** y **Disponibles**.
Rojo Error	#EF4444	Estados **Ocupados**, **Cancelados** o **Eliminación**.

2.2 Tipografía e Iconografía

- **Fuente Principal: Inter** (o alternativa compatible como Roboto/Segoe UI), cumpliendo el requisito de legibilidad.

- **Iconografía:** Se utiliza iconografía sencilla para módulos clave: **Dashboard** 📊, **Reservaciones** 🗓️, **Menús** 🍽️, **Mesas** 🪑, y **Usuarios** 👤.

---

3. Estructura de Pantallas (Mockups y Bocetos)

El diseño utiliza un patrón de **Dashboard** con un Sidebar (Menú Lateral) y un área de contenido principal *responsive*.

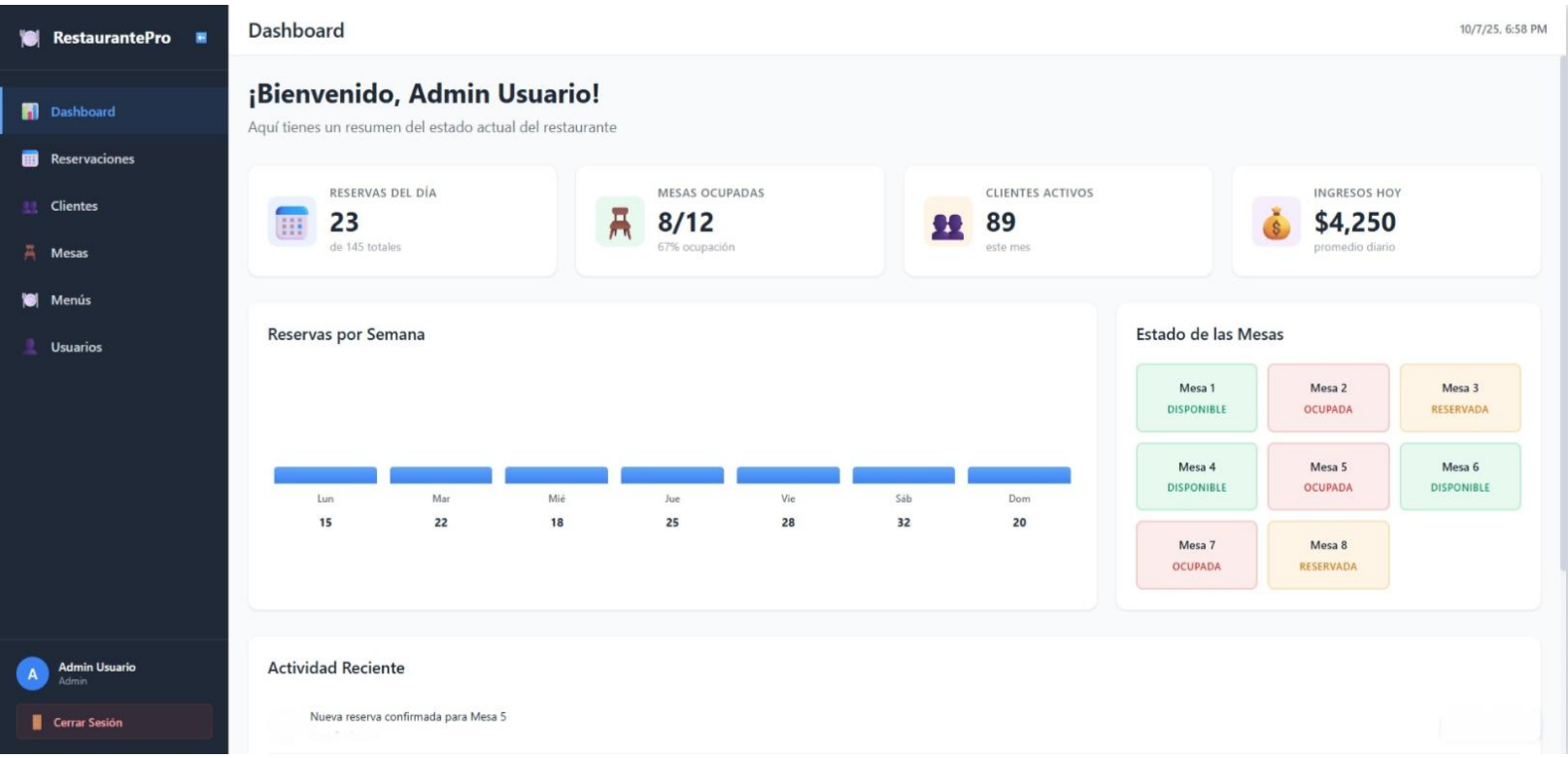
3.1 Pantalla de Login

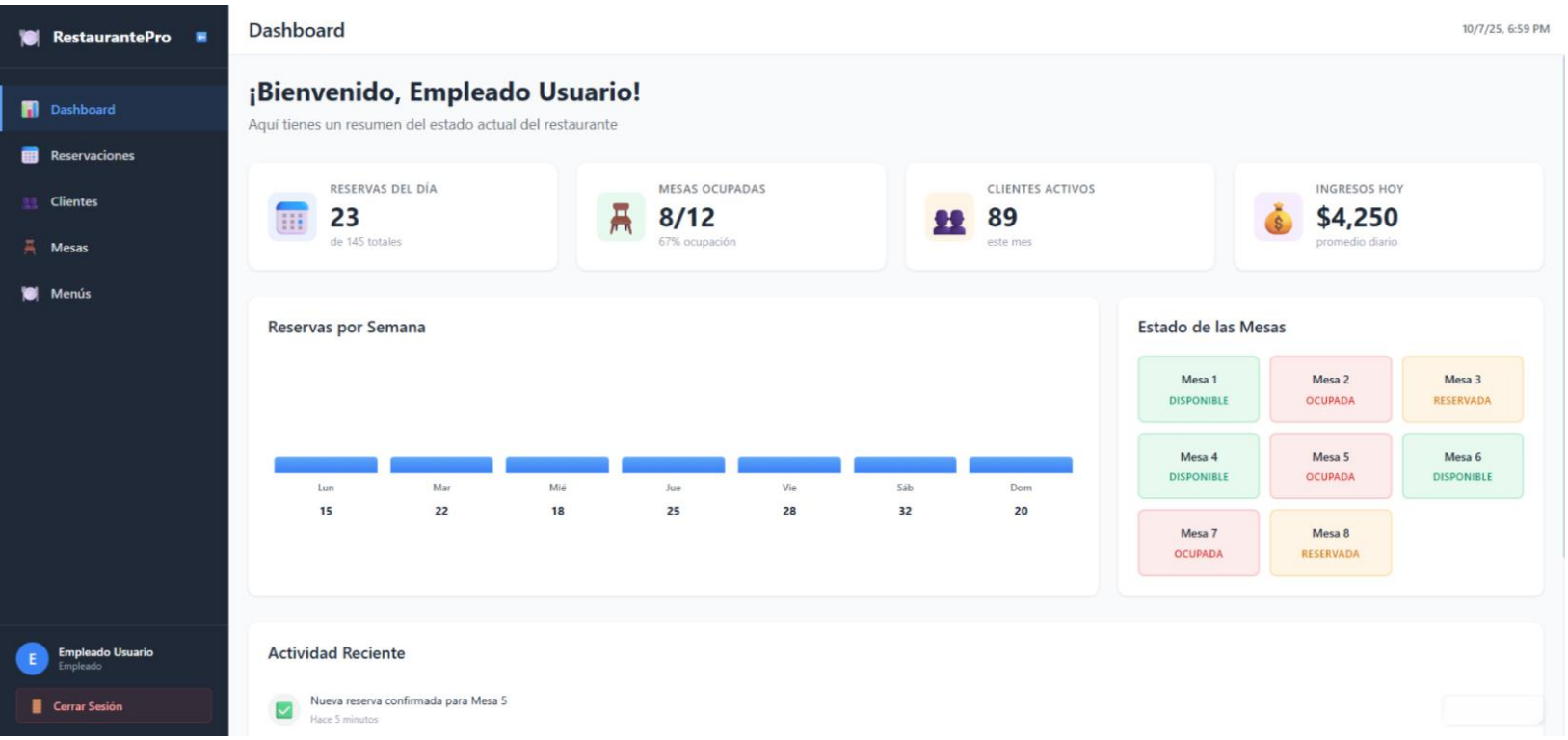
- **Propósito:** Autenticación inicial del usuario.
- **Elementos:** Tarjeta blanca centrada y formulario para *email* y *contraseña*.

3.2 Panel Principal (Dashboard)

- **Propósito:** Resumen ejecutivo de las actividades y estado del restaurante.

- **Elementos: Tarjetas de Estadísticas** (Reservas del Día, Mesas Ocupadas, etc.) y un **Gráfico de Barras** (Reservaciones por Semana).





3.3 Gestión de Reservaciones

- **Propósito:** CRUD de todas las reservaciones.
- **Elementos: Tabla de Datos** con **Badges de Estado** para **Confirmada** (verde), **Pendiente** (amarillo) y **Cancelada** (rojo).

3.4 Gestión de Mesas

- **Propósito:** Administrar la capacidad y disponibilidad de las mesas.
- **Elementos:** Vista de **Tarjetas** o **Tabla** que muestra el estado de las mesas (**Disponible, Ocupada, Reservada**).

3.5 Gestión de Menús

- **Propósito:** Administrar el catálogo de platillos y bebidas.
- **Elementos: Tabla de Datos** con columnas para Nombre, Descripción y **Precio** con formato de moneda.

3.6 Gestión de Usuarios

- **Propósito:** Administrar roles y credenciales del sistema (Solo **Administradores**).
- **Elementos:** Tabla con columnas Nombre, Email y **Rol** (Badges de color diferenciado).

---

4. Arquitectura Técnica y Seguridad Corregida

4.1 Tecnologías Centrales

- **Backend: Spring Boot** (Java), implementando una arquitectura de capas (Controladores, Servicios, Repositorios) para alta escalabilidad y rendimiento.
- **Base de Datos: PostgreSQL** (Relacional), garantizando la integridad de los datos.
- **Frontend: Angular 20** con **Reactive Forms** para validación avanzada.

4.2 Seguridad y Control de Acceso

- **Autenticación:** Implementada en **Spring Boot** a través de **Tokens JWT**.
- **Control de Roles:** La interfaz está diseñada para implementar los roles de **Administrador** y **Empleado**; el módulo de /usuarios solo será accesible por el Administrador, implementando la lógica de autorización en el *backend* de Spring Boot y en el *frontend* con **Angular Guards**.

5. Conclusiones del Pre-Diseño

Este pre-diseño visual es una guía estructural que cumple con todos los requerimientos funcionales del Avance 1. El diseño se alinea con la solidez de la arquitectura **Angular + Spring Boot + PostgreSQL**, garantizando una aplicación moderna, segura y lista para el desarrollo.

ANEXOS

A. Credenciales de Prueba

Administrador:

- Email: admin@restaurant.com

- Contraseña: admin123

Empleado:

- Email: empleado@restaurant.com
- Contraseña: empleado123

B. Rutas de la Aplicación

- /login - Pantalla de autenticación
- /dashboard - Panel principal (requiere auth)
- /reservaciones, /clientes, /mesas, /admin/menus - Requieren autenticación.
- /usuarios - Gestión de usuarios (**Solo Admin**).

1. Definición de la Base de Datos

1.1. Tipo y Propósito

La base de datos del proyecto "RestaurantePro - Sistema de Reservaciones" es de tipo **Relacional**.

Su propósito fundamental es centralizar, organizar y garantizar la **integridad transaccional** de toda la información crítica para la operación del restaurante, incluyendo la gestión de usuarios, clientes, mesas, el catálogo de platos (menús) y, principalmente, los datos detallados de las reservaciones.

1.2. Motor de Base de Datos

El motor de base de datos elegido es **PostgreSQL**.

Justificación: PostgreSQL es un sistema de gestión de bases de datos objeto-relacional (ORDBMS) conocido por su **robustez, alta concurrencia,** y cumplimiento estricto de estándares SQL. Esta elección se alinea con el requisito de construir un sistema escalable y seguro, al ser una base de datos de grado empresarial ideal para aplicaciones críticas.

2. Elementos de Diseño: Modelo Entidad-Relación (MER)

El diseño de la base de datos se estructura alrededor de 7 entidades principales (tablas), que capturan los requerimientos funcionales del sistema (CRUD de Clientes, Mesas, Platos, Reservaciones y Usuarios con Roles).

Diagrama Entidad-Relación (ERD)

A continuación, se presenta una representación visual simplificada del Modelo Entidad-Relación, reflejando las tablas y las relaciones clave implementadas en PostgreSQL.

2.1. Descripción de Entidades y Atributos

La siguiente tabla detalla cada entidad, sus atributos, el tipo de dato utilizado en PostgreSQL y la justificación de las llaves.

2.1. Descripción de Entidades y Atributos

Entidad	Atributo	Tipo de Dato (PostgreSQL)	Llave	Descripción y Justificación
CLIENTE	cliente_id	BIGINT	PK	Identificador único y autonumérico del cliente.
	nombre	VARCHAR(100)		Nombre completo del cliente.
	telefono	VARCHAR(15)		Número de contacto. Permite formatos internacionales.
	email	VARCHAR(100)	Unique	Correo electrónico del cliente. Se usa como clave única.
	direccion	VARCHAR(255)		Domicilio registrado.
MESA	mesa_id	BIGINT	PK	Identificador único y autonumérico.
	numero_mesa	INTEGER	Unique	Número de la mesa física. Es la identificación principal para el usuario.
	capacidad	INTEGER		Número máximo de personas que pueden ocupar la mesa.
	estado	VARCHAR(20)		Estado actual: ‘Disponible’, ‘Ocupada’, ‘Reservada’.
PLATO	plato_id	BIGINT	PK	Identificador único del platillo (ítem del menú).
	nombre	VARCHAR(100)	Unique	Nombre del platillo. Clave única.
	descripcion	VARCHAR(255)		Breve descripción del platillo.

	precio	DECIMAL(10,2)		Precio de venta. Precisión de dos decimales.
	disponible	BOOLEAN		Indica si el plato está activo/visible en el menú.
ROL	rol_id	BIGINT	PK	Identificador único del rol de usuario.
	nombre_rol	VARCHAR(50)	Unique	Nombre descriptivo del rol (ej: 'ADMIN', 'EMPLEADO').
USUARIO	id	BIGINT	PK	Identificador único de la cuenta de acceso.
	username	VARCHAR(50)	Unique	Nombre de usuario o email de acceso. Clave única.
	password	VARCHAR(255)		Contraseña cifrada (hash).
RESERVA	reserva_id	BIGINT	PK	Identificador único de la reservación.
	fecha_hora_inicio	TIMESTAMP		Fecha y hora exactas de inicio.
	duracion_minutos	INTEGER		Duración prevista de la reserva.
	cantidad_personas	INTEGER		Número de personas asistentes.
	estado	VARCHAR(20)		Estado: 'PENDIENTE', 'CONFIRMADA', 'CANCELADA'.
	nota	TEXT		Notas o solicitudes del cliente.
	fecha_confirmacion	TIMESTAMP		Fecha y hora de confirmación.
	fecha_fin_real	TIMESTAMP		Hora final real de ocupación (opcional).
	cliente_id	BIGINT	FK	Referencia al cliente que realiza la reserva.
	mesa_id	BIGINT	FK	Referencia a la mesa asignada.
USUARIO_ROL	usuario_id	BIGINT	PK, FK	Clave foránea a USUARIO. Parte de la PK compuesta.
	rol_id	BIGINT	PK, FK	Clave foránea a ROL. Parte de la PK compuesta.

2.2. Relaciones entre Entidades

Relación	Entidades Involucradas	Cardinalidad	Llaves Foráneas	Justificación
Realiza	CLIENTE - RESERVA	1 a N	RESERVA.cliente_id	Un cliente puede realizar muchas reservas (N), pero cada reserva pertenece a un único cliente (1).
Asigna	MESA - RESERVA	1 a N	RESERVA.mesa_id	Una mesa puede tener muchas reservas asociadas, pero una reserva en particular solo está asignada a una mesa.
Tiene Rol	USUARIO - ROL	N a M	USUARIO_ROL (usuario_id, rol_id)	Un usuario puede tener múltiples roles, y un rol puede ser asignado a múltiples usuarios. Esta relación muchos-a-muchos se resuelve con la tabla de unión USUARIO_ROL.

3. Normalización hasta Tercera Forma Normal (3FN)

El diseño de la base de datos está construido para cumplir con las tres primeras formas normales (1FN, 2FN y 3FN), lo cual es fundamental para **eliminar la redundancia, evitar anomalías en la actualización y asegurar la consistencia** de los datos.

A. Primera Forma Normal (1FN)

- **Verificación:** Todas las tablas aseguran que cada atributo contiene un **valor atómico** (no divisible) y que no existen grupos repetitivos de datos.
- **Conclusión:** Se cumple 1FN.

B. Segunda Forma Normal (2FN)

- **Verificación:** Se aplica a tablas con **claves primarias compuestas**. La única tabla de unión es USUARIO_ROL. Como no tiene atributos adicionales a sus claves foráneas, todos los atributos dependen completamente de la clave compuesta (usuario_id, rol_id).
- **Conclusión:** Se cumple 2FN.

C. Tercera Forma Normal (3FN)

- **Verificación:** Se revisa que no existan **dependencias transitivas**, es decir, que ningún atributo no clave dependa de otro atributo no clave.
    - **Ejemplo:** En la tabla CLIENTE, nombre, telefono, email y direccion dependen directamente de la llave primaria cliente_id. No se almacena información redundante (ej. el nombre del cliente no se repite en RESERVA).
    - **Ejemplo:** La información de los roles se ha separado en la tabla ROL, y solo se referencia su clave rol_id en la tabla de unión.
- **Conclusión:** Se cumple 3FN.

4. Conexión con la Aplicación

4.1. Tecnología de Conexión

La conexión entre el *backend* desarrollado en **Spring Boot** y la base de datos **PostgreSQL** se realiza utilizando:

1. **JDBC Driver** (Java Database Connectivity) para la comunicación de bajo nivel.
2. **Spring Data JPA (Java Persistence API)**, que utiliza **Hibernate** como proveedor de persistencia para mapear las clases Java (Entidades) directamente a las tablas de la base de datos.

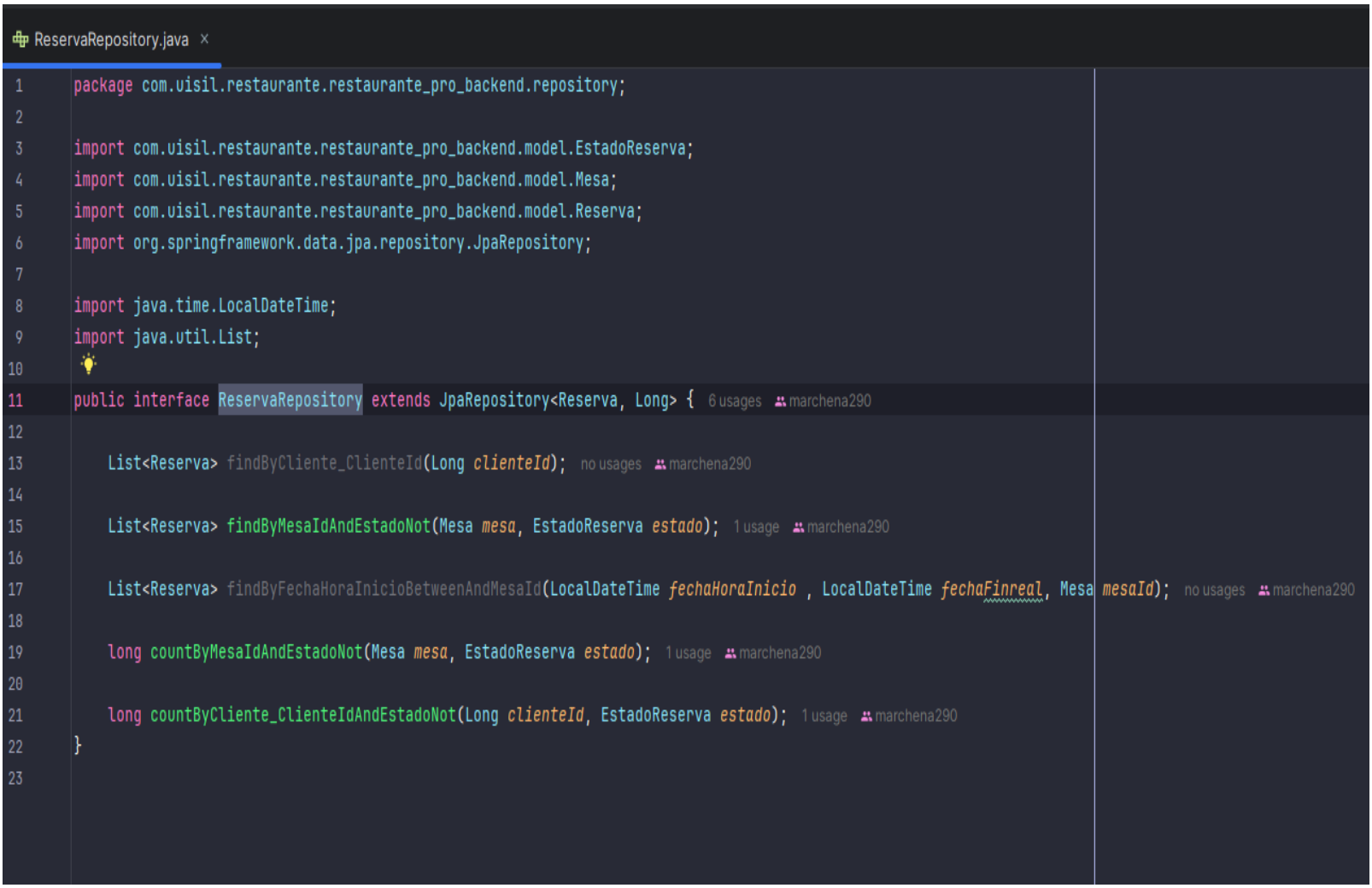
4.2. Evidencia de Conexión Funcional (Configuración)

La configuración para establecer el *Datasource* se gestiona a través de la interfaz de Spring, utilizando **variables de entorno** para mantener las credenciales de la base de datos fuera del código fuente, lo cual es una **práctica esencial de seguridad**.

application.properties

```
# ============================================================
# 1. CONEXIÓN A POSTGRESQL (USANDO VARIABLES DE ENTORNO)
# ============================================================

# URL COMPLETA:
spring.application.name=restaurante-pro-backend
spring.datasource.url=jdbc:postgresql://${DB_HOST}/${DB_RESTAURANTE}

# USUARIO: Usa la variable de entorno DB_USER
spring.datasource.username=${DB_USER}

# CONTRASEÑA: Usa la variable de entorno DB_PASSWORD
spring.datasource.password=${DB_PASSWORD}

# ============================================================
# 2. CONFIGURACIÓN DE JPA / HIBERNATE
# ============================================================

# Le dice a Spring que NO modifique la base de datos (ddl-auto=none)
spring.jpa.hibernate.ddl-auto=update

# Especifica el dialecto para PostgreSQL
spring.jpa.properties.hibernate-dialect=none

# Muestra las consultas SQL
spring.jpa.show-sql=true

# ============================================================
# 3. SEGURIDAD JWT
# ============================================================

api.security.secret=d1p2mQ6wR8nFzK9aJ5cY4bE3tU7vI0sXhLgYgOuZqWpTfCjBvA0xHrM4sN7k
```

4.3. Implementación del Repositorio La funcionalidad CRUD se implementa automáticamente a través de la interfaz **` JpaRepository ` ** de Spring Data JPA.



1. Arquitectura del Backend

Arquitectura Seleccionada: Arquitectura por Capas

La arquitectura seleccionada para el backend es la **Arquitectura por Capas** (o Arquitectura de Tres Capas). Esta decisión se justifica por la necesidad de asegurar el **mantenimiento, la escalabilidad y la separación de responsabilidades** del sistema, cumpliendo con el requisito no funcional de escalabilidad.

El backend desarrollado con Spring Boot se estructura en las siguientes capas lógicas:

1. **Capa de Presentación (Controladores):** Maneja las peticiones HTTP entrantes. En Spring Boot, son las clases anotadas con @RestController. Su única función es recibir peticiones, validar el formato de entrada (DTOs) y delegar la lógica al servicio correspondiente.
2. **Capa de Negocio (Servicios):** Contiene la lógica central de la aplicación. Aquí se implementan las reglas de negocio, como la validación de la disponibilidad de una mesa, la asignación de roles o el cambio de estado de una reservación.
3. **Capa de Datos (Repositorios):** Se encarga de la comunicación con la base de datos PostgreSQL. En Spring Boot, se utiliza Spring Data JPA, que simplifica el CRUD (Crear, Leer, Actualizar, Eliminar) a través de interfaces (@Repository) que extienden JpaRepository.

2. Componentes y Estructura del Código

El backend implementa un patrón de diseño **MVC** (Modelo-Vista-Controlador) para la API REST, donde el "Modelo" son las Entidades (persistencia), y las capas de Servicio y Controlador definen el flujo de negocio.

Estructura del Proyecto

El código está modularizado en paquetes funcionales que reflejan las entidades principales del sistema:

- com.restpro.reservaciones.entity: Contiene los modelos de la base de datos (User, Cliente, Mesa, Menu, Reservacion).
- com.restpro.reservaciones.repository: Contiene las interfaces de JPA para el acceso a datos.
- com.restpro.reservaciones.service: Implementa la lógica de negocio.
- com.restpro.reservaciones.controller: Define los *endpoints* REST.

- com.restpro.reservaciones.security: Maneja la configuración de JWT y autenticación.
- com.restpro.reservaciones.dto: Contiene los DTOs (Data Transfer Objects) para entrada y salida de datos (p. ej., LoginRequest, CreateReservacionDto).

Diagrama de Interacción de Capas (API REST)

El flujo de interacción es el siguiente:

1. El Frontend (Angular) realiza una solicitud HTTP al Backend.
2. El **Controlador** (@RestController) mapea la solicitud y valida la autenticación/autorización.
3. El Controlador llama a un método del **Servicio** (@Service).
4. El Servicio utiliza el **Repositorio** (@Repository) para interactuar con la Base de Datos (PostgreSQL).
5. El Servicio procesa los datos y los devuelve al Controlador, quien finalmente envía la respuesta HTTP al Frontend.

3. Elementos Clave del Backend

3.1 Seguridad: Autenticación con JWT y Control de Roles

La seguridad es un requerimiento no funcional fundamental.

- **Autenticación:** Se implementa con **Tokens JWT** (JSON Web Token). Spring Security se utiliza para interceptar las peticiones. Tras un *login* exitoso, el sistema genera y devuelve un token. El Frontend debe adjuntar este token en el encabezado (Authorization: Bearer <token>) de todas las solicitudes subsiguientes.
- **Cifrado de Contraseñas:** Las contraseñas se encriptan utilizando el algoritmo **BCrypt** antes de ser almacenadas en PostgreSQL.

- **Autorización y Roles:** Se definen los roles **ADMINISTRADOR** y **EMPLEADO**.
    - **Administrador:** Acceso completo a todos los módulos, incluyendo /usuarios.
    - **Empleado:** Acceso a módulos operativos (Clientes, Mesas, Menús, Reservaciones), pero con permisos limitados (p. ej., no puede gestionar otros usuarios).
    - La autorización se aplica a nivel de controlador con anotaciones de Spring Security, por ejemplo: @PreAuthorize("hasRole('ADMINISTRADOR')") en el UserController.

3.2 Gestión de Errores y Escalabilidad

- **Gestión de Errores:** Se implementa un manejo centralizado de excepciones mediante la anotación @ControllerAdvice. Esto permite interceptar excepciones de la capa de servicio (p. ej., MesaNotFoundException o ReservacionConflictException) y mapearlas a respuestas HTTP estandarizadas (p. ej., 404 NOT FOUND o 400 BAD REQUEST), asegurando mensajes de error personalizados y útiles al cliente.
- **Escalabilidad:** La arquitectura por capas facilita la escalabilidad horizontal. Si la carga de trabajo de la lógica de negocio aumenta, solo se replica la capa de servicios. El uso de **JPA** permite cambiar el motor de la base de datos (si fuera necesario) con mínima modificación de código en la capa de persistencia.

4. Conexión con la Base de Datos y el Frontend

Conexión con PostgreSQL

La conexión con la base de datos PostgreSQL se configura a través del archivo application.properties de Spring Boot. Se utiliza **Spring Data JPA** como

tecnología ORM (Object-Relational Mapping), lo que garantiza la integridad y consistencia de los datos

```
# Configuración de PostgreSQL
spring.datasource.url=jdbc:postgresql://localhost:5432/restaurantedb
spring.datasource.username=restaurante_user
spring.datasource.password=password_segura
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Endpoints (Rutas API) para el Frontend

La comunicación con el Frontend de Angular se realiza exclusivamente mediante peticiones API REST. A continuación, se detallan los endpoints clave:

Módulo	**Funcionalidad**	**Método HTTP**	**Ruta (Endpoint)**
Autenticación	Login de usuarios	POST	/api/auth/login
Usuarios	CRUD de empleados y roles	GET, POST, PUT, DELETE	/api/admin/usuarios
Clientes	CRUD de clientes	GET, POST, PUT, DELETE	/api/clientes

Módulo	Funcionalidad	Método HTTP	Ruta (Endpoint)
Mesas	CRUD y consulta de disponibilidad	GET, POST, PUT, DELETE	/api/mesas
Menús	CRUD de platillos	GET, POST, PUT, DELETE	/api/menus
Reservaciones	CRUD de reservaciones	GET, POST, PUT, DELETE	/api/reservaciones
Reservaciones	Cambiar estado (Pendiente, Confirmada, Cancelada) [33]	PATCH	/api/reservaciones/{id}/estado
Reportes	Estadísticas de uso y pedidos	GET	/api/reportes/dashboard

5. Fragmentos de Código

A continuación, se presentan ejemplos de código para ilustrar la implementación de las capas clave de Spring Boot.

5.1 Entity: Reservacion.java

Muestra la definición de la Entidad de Reservación, incluyendo la relación ManyToOne con la Entidad Mesa y el uso de Enum para el estado.

```
// src/main/java/com/restpro/reservaciones/entity/Reservacion.java
@Entity
@Table(name = "reservaciones")
public class Reservacion {

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private LocalDateTime fechaHora; // Fecha y hora de la reserva

    @Column(nullable = false)
    private int cantidadPersonas;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private EstadoReservacion estado = EstadoReservacion.PENDIENTE; // Estados:
PENDIENTE, CONFIRMADA, CANCELADA

    // Relación ManyToOne: Una reservación pertenece a una mesa
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "mesa_id", nullable = false)
    private Mesa mesa;

    // Relación ManyToOne: Una reservación pertenece a un cliente
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "cliente_id", nullable = false)
    private Cliente cliente;
```

Nota no se utilizan getter y setter, ya que se usa la librería de lombok, esta librería genera los getters y setters,con solo importar la biblioteca.

5.2 Service: ReservacionService.java

Muestra la implementación de la lógica de negocio. El método crearReservacion valida la disponibilidad de la mesa antes de guardar el registro, previniendo errores de duplicación

```
// src/main/java/com/restpro/reservaciones/service/ReservacionService.java
@Service
```

```
public class ReservacionService {

    @Autowired
    private ReservacionRepository reservacionRepository;
    @Autowired
    private MesaRepository mesaRepository;

    @Transactional
    public Reservacion crearReservacion(CreateReservacionDto dto) {

        Mesa mesa = mesaRepository.findById(dto.getMesaId())
                            .orElseThrow(() -> new EntityNotFoundException("Mesa no encontrada"));

        // **LÓGICA DE NEGOCIO CRÍTICA:**
        // Se valida la disponibilidad de la mesa en la fecha y hora solicitada.
        if (reservacionRepository.existeConflicto(dto.getMesaId(), dto.getFechaHora())) {
            throw new ReservacionConflictException("La mesa ya está reservada en ese horario.");
        }

        // Mapeo del DTO a la Entidad
        Reservacion nuevaReservacion = new Reservacion();
        nuevaReservacion.setMesa(mesa);
        nuevaReservacion.setFechaHora(dto.getFechaHora());
        nuevaReservacion.setCantidadPersonas(dto.getCantidadPersonas());
        // El estado por defecto es PENDIENTE

        return reservacionRepository.save(nuevaReservacion); // Guarda la reservación
    }
}
```

5.3 Controller: ReservacionController.java

Muestra la capa de presentación, la cual recibe la petición y delega al servicio. Se utiliza @PreAuthorize para el control de acceso.

// src/main/java/com/restpro/reservaciones/controller/ReservacionController.java

```
@RestController
@RequestMapping("/api/reservaciones")
public class ReservacionController {

    @Autowired
    private ReservacionService reservacionService;

    // Solo Empleados y Administradores pueden crear una reserva (ya que no hay reserva de clientes)
    @PreAuthorize("hasAnyRole('ADMINISTRADOR', 'EMPLEADO')")
    @PostMapping
    public ResponseEntity<Reservacion> crearReservacion(@Valid @RequestBody CreateReservacionDto dto) {
        Reservacion nueva = reservacionService.crearReservacion(dto);
        return new ResponseEntity<>(nueva, HttpStatus.CREATED);
    }

    // Solo el Administrador puede listar todas las reservas para reportes.
    @PreAuthorize("hasRole('ADMINISTRADOR')")
    @GetMapping("/todos")
    public ResponseEntity<List<Reservacion>> listarTodasReservaciones() {
        return ResponseEntity.ok(reservacionService.findAll());
    }

    // El control de errores (p. ej., Mesa no encontrada) es gestionado por el @ControllerAdvice.
}
```