Aplicación de BD

Sistema de Gestión de Menú y Pedidos en un Restaurante

Programacion Orientada a Objetos

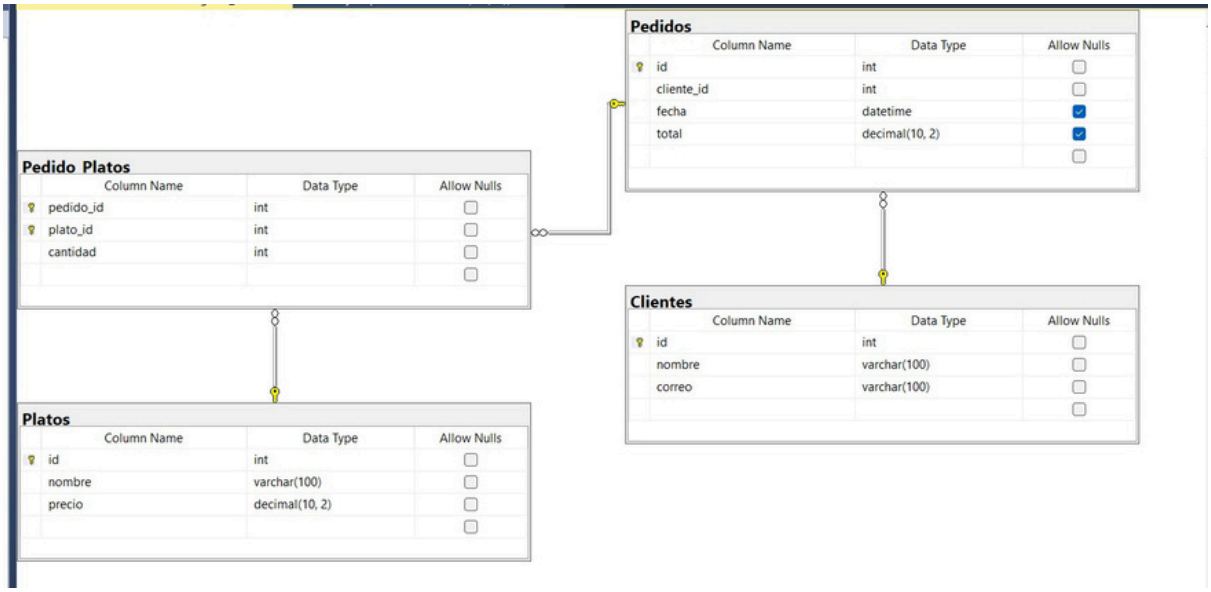
Alumno: Hector Badillo Garcia
IDE: Intellij comunity version
BASE DE DATOS: SQL Server (montado en un contenedor de docker)
LENGUAJE: Java, Maeven
Enlace de git: https://github.com/Inu41123/ProyectoSistemaRestaurante.git

Problema planteado

El restaurante “Delicias Gourmet” desea un sistema de escritorio que permita gestionar su **menú de platillos**, registrar a sus **clientes** y permitir que cada cliente realice **pedidos que incluyan múltiples platillos**. A su vez, cada platillo puede estar presente en varios pedidos.

Modelo relacional de bd

Se construyo una base de datos, simple, es claramente mejorable:



Tablas de la Base de Datos

1. **Clientes**: Esta tabla almacena la información de los clientes.
   - **id**: Es la **clave primaria** (primary key) y es un identificador único para cada cliente. Su tipo de dato es int (entero).
   - **nombre**: Almacena el nombre del cliente y su tipo de dato es varchar(100), que significa que puede guardar una cadena de texto de hasta 100 caracteres.
   - **correo**: Guarda el correo electrónico del cliente, con el mismo tipo de dato `varchar(100)`.

2. **Pedidos**:Esta tabla registra cada pedido realizado por un cliente.
   - **id**:La **clave primaria** de la tabla, un identificador único para cada pedido.

- **cliente_id**: Esta es una **clave foránea** (foreign key). Es la que conecta la tabla Pedidos con la tabla Clientes, indicando qué cliente hizo un pedido específico.
- **fecha**: Almacena la fecha y hora en que se realizó el pedido, usando el tipo de `dato datetime`.
- **total**: Guarda el monto total del pedido. Su tipo de dato es decimal(10, 2), ideal para guardar valores monetarios, con hasta 10 dígitos en total y 2 de ellos para los decimales.

3. **Platos**:Esta tabla contiene el catálogo de los platos disponibles en el restaurante.
   - **id**: La **clave primaria** de la tabla, identificador único para cada plato.
   - **nombre**: Almacena el nombre del plato, con el tipo de dato `varchar(100)`.
   - **precio**: Guarda el precio de cada plato, usando `decimal(10, 2)`.

4. **PedidoPlatos**: Esta tabla es una tabla de unión que resuelve la relación de muchos a muchosentre Pedidos y Platos. Un pedido puede tener muchos platos y un plato puedeestar en muchos pedidos.
   - **pedido_id**: **Clave foránea** que se conecta a la tabla Pedidos.
   - **plato_id**: **Clave foránea** que se conecta a la tabla Platos.
   - **cantidad**: Indica cuántas unidades de un plato específico se pidieron en un pedido dado.

Relaciones entre Tablas

Las líneas que conectan las tablas representan las relaciones. En este caso:

- **Clientes y Pedidos (1 a n)**: Una línea conecta Clientes.id con Pedidos.cliente_id. La simbología indica que un **cliente** puede tener **muchos pedidos**.
- **Pedidos y Pedido Platos (1 a n)**: Una línea conecta Pedidos.id con Pedido Platos.pedido_id. Esto significa que un **pedido** puede incluir **muchos platos**.
- **Platos y Pedido Platos (1 a n)**: Una línea conecta Platos.id con Pedido Platos.plato_id. Esto indica que un **plato** puede ser parte de **muchos pedidos**.

cuando un cliente hace un pedido, se crea un nuevo registro en la tabla Pedidos. Luego, para cada plato que el cliente pide, se crea una fila en la tabla Pedido Platos que asocia ese pedido con el plato correspondiente y la cantidad pedida.

Código de Solución, explicado brevemente

Laestructuraprincipalsigue unaarquitecturaporcapasconelementos de MVC (Modelo-Vista-Controlador):

enlacce del codigo para no hacer demasiado extenso el documento: https://github.com/lnu41123/ProyectoSistemaRestaurante.git:

esta es mi estructura principal:

```
src/
├── main/
│   ├── java/
│   │   └── com/
│   │       └── restaurante/
│   │           ├── controllers/
│   │           │   ├── ClienteController.java
│   │           │   ├── MainController.java
│   │           │   ├── PedidoController.java
│   │           │   └── PlatoController.java
│   │           ├── dao/
│   │           │   ├── ClienteDAO.java
│   │           │   ├── PedidoDAO.java
│   │           │   └── PlatoDAO.java
│   │           ├── models/
│   │           │   ├── Cliente.java
│   │           │   ├── ItemPedido.java
│   │           │   ├── Pedido.java
│   │           │   └── Plato.java
│   │           ├── services/
│   │           │   ├── ClienteService.java
│   │           │   ├── PedidoService.java
│   │           │   └── PlatoService.java
│   │           ├── utils/
│   │           │   └── DatabaseConnection.java
│   │           ├── MainApp.java
│   │           └── module-info.java
│   └── resources/
│       ├── com/restaurante/views/
│       │   ├── clientes.fxml
│       │   ├── main.fxml
│       │   ├── pedidos.fxml
│       │   └── platos.fxml
│       ├── css/
│       │   └── styles.css
│       └── db/
│           └── schema.sql
└── test/
```

en base a ello cada paquete se encarga de una parte específica del problema y en base al trabajo modular de cada módulo (válgame la redundancia) se puede poner en funcionamiento la aplicación

1. models/
    - Contiene las clases que representan entidades del negocio (objetos).
    - `Ejemplo: Cliente.java, Pedido.java, etc.`
    - Qué hace: Define la estructura de datos (atributos, métodos básicos, relaciones entre clases

1. `DetallePedidoItem.java`

- Qué es: Una clase especializada para mostrar información de pedidos en la interfaz gráfica (JavaFX).
- Qué contiene:
    - Propiedades observables (IntegerProperty, StringProperty, etc.) para integrarse con JavaFX.
    - Datos como ID del pedido, nombre del cliente, plato, cantidad, subtotal y fecha.
- Para qué sirve:
    - Facilitar el binding de datos en tablas o listas de FXML.
    - Mostrar detalles de pedidos en la vista de forma reactiva (cambios automáticos en la UI).

2. `Pedido.java`

- Qué es: La entidad principal que representa un pedido en el sistema.
- Qué contiene:
    - ID, ID del cliente, fecha, total, nombre del cliente y lista de ítems `(ItemPedido)`.
    - Método calcularTotal() que suma los subtotales de los ítems.
- Para qué sirve:
    - Gestionar la información completa de un pedido (relación con cliente y ítems).
    - Calcular el total automáticamente basado en los platos y sus cantidades.

3. `ItemPedido.java`

- Qué es: Representa un ítem individual dentro de un pedido (línea de pedido).
- Qué contiene:
  - Relación con un `Plato` y la cantidad solicitada.
  - Propiedad observable `cantidadProperty()` para JavaFX.
- Para qué sirve:
  - Vincular platos específicos con sus cantidades en un pedido.
  - Permitir edición dinámica de cantidades en la UI.

4. `Plato.java`

- Qué es: Modela un plato o producto del menú del restaurante.
- Qué contiene:
  - ID, nombre y precio.
  - Propiedades observables (`nombreProperty`, `precioProperty`) para JavaFX.
- Para qué sirve:
  - Almacenar información de los platos disponibles.
  - Integrarse con interfaces gráficas (ej. mostrar precios en tiempo real).

5. `Cliente.java`

- Qué es: Representa a un cliente del restaurante.
- Qué contiene:
  - ID, nombre y correo electrónico.
  - Propiedades observables (`nombreProperty`, `correoProperty`) para JavaFX.
- Para qué sirve:
  - Gestionar datos de clientes y vincularlos a pedidos.
  - Mostrar información en tablas o formularios de la UI.

Relaciones entre Clases

- `Pedido` → Contiene una lista de `ItemPedido`.
- `ItemPedido` → Referencia a un `Plato` y su cantidad.
- `Pedido` → Referencia a un `Cliente` (por `clienteId` y `clienteNombre`).

- `DetallePedidoItem` → Agrega datos de `Pedido`, `Cliente` y `Plato` para mostrar en la UI.

2. controllers/
    - Manejan la lógica de interacción entre la vista y los servicios.
    - Ejemplo: ClienteController.java gestiona eventos de la interfaz de clientes.
    - Qué hace: Reciben input del usuario (vista), llaman a servicios y actualizan la vista.

1. `MainController.java`

- Qué hace:
    - Controla la ventana principal (TabPane) y coordina la carga de datos al cambiar entre pestañas ("Clientes", "Platos", "Pedidos").
    - Llama a los métodos cargarDatos() de cada controller hijo al activar su pestaña.
- Interacción clave:
    - `Notifica a ClienteController, PlatoController` y `PedidoController` para refrescar sus datos.

2. `ClienteController.java`

- Qué hace:
    - Gestiona la vista de clientes (CRUD: Crear, Leer, Actualizar, Eliminar).
    - Valida datos (nombre y correo electrónico) antes de guardar.
- Componentes principales:
    - TableView<Cliente>: Muestra la lista de clientes.
    - Validator: Valida campos del formulario.
    - ClienteService: Llama a la lógica de negocio y persistencia.
- Flujo típico:
    - Guardar: Valida → Crea/Actualiza cliente → Refresca tabla.
    - Eliminar: Confirma → Elimina cliente → Refresca tabla.

3. `PlatoController.java`

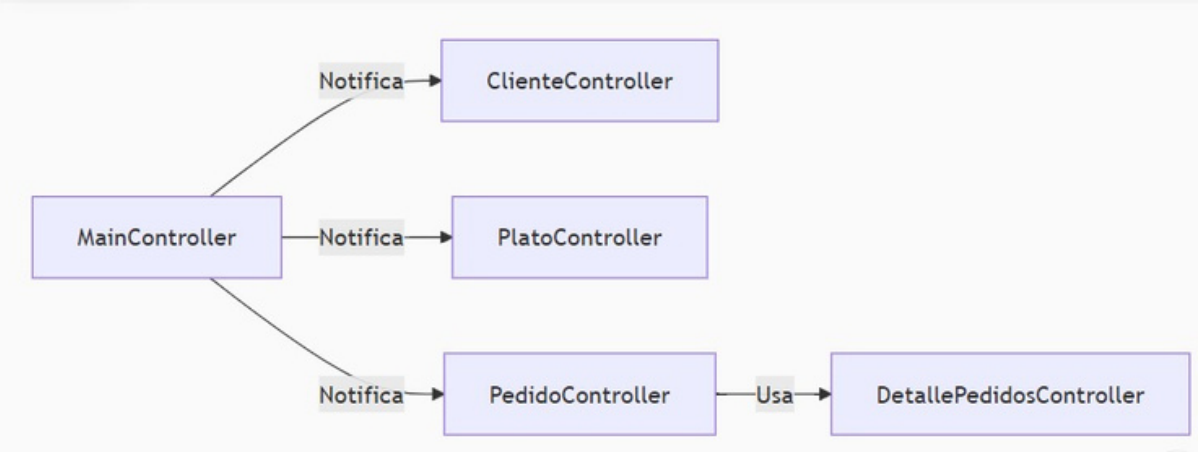
- Qué hace:
  - Administra el menú de platos (CRUD).
  - Valida nombre y precio (número positivo).
- Componentes clave:
  - `TableView<Plato>`: Lista de platos disponibles.
  - `Spinner/TextField`: Para ingresar precio y nombre.
- Funcionalidad destacada:
  - Formatea el precio en la tabla (`$10.00`).
  - Muestra notificaciones de éxito/error.

4. `PedidoController.java`

- Qué hace:
  1. Controla la creación/edición de pedidos:
  - Selección de cliente (ComboBox).
  - Agregar/eliminar platos (ItemPedido).
  - Cálculo automático del total.
- Componentes clave:
  1. Dos `TableView`: Platos disponibles vs. platos seleccionados.
  2. `ObservableList<ItemPedido>`: Mantiene los ítems del pedido actual.
  3. `PedidoService`: Persiste los pedidos.
- Flujo:
  1. Seleccionar cliente + platos.
  2. Calcular total → Guardar/Actualizar.

5. `DetallePedidosController.java`

- Qué hace:
  - Muestra un reporte detallado de todos los pedidos.
  - `Combina datos de Pedido, Cliente, y Plato en DetallePedidoItem.`
- Componentes:
  - `TableView<DetallePedidoItem>`: Muestra ID, cliente, plato, cantidad, subtotal y fecha.
- Origen de datos:
  - Consulta todos los pedidos con `PedidoService` y "aplana" los ítems para la tabla.



3. services/
    - ◯ Contienen la lógica de negocio (reglas, validaciones).
    - ◯ Ejemplo: ClienteService.java podría validar datos antes de guardar un cliente.
    - ◯ Qué hace: Orquestan operaciones complejas y usan los DAOs para persistencia.

1. `PedidoService.java`

- ⬤Qué hace:
    - ◯ Gestiona la lógica de negocio relacionada con los pedidos (crear, actualizar, eliminar, listar).
    - ◯ Valida que los pedidos cumplan con reglas básicas antes de persistirlos.
- ⬤ Métodos clave:
    - ◯ `crearPedido() / actualizarPedido(): Validan y delegan la` persistencia al PedidoDAO.
    - ◯ validarPedido(): Asegura que el pedido tenga un cliente válido y al menos un ítem con cantidad > 0.
    - ◯ obtenerTodos(): Retorna todos los pedidos (usado en reportes).
- ⬤ Interacción:
    - ◯ Usa PedidoDAO para operaciones de base de datos.
    - ◯ Lanzan excepciones si las validaciones fallan.

2. `PlatoService.java`

- Qué hace:
  - Administra la lógica de negocio de los platos (CRUD).
  - Valida nombre y precio antes de guardar.
- Métodos clave:
  - validarPlato(): Verifica que el nombre no esté vacío, el precio sea positivo y el nombre no exista ya.
  - listarPlatos(): Retorna todos los platos para mostrarlos en la UI.
- Reglas de negocio:
  - No permite platos duplicados (por nombre).
  - El precio debe ser mayor a 0.

3. `ClienteService.java`

- Qué hace:
  - Maneja la lógica relacionada con clientes (CRUD).
  - Valida formato de correo electrónico y evita duplicados.
- Métodos clave:
  - `validarCliente()`:
    - Nombre obligatorio.
    - Correo válido (usa Pattern para regex).
    - Correo único en el sistema.
  - listarClientes(): Obtiene todos los clientes para mostrar en tablas.
- Interacción:
  - Usa ClienteDAO para operaciones de base de datos.

4. `dao/` (Data Access Object)
   - Se encargan de interactuar con la base de datos.
   - Ejemplo: ClienteDAO.java ejecuta consultas SQL para CRUD de clientes.
   - Qué hace: Aísla el acceso a la base de datos (ejecuta `INSERT`, `SELECT`, etc.).

1. PedidoDAO.java

- Responsabilidad:
  - Gestiona todas las operaciones de base de datos relacionadas con pedidos y sus ítems.
- Métodos clave:
  - crear(): Inserta un nuevo pedido y sus ítems en transacción (usa RETURN_GENERATED_KEYS para obtener el ID).
  - listarTodos() / listarPorCliente(): Recuperan pedidos con sus ítems y datos del cliente (JOIN con Clientes).
  - actualizar(): Reemplaza todos los ítems del pedido (elimina los antiguos e inserta los nuevos).
  - obtenerItemsPedido(): Método auxiliar para cargar los platos asociados a un pedido.
- Técnicas destacadas:
  - Uso de Batch para insertar múltiples ítems eficientemente.
  - Consultas con JOIN para obtener datos relacionados (ej: nombre del cliente).

2. PlatoDAO.java

- Responsabilidad:
  - Maneja el acceso a datos de la tabla Platos.
- Métodosclave:
  - crear() / actualizar() / eliminar(): Operaciones CRUD básicas.
  - existeNombre(): Verifica duplicados antes de insertar/actualizar.
- Características:
  - Consultas simples con PreparedStatement para prevenir SQL injection.
  - Retorna el ID generado al crear un nuevo plato.

3. ClienteDAO.java

- Responsabilidad:
  - Operasobre la tabla Clientes.
- Métodosclave:
  - existeCorreo(): Valida unicidad del correo electrónico.
  - listarTodos(): Retorna todos los clientes para mostrar en tablas.
- Patronescomunes:
  - Usodetry-with-resources para manejo automático de conexiones.

- Generación de IDs con `RETURN_GENERATED_KEYS`.

5. utils/
    - Clases de utilidad compartidas.
    - Ejemplo: DatabaseConnection.java gestiona la conexión a la DB.
    - Qué hace: Proporciona funcionalidades reutilizables (ej. conexión DB).

DatabaseConnection.java

Función: Clase utilitaria que centraliza la conexión a la base de datos del sistema.

Qué hace:

1. Configura y provee conexiones JDBC a la base de datos DeliciasGourmetDB (SQL Server).
2. Maneja credenciales y URL de conexión de forma segura (aunque en código duro, idealmente deberían estar en un archivo de configuración).
3. Registra en consola el éxito/error de la conexión (útil para depuración).

Características clave:

- Singleton implícito: Todos los DAOs usan el mismo método estático getConnection().
- Manejo de errores: Lanza SQLException si falla la conexión y muestra detalles en consola.
- Parámetros de conexión:
    - Puerto 11433 (típico para SQL Server).
    - encrypt=false y trustServerCertificate=true para desarrollo local (no recomendado en producción).

6. `views/` (en `resources/`)
    - Archivos FXML que definen la interfaz gráfica (JavaFX).
    - Ejemplo: clientes.fxml diseña la pantalla de gestión de clientes.
    - Qué hace: Define la estructura visual (botones, tablas, etc.).

1. main.fxml

- Función:

    - Vista principal que actúa como contenedor de las demás pantallas mediante un `TabPane`.
- Componentes clave:
    - 4 pestañas: Clientes, Platos, Pedidos y Reportes (esta última vacía).
    - Cada pestaña carga una vista secundaria (`clientes.fxml, platos.fxml, etc.`).
- Estilo:
    - Diseño limpio con tamaño fijo (`1200x1200`).

2. `clientes.fxml`

- Función:
    - Gestión de clientes (CRUD).
- Componentes clave:
    - Formulario: Campos para `nombre` y `correo` + botones (Guardar, Eliminar, Limpiar).
    - `TableView`: Muestra lista de clientes registrados.
- Detalles:
    - Botón "Eliminar" se deshabilita inicialmente (`disable="true"`).
    - Estilo con colores distintivos (verde para guardar, rojo para eliminar).

3. `platos.fxml`

- Función:
    - Administración del menú (CRUD de platos).
- Componentes clave:
    - Formulario: Campos para `nombre` y `precio` + botones similares a `clientes.fxml`.
    - `TableView`: Lista de platos con columnas para ID, nombre y precio.
- Diferencia clave:
    - Validación de precio en el controlador (debe ser número positivo).

4. `pedidos.fxml`

- Función:

    - Creación y gestión de pedidos.
- Componentes clave:
    - Sección superior:
        - ComboBox para seleccionar cliente.
        - Dos TableView: Platos disponibles vs. platos agregados al pedido.
        - Spinner para cantidad y botones (Agregar, Eliminar).
    - Sección inferior:
        - TableView con historial de pedidos.
        - Botones para Nuevo Pedido, Actualizar, Refrescar y Ver Detalle.
    - Label dinámico que muestra el total del pedido.
- Complejidad:
    - Vista más elaborada con múltiples interacciones y actualizaciones en tiempo real.

5. `DetallePedidos.fxml`

- Función:
    - Reporte detallado de todos los pedidos (relación cliente-plato).
- Componentes clave:
    - TableView con columnas para:
        - ID del pedido, cliente, fecha, plato, cantidad y subtotal.
- Uso:
    - Se abre desde el botón "Ver Detalle" en pedidos.fxml.

7. MainApp.java
    - Punto de entrada de la aplicación.
    - Qué hace: Inicia la aplicación JavaFX y carga la vista principal.

2. `MainApp.java`

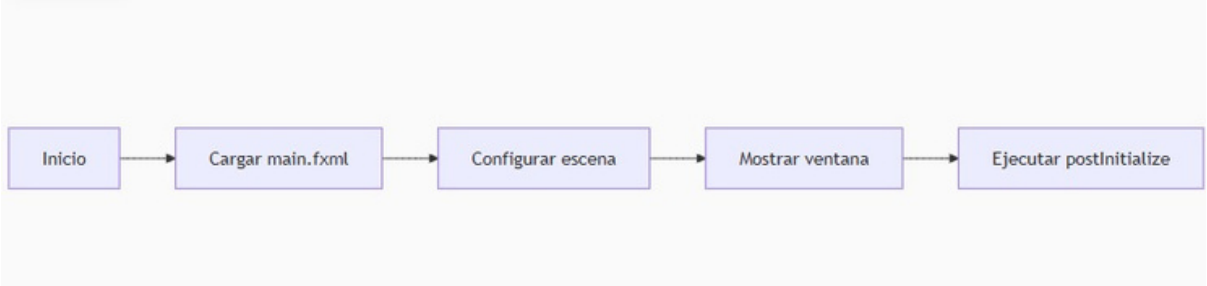
Función:

Clase principal que inicia la aplicación JavaFX y configura la ventana inicial.

Qué hace:

1. Carga la vista principal (main.fxml) y su controlador (`MainController`).

2. Configura la escena:
    - ◯ Tamaño inicial de `800x600` (pero se maximiza).
    - ◯ Aplica estilos CSS (`styles.css`).
3. Muestra la ventana:
    - ◯ Maximizada y con título *"Sistema Restaurante - Delicias Gourmet"*.
4. Inicialización tardía:
    - ◯ Llama a `postInitialize()` del controlador después de mostrar la ventana (para cargar datos iniciales).



8. module-info.java
    - ◯ Configuración de módulos (Java 9+).
    - ◯ Qué hace: Define dependencias y exporta paquetes.

1. `module-info.java`

Función:

Archivo de configuración de módulos (Java 9+) que define las dependencias y permisos del proyecto.

Qué hace:

- ● Declara dependencias (`requires`):
    - ◯ `javafx.controls y javafx.fxml: Para JavaFX (UI).`
    - ◯ `java.sql`: Para acceso a bases de datos.
    - ◯ `org.controlsfx.controls y net.synedra.validatorfx: Librerías` externas (controles UI y validación).
    - ◯ `lombok`: Solo en tiempo de compilación (`requires static`).
- ● Permisos (`opens`):
    - ◯ Abre paquetes a JavaFX (`com.restaurante.controllers` para FXML, `com.restaurante.models` para binding de datos).
- ● Exporta (`exports`):

- Paquetes accesibles para otros módulos (`com.restaurante.controllers`).

Importante: Garantiza que JavaFX y las librerías externas funcionen correctamente en un proyecto modular.

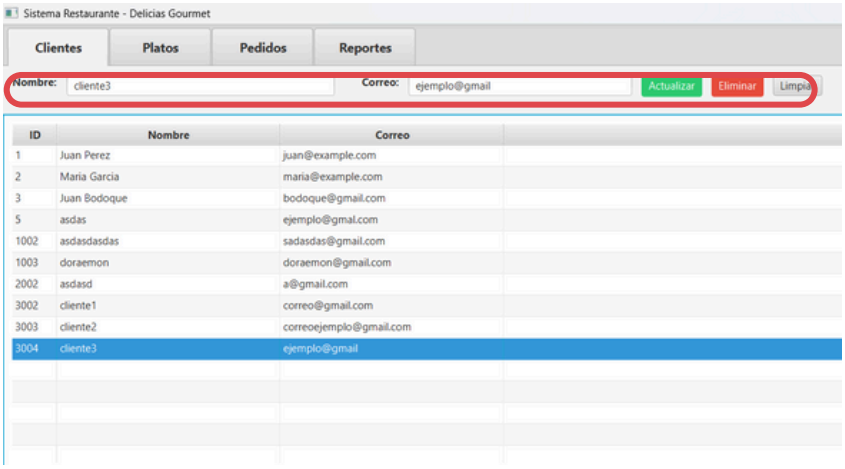
9. resources/db/schema.sql
    - Scripts SQL para crear tablas.
    - Qué hace: Define la estructura inicial de la base de datos.

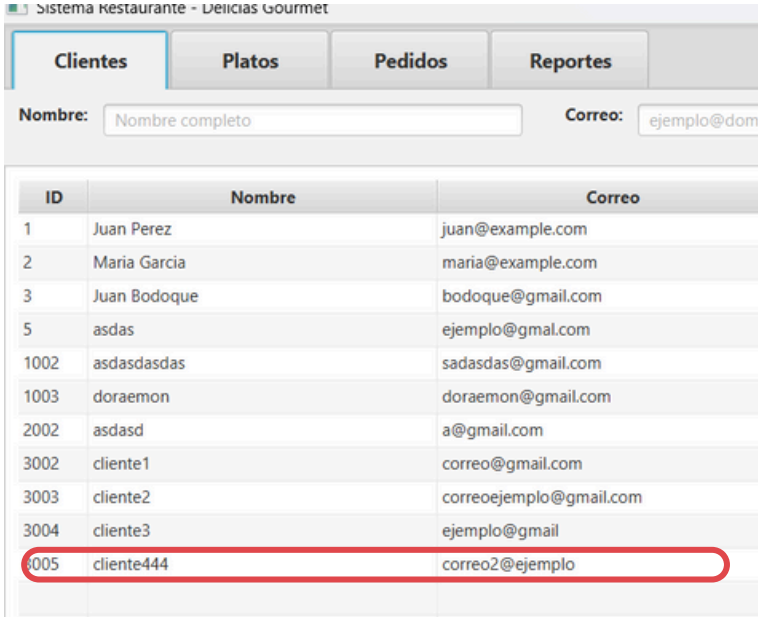
Pruebas con los casos de prueba mencionados y Resultados obtenidos

Casos de prueba mínimos

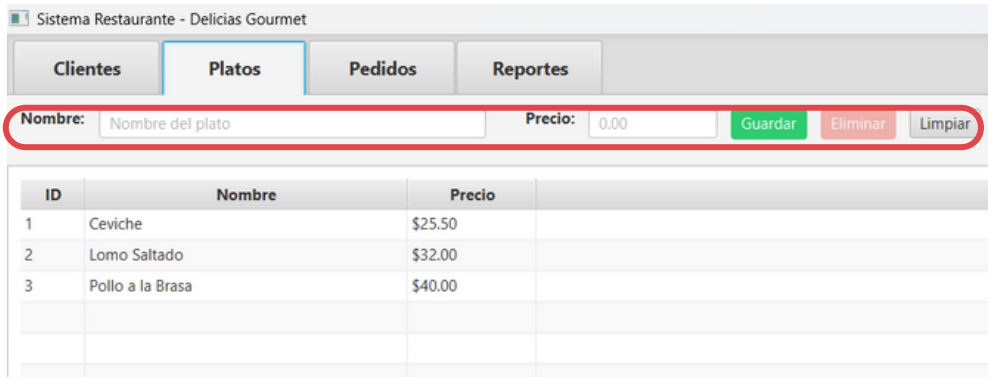
1. Crear un cliente y tres platillos, luego generar un pedido con dos de ellos.

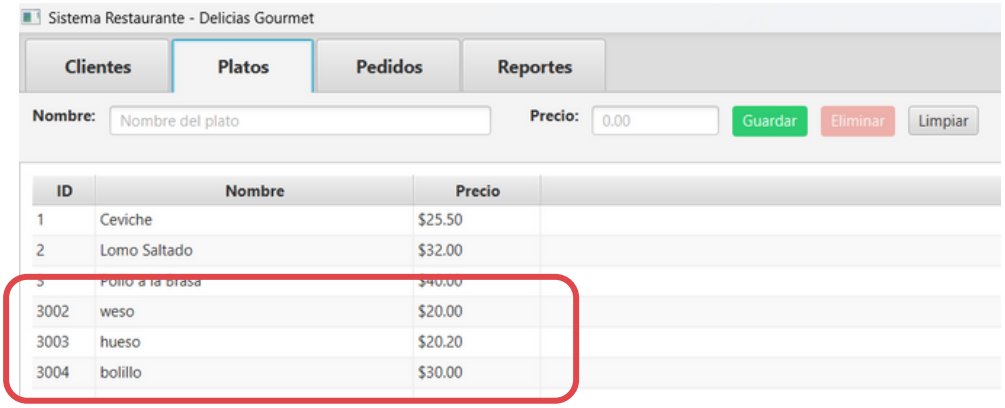
Creamos el cliente nuevo



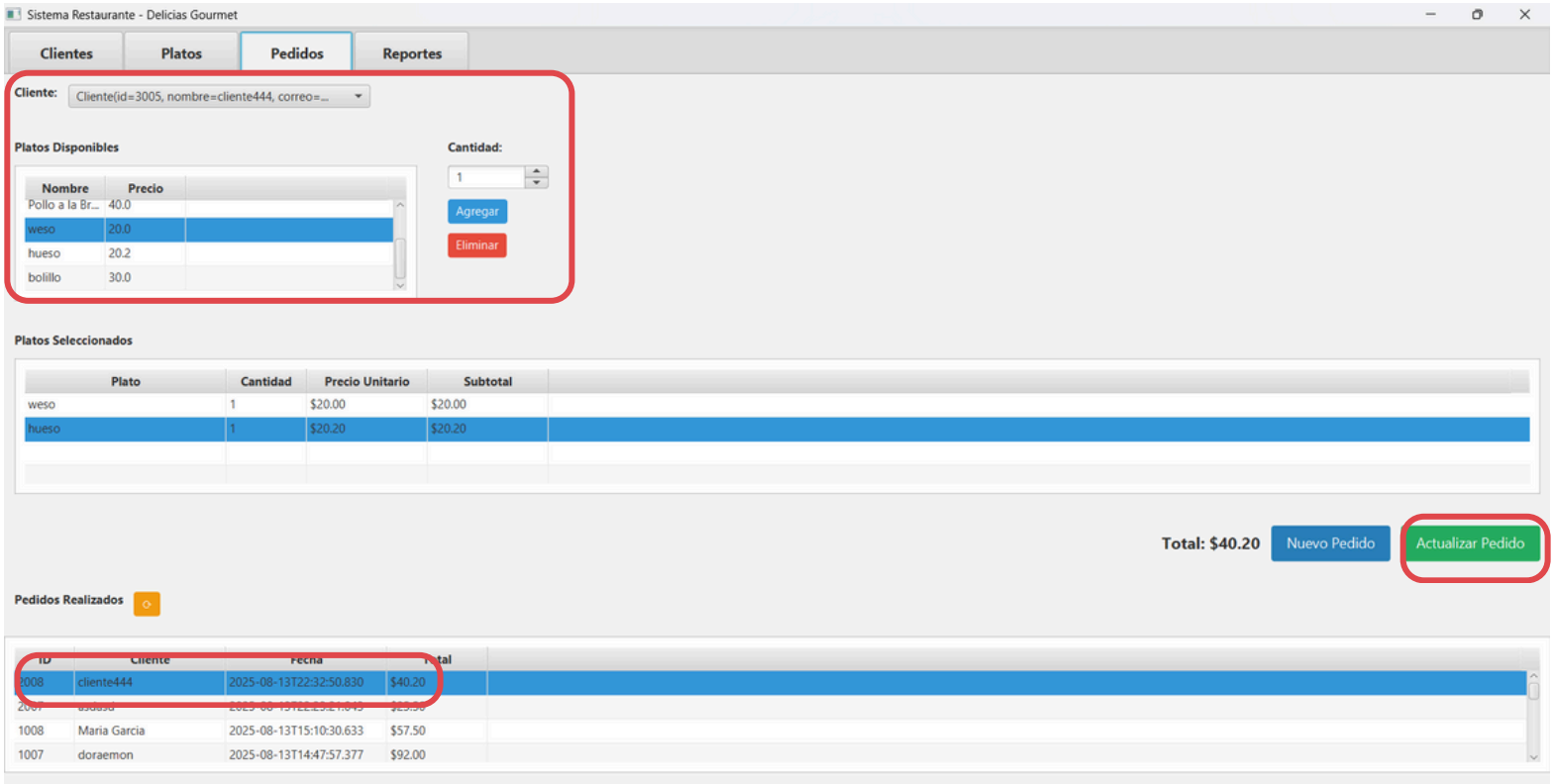


Creamos los platillos





Creamos los pedidos



2. Ver la lista de pedidos del cliente.

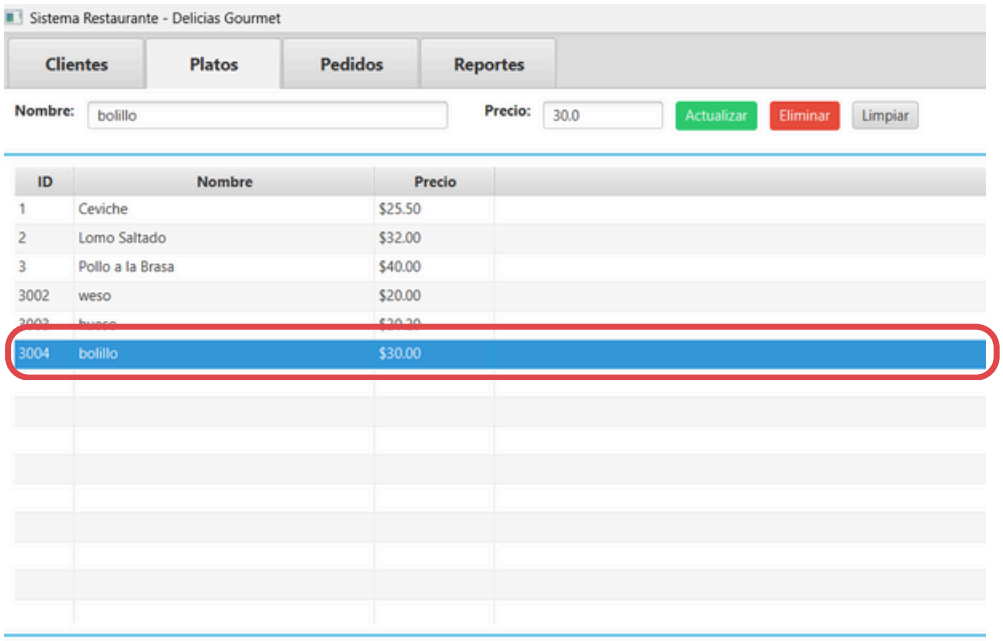
Al hacer clic sobre el pedido podremos revisar de que cosas se compone a su vez podemos editarlas

3. Ver el total pagado por pedido.

Aqui podemos revisar el total a pagar, respecto a nustro pedido

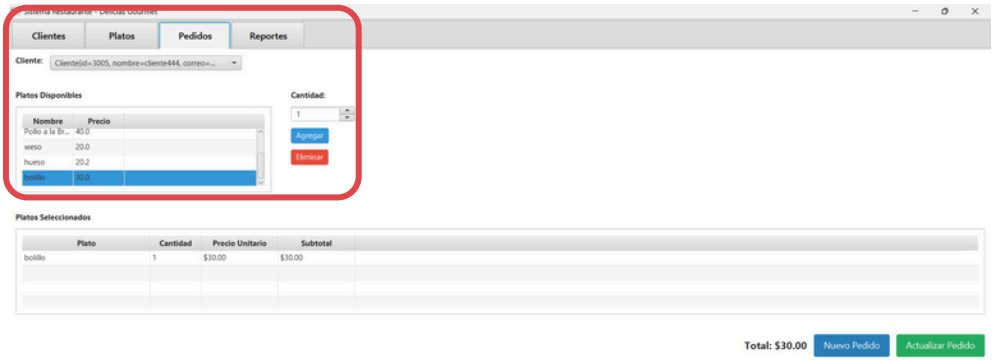
4. Eliminar un platillo y verificar que se borra de los pedidos.

Revisamos el platillo que vamos a usar

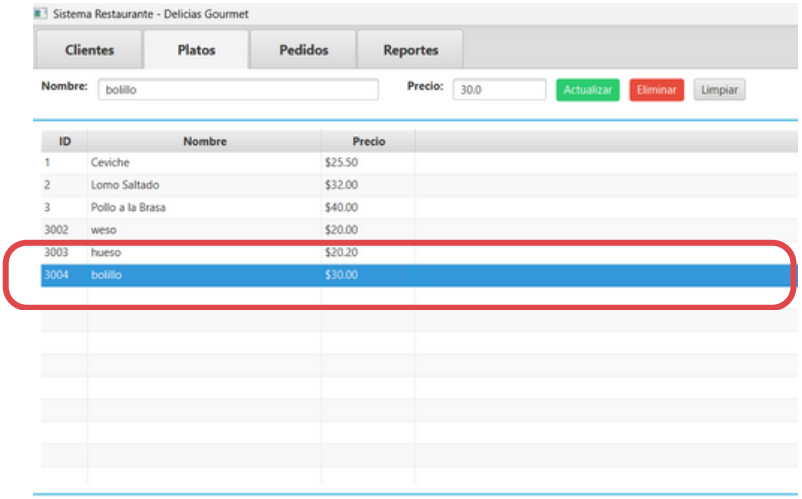


4. Eliminar un platillo y verificar que se borra de los pedidos.

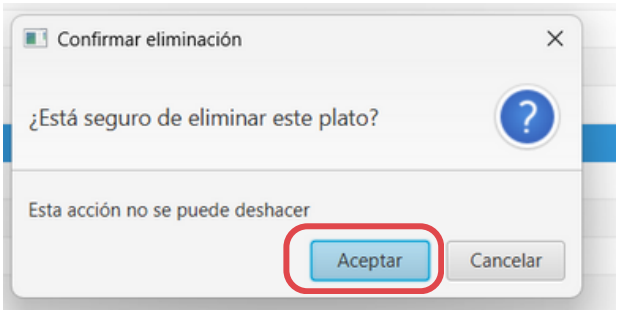
creamos un pedido pero no lo subimos, se quedara ahi de momento



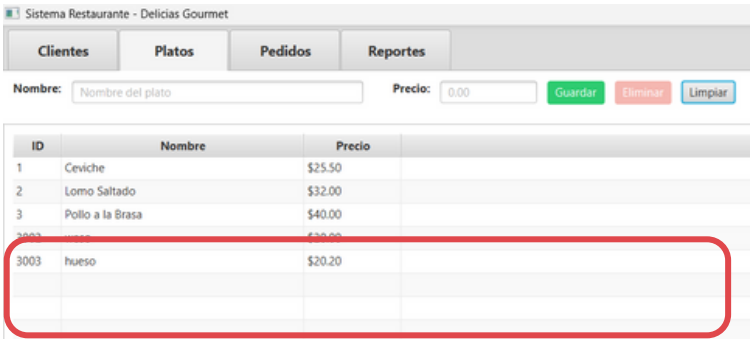
Ubicamos el platillo en nuestra pantalla de platillos y precionamos el de eliminar



lo eliminamos

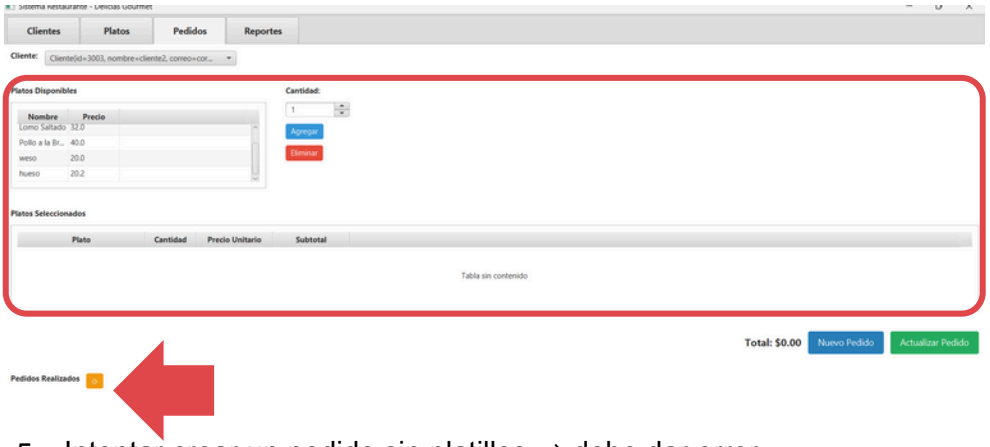


revisamos



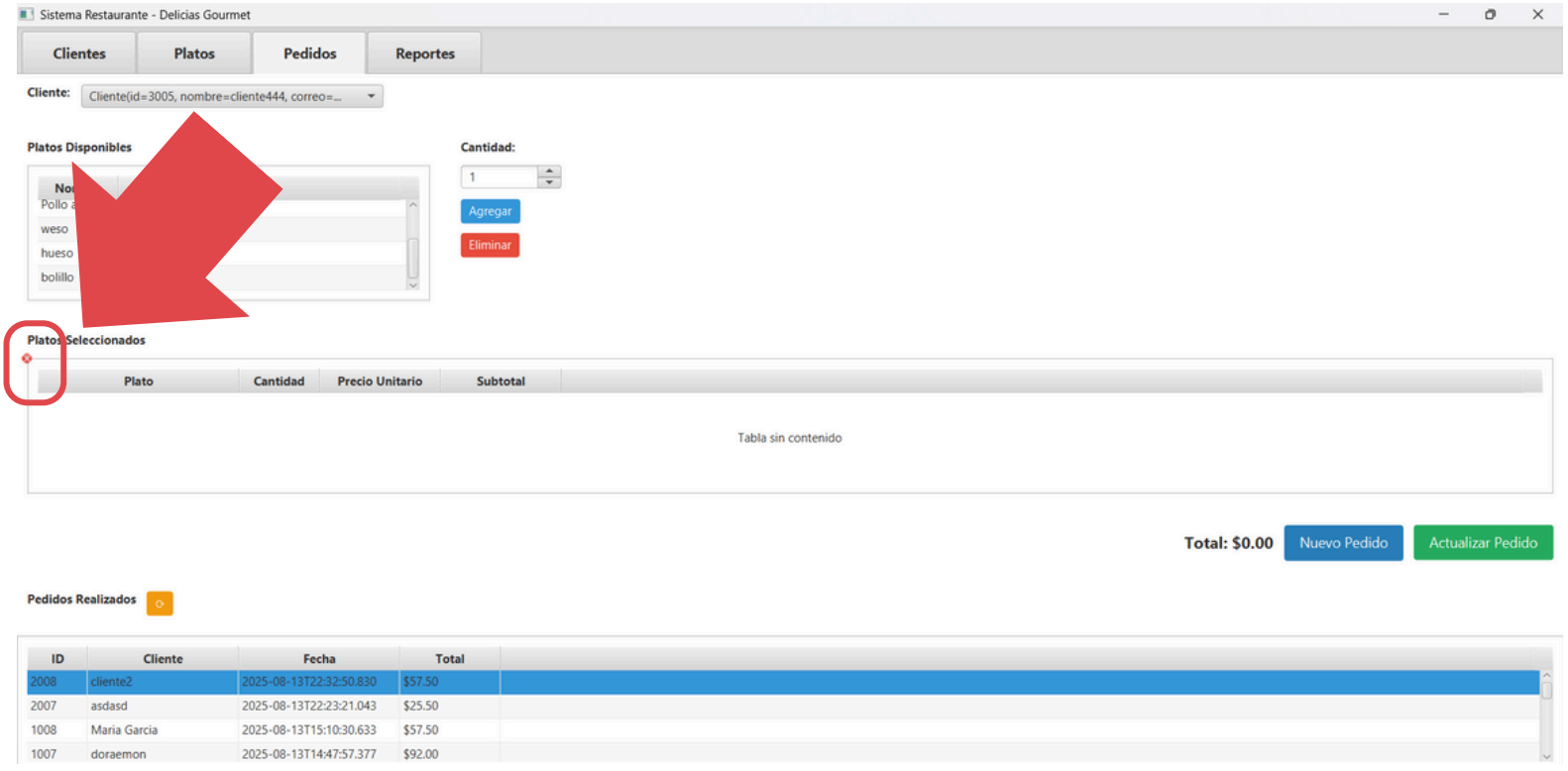
Como vemos bolillo ya no existe, de igual manera aqui no se aprecia, pero si un cliente ya habia pedido lo que eliminamos se borra de ahi mismo

IMPORTANTE, AQUI PRESIONAR EL BOTON DE RELOAD, EL AMARILLO



5. Intentar crear un pedido sin platillos → debe dar error.

Si bien no es muy evidente a primera vista, al intentar ingresar un pedido sin platillos podemos apreciar como una perqueña cruceta se coloca sobre el platillos seleccioandos, y no continua con el proceso



Manual de usuario

🛠️ Instrucciones Generales de Uso

1. Requisitos del Sistema

Antes de ejecutar el sistema, asegúrate de cumplir con los siguientes requisitos:

- Java instalado (versión 8 o superior)
- Entorno de desarrollo como Eclipse, NetBeans o IntelliJ (opcional)
- Conexión a una base de datos configurada (MySQL o la que esté especificada en `package com.restaurante.utils;`)
- Archivo de base de datos importado (por ejemplo, schema.sql)

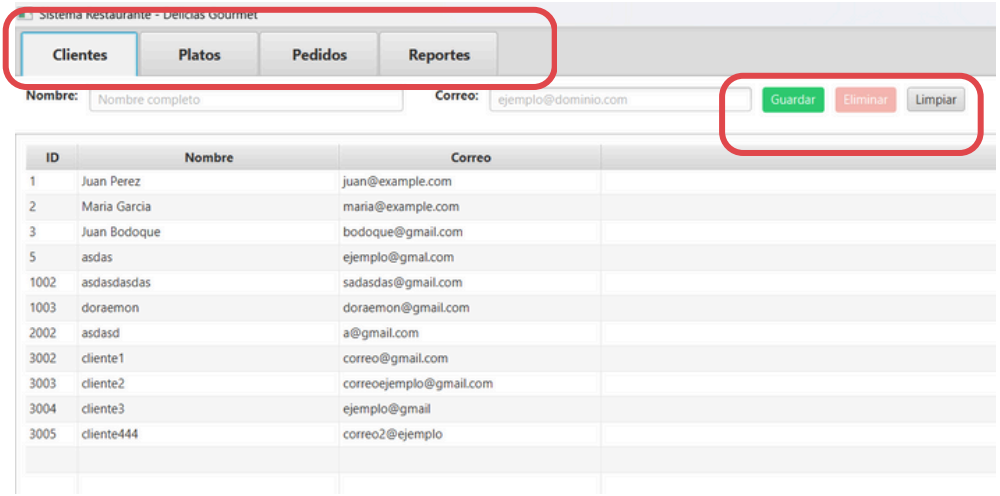
2. Inicio del Sistema

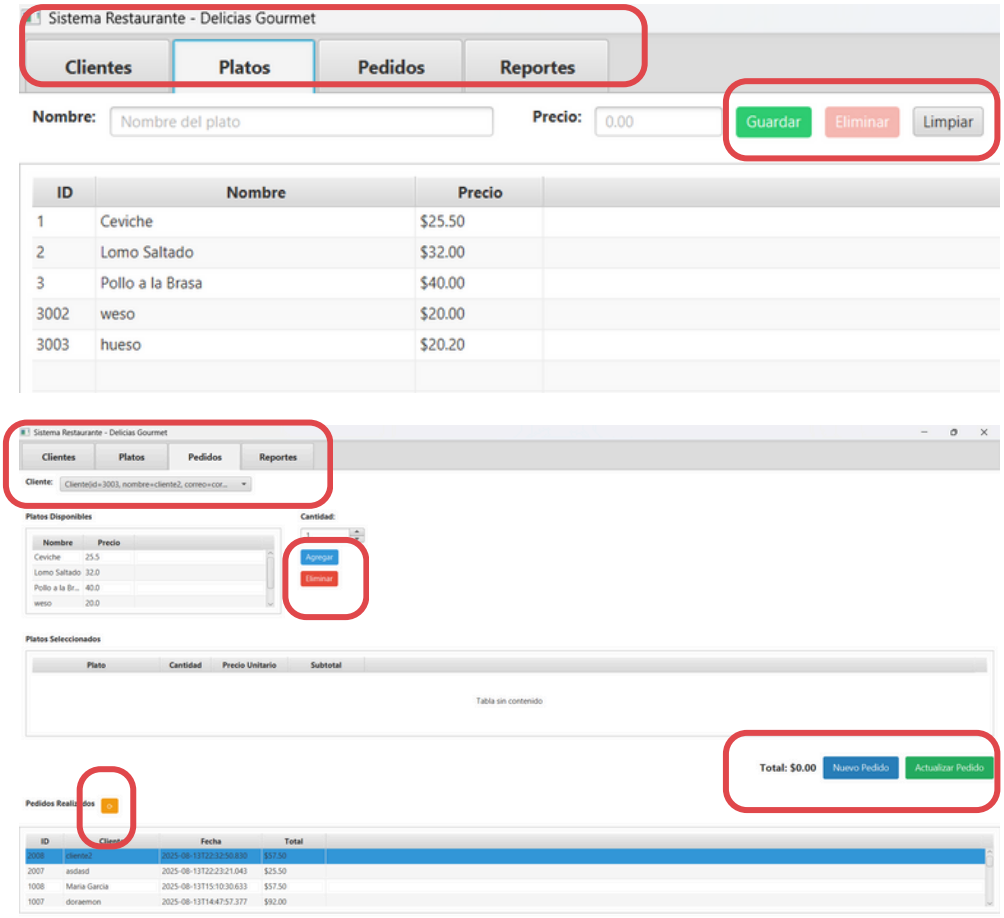
Para ejecutar el sistema:

1. Abre tu entorno de desarrollo o terminal.
2. Compila y ejecuta la clase principal AppMain.java.(ubicado en package com.restaurante;)
3. Seabrirá una ventana con el **menú de gestión de clientes**, desde donde puedes acceder alagestión de platillos o pedidos .

3. Navegación General

- Cada módulo cuenta con una interfaz clara y campos organizados.
- Las acciones están disponibles mediante botones: **Guardar**, **Eliminar**, **Limpiar**, entre otros.
- Puedes navegar entre pantallas usando los botones **"del contenedor superior"**.

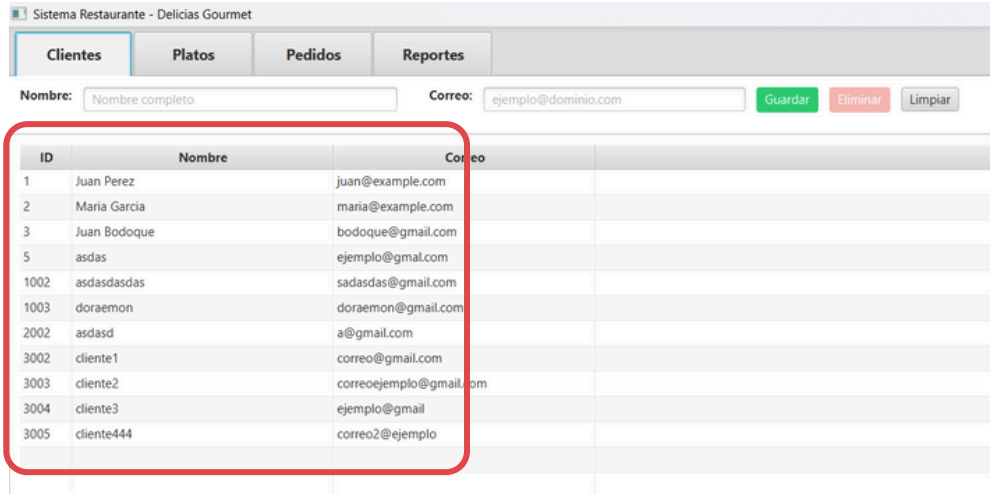


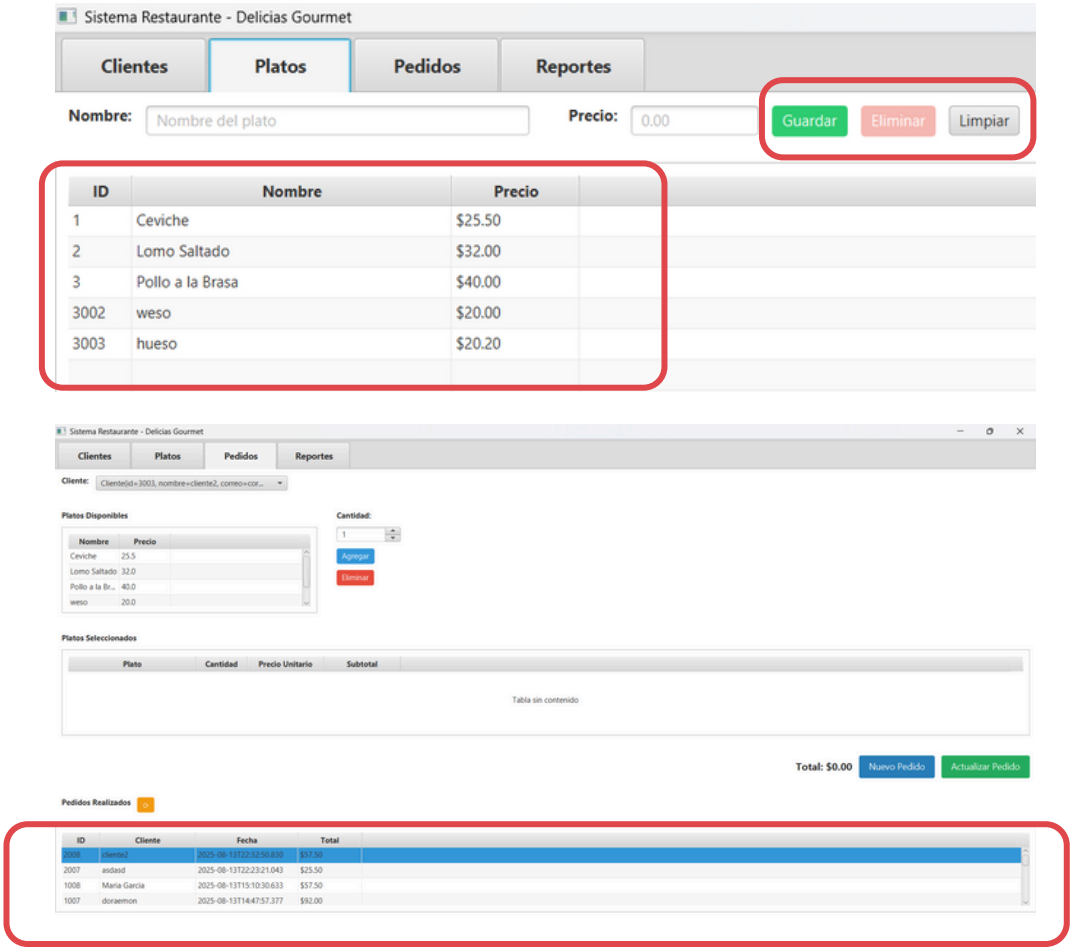


EL caso de pedidos es especial, cuanta con su boton de recarga, este lo tienes que actualizar, si bien los cambios se reflejan en la base de datos, es necesario hacelrlo garficamente, luego el de nuevo pedido funciona como nuesto “limpiar”

4. Estados de los Registros

- Tanto en clientes como en platillos se encuentra una lista con los clientes y una con los platos respectivamente, la de pedidos se muestra la lista de los pedidos





5. Operaciones CRUD

platilos

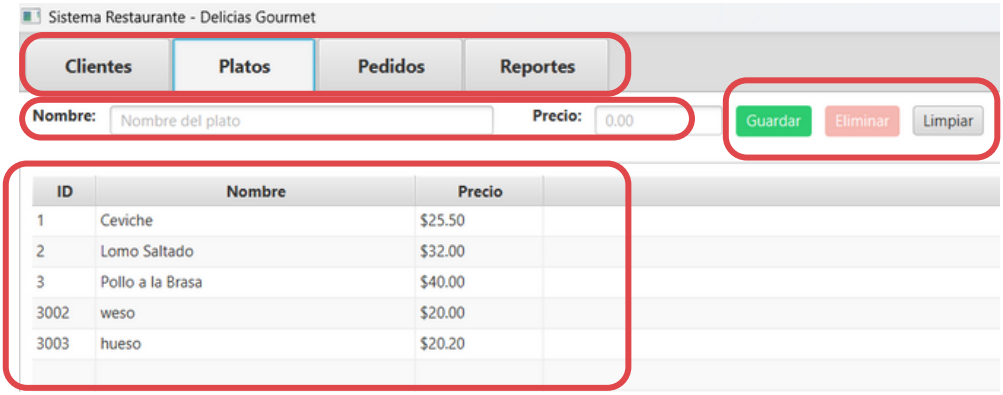
Elementos de la interfaz:
Campo “Nombre”
Permite ingresar el nombre del plato a registrar o editar.

Campo “Precio”
Permite ingresar el precio del plato.

Botón “Guardar” (verde)
Si los campos están vacíos → crea un nuevo plato en la base de datos.
Si hay un plato seleccionado en la tabla → actualiza sus datos.

Botón “Eliminar” (rojo)
Elimina de forma permanente el plato seleccionado en la tabla.
.
Botón “Limpiar” (gris)
Limpia los campos “Nombre” y “Precio” para permitir un nuevo registro.
También deselecciona cualquier plato marcado en la tabla.

Tabla de Platos
Muestra todos los platos almacenados en la base de datos con sus columnas:
ID: identificador único.
Nombre: nombre del plato.
Precio: valor en moneda local.
Permite seleccionar un plato para editar o eliminar.



5. Operaciones CRUD

Clientes

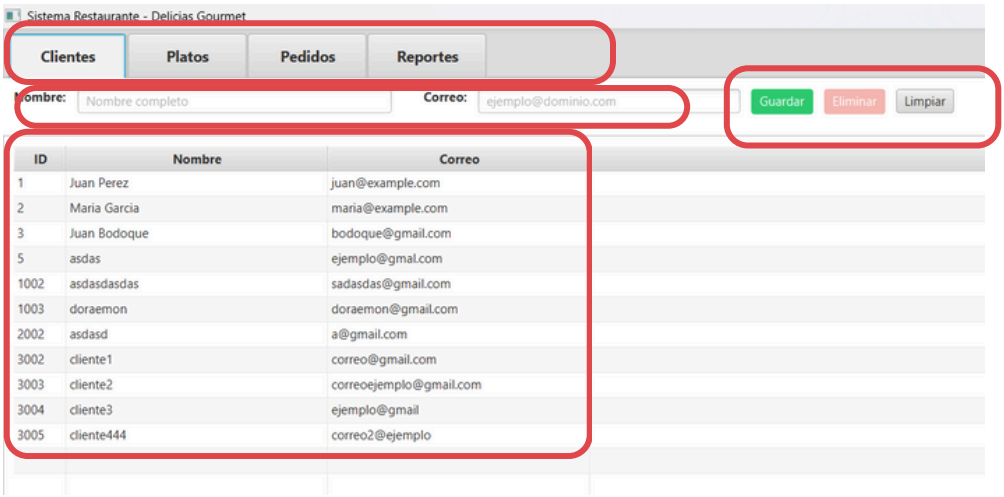
Elementos de la interfaz:
Campo “Nombre”
Permite ingresar el nombre del usuario a registrar o editar.

Campo “Correo”
Permite ingresar el correo del usuario

Botón “Guardar” (verde)
Si los campos están vacíos → crea un nuevo cliente en la base de datos.
Si hay un cliente seleccionado en la tabla → actualiza sus datos.

Botón “Eliminar” (rojo)
Elimina de forma permanente el cliente seleccionado en la tabla.
.
Botón “Limpiar” (gris)
Limpia los campos “Nombre” y “correo” para permitir un nuevo registro.
También deselecciona cualquier plato marcado en la tabla.

Tabla de Platos
Muestra todos los platos almacenados en la base de datos con sus columnas:
ID: identificador único.
Nombre: nombre del plato.
correo: su correo electronico
Permite seleccionar un plato para editar o eliminar.



Pantalla de Gestión de Pedidos
Esta sección permite registrar, actualizar y visualizar los pedidos de los clientes, seleccionando los platos y cantidades que se deseen.
Elementos de la interfaz:
1. Selector de Cliente
Muestra una lista desplegable con todos los clientes registrados.
Al elegir uno, se asocia el pedido al cliente seleccionado.

2. Platos Disponibles
Lista de todos los platos registrados en el sistema con su nombre y precio.
Se selecciona un plato para añadirlo al pedido.

3. Cantidad
Campo numérico para indicar cuántas unidades del plato seleccionado se añadirán.

4. Botón “Agregar” (azul)
Añade el plato seleccionado con la cantidad indicada a la lista de Platos Seleccionados.

5. Botón “Eliminar” (rojo)
Elimina un plato de la lista de Platos Seleccionados antes de guardar el pedido.

6. Platos Seleccionados
Tabla que muestra los platos añadidos al pedido actual con:
Plato
Cantidad
Precio Unitario
Subtotal (Cantidad × Precio Unitario)

7. Total
Muestra el monto total del pedido calculado automáticamente.

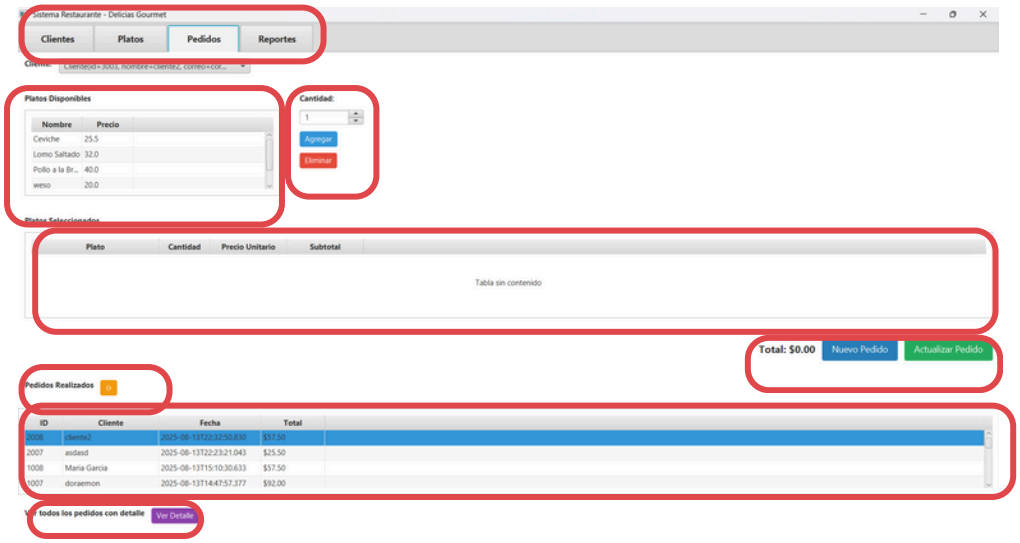
8. Botón “Nuevo Pedido” (azul)
Guarda un nuevo pedido en la base de datos con los platos y cantidades indicadas.

9. Botón “Actualizar Pedido” (verde)
Modifica un pedido existente seleccionado en la lista de Pedidos Realizados.

10. Pedidos Realizados
Tabla con los pedidos existentes, que muestra:
ID del pedido
Cliente
Fecha del pedido
Total del pedido
Permite seleccionar un pedido para actualizarlo o revisarlo.

11. Botón de Recarga (icono naranja con flecha circular)
Actualiza la lista de Pedidos Realizados mostrando los más recientes.

12. Botón “Ver Detalle” (morado)
Muestra un informe detallado del pedido seleccionado, incluyendo todos los platos, cantidades y precios.



6. Recomendaciones

- Actualiza la pantalla de pedido simpre que se vaya a usar
- No dejes campos vacíos al agregar o editar registros.
- Para evitar errores, selecciona un registro antes de usar los botones de acción.

Contacto

Para dudas, problemas técnicos o sugerencias sobre el sistema de Gestión de restaurante, comuníquese con el equipo de desarrollo:

Equipo de Desarrollo: Number 81: Superdreadnought Rail Cannon Super Dora
Correo: hectrgarc00@gmail.com
Teléfono:773 680 14 69
Horario de atención: Lunes a Viernes de 9:00 a 18:00 hrs

Por favor incluya en su mensaje una descripción detallada del problema y, de ser posible, una captura de pantalla del error.

Conclusiones

Laimplementación del módulo de gestión de pedidos en el sistema Delicias Gourmet ha permitido consolidar una interfaz intuitiva, funcional y adaptable a las necesidades del restaurante. Se logró integrar la creación, edición y visualización detallada de pedidos,

garantizando una experiencia fluida tanto para el usuario como para el equipo administrativo.

Además, se fortaleció la estructura modular del sistema, facilitando futuras extensiones como reportes, filtros avanzados o exportación de datos. El enfoque práctico y colaborativo durante el desarrollo permitió resolver desafíos técnicos con eficiencia y mantener una comunicación clara entre capas de servicio, controlador y vista.

Este avance representa no solo una mejora operativa, sino también un paso firme hacia la profesionalización del sistema, manteniendo siempre el sabor humano que distingue a Delicias Gourmet.

hay que mencionar que fue un camino trillado durante 3 dias, siendo dedicado cada uno a una cosa especifica, en el primer dia intente dejar la estructura correcta para poder trabajar, el segundo dia fue la creacion de codigo dentro de los modulos, el ultimo dia siendo la conexion a base de datos.

Como mencione todo fue algo “caotico”, en primer lugar fue hacer que el pom.xml, pudiera importar de manera correcta las dependencias, la construcción fue la parte enredada, porque en momentos el programa no hacia lo que deberia entonces son cosas que se deberian de corregir, finalmente resaltamos la conexion a base de datos, se opto por un contenedor en Docker debido a al inconveniente que tenia la conexion local de SQL Server