

Universidad Mariano Gálvez de Guatemala

Facultad: Ingeniería en Sistemas de Información y Ciencias de la Computación

Catedrático: David Álvarez

Curso: Programación II

1966

Proyecto Final

CONOCEREIS LA VERDAD
Y LA VERDAD OS HARA LIBRES

Brayan Omar Hernández Cabrera 9941-24-2010

Ricardo Alfonso Chan López 9989 21 7614

Emily Yamileth Herrera Monterroso 9941-24-1489

INDICE

Caratula	1
Índice	2
Introducción	3
Desarrollo del tema	4-28
Conclusiones	29

INTRODUCCIÓN

El presente proyecto consiste en el desarrollo de un Sistema de Control Automotriz orientado a la gestión integral de un taller mecánico. El sistema fue implementado en el entorno IntelliJ IDEA utilizando Java como lenguaje principal y MariaDB como sistema gestor de base de datos relacional.

El sistema cuenta con módulos independientes e interconectados que permiten administrar clientes, vehículos, servicios, repuestos y facturación, ofreciendo además un módulo de reportes de historial de reparaciones por vehículo. Esta arquitectura modular facilita el mantenimiento y la escalabilidad del software, garantizando la integridad de la información y una operación eficiente.

El backend se implementa siguiendo principios de Programación Orientada a Objetos (POO) y una arquitectura en capas (controlador — servicio — repositorio) así como manejamos el front-end con función principal de consumir los endpoints expuestos por el backend para demostrar flujos clave como login, CRUD y generación de reportes.

Sistema de Control del Taller Automotriz

Controladores

Cliente Controlador

El archivo **ClienteControlador.java** es el encargado de **recibir las peticiones** que llegan desde el navegador, el celular o cualquier programa que quiera usar tu API. Su trabajo es **escuchar las solicitudes** que hacen los usuarios y **decidir qué hacer** con ellas.

Piensa en él como **el recepcionista del taller**:

- Atiende a quien llega (una petición HTTP),
- Escucha qué quiere (crear, ver, editar o borrar un cliente)
- Y le pasa el mensaje al área que se encarga (el servicio).

@RestController

Le dice a Spring que esta clase será un “controlador REST”.

Eso significa que responderá con **datos en formato JSON**, no con páginas web.

@RequestMapping("/api/clientes")

Define la **ruta base**.

Todo lo que se haga dentro de esta clase empezará con esa dirección.

Por ejemplo:

- GET /api/clientes → lista todos los clientes
- POST /api/clientes → crea un cliente
- GET /api/clientes/5 → busca al cliente con ID 5

@Autowired private ClienteServicio clienteServicio;

Aquí se **conecta** el controlador con la parte del sistema que hace el trabajo real (el servicio).

```
1 package umg.edu.proyectofinaltaller mecanico.controlador;
2
3 import umg.edu.proyectofinaltaller mecanico.modelo.Cliente;
4 import umg.edu.proyectofinaltaller mecanico.servicio.ClienteServicio;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.*;
7
8 import java.util.List;
9
10 @RestController
11 @RequestMapping("/api/clientes")
12 public class ClienteControlador {
13
14     @Autowired
15     private ClienteServicio clienteServicio;
16
17     @GetMapping("/")
18     public List<Cliente> listar() { return clienteServicio.listarTodos(); }
19
20     @PostMapping("/")
21     public Cliente guardar(@RequestBody Cliente cliente) { return clienteServicio.guardar(cliente); }
22
23     @GetMapping("/{id}")
24     public Cliente obtenerPorId(@PathVariable Long id) { return clienteServicio.buscarPorId(id); }
25
26     @PutMapping("/{id}")
27     public Cliente actualizar(@PathVariable Long id, @RequestBody Cliente cliente) {
28         return clienteServicio.actualizar(id, cliente);
29     }
30
31     @DeleteMapping("/{id}")
32     public void eliminar(@PathVariable Long id) { clienteServicio.eliminar(id); }
```

Factura Controlador

Este código es el controlador de las facturas del sistema del taller mecánico. Su función principal es recibir las peticiones del usuario (por ejemplo: ver facturas, crear una nueva, borrar una, etc.) y comunicarse con la parte del sistema que sabe cómo hacerlo (el servicio FacturaServicio).

Podemos decir que este controlador es como el encargado del área de facturación del taller:

- Recibe las solicitudes (crear, ver, modificar o eliminar facturas).
- Las envía al servicio que sabe cómo hacerlo.
- Y devuelve una respuesta al usuario en formato JSON

@RestController

Indica que esta clase será un **controlador REST**, es decir, responderá con datos (JSON) y no con páginas web.

@RequestMapping("/api/facturas")

Define la **ruta base** de este controlador. Todas las operaciones relacionadas con las facturas empezarán con esta dirección.

@Autowired private FacturaServicio facturaServicio;

Aquí el controlador se conecta con el servicio (FacturaServicio) que contiene las operaciones para trabajar con las facturas.

Es como tener un asistente al que le pide hacer las tareas de guardar, buscar o eliminar.

@RequestMapping("/api/servicios")

Define la ruta base del controlador. Todas las operaciones que se hagan dentro de esta clase usarán esta ruta como punto de inicio.

@Autowired private ServicioServicio servicioServicio;

Aquí se establece la conexión con el servicio lógico (ServicioServicio).

Gracias a la anotación @Autowired, Spring Boot inyecta automáticamente la clase que se encarga de la lógica de negocio, permitiendo que el controlador solo se concentre en recibir y devolver las peticiones.

```

1 package umg.edu.proyectofinaltallermecanico.controlador;
2
3 > import ...
4
5 @RestController
6 @RequestMapping("/api/facturas")
7 public class FacturaControlador {
8
9     @Autowired
10     private FacturaServicio facturaServicio;
11
12     @GetMapping("/")
13     public List<Factura> listar() { return facturaServicio.listarTodos(); }
14
15     @PostMapping("/")
16     public Factura guardar(@RequestBody Factura factura) { return facturaServicio.guardar(factura); }
17
18     @GetMapping("/{id}")
19     public Factura obtenerPorId(@PathVariable Long id) { return facturaServicio.buscarPorId(id); }
20
21     @PutMapping("/{id}")
22     public Factura actualizar(@PathVariable Long id, @RequestBody Factura factura) {
23         return facturaServicio.actualizar(id, factura);
24     }
25
26     @DeleteMapping("/{id}")
27     public void eliminar(@PathVariable Long id) { facturaServicio.eliminar(id); }
28 }

```

Servicio Controlador

El archivo ServicioControlador.java es el encargado de manejar todas las peticiones HTTP relacionadas con los servicios del taller mecánico. Su función principal es recibir las solicitudes del usuario (por ejemplo, listar los servicios disponibles, crear uno nuevo, actualizarlo o eliminarlo) y enviar esas solicitudes al servicio lógico (ServicioServicio), que realiza las operaciones directamente con la base de datos.

Podemos compararlo con el encargado del área de servicios del taller:

- Recibe las órdenes de los clientes o del sistema,
- Lo pasa al departamento correcto (el servicio),
- Y devuelve una respuesta con el resultado (en formato JSON)

@RestController

Indica que esta clase es un **controlador REST**.

Esto significa que responde con datos (en formato JSON) y no con páginas HTML.

```

1 import umg.edu.proyectofinaltallermecanico.modelo.Servicio;
2 import umg.edu.proyectofinaltallermecanico.servicio.ServicioServicio;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.web.bind.annotation.*;
5
6 import java.util.List;
7
8 @RestController
9 @RequestMapping("/api/servicios")
10 public class ServicioControlador {
11
12     @Autowired
13     private ServicioServicio servicioServicio;
14
15     @GetMapping("/")
16     public List<Servicio> listar() { return servicioServicio.listarTodos(); }
17
18     @PostMapping("/")
19     public Servicio guardar(@RequestBody Servicio servicio) { return servicioServicio.guardar(servicio); }
20
21     @GetMapping("/{id}")
22     public Servicio obtenerPorId(@PathVariable Long id) { return servicioServicio.buscarPorId(id); }
23
24     @PutMapping("/{id}")
25     public Servicio actualizar(@PathVariable Long id, @RequestBody Servicio servicio) {
26         return servicioServicio.actualizar(id, servicio);
27     }
28
29     @DeleteMapping("/{id}")
30     public void eliminar(@PathVariable Long id) { servicioServicio.eliminar(id); }
31 }

```

Servicio Controlador

Este código es un controlador de vehículos hecho con Spring Boot. Su trabajo es recibir las peticiones del usuario (por ejemplo, desde Postman o un frontend) y comunicarse con el servicio que maneja la lógica del programa.

@RestController

Lo que significa que responde a solicitudes web (HTTP) y devuelve datos en formato JSON.

@RequestMapping("/api/vehiculos")

Se indica que todas las rutas (endpoints) de esta clase empezarán con esa dirección.

Por ejemplo:

- /api/vehiculos
- /api/vehiculos/5

Esto hace que el controlador tenga acceso a la clase VehiculoServicio, donde están las funciones reales para guardar, eliminar, actualizar o listar vehículos.

A partir de ahí, hay varios métodos que responden a distintos tipos de peticiones:

1. **)-usa @GetMapping**
 - a. Muestra todos los vehículos registrados.
 - b. Es como cuando haces “ver todos los datos”.
2. **guardar()-usa @PostMapping**
 - a. Sirve para **agregar un vehículo nuevo**.
 - b. Recibe la información en formato JSON (por ejemplo, marca, modelo, año).
3. **obtenerPorId() -usa @GetMapping("/{id}")**
 - a. Busca un vehículo específico por su **ID** (número identificador).
 - b. Ejemplo: /api/vehiculos/3 muestra el vehículo con ID 3.
4. **actualizar()-usa @PutMapping("/{id}")**
 - a. Permite **modificar** los datos de un vehículo existente.
 - b. Se envía el ID en la URL y los nuevos datos en el cuerpo del mensaje.
5. **eliminar() -usa @DeleteMapping("/{id}")**
 - a. **Borra** el vehículo que tenga ese ID.

Vista

```
1 ClienteControlador.java 2 FacturaControlador.java 3 ServicioControlador.java 4 VehiculoControlador.java
5 package umg.edu.proyectoFinalTallerMecanica.controlador;
6
7 import java.util.List;
8
9 @RestController
10 @RequestMapping("/api/vehiculos")
11 public class VehiculoControlador {
12
13     @Autowired
14     private VehiculoServicio vehiculoServicio;
15
16     @GetMapping("/")
17     public List<Vehiculo> listar() { return vehiculoServicio.listarTodos(); }
18
19     @PostMapping("/")
20     public Vehiculo guardar(@RequestBody Vehiculo vehiculo) { return vehiculoServicio.guardar(vehiculo); }
21
22     @GetMapping("/{id}")
23     public Vehiculo obtenerPorId(@PathVariable Long id) { return vehiculoServicio.buscarPorId(id); }
24
25     @PutMapping("/{id}")
26     public Vehiculo actualizar(@PathVariable Long id, @RequestBody Vehiculo vehiculo) {
27         return vehiculoServicio.actualizar(id, vehiculo);
28     }
29
30     @DeleteMapping("/{id}")
31     public void eliminar(@PathVariable Long id) { vehiculoServicio.eliminar(id); }
32 }
```

Controlador

Este código sirve para mostrar una página web cuando alguien entra al sitio, por ejemplo, desde el navegador.

La anotación `@Controller` le dice a Spring que esta clase no devuelve datos en JSON (como el otro controlador), sino que muestra páginas HTML.

Dentro de la clase hay un método llamado `inicio()` que usa `@GetMapping("/")`, lo que significa que se ejecuta cuando entras a la página principal.

```
1 package umg.edu.proyectofinaltallermecanico.controlador;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.GetMapping;
6
7 @Controller
8 public class VistaControlador {
9
10     @GetMapping("/")
11     public String inicio(Model model) {
12         model.addAttribute("mensaje", "¡Bienvenido al Taller Mecánico!");
13         return "index"; // Carga index.html desde /templates
14     }
15 }
```

MODELO

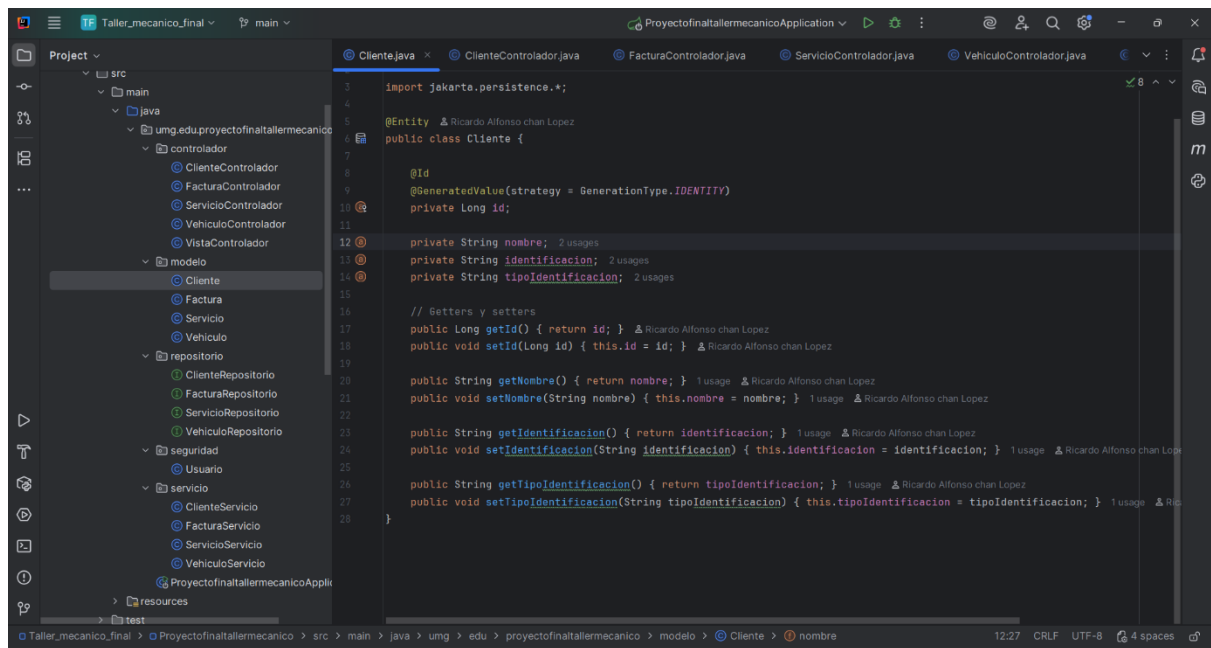
El apartado modelo encontraremos un conjunto de clases que representan las entidades principales del sistema. En él se definen los atributos y estructuras de los datos que maneja la aplicación, aplicando principios de Programación Orientada a Objetos (POO). Este apartado es la base para la comunicación con la base de datos y es utilizado por otras capas del proyecto, como el controlador y el servicio, para gestionar la información de manera organizada y coherente.

En nuestro proyecto encontraremos las siguientes clases:

1. Clientes
2. Factura
3. Servicio
4. Vehículo

Veamos una breve descripción de cada una:

1. Clientes



La primera imagen muestra la clase Cliente que representa a las personas que utilizan los servicios del taller. Contiene atributos como el nombre, la identificación y el tipo de identificación en nuestro caso ingresamos el código de identificación único DPI.

Cada cliente tiene un ID único generado automáticamente. Esta clase sirve como base para relacionar a los clientes con sus vehículos, servicios y facturas en la base de datos.

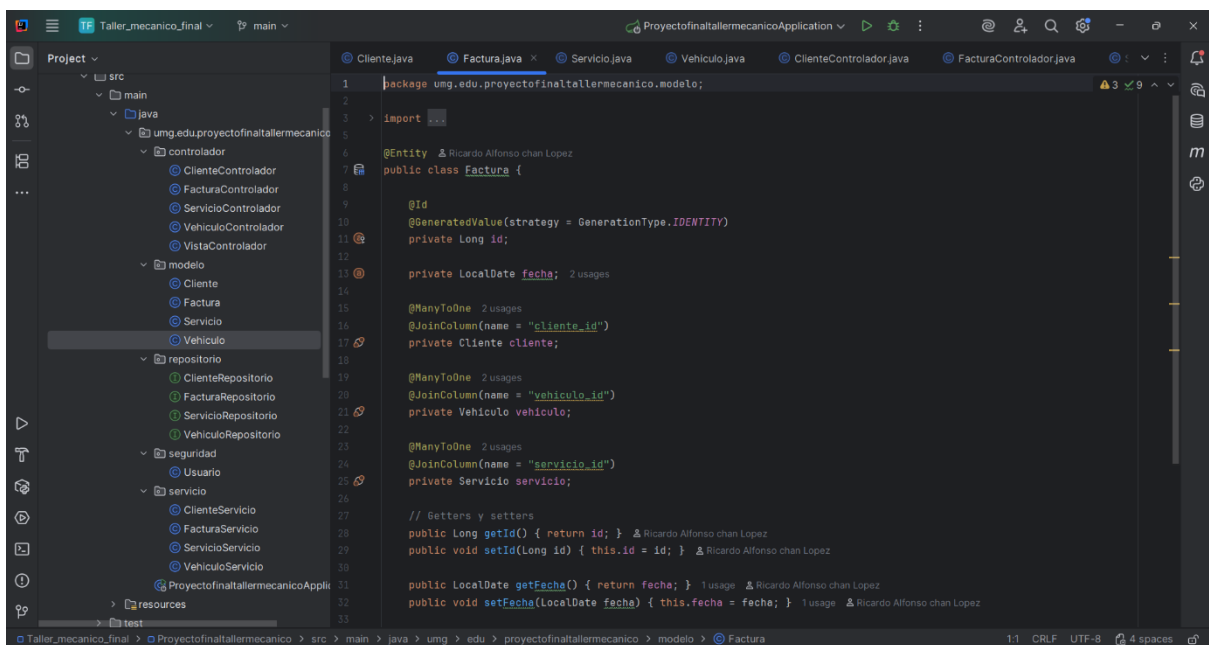
Esta clase está anotada con `@Entity`, lo que indica que representa una entidad persistente dentro de la base de datos y se gestionará mediante JPA/Hibernate.

Cuenta con los siguientes elementos técnicos:

- `@Id` y `@GeneratedValue(strategy = GenerationType.IDENTITY)`: definen la llave primaria y permiten que el campo `id` se genere automáticamente por el motor de la base de datos.
- Atributos: `nombre`, `identificacion` y `tipoidentificacion`, que almacenan los datos principales del cliente.
- Métodos Getters y Setters: implementan el principio de encapsulamiento, permitiendo el acceso controlado a los atributos.

En conjunto, esta clase modela la tabla `Cliente` dentro del esquema del sistema.

2. Factura



La segunda imagen muestra la clase `Factura` que gestiona la información de las facturas emitidas por el taller automotriz.

Incluye la fecha y las relaciones con las otras entidades: `Cliente`, `Vehículo` y `Servicio`, utilizando anotaciones `@ManyToOne` lo quiere decir que es un JPA que establece la relación de muchos a uno entre clases de entidad demostrando que una entidad puede relacionarse con una instancia de otra entidad para indicar que una factura pertenece a un cliente, un vehículo y un servicio específicos.

Su función principal es registrar las transacciones realizadas por el usuario en el sistema.

La clase Factura también está anotada con `@Entity`, lo que la convierte en una tabla gestionada por JPA.

Sus características técnicas son:

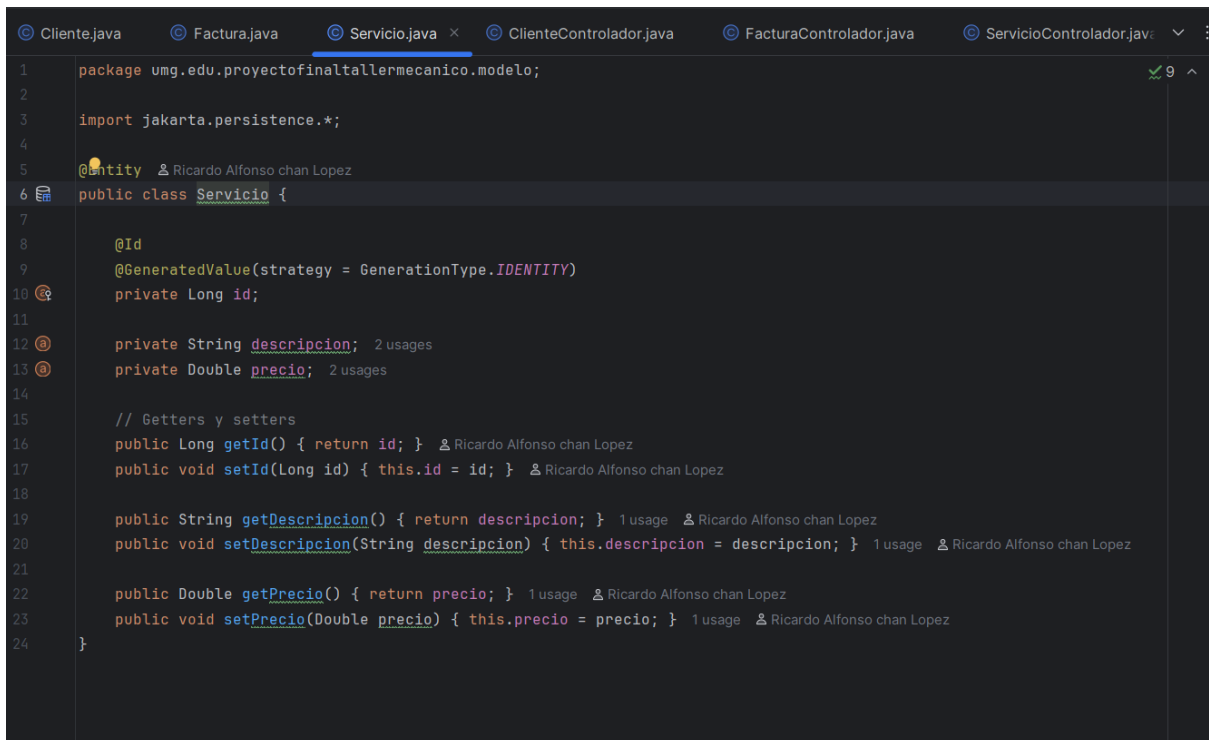
- Definido con `@Id` y `@GeneratedValue(strategy = GenerationType.IDENTITY)`, que genera un identificador único para cada factura.
- Integramos el tipo `LocalDate`, utilizado para registrar la fecha de emisión de la factura.

Relaciones con otras entidades:

- `@ManyToOne` con `Cliente`, `Vehiculo` y `Servicio`, usando `@JoinColumn` para especificar las claves foráneas (`cliente_id`, `vehiculo_id`, `servicio_id`).
- Métodos Getters y Setters: son los que permiten acceder y modificar los datos de cada campo.

Técnicamente, esta clase consolida la información de facturación enlazando las demás entidades mediante relaciones relacionales de tipo muchos a uno.

3. Servicio



```
1 package umg.edu.proyectofinaltallermechanico.modelo;
2
3 import jakarta.persistence.*;
4
5 @Entity
6 public class Servicio {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private Long id;
11
12    private String descripcion;
13    private Double precio;
14
15    // Getters y setters
16    public Long getId() { return id; }
17    public void setId(Long id) { this.id = id; }
18
19    public String getDescripcion() { return descripcion; }
20    public void setDescripcion(String descripcion) { this.descripcion = descripcion; }
21
22    public Double getPrecio() { return precio; }
23    public void setPrecio(Double precio) { this.precio = precio; }
24 }
```

La tercera imagen muestra la clase Servicio la cual define los servicios que ofrece el taller, como mantenimiento o reparación.

Contiene atributos como la descripción y el precio, los cuales son utilizados al generar facturas.

Cada servicio tiene su propio ID único y se relaciona con la factura al momento de registrar un trabajo realizado.

Representa los diferentes trabajos o reparaciones que ofrece el taller.

Acá podemos encontrar los aspectos técnicos siguientes:

@Entity el cual define que es una entidad persistente en la base de datos.

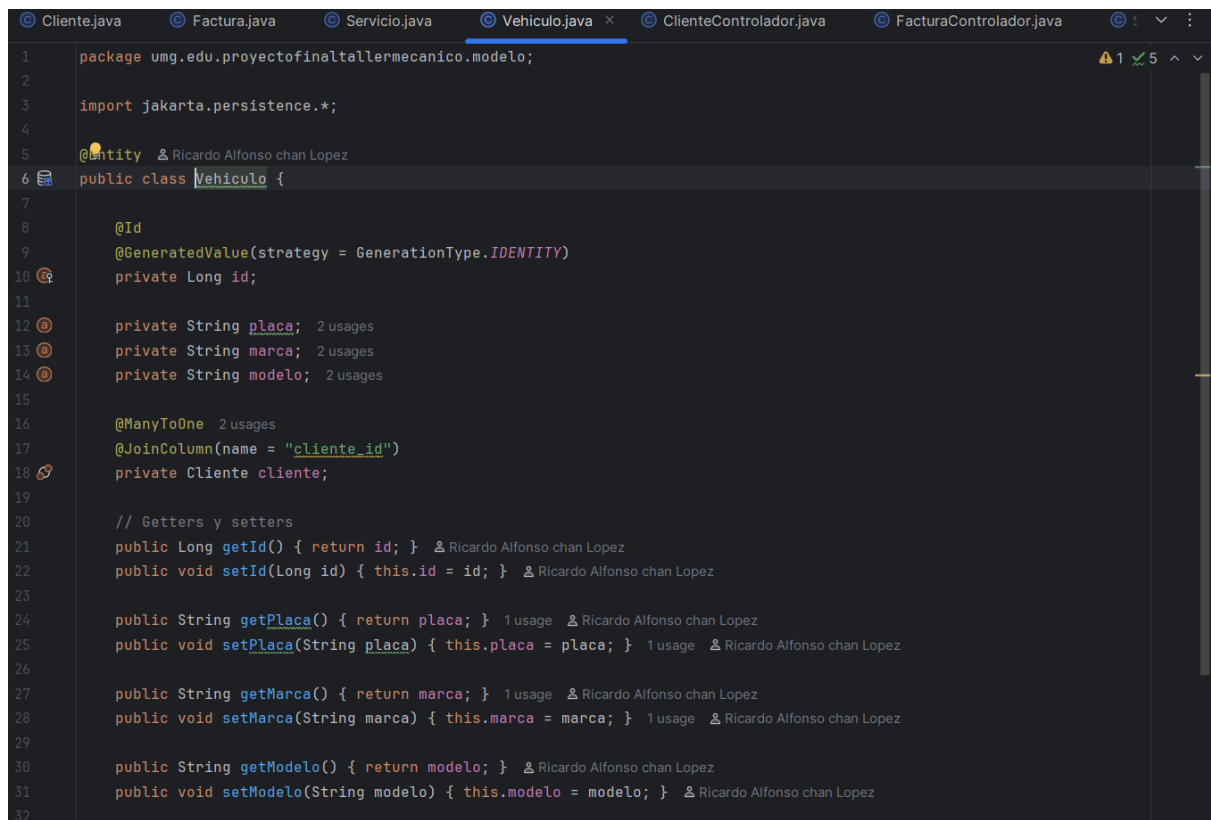
@Id y @GeneratedValue quienes generan un identificador único (id) para cada servicio.

En los tributos principales podemos encontrar la descripción o detalle del servicio y precio asociado.

Métodos Getters y Setters que permiten la manipulación de los atributos de manera segura y encapsulada.

En el modelo relacional, esta clase se asocia con Factura mediante una relación ManyToOne, indicando que un servicio puede estar incluido en varias facturas.

4. Vehículo



```
1 package umg.edu.proyectofinaltallermechanico.modelo;
2
3 import jakarta.persistence.*;
4
5 @Entity
6 public class Vehiculo {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private Long id;
11
12    private String placa;
13    private String marca;
14    private String modelo;
15
16    @ManyToOne
17    @JoinColumn(name = "cliente_id")
18    private Cliente cliente;
19
20    // Getters y setters
21    public Long getId() { return id; }
22    public void setId(Long id) { this.id = id; }
23
24    public String getPlaca() { return placa; }
25    public void setPlaca(String placa) { this.placa = placa; }
26
27    public String getMarca() { return marca; }
28    public void setMarca(String marca) { this.marca = marca; }
29
30    public String getModelo() { return modelo; }
31    public void setModelo(String modelo) { this.modelo = modelo; }
32}
```

La clase Vehículo representa los automóviles registrados por los clientes.

Incluye atributos como placa, marca, modelo y una relación con la clase Cliente usando @ManyToOne, lo que indica que un cliente puede tener varios vehículos.

Esta entidad permite asociar fácilmente cada servicio y factura al vehículo correspondiente.

Está mapeada como entidad de base de datos mediante @Entity.

Podemos encontrar los elementos técnicos relevantes siguientes:

@Id y @GeneratedValue(strategy = GenerationType.IDENTITY): definen el campo id como clave primaria autogenerada.

Atributos: placa, marca, modelo, los cuales representan la información esencial de cada automóvil.

Relación con Cliente: usa la anotación @ManyToOne junto con @JoinColumn(name = "cliente_id"), indicando que varios vehículos pueden pertenecer a un mismo cliente

Métodos Getters y Setters: implementan el acceso controlado a los campos, cumpliendo con las buenas prácticas de POO.

Desde un punto de vista técnico, esta clase establece la relación entre los vehículos y sus propietarios (clientes) dentro de la base de datos.

REPOSITORIO

Este paquete se encarga de la comunicación con la base de datos, es decir, de guardar, buscar, actualizar o eliminar registros.

1. ClienteRepositorio



```
1 package umg.edu.proyectofinaltallermecanico.repositorio;
2
3 import umg.edu.proyectofinaltallermecanico.modelo.Cliente;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface ClienteRepositorio extends JpaRepository<Cliente, Long> {
7 }
```

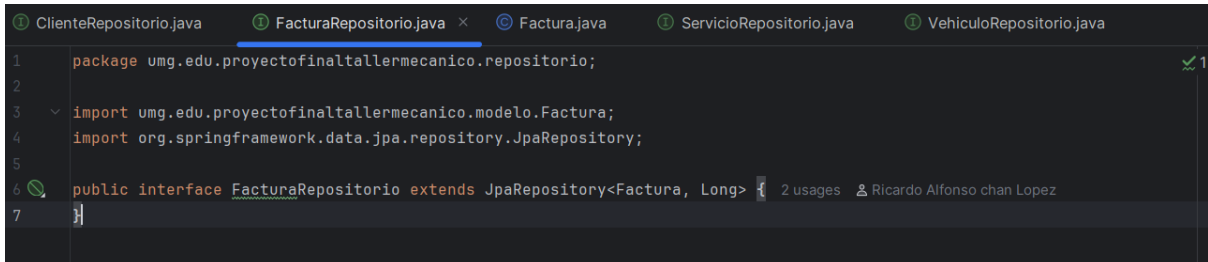
La clase Cliente, que es la entidad (tabla) con la que este repositorio trabajará.

La interfaz JpaRepository, proporcionada por Spring Data JPA, que ofrece métodos listos para usar (como save(), findById(), findAll(), deleteById(), etc.).

```
public interface ClienteRepositorio extends JpaRepository<Cliente, Long> {
}
```

Al extender de JpaRepository, Spring Boot genera automáticamente las operaciones CRUD (Crear, Leer, Actualizar, Eliminar).

2. FacturaRepository



```
1 package umg.edu.proyectofinaltallermecanico.repositorio;
2
3 import umg.edu.proyectofinaltallermecanico.modelo.Factura;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface FacturaRepository extends JpaRepository<Factura, Long> {
7 }
```

Es la entidad que representa la tabla de facturas en la base de datos.

JpaRepository: es la interfaz de Spring que proporciona métodos genéricos para realizar operaciones de persistencia (guardar, buscar, actualizar y eliminar).

```
public interface FacturaRepository extends JpaRepository<Factura, Long> {
}
```

Esta línea declara una interfaz llamada FacturaRepository que extiende JpaRepository.

Los parámetros genéricos indican:

Factura: la clase o entidad que gestionará este repositorio.

Long: el tipo de dato del campo identificador (id) de la entidad.

Spring Boot detecta esta interfaz automáticamente gracias al componente de inyección de dependencias.

Al extender JpaRepository, el repositorio hereda todos los métodos CRUD.

3. ServicioRepository



```
1 package umg.edu.proyectofinaltallermecanico.repositorio;
2
3 import umg.edu.proyectofinaltallermecanico.modelo.Servicio;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface ServicioRepository extends JpaRepository<Servicio, Long> {
7 }
```

Se encarga de gestionar la persistencia de datos de las entidades del sistema a través de Spring Data JPA.

Esta interfaz extiende JpaRepository<Servicio, Long>, lo que significa que:

Trabaja con la entidad Servicio, que representa los servicios ofrecidos por el taller.

Usa un identificador de tipo Long como clave primaria (id).

Gracias a la herencia de JpaRepository, no es necesario implementar manualmente consultas SQL. Spring Boot genera automáticamente todos los métodos comunes.

4. VehiculoRepositorio



```
1 package umg.edu.proyectofinaltallermecanico.repositorio;
2
3 import umg.edu.proyectofinaltallermecanico.modelo.Vehiculo;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface VehiculoRepositorio extends JpaRepository<Vehiculo, Long> {
7 }
```

Esta interfaz extiende JpaRepository<Vehiculo, Long>, lo que significa que:

Administra la entidad Vehiculo, que representa los automóviles registrados por los clientes.

Utiliza un identificador de tipo Long como clave primaria.

Gracias a esta extensión, Spring Boot genera automáticamente las operaciones CRUD (Crear, Leer, Actualizar y Eliminar) sin necesidad de implementar consultas SQL manuales.

SERVICIO

Cliente Servicio

Su función es encargarse de la lógica del negocio relacionada con los clientes: guardar, buscar, listar y actualizar clientes.

@Service:

indica que esta clase es un servicio, es decir, una parte del código que se encarga de la lógica principal del sistema (no de las rutas ni de la base de datos directamente).

ClienteServicio es el nombre de la clase.

@Autowired:

Le dice a Spring que inyecte automáticamente el repositorio (o sea, que lo conecte sin que tú tengas que crear el objeto manualmente).

ClienteRepositorio es la clase que se encarga de hablar directamente con la base de datos.

- Llama al método findAll() del repositorio.
- Devuelve una lista con todos los clientes guardados en la base de datos.
- Guarda un nuevo cliente en la base de datos.
- Si el cliente ya existe, lo actualiza.
- save() es una función de Spring Data que se encarga de eso automáticamente.
- Usa findById(id) para buscar un cliente específico.
- Si lo encuentra, lo devuelve.
- Si no lo encuentra, lanza un error con el mensaje "Cliente no encontrado".
- Optional se usa para evitar errores si el cliente no existe.
- Primero busca el cliente que ya existe en la base.
- Luego actualiza sus datos (nombre, identificación, tipo de identificación).
- Finalmente guarda los cambios con save ().

```

1 package umg.edu.proyectofinaltaller mecanico.servicio;
2
3 import umg.edu.proyectofinaltaller mecanico.modelo.Cliente;
4 import umg.edu.proyectofinaltaller mecanico.repositorio.ClienteRepositorio;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import java.util.List;
9 import java.util.Optional;
10
11 @Service
12 public class ClienteServicio {
13
14     @Autowired
15     private ClienteRepositorio clienteRepositorio;
16
17     public List<Cliente> listarTodos() { return clienteRepositorio.findAll(); }
18
19     public Cliente guardar(Cliente cliente) { return clienteRepositorio.save(cliente); }
20
21     public Cliente buscarPorId(Long id) {
22         Optional<Cliente> resultado = clienteRepositorio.findById(id);
23         return resultado.orElseThrow(() -> new RuntimeException("Cliente no encontrado"));
24     }
25
26     @
27     public Cliente actualizar(Long id, Cliente nuevoCliente) {
28         Cliente existente = buscarPorId(id);
29         existente.setNombre(nuevoCliente.getNombre());
30         existente.setIdentificacion(nuevoCliente.getIdentificacion());
31         existente.setTipoIdentificacion(nuevoCliente.getTipoIdentificacion());
32         return clienteRepositorio.save(existente);
33     }
34 }

```

Factura Servicio

Este código es el encargado de manejar todo lo que pasa con las facturas dentro del sistema del taller.

El controlador le pide cosas a este servicio (por ejemplo, guarda esta factura o muéstrame todas), y este servicio se comunica con el repositorio para hacer las operaciones en la base de datos.

- La anotación `@Service` indica que esta clase es un servicio, es decir, que se encarga de la lógica del negocio, no de recibir peticiones ni de conectarse directamente con la base de datos.
- FacturaServicio es el nombre de la clase que gestionará todo lo que tenga que ver con las facturas.
- `@Autowired`: le dice a Spring que cree automáticamente un objeto de tipo FacturaRepositorio para que el servicio pueda usarlo sin tener que instanciarlo manualmente.
- FacturaRepositorio es el que habla directamente con la base de datos.
- Llama al método `findAll()` del repositorio.
- Devuelve una lista con todas las facturas registradas.
- Guarda una factura nueva en la base de datos.
- Si la factura ya existe, la actualiza.
- El método `save()` hace ambas cosas automáticamente.
- Busca una factura específica por su número o ID.
- Si la encuentra, la devuelve.
- Si no la encuentra, lanza un error con el mensaje Factura no encontrada.
- Se usa `Optional` para evitar errores cuando el resultado es nulo.
- `Pr0069` busca la factura que ya existe.
- Luego actualiza sus datos: la fecha, el cliente, el vehículo y el servicio.
- Finalmente, guarda los cambios con `save()`.

```
FacturaServicio.java  ClienteServicio.java
1 package umg.edu.proyectofinaltallermecanico.servicio;
2
3 import umg.edu.proyectofinaltallermecanico.modelo.Factura;
4 import umg.edu.proyectofinaltallermecanico.repositorio.FacturaRepositorio;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import java.util.List;
9 import java.util.Optional;
10
11 @Service
12 public class FacturaServicio {
13
14     @Autowired
15     private FacturaRepositorio facturaRepositorio;
16
17     public List<Factura> listarTodos() { return facturaRepositorio.findAll(); }
18
19     public Factura guardar(Factura factura) { return facturaRepositorio.save(factura); }
20
21     public Factura buscarPorId(Long id) {
22         Optional<Factura> resultado = facturaRepositorio.findById(id);
23         return resultado.orElseThrow(() -> new RuntimeException("Factura no encontrada"));
24     }
25
26     public Factura actualizar(Long id, Factura nuevaFactura) {
27         Factura existente = buscarPorId(id);
28         existente.setFecha(nuevaFactura.getFecha());
29         existente.setCliente(nuevaFactura.getCliente());
30         existente.setVehiculo(nuevaFactura.getVehiculo());
31         existente.setServicio(nuevaFactura.getServicio());
32         return facturaRepositorio.save(existente);
33     }
34 }
```

Servicio

Servicio

Este código se encarga de toda la parte lógica de los servicios del taller. El controlador le pide cosas (como muéstrame todos los servicios” o “borra este servicio), y este servicio se encarga de hablar con la base de datos a través del repositorio para realizar esas acciones. Dentro de la clase hay una conexión con el repositorio, que es el encargado de comunicarse directamente con la base de datos. El servicio usa ese repositorio para guardar, buscar, modificar o eliminar información.

- **Listar todos los servicios**
Devuelve una lista con todos los servicios registrados en la base de datos.
- **Guardar un servicio nuevo**
Permite agregar un servicio nuevo, por ejemplo, cuando el taller empieza a ofrecer algo diferente.
- **Buscar un servicio por su ID**
Localiza un servicio específico usando su identificador único. Si no existe, muestra un mensaje indicando que no se encontró.
- **Actualizar un servicio**
Toma un servicio existente y cambia sus datos, como la descripción o el precio.
- **Eliminar un servicio**
Borra un servicio de la base de datos si ya no se ofrece o fue creado por error.

```
1 package umg.edu.proyectofinaltallermechanico.servicio;
2
3 import umg.edu.proyectofinaltallermechanico.modelo.Servicio;
4 import umg.edu.proyectofinaltallermechanico.repositorio.ServicioRepositorio;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import java.util.List;
9 import java.util.Optional;
10
11 @Service 2 usages Ricardo Alfonso chan Lopez
12 public class ServicioServicio {
13
14     @Autowired
15     private ServicioRepositorio servicioRepositorio;
16
17     public List<Servicio> listarTodos() { return servicioRepositorio.findAll(); }
18
19     public Servicio guardar(Servicio servicio) { return servicioRepositorio.save(servicio); }
20
21     public Servicio buscarPorId(Long id) { 2 usages Ricardo Alfonso chan Lopez
22         Optional<Servicio> resultado = servicioRepositorio.findById(id);
23         return resultado.orElseThrow(() -> new RuntimeException("Servicio no encontrado"));
24     }
25
26     public Servicio actualizar(Long id, Servicio nuevoServicio) { 1 usage Ricardo Alfonso chan Lopez
27         Servicio existente = buscarPorId(id);
28         existente.setDescripcion(nuevoServicio.getDescripcion());
29         existente.setPrecio(nuevoServicio.getPrecio());
30         return servicioRepositorio.save(existente);
31     }
32 }
```

Vehículos Servicio

La clase está marcada como un servicio en Spring Boot, lo que significa que se ocupa de manejar la información del negocio, mientras que otro componente (el repositorio) se encarga de comunicarse con la base de datos. De la clase hay una conexión con el repositorio de vehículos, que es quien realmente guarda y recupera la información desde la base.

- **Listar todos los vehículos:**
devuelve una lista con todos los autos registrados en la base de datos.
- **Guardar un vehículo nuevo:**
permite agregar un nuevo vehículo, por ejemplo, cuando un cliente lleva su auto por primera vez al taller.
- **Buscar un vehículo por su ID:**
localiza un vehículo específico usando su número identificador; si no existe, muestra un mensaje de que no fue encontrado.
- **Actualizar los datos de un vehículo:**

```
import umg.edu.proyectofinaltallermechanico.repositorio.VehiculoRepositorio;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class VehiculoServicio {

    @Autowired
    private VehiculoRepositorio vehiculoRepositorio;

    public List<Vehiculo> listarTodos() { return vehiculoRepositorio.findAll(); }

    public Vehiculo guardar(Vehiculo vehiculo) { return vehiculoRepositorio.save(vehiculo); }

    public Vehiculo buscarPorId(Long id) {
        Optional<Vehiculo> resultado = vehiculoRepositorio.findById(id);
        return resultado.orElseThrow(() -> new RuntimeException("Vehiculo no encontrado"));
    }

    public Vehiculo actualizar(Long id, Vehiculo nuevoVehiculo) {
        Vehiculo existente = buscarPorId(id);
        existente.setPlaca(nuevoVehiculo.getPlaca());
        existente.setMarca(nuevoVehiculo.getMarca());
        existente.setModelo(nuevoVehiculo.getModelo());
        existente.setCliente(nuevoVehiculo.getCliente());
        return vehiculoRepositorio.save(existente);
    }

    public void eliminar(Long id) { vehiculoRepositorio.deleteById(id); }
```

busca un vehículo que ya existe y cambia su información, como la placa, la marca, el modelo o el cliente al que pertenece.

- **Eliminar un vehículo:**
borra un vehículo de la base de datos, por ejemplo, si se registró por error o ya no se necesita.

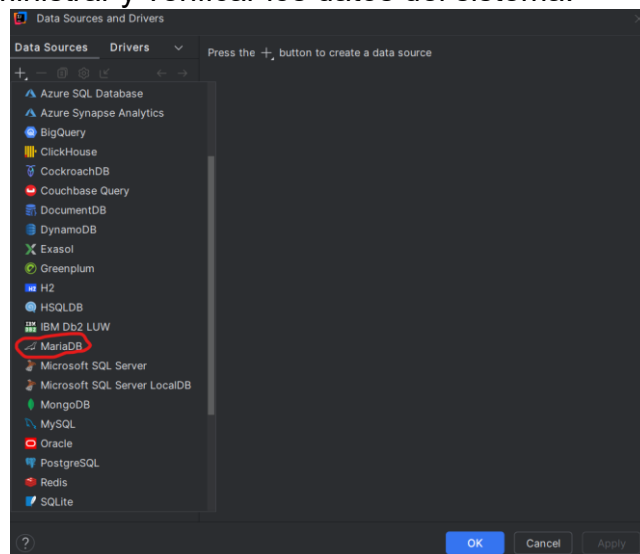
Conexión de Heidy SQL con MariaDB

Para establecer la conexión entre la aplicación desarrollada en Java con Spring Boot y la base de datos, se agregó al proyecto la librería MariaDB Java Client. Esta librería actúa como un conector o puente que permite que el lenguaje Java se comuniquen con MariaDB. Gracias a este conector, la aplicación puede enviar y recibir información sin necesidad de escribir comandos SQL de forma manual.

La configuración de la conexión se realiza en un archivo especial del proyecto donde se especifica la dirección de la base de datos, el nombre del usuario y la contraseña de acceso. Con esta información, Spring Boot puede conectarse automáticamente a la base de datos cada vez que se ejecuta el sistema.

Para la administración visual de la base de datos se utilizó la herramienta HeidiSQL, la cual facilita la creación y gestión de las tablas, la visualización de los registros almacenados y la ejecución de consultas SQL.

En resumen, MariaDB es la base de datos utilizada, Spring Boot se encarga de conectarse a ella mediante el conector, y HeidiSQL se usa como interfaz visual para administrar y verificar los datos del sistema.

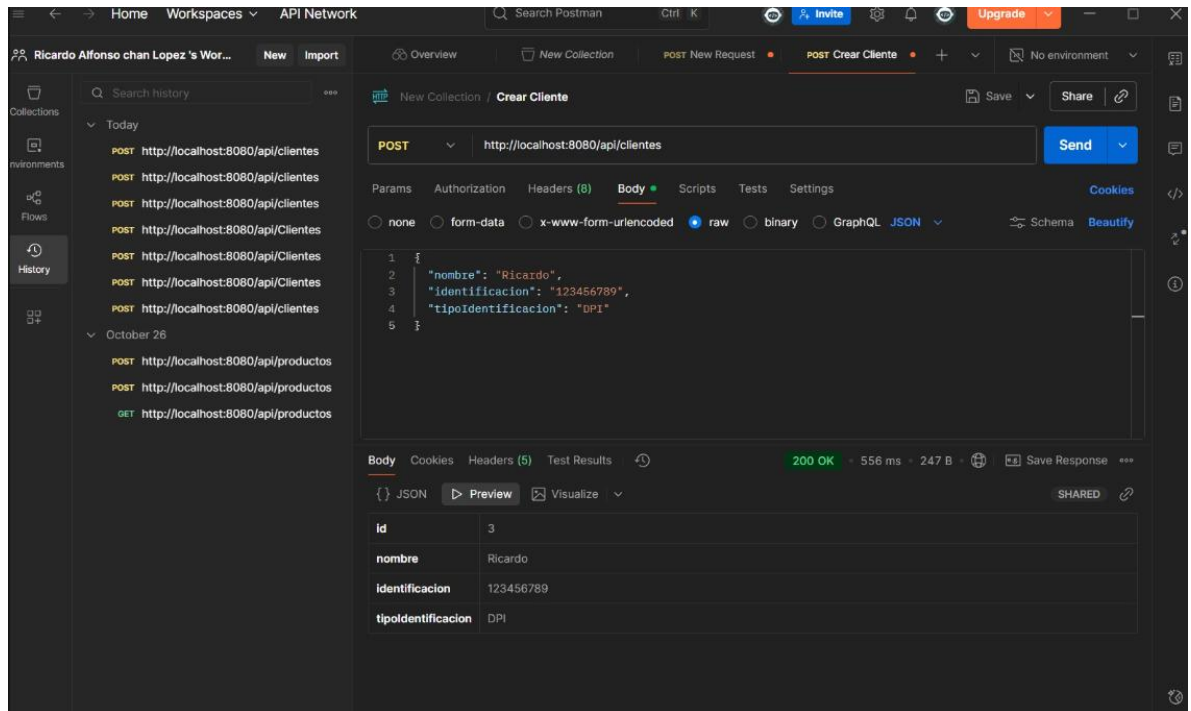




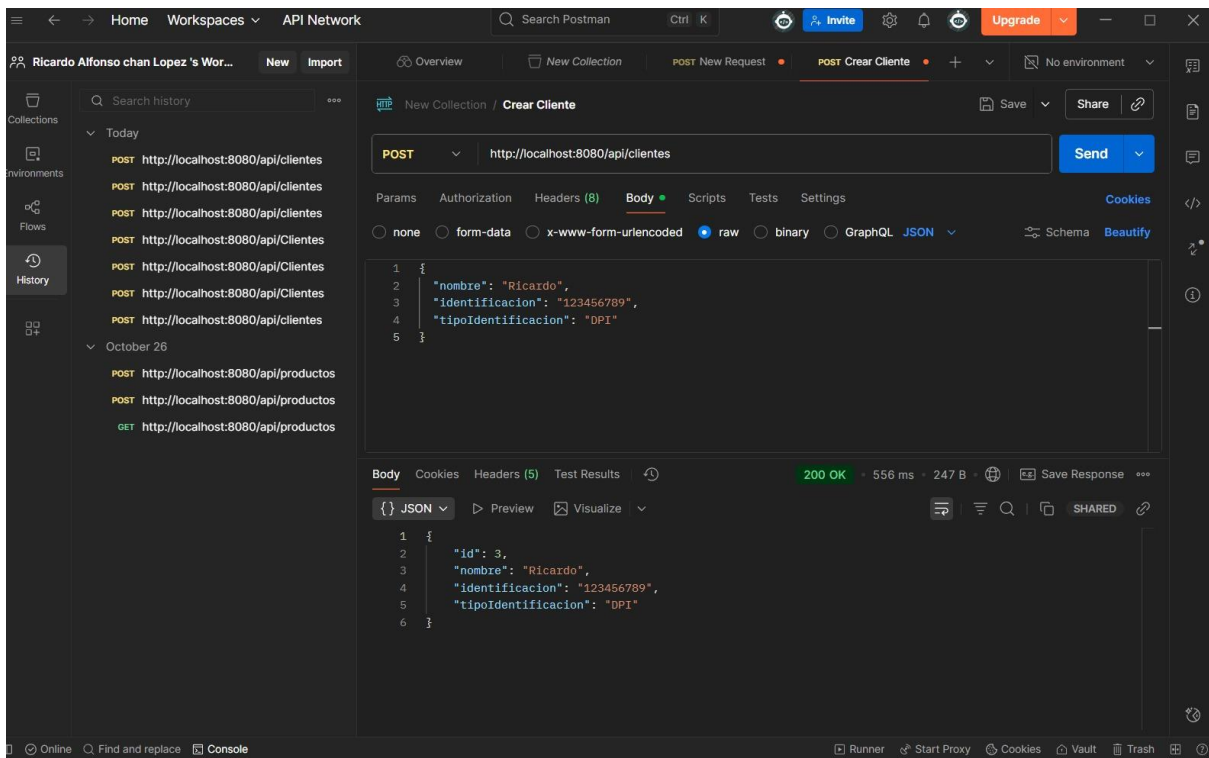
Postman

Postman está actuando como una herramienta que simula a un usuario o aplicación que se comunica con tu API para asegurarte de que funcione correctamente.

- **Seleccionas el tipo de petición**
En este caso “POST”, que sirve para enviar información nueva al servidor.
- **Indicas la dirección del servicio,**
Que es localhost:8080/api/clientes; eso significa que el servidor está funcionando en tu computadora, en el puerto 8080.
- **En la sección “Body”,**
Escribes los datos del cliente que quieres registrar (como nombre e identificación).
- Al presionar **Send**,
Postman envía esa información al servidor.
- El servidor procesa la solicitud, guarda los datos en la base de datos y responde confirmando que todo salió bien.
- En la parte inferior ves la **respuesta**, donde el servidor te muestra el cliente creado y el estado “200 OK”, lo que indica que la operación fue exitosa.



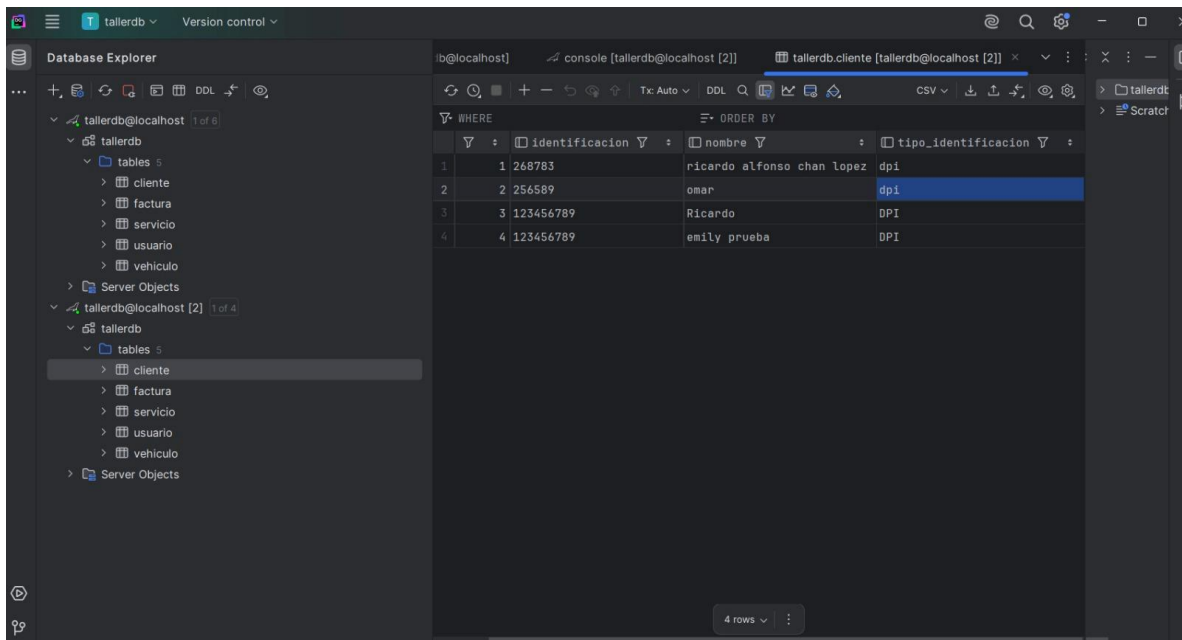
JSON



El JSON de desarrollo se basa en una arquitectura RESTful, implementada con Spring Boot y probada mediante Postman. El backend expone un endpoint tipo

POST (/api/clientes) que recibe datos en formato JSON para registrar nuevos clientes. La información se envía al controlador, que a su vez interactúa con el servicio y el repositorio JPA para almacenar los datos en la base de datos. El servidor responde con un objeto JSON que incluye el ID autogenerated y los datos ingresados, confirmando la persistencia. La comunicación es síncrona y responde con el código HTTP 200 OK.

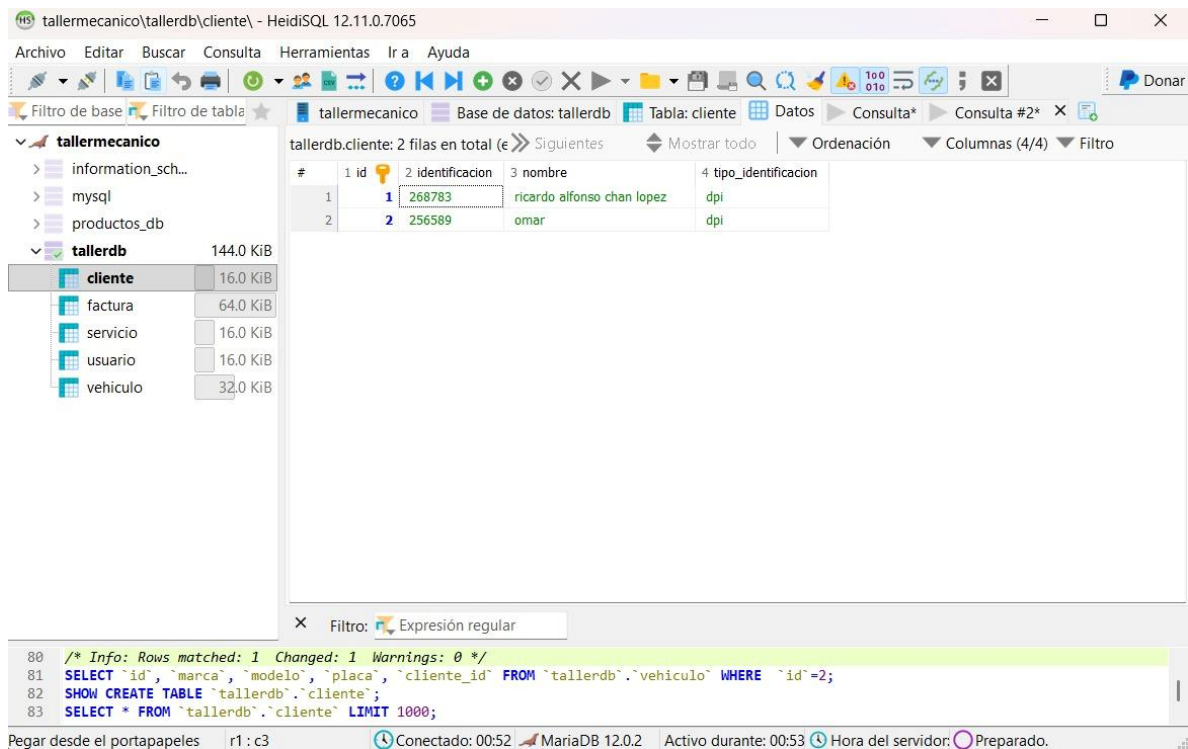
TABLAS EN EJECUCIÓN



The screenshot shows a database client interface with a 'Database Explorer' on the left and a query results pane on the right. The query results pane displays a table with 4 rows. The table has columns: 'identificacion', 'nombre', and 'tipo_identificacion'. The data is as follows:

	identificacion	nombre	tipo_identificacion
1	1 268783	ricardo alfonso chan lopez	dpi
2	2 256589	omar	dpi
3	3 123456789	Ricardo	DPI
4	4 123456789	emily prueba	DPI

EJECUCION EN HEIDISQL



The screenshot shows the HeidiSQL interface. On the left, the database structure is visible, including the 'tallerdb' database and its tables: 'cliente', 'factura', 'servicio', 'usuario', and 'vehiculo'. The 'cliente' table is selected, showing 2 rows in total. The main window displays the data for the 'cliente' table:

#	1 id	2 identificacion	3 nombre	4 tipo_identificacion
1	1	268783	ricardo alfonso chan lopez	dpi
2	2	256589	omar	dpi

The bottom panel shows the following SQL queries:

```
80 /* Info: Rows matched: 1 Changed: 1 Warnings: 0 */
81 SELECT `id`, `marca`, `modelo`, `placa`, `cliente_id` FROM `tallerdb`.`vehiculo` WHERE `id`=2;
82 SHOW CREATE TABLE `tallerdb`.`cliente`;
83 SELECT * FROM `tallerdb`.`cliente` LIMIT 1000;
```

The status bar at the bottom indicates: 'Pegar desde el portapapeles', 'r1: c3', 'Conectado: 00:52', 'MariaDB 12.0.2', 'Activo durante: 00:53', 'Hora del servidor: Preparado'.

La base de datos tallerdb es el núcleo de almacenamiento del sistema del taller mecánico, encargada de gestionar toda la información relacionada con los clientes, vehículos, servicios, facturas y usuarios del sistema. Está diseñada bajo el modelo relacional, optimizando la integridad de los datos mediante llaves primarias y foráneas que vinculan las tablas. La tabla cliente almacena datos personales como nombre, identificación y tipo de documento, permitiendo diferenciar a cada usuario del taller. Vehículo mantiene información esencial como marca, modelo, placa e identificación del cliente propietario, garantizando un registro preciso de los automóviles atendidos. La tabla servicio detalla los tipos de trabajos realizados, incluyendo su descripción y costo, mientras que factura vincula al cliente, vehículo y servicio, junto con la fecha de emisión, para asegurar un control financiero completo. Finalmente, la tabla usuario gestiona las credenciales de acceso, fortaleciendo la seguridad del sistema. En conjunto, esta base de datos proporciona una estructura sólida, escalable y eficiente que soporta el funcionamiento integral del taller, facilitando consultas rápidas, relaciones bien definidas y mantenimiento simplificado, ideal para una aplicación web moderna basada en Spring Boot y MariaDB.

EJECUCION EN INTERFAZ WEB

Bienvenidos al Taller Mecánico de Ricardo, Omar y Emily

Clientes

Nombre

Identificación

Tipo

Agregar Cliente

Vehículos

Placa

Marca

Modelo

ID Cliente

Agregar Vehículo

Servicios

Descripción

Precio

Agregar Servicio

Facturas

dd/mm/aaaa

ID Cliente

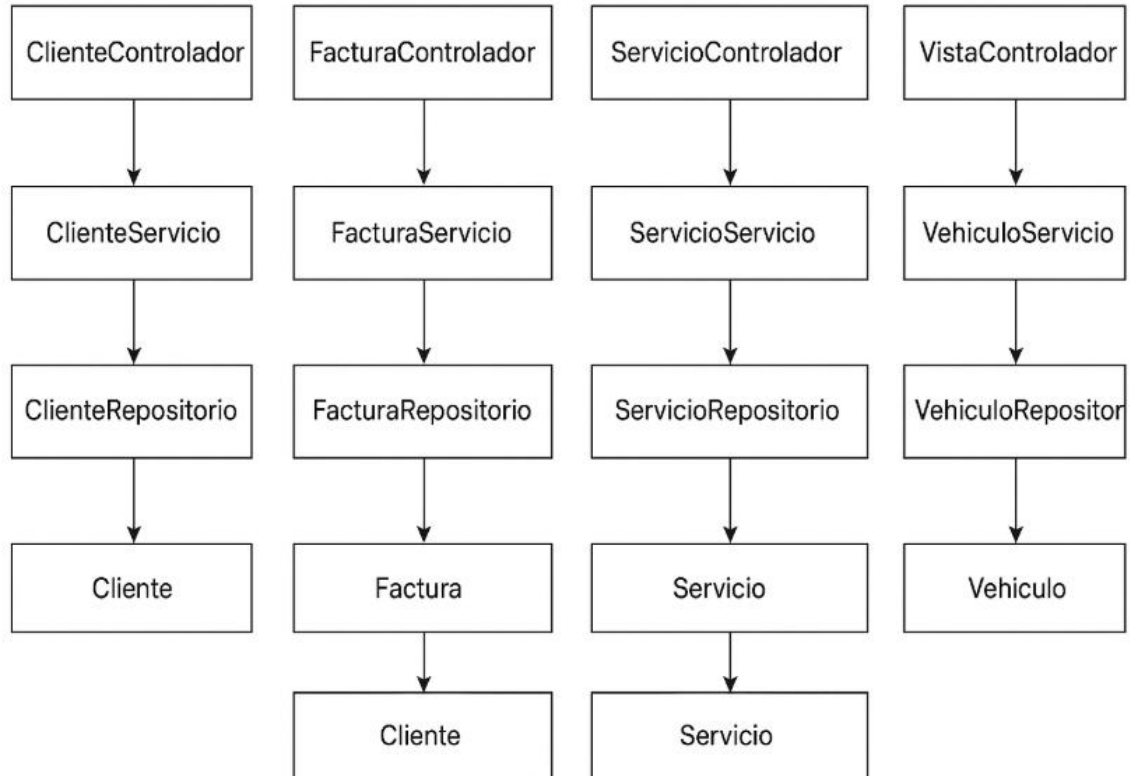
ID Vehículo

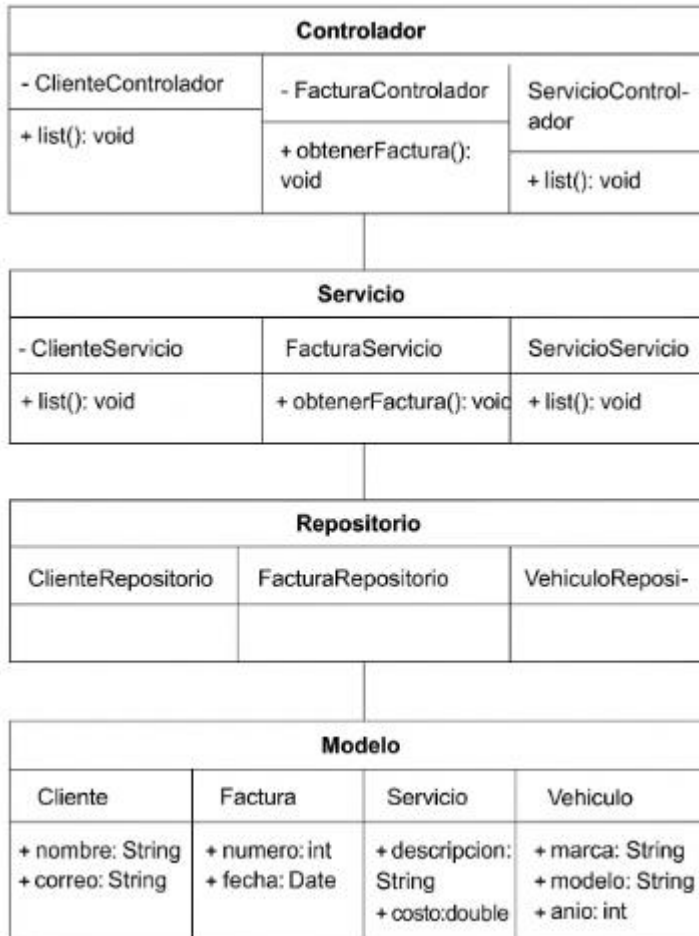
ID Servicio

Generar Factura

La interfaz web del Taller Mecánico de Ricardo, Omar y Emily está diseñada para brindar una experiencia sencilla, intuitiva y eficiente tanto para el personal administrativo como para los usuarios del sistema. Su objetivo principal es permitir la gestión completa de clientes, vehículos, servicios y facturas desde un entorno unificado y visualmente agradable. En la sección Clientes, se pueden registrar los datos personales de cada cliente, incluyendo nombre, identificación y tipo de documento. En Vehículos, se ingresan detalles técnicos como placa, marca, modelo e identificación del propietario, vinculando cada automóvil a su respectivo cliente. La sección Servicios permite registrar las reparaciones o mantenimientos realizados, especificando descripción y precio, mientras que Facturas integra todos los módulos permitiendo generar comprobantes con fecha, cliente, vehículo y servicio asociado. Visualmente, la interfaz utiliza paneles transparentes sobre un fondo de taller mecánico, ofreciendo una estética moderna y profesional. Los formularios son simples, con botones destacados para realizar acciones clave como “Agregar Cliente” o “Generar Factura”. Esta interfaz, desarrollada con tecnologías web responsivas, conecta directamente con la base de datos mediante el backend en Spring Boot, garantizando fluidez, seguridad y una administración digital completa del taller.

UML





CONCLUSIONES

La elaboración del sistema del taller mecánico permitió comprobar la eficacia de la integración entre herramientas modernas de desarrollo como IntelliJ IDEA y tecnologías de gestión de bases de datos como MariaDB y HeidiSQL. IntelliJ IDEA facilitó la construcción del proyecto con el framework Spring Boot, ofreciendo un entorno robusto para la programación orientada a objetos y la estructuración por capas (modelo, servicio, repositorio y controlador). Por otro lado, MariaDB se consolidó como una base de datos relacional confiable y de alto rendimiento, mientras que HeidiSQL fue de gran utilidad para la administración gráfica de los datos, simplificando tareas de consulta, edición y validación. La interacción fluida entre estas herramientas permitió un flujo de trabajo ágil, estableciendo una comunicación directa entre la aplicación y la base de datos. En conjunto, este ecosistema tecnológico garantizó la coherencia de los datos, la escalabilidad del sistema y la eficiencia en la implementación del proyecto.

La arquitectura implementada en capas permitió separar las responsabilidades del sistema, logrando una estructura más limpia, modular y fácil de mantener. Mediante IntelliJ IDEA, se implementaron controladores, servicios y repositorios que interactúan con las entidades de la base de datos, lo que demostró la importancia del diseño orientado a objetos y del patrón MVC (Modelo–Vista–Controlador). Además, el uso de MariaDB reforzó los conocimientos en modelado de bases de datos, manejo de relaciones entre tablas y ejecución de consultas SQL. Finalmente, el empleo de HeidiSQL facilitó la visualización y depuración de datos, confirmando que una adecuada gestión de información es esencial para el correcto funcionamiento del sistema.