

---

## Indice

---

➤ 1. Descripción del Proyecto.....	2
➤ 2. Diseño de Tablas.....	2
➤ 3. Entidades.....	3
• 3.1. Cliente.....	3
• 3.2. Factura.....	4
• 3.3. ItemFactura.....	4
• 3.4. Producto.....	5
➤ 4. Funcionalidades del Proyecto.....	5
• 4.1. Alta de Clientes.....	7
• 4.2. Listado Clientes.....	9
– 4.2.1. Paginación del listado.....	10
• 4.3. Modificar Cliente.....	11
• 4.4. Datos Cliente.....	12
• 4.5. Borrar Cliente.....	13
• 4.6. Crear Factura.....	14
– 4.6.1. Buscar Productos.....	17
• 4.7. Datos Factura.....	19
• 4.8. Borrar Factura.....	20
➤ 5. Configuración de Despliegue.....	21

## 1. Descripción del Proyecto

El proyecto trata de una aplicación de gestión de facturación de clientes en base a una tienda que vende productos para los podcasts, estudios de sonido...

## 2. Diseño de Tablas

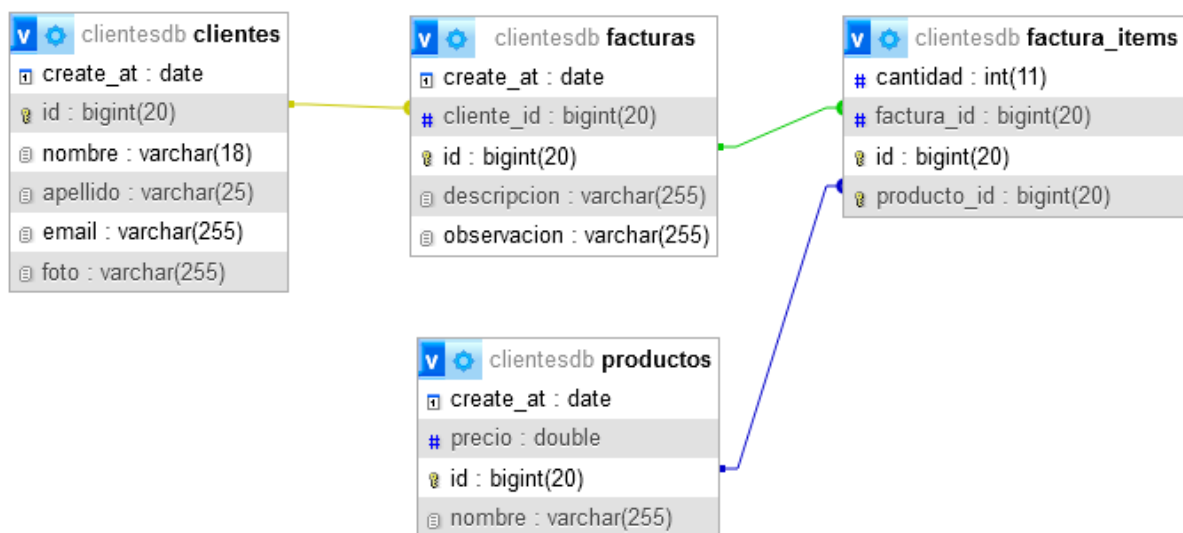
La base de datos contiene 4 tablas:

**Clientes:** Almacena los datos de contacto de un cliente.

**Facturas:** Guarda los pedidos que ha realizado un cliente.

**Factura\_Items:** Son los productos pedidos en cada factura con la cantidad solicitada.

**Productos:** Donde están los productos disponibles de la tienda que gestionamos.



## 3. Entidades

Todas estas entidades para que las reconozca SpringBoot necesitan las Java Annotations mediante la anotación **@Entity**. Cada entidad usa unas anotaciones en cada una de sus propiedades para validar dicha entidad.

### 3.1. Cliente

Esta entidad pertenece a la tabla **clientes** y tiene las siguientes propiedades:

- **ID:** Es la clave primaria de la entidad y se genera de manera automática por cada registro insertado
- **Nombre:** Es el nombre del cliente que no puede estar vacío (@NotEmpty) y su tamaño (@Size) está entre 4 y 18 caracteres.
- **Apellidos:** Son los apellidos del cliente que no puede estar vacío (@NotEmpty) y su tamaño (@Size) está entre 4 y 25 caracteres.
- **Email:** Es el correo del cliente que no puede estar vacío (@NotEmpty) y tiene que tener un formato válido de correo (@Email).
- **CreateAt:** Esta propiedad almacena la fecha de cuando se dio de alta a un cliente. Su campo en la BD se almacena como create\_at con la anotación @Column, se guarda solo con la fecha sin la hora mediante @Temporal(TemporalType.DATE) con el formato de YYYY-MM-DD y no puede estar vacía.
- **Facturas:** Esto es la relación que tiene con la entidad Facturas que es que un cliente puede tener muchas facturas.
- **Foto:** Almacena la ruta de la foto del cliente.

## 3.2. Factura

Esta entidad pertenece a la tabla **facturas** y tiene las siguientes propiedades:

- **ID:** Lo mismo que en la entidad cliente
- **Descripción:** Donde se rellena la descripción principal de la factura que no puede estar vacío (@NotEmpty).
- **Observación:** Donde se dan los detalles de la factura (opcional).
- **CreateAt:** Esta propiedad almacena la fecha de cuando se creo la factura ( lo hace de con la fecha actual gracias a @PrePersist actuando antes de persistir en la entidad). Su campo en la BD se almacena como create\_at con la anotación @Column y se guarda solo con la fecha sin la hora mediante @Temporal(TemporalType.DATE).
- **Cliente:** Esto es la relación que tiene con la entidad Cliente ya que una o muchas facturas pueden pertenecer solo a un cliente, es decir, la misma factura no puede pertenecer a dos clientes.
- **Items:** Es la relación que tiene con la entidad ItemFactura que indica que en una factura puede haber uno o varios productos y se crea un campo en la BD con el nombre factura\_id gracias a esta relación.

Tiene una función para calcular el total de la factura cogiendo de cada item de la factura su subtotal.

## 3.3. ItemFactura

Esta entidad pertenece a la tabla **factura\_items** y tiene las siguientes propiedades:

- **ID:** Lo mismo que las entidades anteriores.
- **Cantidad:** Es el total de veces que se ha pedido un producto.
- **Producto:** Es la relación que tiene con la entidad Producto que indica que en un item de la factura puede haber solo un producto y se crea un campo en la BD con el nombre producto\_id gracias a esta relación.

Tiene una función para calcular el importe de cada precio del producto por la cantidad comprada, es decir, el subtotal de una factura.

### 3.4. Producto

Esta entidad pertenece a la tabla **productos** y tiene las siguientes propiedades:

- **ID:** Lo mismo que las entidades anteriores.
- **Nombre:** Es el nombre del producto a comprar.
- **Precio:** Lo que vale cada producto.
- **CreateAt:** Igual que la entidad de Factura
- **Cliente:** Esto es la relación que tiene con la entidad Cliente ya que una o muchas facturas pueden pertenecer solo a un cliente, es decir, la misma factura no puede pertenecer a dos clientes.
- **Items:** Es la relación que tiene con la entidad ItemFactura que indica que en una factura puede haber uno o varios productos y se crea un campo en la BD con el nombre factura\_id gracias a esta relación.

## 4. Funcionalidades del Proyecto

Antes de ver las funcionalidades del proyecto se muestran las interfaces de las cuales salen todas las funcionalidades del proyecto.

**IClienteDao:** Utilizado para el CRUD de Clientes y la paginación.

```
public interface IClienteDao extends CrudRepository<Cliente, Long>, PagingAndSortingRepository<Cliente, Long> { }
```

**IFacturaDaoJPA:** Utilizado para el CRUD de Facturas. Está también un IFacturaDao que extiende de CrudRepository y sus funciones hechas pero se ha utilizado el otro DAO por JPA.

```
public interface IFacturaDaoJPA
{
    public void saveFactura(Factura factura);
    public Factura findFactura(Long id);
    public void deleteFactura(Long id);
}
```

**IProductoDao:** Utilizado para encontrar los productos por ID o por el nombre a través de una query con un LIKE.

```
public interface IProductoDao extends CrudRepository<Producto, Long> {  
  
    @Query("select p from Producto p where p.nombre like %?1%")  
    public List<Producto> findByNombre(String term);  
  
}
```

**IClienteService:** Se utiliza para declarar todas las funcionalidades del proyecto.

```
public interface IClienteService  
{  
    public List<Cliente> findAll();  
    public Page<Cliente> findAll(Pageable pageable);  
    public void save(Cliente cliente);  
    public Cliente findOne(Long id);  
    public void delete(Long id);  
    public List<Producto> findByNombre(String term);  
    public void saveFactura(Factura factura);  
    public Producto findProductoById(Long id);  
    public Factura findFacturaById(Long id);  
    public void deleteFactura(Long id);  
}
```

**IUploadFileService:** Para la gestión de fotos

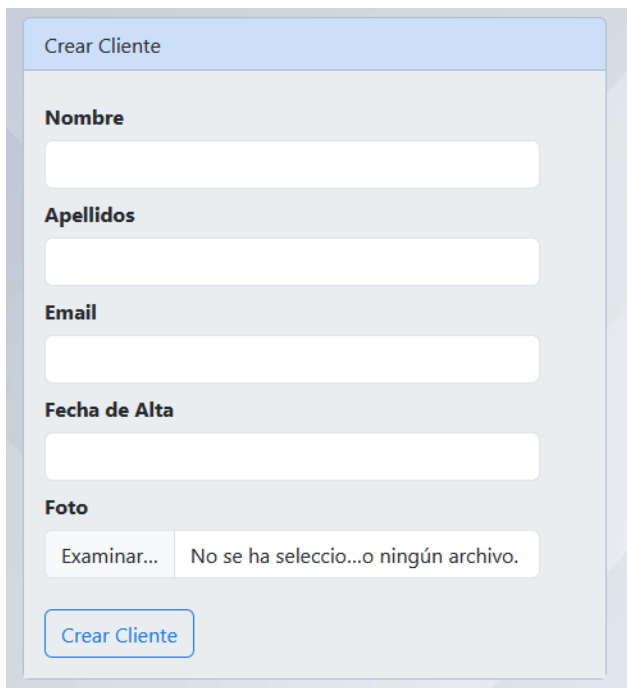
```
public interface IUploadFileService  
{  
    public Resource load(String filename) throws MalformedURLException;  
    public String copy(MultipartFile file) throws IOException;  
    public boolean delete(String filename);  
    public void deleteAll();  
    public void init() throws IOException;  
}
```

## 4.1. Alta de Clientes

Para la alta de clientes el siguiente método coge una petición **GET** desde el controlador si se ha buscado /form y crea un nuevo cliente y lo manda a la vista a través del model

```
@RequestMapping(value = "form") // Por defecto es un GET
public String crear(Map<String, Object> model)
{
    Cliente cliente = new Cliente();
    model.put(key:"titulo", value:"Crear Cliente");
    model.put(key:"cliente", cliente);
    return "form";
}
```

La vista muestra un formulario para crear un cliente

El formulario tiene un encabezado azul con el título "Crear Cliente". A continuación, hay campos de entrada para "Nombre", "Apellidos", "Email" y "Fecha de Alta". Debajo de estos, hay una sección para la "Foto" que incluye un botón "Examinar..." y un mensaje "No se ha seleccionado ningún archivo.". Al final del formulario, hay un botón azul "Crear Cliente".

Crear Cliente	
<b>Nombre</b>	
<input type="text"/>	
<b>Apellidos</b>	
<input type="text"/>	
<b>Email</b>	
<input type="text"/>	
<b>Fecha de Alta</b>	
<input type="text"/>	
<b>Foto</b>	
<input type="button" value="Examinar..."/>	No se ha seleccionado ningún archivo.
<input type="button" value="Crear Cliente"/>	

El formulario se envía a través de una petición **POST** que la recoge el controlador. Si tiene errores vuelve a mandar al formulario. Tiene un control para las fotos. Si el cliente cumple las condiciones que tiene la entidad lo inserta en la BD y manda a la vista de listar que verás a continuación.

```
@RequestMapping(value = "form", method = RequestMethod.POST)
public String guardar(@Valid Cliente cliente, BindingResult result, Model model, SessionStatus status,
    RedirectAttributes flash, @RequestParam("file") MultipartFile foto)
{
    if (result.hasErrors()) {
        model.addAttribute("titulo", "Crear Cliente");
        return "form";
    }

    if (!foto.isEmpty())
    {
        if (cliente.getId() != null && cliente.getId() > 0 && cliente.getFoto() != null && cliente.getFoto().length() > 0)
        {
            uploadFileService.delete(cliente.getFoto());
        }
    }

    String uniqueFilename = null;

    try
    {
        uniqueFilename = uploadFileService.copy(foto);
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }

    flash.addFlashAttribute("info", "Has subido correctamente: '" + uniqueFilename + "'");

    cliente.setFoto(uniqueFilename);

    String mensajeFlash = (cliente.getId() != null)? "Cliente editado con exito" : "Cliente creado con exito";

    clienteService.save(cliente);
    status.setComplete();
    flash.addFlashAttribute("success", mensajeFlash);
    return "redirect:listar";
}
```

Utiliza tanto los métodos de los servicios de gestión de fotos (UploadFileServiceImplementado) para borrar la foto si se da ciertas condiciones o copiar la foto a la carpeta uploads como el servicio de cliente (ClienteServiceImplementado) para guardar el cliente en la BD.

```
@Override
public boolean delete(String filename)
{
    Path rootPath = getPath(filename);
    File archivo = rootPath.toFile();

    if(archivo.exists() && archivo.canRead())
    {
        if (archivo.delete())
        {
            return true;
        }
    }
    return false;
}
```

```
@Override
public String copy(MultipartFile file) throws IOException
{
    String uniqueFilename = UUID.randomUUID().toString() + "_" + file.getOriginalFilename();
    Path rootPath = getPath(uniqueFilename);

    log.info("rootPath: " + rootPath);

    Files.copy(file.getInputStream(), rootPath);

    return uniqueFilename;
}
```

```
@Override
@Transactional
public void save(Cliente cliente) {
    clienteDao.save(cliente);
}
```



## 4.2. Listado Clientes

Cuando se busca la dirección **/listar** como es una petición **GET** llega al controlador del **Cliente** por el método **listar** que lo que hace es encontrar todos los clientes de la BD y los manda a la vista listar con una paginación de 6 por página y que la paginación por defecto mediante parámetro que se carga es la 0.

```
@RequestMapping(value = "/listar", method = RequestMethod.GET)
public String listar(@RequestParam(name = "page", defaultValue = "0") int page, Model model)
{
    Pageable pageRequest = PageRequest.of(page, 6);

    Page<Cliente> clientes = clienteService.findAll(pageRequest);
    PageRenderer<Cliente> pageRender = new PageRenderer<>(url: "/listar", clientes);

    model.addAttribute("titulo", "Listado de clientes");
    model.addAttribute("clientes", clientes);
    model.addAttribute("page", pageRender);
    return "listar";
}
```

Vista de la lista de clientes.

Nombre	Apellidos	Factura	Detalles Cliente	Editar Cliente	Eliminar Cliente
Santiago	Gonzalez Perez	<a href="#">Crear Factura</a>	<a href="#">Ver</a>	<a href="#">Editar</a>	<a href="#">Borrar</a>
Valentina	Rodriguez Mendoza	<a href="#">Crear Factura</a>	<a href="#">Ver</a>	<a href="#">Editar</a>	<a href="#">Borrar</a>
Mateo	Garcia Vargas	<a href="#">Crear Factura</a>	<a href="#">Ver</a>	<a href="#">Editar</a>	<a href="#">Borrar</a>
Isabella	Fernandez Lopez	<a href="#">Crear Factura</a>	<a href="#">Ver</a>	<a href="#">Editar</a>	<a href="#">Borrar</a>
Sebastian	Martinez Herrera	<a href="#">Crear Factura</a>	<a href="#">Ver</a>	<a href="#">Editar</a>	<a href="#">Borrar</a>
Camila	Diaz Castro	<a href="#">Crear Factura</a>	<a href="#">Ver</a>	<a href="#">Editar</a>	<a href="#">Borrar</a>

« 1 2 »

Tiene botones que enlazan a otras vistas que se verán más adelante. Utiliza el servicio de cliente (ClienteServiceImplementado) para encontrar todos los clientes en la BD.

```
@Override
public Page<Cliente> findAll(Pageable pageable) {
    return clienteDao.findAll(pageable);
}

@Override
```

### 4.2.1. Paginación del listado

Para la paginación se ha definido una entidad para los clientes que se van a mostrar por cada página ( 6 clientes por página como se ha visto en el listado )

```
public class PageItem {

    private int numero;
    private boolean actual;

    // Constructores
    public PageItem(int numero, boolean actual) {
        this.numero = numero;
        this.actual = actual;
    }

    // Getters y Setters
    public int getNumero() {
        return numero;
    }

    public boolean isActual() {
        return actual;
    }

}
```

Y luego otra entidad para saber que limites de contenido se establecen en la paginación. El primer **IF** contiene todo, después del **ELSE** y primer **IF** es el limite inicial, el **ELSE IF** es el final y el último **ELSE** es el intermedio de la paginación. También se han definido una serie de funciones para saber cual es la primera página, la última, si tiene siguiente o si tiene anterior.

```
public PageRender(String url, Page<T> page) {
    this.url = url;
    this.page = page;
    this.paginas = new ArrayList<PageItem>();

    elementosPorPagina = page.getSize();
    totalPaginas = page.getTotalPages();
    paginaActual = page.getNumber() + 1;

    int desde, hasta;

    if (totalPaginas ≤ elementosPorPagina)
    {
        desde = 1;
        hasta = totalPaginas;
    }
    else
    {
        if (paginaActual ≤ elementosPorPagina / 2)
        {
            desde = 1;
            hasta = elementosPorPagina;
        }
        else if (paginaActual ≥ totalPaginas - elementosPorPagina / 2)
        {
            desde = totalPaginas - elementosPorPagina + 1;
            hasta = elementosPorPagina;
        }
        else
        {
            desde = paginaActual - elementosPorPagina / 2;
            hasta = elementosPorPagina;
        }
    }

    for (int i = 0; i < hasta; i++)
    {
        paginas.add(new PageItem(desde + i, paginaActual == desde + i));
    }
}
```

```
// Funciones
public boolean isFirst()
{
    return page.isFirst();
}

public boolean isLast()
{
    return page.isLast();
}

public boolean hasNext()
{
    return page.hasNext();
}

public boolean hasPrevious()
{
    return page.hasPrevious();
}
```

### 4.3. Modificar Cliente

La manera de acceder a la modificación del cliente es a través del botón **Editar** de la vista listar que lo que hace es mandar una petición **GET** que llega al método editar del controlador Cliente. Este método encuentra el cliente si existe mas de un id y manda a la misma vista que la alta de clientes.

```
<td><a class="btn btn-outline-success btn-xs" th:href="@{/form/} + ${cliente.id}" th:text="Editar"></a></td>
```

```
@RequestMapping(value = "/form/{id}")
public String editar(@PathVariable(value = "id") Long id, Map<String, Object> model, RedirectAttributes flash)
{
    Cliente cliente = new Cliente();

    if (id > 0)
    {
        cliente = clienteService.findOne(id);

        if (cliente == null)
        {
            flash.addFlashAttribute("error", "El ID del cliente no existe en la BD");
            return "redirect:/listar";
        }
    }
    else
    {
        flash.addFlashAttribute("error", "El ID del cliente no puede ser 0");
        return "redirect:/listar";
    }

    model.put(key:"cliente", cliente);
    model.put(key:"titulo", value:"Editar cliente");

    return "form";
}
```

Utiliza el servicio de cliente (ClienteServiceImplementado) para encontrar un cliente en la BD.

```
@Override
@Transactional(readOnly = true)
public Cliente findOne(Long id) {
    return clienteDao.findById(id).orElse(null);
}
```

## 4.4. Datos Cliente

Para acceder a los datos de un cliente se hace una petición **GET** a `/ver/{id}` del cliente que eso lo hace el botón **Ver** de la vista listar.

```
<td><a class="btn btn-outline-primary btn-xs" th:href="@{/ver/} + ${cliente.id}" th:text="Ver"></a></td>
```

Una vez se ha clickado en el botón nos lleva al método ver del controlador Cliente y lo que hace es encontrar el cliente por el id y nos manda a la vista de ver

```
@GetMapping(value = "/ver/{id}")
public String ver(@PathVariable(value = "id") Long id, Map<String, Object> model, RedirectAttributes flash)
{
    Cliente cliente = clienteService.findOne(id);

    if (cliente == null)
    {
        flash.addFlashAttribute("error", "El cliente no existe en la BD");
        return "redirect:/listar";
    }

    model.put(key:"cliente", cliente);
    model.put(key:"titulo", "Detalles del cliente: " + cliente.getNombre());

    return "ver";
}
```

Utiliza el servicio de cliente (ClienteServiceImplementado) para encontrar un cliente en la BD para poder visualizarlo.

```
@Override
@Transactional(readOnly = true)
public Cliente findOne(Long id) {
    return clienteDao.findById(id).orElse(other:null);
}
```

La vista nos muestra la información con la que se ha registrado el cliente. Si tiene foto la muestra sino no. Y además si el cliente tiene facturas muestra lo básico de la factura con dos funcionalidades que se verán más adelante.

Detalles del cliente: Santiago

<b>Nombre</b>	Santiago
<b>Apellidos</b>	Gonzalez Perez
<b>Correo</b>	santiago.gonzalez@email.com
<b>Fecha de Alta</b>	1990-05-15

Folio	Descripción	Fecha	Total	Ver Factura	Borrar Factura
1	Factura altavoces sala espera	2024-02-16	860.0	<a href="#">Detalles</a>	<a href="#">Eliminar</a>

## 4.5. Borrar Cliente

Para borrar los datos de un cliente se hace una petición **GET** a `/eliminar/{id}` del cliente que eso lo hace el botón **Borrar** de la vista listar.

Esta petición llega al método eliminar del controlador Cliente y lo que hace es encontrar al cliente si hay mas de un id y lo borra utilizando el servicio de cliente (`ClienteServiceImpl`). En caso de que el cliente tenga una foto también la borra de la carpeta uploads utilizando el servicio de gestión de fotos (`UploadFileServiceImpl`).

```
@RequestMapping(value = "/eliminar/{id}")
public String eliminar(@PathVariable(value = "id") Long id, RedirectAttributes flash)
{
    if (id > 0 )
    {
        Cliente cliente = clienteService.findOne(id);

        clienteService.delete(id);
        flash.addFlashAttribute("success", "Cliente eliminado con éxito");

        if(uploadFileService.delete(cliente.getFoto()))
        {
            flash.addFlashAttribute("info", "Foto " + cliente.getFoto() + " ha sido eliminada");
        }
    }
    return "redirect:/listar";
}
```

```
@Override
@Transactional(readOnly = true)
public Cliente findOne(Long id) {
    return clienteDao.findById(id).orElse(other:null);
}
```

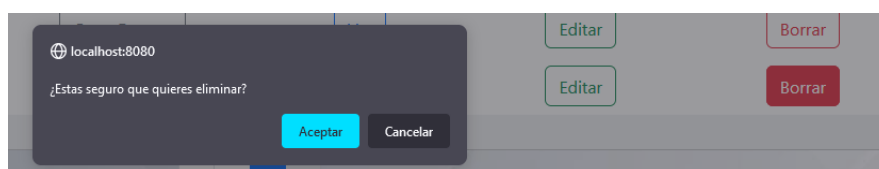
```
@Override
@Transactional
public void delete(Long id) {
    clienteDao.deleteById(id);
}
```

```
@Override
public boolean delete(String filename)
{
    Path rootPath = getPath(filename);
    File archivo = rootPath.toFile();

    if(archivo.exists() && archivo.canRead())
    {
        if (archivo.delete())
        {
            return true;
        }
    }
    return false;
}
```

En la vista listar el botón de Borrar da un aviso de si se quiere confirmar el borrado del cliente

```
<td><a class="btn btn-outline-danger btn-xs" th:href="@{/eliminar/} + ${cliente.id}"
th:text="Borrar" onclick="return confirm('¿Estas seguro que quieres eliminar?')"></a></td>
```



## 4.6. Crear Factura

En la vista de listar hay un botón de **Crear Factura** que realiza una petición **GET** a `/factura/form/{clienteID}` con el id del cliente. La petición llega al método **crear** del controlador **Factura** que lo hace es encontrar el cliente, crea una factura, se la asigna al cliente y la manda a la vista `factura/form`

```
@GetMapping("/form/{clienteId}")
public String crear(@PathVariable(value = "clienteId") Long clienteId, Map<String, Object> model, RedirectAttributes flash)
{
    Cliente cliente = clienteService.findOne(clienteId);

    if (cliente == null)
    {
        flash.addFlashAttribute("error", "El cliente no existe en la base de datos");
        return "redirect:/listar";
    }

    Factura factura = new Factura();
    factura.setCliente(cliente);

    model.put(key:"factura", factura);
    model.put(key:"titulo", value:"Crear Factura");

    return "factura/form";
}
```

La vista contiene un formulario para crear la factura. En la vista hay un campo para buscar los productos existentes de SoundHub y una tabla que calcula automáticamente el total de la factura de los productos seleccionados por su cantidad solicitada

Crear Factura

Cliente  
Santiago Gonzalez Perez

Descripción

Observación

Buscar Producto

Nombre	Precio	Cantidad	Total	Eliminar
<div>Total Factura: 0</div> <div>Crear Factura</div>				



Ahora una vez creada hace falta guardarla en la BD. Para ello se hace una petición **POST** cuando se envía el formulario y esa petición llega al método **guardar** del controlador Factura.

```
@PostMapping("form")
public String guardar(@Valid Factura factura, BindingResult result, Model model,
    @RequestParam(name = "item_id[]", required = false) Long[] itemId,
    @RequestParam(name = "cantidad[]", required = false) Integer[] cantidad,
    RedirectAttributes flash, SessionStatus status )
{
    if (result.hasErrors())
    {
        model.addAttribute("titulo", "Crear Factura");
        return "factura/form";
    }

    if (itemId == null || itemId.length == 0)
    {
        model.addAttribute("titulo", "Crear Factura");
        model.addAttribute("error", "Error: La factura No puede no tener lineas");
        return "factura/form";
    }

    for (int i = 0; i < itemId.length; i++)
    {
        Producto producto = clienteService.findProductoById(itemId[i]);

        ItemFactura linea = new ItemFactura();
        linea.setCantidad(cantidad[i]);
        linea.setProducto(producto);
        factura.addItemFactura(linea);

        log.info("ID: " + itemId[i].toString() + ", cantidad: " + cantidad[i].toString());
    }

    //clienteService.saveFactura(factura);
    facturaDao.saveFactura(factura);
    status.setComplete();

    flash.addFlashAttribute("success", "Factura creada con exito");

    return "redirect:/ver/" + factura.getCliente().getId();
}
```

Este método lo que hace es comprobar si la factura no tiene errores y en caso de que todo este correcto recorre la lista de productos encontrando dicho producto mediante el servicio de cliente (ClienteServiceImplementado).

```
@Override
@Transactional(readOnly = true)
public Producto findProductoById(Long id) {
    return productoDao.findById(id).orElse(other:null)
}
```

Una vez encontrado crea una línea de factura a la que se le asigna el producto y la cantidad seleccionada y luego se añade a la factura.

Cuando se hayan seleccionado los productos que queremos se guardará la factura en la BD con el método saveFactura de FacturaDaoImplementado que comprueba que si el id no es nulo y si es mayor que 0 que lo hace es actualizar el registro sino lo inserta en la BD.

```
@Repository("facturaDaoJPA")
public class FacturaDaoImplementado implements IFacturaDaoJPA {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    @Transactional
    public void saveFactura(Factura factura)
    {
        if (factura.getId() != null && factura.getId() > 0)
        {
            entityManager.merge(factura);
        }
        else
        {
            entityManager.persist(factura);
        }
    }
}
```



#### 4.6.1. Buscar Productos

Para buscar productos se ha tenido que utilizar una plantilla para mostrar los productos de la tabla y luego en base a utilizar JavaScript se ha hecho que en el id buscar\_producto cargue una lista según el producto que se haya buscado y una vez seleccionado muestra los detalles de ese producto y calcula cuanto va a ser el total de la factura.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" lang="en">
<body>

<table th:fragment="itemsFactura" class="d-none">
  <tbody id="plantillaItemsFactura">
    <tr id="row_{ID}">
      <td class="d-none"><input type="hidden" value="{ID}" name="item_id[]"></td>
      <td>{NOMBRE}</td>
      <td>{PRECIO}</td>
      <td class="w-25"><input type="number" value="1" name="cantidad[]" id="cantidad_{ID}" class="form-control col-sm-2"
        |  onchange="itemsHelper.calcularImporte({ID}, {PRECIO}, this.value);"></td>
      <td><span id="total_importe_{ID}">0</span></td>
      <td><a href="#" class="btn btn-danger btn-xs" onclick="itemsHelper.eliminarLineaFactura({ID});">X</a></td>
    </tr>
  </tbody>
</table>
```

```
<script type="text/javascript" th:fragment="javascript">

$(document).ready(function() {

  $("#buscar_producto").autocomplete({

    source: function(request, response){

      $.ajax({
        url: "/factura/cargar-productos/" + request.term,
        dataType: "json",
        data: { term: request.term },
        success: function(data) {
          response($.map(data, function(item){
            return {
              value: item.id,
              label: item.nombre,
              precio: item.precio,
            };
          }));
        },
      });
    },
    select: function(event, ui) {
      //$("#buscar_producto").val(ui.item.label)

      if (itemsHelper.hasProducto(ui.item.value))
      {
        itemsHelper.incrementaCantidad(ui.item.val, ui.item.precio);
        return false;
      }

      var linea = $("#plantillaItemsFactura").html();

      linea = linea.replace(/ {ID}/g, ui.item.value);
      linea = linea.replace(/ {NOMBRE}/g, ui.item.label);
      linea = linea.replace(/ {PRECIO}/g, ui.item.precio);
    }
  });
});
```

```

        $("#cargarItemProductos tbody").append(linea);
        itemsHelper.calcularImporte(ui.item.value, ui.item.precio, 1)

        return false;
    }
});

$("form").submit(function() {
    $("#plantillaItemsFactura").remove();
    return;
});

});

var itemsHelper =
{
    calcularImporte: function(id, precio, cantidad)
    {
        $("#total_importe_" + id).html(parseInt(precio) * parseInt(cantidad));
        this.calcularGranTotal();
    },
    hasProducto: function(id)
    {
        var resultado = false;

        $('input[name="item_id[]"]').each(function() {
            if(parseInt(id) == parseInt($(this).val())) {
                resultado = true;
            }
        });

        return resultado;
    },
    incrementaCantidad: function(id, precio)
    {
        var cantidad = $("#cantidad_" + id).val() ? parseInt($("#cantidad_" + id).val()) : 0;
        $("#cantidad_" + id).val(++cantidad);
        this.calcularImporte(id, precio, cantidad);
    },

```

```

    },
    eliminarLineaFactura: function(id)
    {
        $("#row_" + id).remove();
        this.calcularGranTotal();
    },
    calcularGranTotal: function()
    {
        var total = 0;

        $('span[id^="total_importe_"]').each(function() {
            total += parseInt($(this).html());
        });

        $('#gran_total').html(total);
    }
}

```

</script>

## 4.7. Datos Factura

En la vista de los detalles del cliente había dos botones que se mencionaron anteriormente. Uno de ellos pertenece a los **Detalles** de la factura que realiza una petición **GET** a `/factura/ver/{id}` y llega al método **ver** del controlador **Factura**.

```
@GetMapping("/ver/{id}")
public String ver(@PathVariable(value = "id") Long id, Model model, RedirectAttributes flash)
{
    //Factura factura = clienteService.findFacturaById(id);
    Factura factura = facturaDao.findFactura(id);

    if (factura == null)
    {
        flash.addFlashAttribute("error", "La factura no existe en la base de datos");
        return "redirect:/listar";
    }

    model.addAttribute("factura", factura);
    model.addAttribute("titulo", "Factura: ".concat(factura.getDescripcion()));

    return "factura/ver";
}
```

Este método encuentra la factura mediante el FacturaDaoImplementado (utilizando el crud DAO JPA) y manda a la vista factura/ver dicha factura encontrada

```
@Override
@Transactional
public Factura findFactura(Long id)
{
    return entityManager.find(entityClass:Factura.class, id);
}
```

## 4.8. Borrar Factura

El otro botón pertenece a **Eliminar** la factura que realiza una petición **GET** a `/factura/eliminar/{id}` y llega al método **eliminar** del controlador **Factura**.

```
@GetMapping("/eliminar/{id}")
public String eliminar(@PathVariable(value = "id") Long id, RedirectAttributes flash)
{
    //Factura factura = clienteService.findFacturaById(id);
    Factura factura = facturaDao.findFactura(id);

    if (factura != null)
    {
        //clienteService.deleteFactura(id);
        facturaDao.deleteFactura(id);
        flash.addFlashAttribute("success", "Factura eliminada");
        return "redirect:/ver/" + factura.getCliente().getId();
    }

    flash.addFlashAttribute("error", "La factura no existe en la base de datos o no se pudo eliminar");
    return "redirect:/listar";
}
```

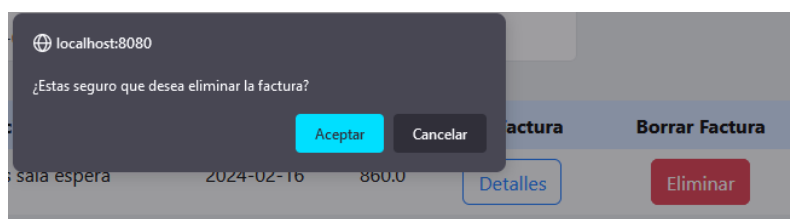
Este método encuentra la factura mediante el FacturaDaoImplementado (utilizando el crud DAO JPA) y si existe borra la factura con el método deleteFactura también del DAO de la Factura.

```
@Override
@Transactional
public Factura findFactura(Long id)
{
    return entityManager.find(entityClass:Factura.class, id);
}

@Override
@Transactional
public void deleteFactura(Long id)
{
    entityManager.remove(findFactura(id));
}
```

Una vez se clicka el botón salta una alerta para confirmar si se desea borrar la factura.

```
<td><a class="btn btn-outline-danger btn-xs" th:href="@{'/factura/eliminar/' + ${factura.id}}" th:text="'Eliminar'"
onclick="return confirm('¿Estas seguro que desea eliminar la factura?')"></a></td>
```



## 4.9. Listar Productos

Para ver los productos disponibles hay un enlace vinculado en la barra de navegación llamada Productos que hace una petición **GET** a /productos.

```
<li class="nav-item">
  <a class="nav-link" th:href="@{/productos}">Productos</a>
</li>
```

Esa petición llega al método **listarProductos** de ProductoController que encuentra todos los productos disponibles y lo manda a la vista **productos.html**

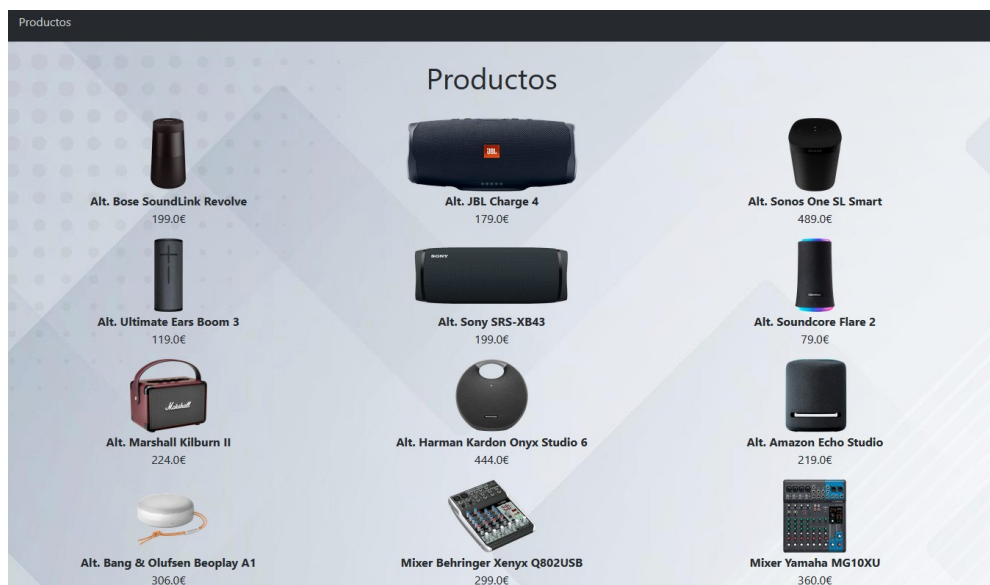
```
@Controller
@SessionAttributes("producto")
public class ProductoController {

    @Autowired
    private IClienteService clienteService;

    @RequestMapping(value = "/productos", method = RequestMethod.GET)
    public String listarProductos(Model model)
    {
        List<Producto> productos = clienteService.findAllProductos();

        model.addAttribute("productos", productos);
        return "productos";
    }
}
```

La vista recorre cada producto y muestra su imagen, nombre del producto y el precio





## 5. Instalación y Configuración de Despliegue

Para utilizar este proyecto es necesario tener **SpringToolSuite**.

Tienes dos formas de instalarlo:

- Mediante la extensión de VSCode llamada **Spring Boot Tools** ( la puedes encontrar en la sección de extensiones de VSCode o en la web de [SpringTools](#) )
- Otra forma es descargarlo para eclipse desde la web mencionada ( es necesario una versión de java ). Para instalarlo ejecutas el jar que has descargado y extraes el contenido del jar. Borra las carpetas META-INF, ui, contents.zip y el fichero SelfExtractor.class. Una vez este todo listo puedes ejecutar **SpringToolSuite4.exe**.

Para poder lanzar este proyecto es necesario un servidor web y un gestor de Base de Datos. La opción más utilizada y sencilla es XAMPP con la cual se lanza Apache y MySQL para acceder a PHPMYADMIN una aplicación web para administrar la BD.

Una vez se haya lanzado ejecutaras la aplicación de Springboot ( el archivo [nombre\_proyecto]Application.java )que lanzará un Tomcat para poder visualizar el proyecto.

```
@SpringBootApplication
public class EstudioSonidoApplication implements CommandLineRunner {

    @Autowired
    IUploadFileService uploadFileService;

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(EstudioSonidoApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        uploadFileService.deleteAll();
        uploadFileService.init();
    }
}
```

Cuando esté todo listo buscas el proyecto en la siguiente url: **localhost:8080** ( 8080 es el puerto por defecto que utiliza Springboot para lanzar sus aplicaciones ).

Para poder desplegar el proyecto abriremos una terminal y crearemos el fichero .jar de despliegue con el siguiente comando:

## .\mvnw.cmd package

Recuerda que este fichero se ejecuta cuando estas en la raíz del proyectos

```
> .\mvnw package
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.afrancop.springboot:estudio-sonido >-----
[INFO] Building estudio-sonido 0.0.1-SNAPSHOT
[INFO] from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ estudio-sonido ---
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO] Copying 52 resources from src\main\resources to target\classes
[INFO]
[INFO] --- compiler:3.11.0:compile (default-compile) @ estudio-sonido ---
[INFO] Changes detected - recompiling the module! :input tree
[INFO] Compiling 21 source files with javac [debug release 17] to target\classes
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ estudio-sonido ---
[INFO] skip non existing resourceDirectory D:\DAM 2\Acceso a Datos\2 EV\U4\Proyecto\estudio-sonido\src\test\resources
[INFO]
[INFO] --- compiler:3.11.0:testCompile (default-testCompile) @ estudio-sonido ---
[INFO] Changes detected - recompiling the module! :dependency
[INFO] Compiling 1 source file with javac [debug release 17] to target\test-classes
```

Una vez tengas desplegado el proyecto puedes probar a lanzarlo con el siguiente comando:

**java -jar [ruta del fichero.jar]**

[illegible]

Es necesario que en las variables de entorno del sistema tengas que crear una variable llamada JAVA\_HOME y con el valor de la ruta del jdk de java para que funcione.

