ALBARANCHECK

Proyecto de Desarrollo de Aplicaciones Multiplataformas

Autor: Leonardo Manuel Méndez Pérez

Fecha: 24/06/2024

C.F.G.S. Desarrollo de Aplicaciones Multiplataforma

Comunidad de Madrid - España

Índice

1. Objetivo General

Como parte de mi camino educativo, he decidido desarrollar un proyecto completo para una aplicación con utilidad práctica y que cubra una necesidad actual en un grupo de empresas.

El objetivo principal es recorrer todas las fases del diseño de software, utilizando diversas tecnologías en un único proyecto. Esto me permitirá evaluar las diferencias en cuanto al uso y aplicabilidad de cada tecnología según el contexto y tipo de proyecto.

Mi visión es crear una aplicación intuitiva y eficiente para la recepción de productos. La aplicación permitirá a los usuarios:

- Cargar albaranes en formato PDF.
- Escanear códigos de barras de productos o bultos.
- Verificar si todos los productos del albarán han sido recibidos.
- Generar un informe detallado indicando los productos recibidos, faltantes o con errores.

La aplicación final será accesible tanto en dispositivos Android como iOS, descargable desde las respectivas tiendas de aplicaciones, y podrá utilizarse con la cámara del dispositivo o un lector de códigos de barra Bluetooth. Para el análisis y extracción de los datos de los albaranes se utilizará una API de una IA.

Para lograr esto, seguiré un enfoque iterativo, desarrollando distintas versiones que evolucionarán en complejidad y funcionalidad:

Versión	**Tecnología**	**Plataforma**	**Objetivo**	**Características**
V 0.1	Java (sin frameworks)	Escritorio	Validar la idea	Lectura de códigos de barras, informe básico.
V 0.2	Java + MySQL	Escritorio	Persistencia básica	Guardar y recuperar datos de albaranes.
V 1.0	Java + Hibernate	Escritorio	ORM	Mapeo O/R, informes completos.
V 2.0	Spring Boot + Thymeleaf	Web	Acceso multiplataforma	Interfaz web, seguridad, informes.
V 3.0	FlutterFlow + Supabase	Android/iOS	Experiencia móvil	Escaneo con cámara, sincronización en la nube.
V 4.0	Android Studio + SQLite	Android	App nativa optimizada	Uso de Room, Bluetooth y cámara nativa.
V 5.0	IA + FlutterFlow + Supabase	Android/iOS	Automatización con IA	OCR, extracción automática de datos.

Cuadro 1: Plan de versiones del proyecto AlbaranCheck

2. Versión 0.1: Prototipo Básico (Java sin frameworks y sin persistencia)

2.1. Objetivo de esta versión

Desarrollar una aplicación de escritorio que permita gestionar la recepción de albaranes de entrega enviados por una compañía.

Las principales funcionalidades del sistema incluyen:

- Permitir la carga de un archivo PDF que contenga el albarán.
- Leer y procesar el contenido del albarán para mostrar la información resumida en pantalla (número de albarán, fecha, proveedor, lista de productos y cantidades).
- Habilitar un campo para escanear los códigos de barras de los productos recibidos, generando una lista de productos verificados.
- Comparar los productos escaneados con la lista del albarán para identificar discrepancias.
- Emitir un informe final que detalle los productos recibidos, faltantes y discrepancias.

2.2. Alcance y limitaciones

Esta versión inicial (V 0.1) tiene un alcance limitado:

- No se guardará un histórico de albaranes, verificaciones ni usuarios.
- La lista de productos con sus códigos EAN se almacenará en un archivo `.dat`.
- Sin persistencia en bases de datos.
- Sin manejo avanzado de excepciones.

2.3. Requisitos Funcionales

- RF.1 Carga de albaranes en PDF.
- RF.2 Lectura y visualización del contenido del albarán.
- RF.3 Escaneo de códigos de barras.
- RF.4 Verificación de recepción.
- RF.5 Generación de informe de recepción.
- RF.6 Crear nuevos productos y modificar los existentes.
- RF.7 Mostrar lista de productos.

2.4. Requisitos No Funcionales

- RNF.1 Interfaz intuitiva y fácil de usar.
- RNF.2 Procesamiento rápido de albaranes y escaneo.
- RNF.3 Portabilidad en diferentes máquinas.

2.5. Casos de Uso

- CU.01: Cargar albarán.
- CU.02: Ver albarán.
- CU.03: Escanear producto.
- CU.04: Verificar recepción.
- CU.05: Generar informe.
- CU.06: Crear/Modificar producto.
- CU.07: Mostrar lista de productos.

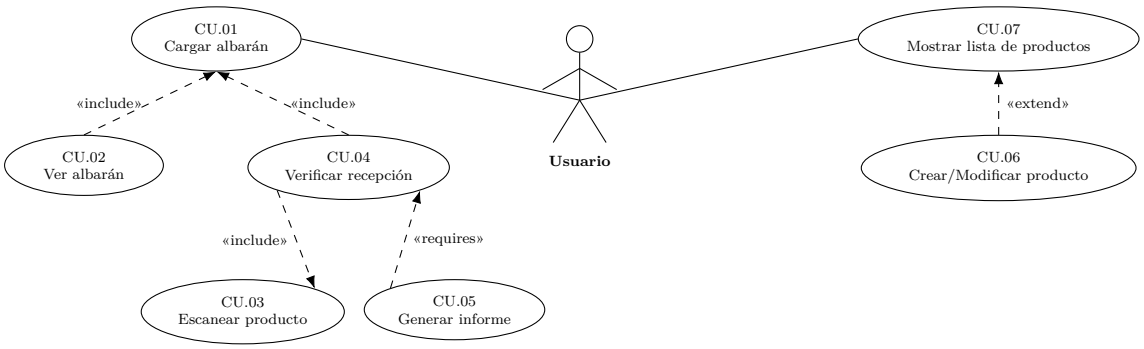

Figura 1: Casos de Uso.

2.6. Diseño

Diagrama de clases (UML): en esta versión se modela con las siguientes clases principales:

- **Main:** Clase principal que inicia la aplicación y crea una instancia de la interfaz gráfica (GUI).
- **GUI:** Clase encargada de la interfaz de usuario, que incluye ventanas para cargar el albarán, escanear códigos de barras y mostrar el informe. Interactúa con el usuario y delega las operaciones al Controlador.
- **Controlador:** Clase que actúa como intermediario entre la interfaz gráfica y la lógica del negocio. Gestiona la carga de albaranes, el escaneo de productos y la generación de informes.

- **Albarán:** Clase que representa un albarán con sus atributos (número, fecha, proveedor y lista de productos esperados).
- **ProductoEnAlbarán:** Clase que asocia un producto con la cantidad esperada en un albarán específico.
- **Producto:** Clase que contiene la información básica de un producto (código de barras y descripción).
- **ProductoVerificado:** Clase que representa un producto que ha sido verificado en un albarán, incluyendo la cantidad recibida.

El flujo de datos comienza en la GUI, pasa al Controlador para procesar la lógica, y utiliza las clases de modelo (Albarán, Producto, etc.) para estructurar la información.

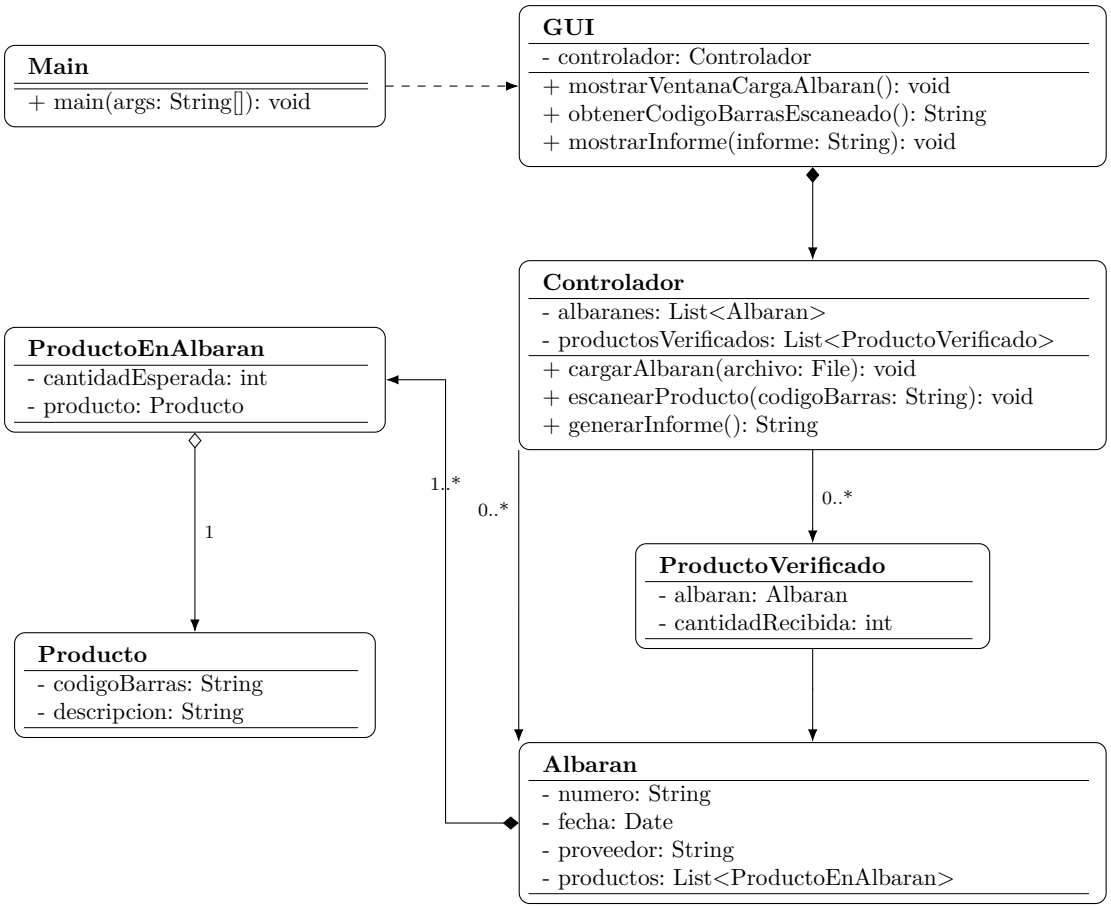

Figura 2: Diagrama de clases del sistema *AlbaranCheck*.

2.7. Implementación

Detalles de la implementación de cada componente:

- **Main:** Implementado como un punto de entrada simple que instancia la clase GUI y muestra la ventana principal.
- **GUI:** Utiliza la biblioteca Swing de Java para crear una interfaz gráfica básica. Incluye un botón para cargar el archivo PDF del albarán, un campo de texto para

introducir códigos de barras manualmente o mediante un lector, y un área de texto para mostrar el informe.

- **Controlador:** Gestiona la lógica principal utilizando métodos para cada una de las acciones necesarias.
- **Albarán, ProductoEnAlbarán, Producto, ProductoVerificado:** Clases POJO (Plain Old Java Objects) con constructores, getters y setters.

Librerías y herramientas utilizadas:

- Java SE (Standard Edition): Versión 8 o superior, utilizada como base para el desarrollo.
- Swing: Biblioteca nativa de Java para la creación de la interfaz gráfica.
- Apache PDFBox: Para la lectura de los albaranes en formato PDF.
- Eclipse/IntelliJ IDEA: Entorno de desarrollo integrado (IDE) utilizado para escribir, depurar y compilar el código.
- Archivo `.dat`: Usado como sustituto temporal para simular la carga de datos de productos.

2.8. Pruebas y Resultados

Pruebas unitarias y de integración realizadas:

- **Carga de albarán:** Se probó la capacidad del sistema para cargar un archivo PDF con un albarán y mostrarlos en la GUI.
- **Escaneo de productos:** Se simularon escaneos de códigos de barras válidos e inválidos para verificar que el sistema los registra correctamente.
- **Generación de informe:** Se comprobó que el informe refleja correctamente los productos escaneados y faltantes.
- **Carga de productos:** Se verificó que el sistema puede cargar y mostrar la lista de productos desde el archivo `.dat`.

Pruebas de integración:

- Se verificó que la GUI interactúa correctamente con el Controlador al cargar un albarán y escanear productos.
- Se probó el flujo completo: cargar albarán → escanear productos → generar informe.

Resultados de las pruebas y análisis de rendimiento:

- La carga de datos desde el archivo `.dat` fue exitosa en el 100 % de los casos probados.
- El escaneo de productos identificó correctamente los productos recibidos y faltantes.
- El informe se generó correctamente, mostrando discrepancias en todos los casos probados.
- Se crearon y modificaron productos sin errores.

Rendimiento:

- Tiempo promedio de carga de un albarán: 0.5 segundos.
- Tiempo de procesamiento por escaneo: <0.1 segundos.
- **Limitación:** El sistema se ralentiza con listas de más de 100 productos debido a la falta de optimización en la búsqueda.

3. Versión 0.2: Almacenamiento en Base de Datos (Java sin frameworks, MySQL)

3.1. Objetivos de esta versión

El objetivo principal de esta versión es implementar el almacenamiento persistente de datos utilizando una base de datos MySQL. Esto permitirá que la aplicación conserve información de los albaranes, productos escaneados, verificaciones y resultados, incluso después de cerrar la aplicación.

Se busca mantener una estructura de datos eficiente y relacional para garantizar la integridad de la información y facilitar futuras consultas y operaciones.

3.2. Requisitos Funcionales adicionales

- RF.8 Persistencia de albaranes en la base de datos.
- RF.9 Consulta de productos vinculados a un albarán desde la base de datos.
- RF.10 Registro y almacenamiento de verificaciones en la base de datos.
- RF.11 Generación de informes basados en la información persistida.
- RF.12 Gestión de usuarios en la aplicación (alta, baja, modificación).
- RF.13 Autenticación de usuarios para acceso al sistema.
- RF.14 Almacenamiento seguro de contraseñas utilizando hashing.

3.3. Requisitos No Funcionales adicionales

- RNF.4 La base de datos debe garantizar integridad referencial mediante claves foráneas.
- RNF.5 El tiempo de respuesta de las consultas debe ser inferior a 1 segundo con un volumen de hasta 10.000 registros.
- RNF.6 La conexión debe manejar adecuadamente errores de red y credenciales inválidas.
- RNF.7 Las contraseñas de usuarios deben almacenarse de manera segura (hashing y salting).

3.4. Casos de Uso adicionales

- CU.08: Registrar usuario. Permite crear un nuevo usuario en el sistema, asignándole un nombre de usuario y una contraseña que se almacena de forma segura mediante hashing.

- CU.09: Autenticación de usuario. El sistema valida las credenciales ingresadas contra los datos almacenados y concede acceso solo a usuarios válidos.
- CU.10: Gestión de usuarios. Posibilita la modificación y eliminación de usuarios existentes, garantizando integridad en las relaciones con albaranes y verificaciones.

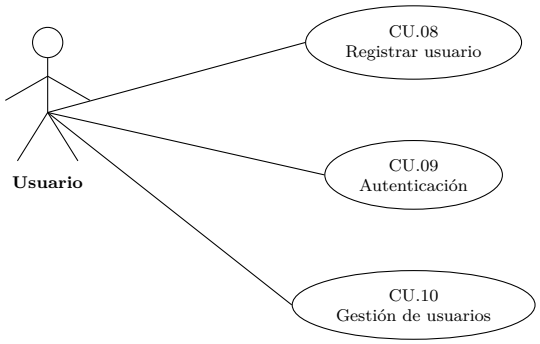

Figura 3: Diagrama de Casos de Uso para la gestión de usuarios (Versión 0.2, versión compacta).

3.5. Diseño

3.5.1. Interfaz de usuario

La interfaz gráfica se modificará para incluir:

- Formulario para cargar datos del albarán y almacenarlos en la base de datos.
- Visualización de productos vinculados a un albarán desde la base de datos.
- Botón para generar informes de verificación directamente desde los datos almacenados.

3.5.2. Modelo entidad-relación (ER)

El modelo entidad-relación está diseñado para representar las entidades clave de la aplicación: Albarán, Producto, ProductoEnAlbaran, ProductoVerificado y Usuario. Las relaciones entre estas entidades permiten realizar operaciones de consulta y almacenamiento de manera eficiente.

- **Albarán:** Representa los documentos de entrega cargados en el sistema.
- **Producto:** Contiene la información de cada producto registrado.
- **ProductoEnAlbaran:** Representa la relación entre un albarán y los productos que contiene.
- **ProductoVerificado:** Almacena los productos escaneados y su estado de verificación.
- **Usuario:** Representa a los usuarios que interactúan con el sistema.

3.6. Descripción de las tablas y relaciones

Tabla Albarán

- `id_albaran (PK)`: Identificador único del albarán.
- `numero (VARCHAR)`: Número del albarán.
- `fecha (DATE)`: Fecha del albarán.
- `proveedor (VARCHAR)`: Nombre del proveedor.

Tabla Producto

- `id_producto (PK)`: Identificador único del producto.
- `codigo_barras_unidad (VARCHAR)`: Código de barras del producto.
- `codigo_barras_bulto (VARCHAR)`: Código de barras del bulto.
- `descripcion (VARCHAR)`: Descripción del producto.

Tabla ProductoEnAlbaran

- `id_relacion (PK)`: Identificador único de la relación.
- `id_albaran (FK)`: Relación con el albarán asociado.
- `id_producto (FK)`: Relación con el producto asociado.
- `cantidad_esperada (INT)`: Cantidad esperada según el albarán.

Tabla ProductoVerificado

- `id_verificacion (PK)`: Identificador único de la verificación.
- `id_albaran (FK)`: Relación con el albarán asociado.
- `id_producto (FK)`: Relación con el producto escaneado.
- `cantidad_recibida (INT)`: Cantidad recibida del producto.

Tabla Usuario

- `id_usuario (PK)`: Identificador único del usuario.
- `nombre_usuario (VARCHAR)`: Nombre del usuario.
- `contrasena (VARCHAR)`: Contraseña del usuario.

3.6.1. Relaciones principales

- Albarán (1:N) ProductoEnAlbaran.
- Producto (1:N) ProductoEnAlbaran.
- ProductoEnAlbaran (1:N) ProductoVerificado.
- Producto (1:N) ProductoVerificado.

3.7. Implementación

3.7.1. Código SQL para la creación de la Base de Datos

```sql
CREATE TABLE Albaran (
    id_albaran INT AUTO_INCREMENT PRIMARY KEY,
    numero VARCHAR(255) NOT NULL,
    fecha DATE NOT NULL,
    proveedor VARCHAR(255) NOT NULL
);

CREATE TABLE Producto (
    id_producto INT AUTO_INCREMENT PRIMARY KEY,
    codigo_barras_unidad VARCHAR(255) NOT NULL,
    codigo_barras_bulto VARCHAR(255),
    descripcion VARCHAR(255) NOT NULL
);

CREATE TABLE ProductoEnAlbaran (
    id_relacion INT AUTO_INCREMENT PRIMARY KEY,
    id_albaran INT NOT NULL,
    id_producto INT NOT NULL,
    cantidad_esperada INT NOT NULL,
    FOREIGN KEY (id_albaran) REFERENCES Albaran(id_albaran) ON DELETE CASCADE,
    FOREIGN KEY (id_producto) REFERENCES Producto(id_producto) ON DELETE CASCADE
);

CREATE TABLE ProductoVerificado (
    id_verificacion INT AUTO_INCREMENT PRIMARY KEY,
    id_albaran INT NOT NULL,
    id_producto INT NOT NULL,
    cantidad_recibida INT NOT NULL,
    FOREIGN KEY (id_albaran) REFERENCES Albaran(id_albaran) ON DELETE CASCADE,
    FOREIGN KEY (id_producto) REFERENCES Producto(id_producto) ON DELETE CASCADE
);
```

Listing 1: Script de creación de tablas en MySQL

3.7.2. Conexión a la Base de Datos (Java con JDBC)

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public class ConexionDB {
    private static final String URL = "jdbc:mysql://localhost:3306/gestion_albaranes";
    private static final String USUARIO = "root";
    private static final String CONTRASENA = "password";

    public static Connection conectar() throws SQLException {
        return DriverManager.getConnection(URL, USUARIO, CONTRASENA);
    }
}
```

Listing 2: Clase de conexión a la base de datos con JDBC

3.7.3. Consultas SQL para guardar y recuperar datos

Insertar un albarán:

```
INSERT INTO Albaran (numero, fecha, proveedor) VALUES (?, ?, ?);
```

Insertar un producto en albarán:

```
INSERT INTO ProductoEnAlbaran (id_albaran, id_producto, cantidad_esperada)
VALUES (?, ?, ?);
```

Recuperar productos en albarán con verificaciones:

```
SELECT p.codigo_barras, p.descripcion, pa.cantidad_esperada, pv.cantidad_recibida
FROM ProductoEnAlbaran pa
JOIN Producto p ON pa.id_producto = p.id_producto
LEFT JOIN ProductoVerificado pv
  ON pa.id_producto = pv.id_producto AND pa.id_albaran = pv.id_albaran
WHERE pa.id_albaran = ?;
```

Registrar una verificación:

```
INSERT INTO ProductoVerificado (id_albaran, id_producto, cantidad_recibida)
VALUES (?, ?, ?);
```

3.8. Gestión de errores y seguridad

- Manejo de errores en la conexión a la base de datos mediante bloques `try-catch`.
- Uso de `PreparedStatement` en lugar de concatenación de cadenas para prevenir inyección SQL.
- Almacenamiento de contraseñas de usuario mediante algoritmos de hashing (p. ej. `bcrypt`).
- Validación de datos de entrada para evitar registros inconsistentes o incompletos.

3.9. Pruebas y Resultados

- Inserción y recuperación de albaranes: comprobación de que los datos se almacenan y se recuperan correctamente.

- Verificación de integridad referencial: eliminación de un albarán borra sus relaciones en cascada.
- Inserción de productos y asociación a albaranes probada correctamente.
- Registro de verificaciones probado con cantidades correctas e incorrectas.
- Tiempo de respuesta medio: <0.5 segundos en operaciones CRUD con hasta 5.000 registros.

4. Versión 1.0: Mapeo Objeto-Relacional (Java + MySQL + Hibernate)

4.1. Objetivo de esta versión

El objetivo principal es introducir un **mapeo objeto-relacional** mediante el uso de *Hibernate* y JPA, con el fin de simplificar la interacción con la base de datos, mejorar la mantenibilidad del código y permitir consultas más eficientes.

Los objetivos específicos son:

- **Simplificar el acceso a la base de datos:** Utilizar Hibernate para abstraer y automatizar el mapeo entre las clases Java y las tablas de MySQL.
- **Facilitar la gestión de entidades:** Reducir la cantidad de código repetitivo mediante anotaciones JPA y repositorios.
- **Optimizar consultas:** Usar HQL (*Hibernate Query Language*) para búsquedas complejas y eficientes.

4.2. Diseño

4.2.1. Interfaz de usuario

Se mantendrá una interfaz gráfica similar a la versión anterior, con las siguientes mejoras:

- Visualización directa de datos recuperados desde la base de datos.
- Incorporación de formularios para la **inserción**, **edición** y **eliminación** de registros de albaranes y productos.
- Integración con los servicios de Hibernate para operar sobre entidades.

4.2.2. Estructura del código

La aplicación se organizará en capas siguiendo buenas prácticas de arquitectura:

- **Entidades JPA:** Clases anotadas con `@Entity` que representan las tablas de la base de datos (Albarán, Producto, ProductoEnAlbaran, ProductoVerificado, Usuario).
- **Repositorios (DAO):** Interfaces o clases que implementan operaciones CRUD sobre las entidades, aprovechando la abstracción de Hibernate.
- **Servicios:** Capas intermedias que contienen la lógica de negocio y coordinan la interacción entre interfaz gráfica y repositorios.
- **Configuración de Hibernate:** Mediante el archivo `hibernate.cfg.xml`, incluyendo parámetros de conexión, dialecto de MySQL y opciones de mapeo.

4.3. Implementación

4.3.1. Funcionalidades clave

- **Carga de albaranes en PDF:** Integración con `Apache PDFBox` para extraer datos de documentos y transformarlos en entidades persistentes.
- **Gestión de entidades:** Uso de Hibernate para guardar, actualizar, eliminar y consultar registros en la base de datos.
- **Generación de informes:** Consultas con HQL que permiten obtener listados detallados de verificaciones y discrepancias.

4.3.2. Ejemplo de entidad JPA

A continuación, un ejemplo de cómo se representa la tabla `Albaran` como entidad en Hibernate:

```
import jakarta.persistence.*;
import java.util.Date;
import java.util.List;

@Entity
@Table(name = "Albaran")
public class Albaran {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idAlbaran;

    private String numero;

    @Temporal(TemporalType.DATE)
    private Date fecha;

    private String proveedor;

    @OneToMany(mappedBy = "albaran", cascade = CascadeType.ALL)
    private List<ProductoEnAlbaran> productos;

    // Getters y Setters
}
```

Listing 3: Entidad JPA para Albarán

4.3.3. Comparativa con JDBC tradicional

El uso de Hibernate proporciona varias ventajas frente al enfoque de conexión manual con JDBC:

- Elimina gran parte del código repetitivo de SQL embebido en Java.
- Gestiona automáticamente las relaciones entre entidades (1:N, N:M).

- Permite consultas orientadas a objetos mediante HQL o Criteria API.
- Asegura mayor mantenibilidad y escalabilidad en proyectos de mediana y gran envergadura.

4.4. Requisitos Funcionales

- RF.8 Persistencia de albaranes mediante Hibernate.
- RF.9 Gestión de entidades con operaciones CRUD (crear, leer, actualizar, eliminar).
- RF.10 Consultas complejas utilizando HQL.
- RF.11 Formularios para gestión de albaranes y productos desde la GUI.

4.5. Requisitos No Funcionales

- RNF.4 Separación clara en capas (entidades, repositorios, servicios).
- RNF.5 Configuración centralizada de la base de datos en `hibernate.cfg.xml`.
- RNF.6 Uso de anotaciones JPA estándar para garantizar portabilidad.

4.6. Tecnologías utilizadas

- **Java SE 8+**: Lenguaje de programación principal.
- **Hibernate y JPA**: Mapeo objeto-relacional.
- **MySQL**: Sistema de gestión de base de datos.
- **Apache PDFBox**: Lectura de archivos PDF.
- **IDE (Eclipse o IntelliJ IDEA)**: Entorno de desarrollo.

5. Versión 2.0: Aplicación Web (Java + Spring Boot + Thymeleaf + MySQL)

5.1. Objetivo de esta versión

El objetivo principal de esta versión es transformar la aplicación en un sistema accesible desde cualquier navegador web, utilizando **Spring Boot** y el patrón *MVC* para estructurar la lógica de presentación, negocio y acceso a datos.

Los objetivos específicos son:

- **Accesibilidad:** Permitir el acceso a la aplicación desde cualquier dispositivo conectado a la red.
- **Modernización del desarrollo:** Aprovechar las capacidades de **Spring Boot** para simplificar la configuración y despliegue.
- **Mejora en la experiencia de usuario:** Crear una interfaz web atractiva y responsive mediante **Thymeleaf**.
- **Seguridad:** Incorporar autenticación de usuarios y gestión de roles para controlar el acceso a las distintas secciones.

5.2. Diseño

5.2.1. Interfaz de usuario

La aplicación web contará con:

- **Página de inicio (Dashboard):** Panel principal con resumen de albaranes y notificaciones.
- **Gestión de albaranes:** Formularios web para cargar documentos PDF, verificar productos y editar información.
- **Generación de informes:** Descarga de reportes en formatos PDF y CSV.
- **Seguridad integrada:** Acceso diferenciado según roles (`ADMIN`, `USER`).

5.2.2. Arquitectura

Se emplea el patrón **Modelo–Vista–Controlador (MVC)** mediante Spring:

- **Controladores (Controllers):** Gestionan peticiones HTTP y delegan en los servicios.
- **Servicios (Services):** Contienen la lógica de negocio.
- **Repositorios (Repositories):** Se implementan con **Spring Data JPA** para interactuar con MySQL.

- **Vistas (Views):** Plantillas dinámicas con **Thymeleaf**.

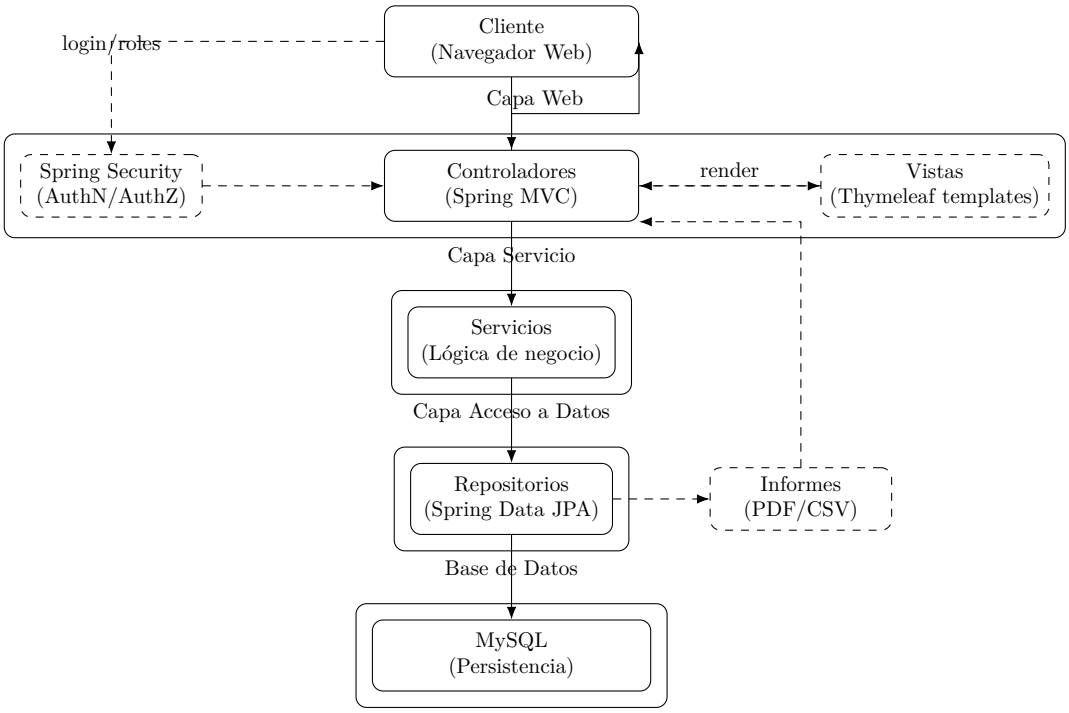

Figura 4: Arquitectura MVC en Spring Boot.

5.3. Implementación

5.3.1. Funcionalidades clave

- **Carga de albaranes y verificación en línea:** Subida de archivos PDF y procesamiento inmediato.
- **Consulta y gestión de datos:** Edición de albaranes y productos desde formularios web.
- **Informes descargables:** Generación de informes en PDF o CSV.
- **Seguridad:** Login con Spring Security y control de acceso basado en roles.

5.3.2. Ejemplo de Controlador Spring MVC

```
@Controller
@RequestMapping("/albaranes")
public class AlbaranController {

    @Autowired
    private AlbaranService albaranService;

    @GetMapping
    public String listarAlbaranes(Model model) {
        model.addAttribute("albaranes", albaranService.obtenerTodos());
```

```
        return "albaranes/lista"; // Vista Thymeleaf
    }

    @PostMapping("/nuevo")
    public String guardarAlbaran(@ModelAttribute Albaran albaran) {
        albaranService.guardar(albaran);
        return "redirect:/albaranes";
    }
}
```

Listing 4: Ejemplo de controlador Spring MVC con Thymeleaf

5.3.3. Seguridad con Spring Security

La aplicación contará con un sistema de autenticación y autorización basado en Spring Security:

- Configuración de usuarios y roles en base de datos (`Usuario`, `Rol`).
- Login con formulario personalizado.
- Restricciones de acceso a rutas: por ejemplo, sólo los administradores pueden modificar productos o eliminar albaranes.

5.4. Requisitos Funcionales

- RF.12 Carga de albaranes desde el navegador.
- RF.13 Verificación de productos en línea.
- RF.14 Edición y consulta de datos mediante formularios web.
- RF.15 Generación de informes en PDF y CSV.
- RF.16 Autenticación y control de acceso por roles.

5.5. Requisitos No Funcionales

- RNF.7 Aplicación web responsive y multiplataforma.
- RNF.8 Arquitectura en capas basada en MVC.
- RNF.9 Uso de plantillas Thymeleaf para mantener separación de lógica y presentación.
- RNF.10 Integración con Spring Security para robustez en la gestión de usuarios.

5.6. Tecnologías utilizadas

- **Java SE 8+**
- **Spring Boot y Spring MVC**

- **Spring Data JPA**
- **Thymeleaf**
- **MySQL**
- **Bootstrap (opcional)** para mejorar la apariencia.

6. Versión 3.0: Aplicación Multiplataforma (FlutterFlow + Supabase)

6.1. Objetivo de esta versión

Los objetivos principales son:

- **Experiencia multiplataforma:** Entregar una única base de UI y lógica que se publique en **Android**, **iOS** y **Web/PWA**.
- **Sincronización en la nube:** Utilizar **Supabase** para autenticación, almacenamiento y sincronización en tiempo real.
- **Interacción nativa:** Aprovechar cámara y capacidades del dispositivo (incl. lector externo) mediante acciones y extensiones de Flutter/FlutterFlow.

6.2. Diseño

6.2.1. Interfaz de usuario

- **Pantalla principal (Dashboard):** Acceso rápido a carga de albaranes, escaneo de productos y generación de informes.
- **Flujos de navegación:** Minimizan el número de taps/clics hasta cada función.
- **Diseño adaptativo:** *Responsive* para móviles y *adaptive* para tablet/escritorio (breakpoints de FlutterFlow y layouts reactivos).

6.2.2. Arquitectura

- **FlutterFlow (Flutter):** Construcción visual de pantallas, acciones y estados; exportación a Android, iOS y Web/PWA.
- **Supabase:** Backend (Postgres administrado), *Auth*, *Storage* y *Realtime*.
- **Persistencia local: SQLite** para modo *offline-first* y sincronización diferida.

6.3. Implementación

6.3.1. Funcionalidades clave

- **Carga y procesamiento de albaranes:** Captura (cámara) o subida (archivo) de PDF/imagen.
- **Escaneo de códigos:** Lectura de EAN/UPC/QR desde la cámara (y soporte para lector externo donde aplique).

- **Sincronización de datos:** Supabase como fuente de verdad y base local para operar sin conexión.

6.3.2. Estrategia *offline-first* y sincronización

- **Cola de operaciones:** Altas/cambios se guardan en local con estado `PENDIENTE`.
- **Reconciliación:** Al recuperar conectividad, un proceso de sincronización aplica cambios en Supabase y marca `SINCRONIZADO`.
- **Conflictos:** Política *last-write-wins* con marcas temporales; para albaranes críticos, historial de cambios.
- **Suscripciones:** Canales *realtime* para refrescar vistas en dispositivos conectados.

6.3.3. Publicación multiplataforma

- **Android/iOS:** Generación de binarios desde el proyecto exportado de FlutterFlow (firmado y subida a tiendas).
- **Web/PWA:** Despliegue de `flutter build web`; configuración de `manifest.json` y `service worker` para modo PWA (instalable y con caché).

6.3.4. Seguridad y privacidad

- **Autenticación:** Email/contraseña (y proveedores sociales si se habilitan) con sesiones seguras.
- **Autorización:** RLS (Row-Level Security) en Supabase para aislar datos por usuario/organización.
- **Credenciales:** Tokens en *secure storage* del dispositivo/navegador (Keychain/-Keystore; en Web usar almacenamiento seguro y expiración).
- **Permisos:** Solicitud de cámara/archivos conforme a plataforma/ navegador (HTTPS requerido para Web).

6.4. Requisitos Funcionales (V3.0)

- RF.17 Inicio/cierre de sesión en Android, iOS y Web.
- RF.18 Carga de albaranes desde cámara o archivos (móvil y escritorio).
- RF.19 Escaneo de códigos de barras con cámara del dispositivo.
- RF.20 Operación *offline* con sincronización posterior.
- RF.21 Consulta y edición de albaranes/productos desde móvil y navegador.
- RF.22 Generación/descarga de informes (móvil y Web/PWA).
- RF.23 Despliegue Web con instalación como PWA.

6.5. Requisitos No Funcionales (V3.0)

- RNF.11 **Codebase única** para Android/iOS/Web.
- RNF.12 Tiempo de arranque < 2 s en móviles objetivo; < 3 s en Web con caché PWA.
- RNF.13 *Offline-first* con recuperación transparente y manejo de conflictos.
- RNF.14 UI *responsive/adaptive* con rendimiento fluido en listas largas (*lazy loading*).
- RNF.15 Seguridad de sesiones y RLS activas en Supabase.

6.6. Tecnologías

- **FlutterFlow (Flutter):** Construcción visual + exportación multiplataforma.
- **Supabase:** Auth, Postgres, Storage, Realtime.
- **SQLite/Isar:** Base local para modo sin conexión.
- **PWA (Web):** `manifest.json`, `service worker` y despliegue static hosting.

6.6.1. Esquema mínimo de tablas en Supabase (SQL)

```
create table albaran (
  id_albaran bigserial primary key,
  numero text not null,
  fecha date not null,
  proveedor text not null,
  owner uuid not null references auth.users(id)
);

create table producto (
  id_producto bigserial primary key,
  codigo_barras_unidad text not null,
  codigo_barras_bulto text,
  descripcion text not null,
  owner uuid not null references auth.users(id)
);

create table producto_en_albaran (
  id_relacion bigserial primary key,
  id_albaran bigint not null references albaran(id_albaran) on delete cascade,
  id_producto bigint not null references producto(id_producto) on delete cascade,
  cantidad_esperada int not null
);

create table producto_verificado (
  id_verificacion bigserial primary key,
  id_albaran bigint not null references albaran(id_albaran) on delete cascade,
  id_producto bigint not null references producto(id_producto) on delete cascade,
  cantidad_recibida int not null,
  ts timestamptz default now()
```

```
);

-- RLS (ejemplo)
alter table albaran enable row level security;
create policy "owner-only"
on albaran for all
using (auth.uid() = owner) with check (auth.uid() = owner);
```

Listing 5: Tablas principales en Supabase (simplificado)

6.7. Pruebas y resultados esperados

- **Pruebas de conectividad:** Login/logout, expiración de sesión y reautenticación.
- **Pruebas *offline*:** Insertar escaneos sin red y verificar sincronización al reconectar.
- **Rendimiento de listas:** Carga de > 1000 filas con *lazy loading*.
- **Cámara y permisos:** Solicitud y revocación de permisos, manejo de errores del escáner.
- **Integridad:** Consistencia entre SQLite y Supabase tras múltiples conflictos.

7. Versión 4.0: Aplicación Nativa Android (Android Studio + SQLite)

7.1. Objetivo de esta versión

El propósito de esta versión es desarrollar una aplicación nativa en Android que aproveche al máximo las capacidades del sistema operativo y del hardware del dispositivo. Los objetivos específicos incluyen:

- **Optimización en Android:** Uso de librerías y componentes nativos para maximizar el rendimiento.
- **Rendimiento y usabilidad:** Experiencia fluida y optimizada con interfaz adaptada al entorno Android.
- **Gestión local robusta:** Almacenamiento confiable en SQLite, con soporte para operación sin conexión.

7.2. Diseño

7.2.1. Interfaz de usuario

- **Pantalla principal:** Accesos directos a carga de albaranes, escaneo de códigos y generación de informes.
- **Navegación optimizada:** Uso de `Activity` y `Fragment` para una navegación fluida.
- **Indicadores visuales:** Gráficos y notificaciones que muestran estado de verificación (recibidos vs. faltantes).

7.2.2. Arquitectura

Se implementa el patrón **MVVM (Model–View–ViewModel)** para separar responsabilidades:

- **Model:** Entidades y acceso a datos mediante **Room (SQLite)**.
- **View:** Pantallas y fragmentos con `XML layouts`.
- **ViewModel:** Maneja la lógica de negocio y expone datos reactivos con **LiveData**.

Características adicionales:

- **Persistencia:** Uso de **Room** para simplificar el acceso a SQLite.
- **Cámara nativa:** Integración de `CameraX API`.
- **Bluetooth API:** Conexión con lectores externos de códigos de barras.

7.3. Implementación

7.3.1. Funcionalidades clave

- **Carga de albaranes:** Captura de PDF desde almacenamiento local.
- **Escaneo de códigos:** Lectura con la cámara o dispositivos Bluetooth usando ZXing.
- **Persistencia local:** Almacenamiento robusto en SQLite mediante Room.
- **Generación de informes:** Creación de reportes en PDF/CSV desde los datos almacenados.

7.3.2. Ejemplo de entidad Room

```
@Entity(tableName = "productos")
public class Producto {
    @PrimaryKey(autoGenerate = true)
    private int id;

    @ColumnInfo(name = "codigo_barras")
    private String codigoBarras;

    @ColumnInfo(name = "descripcion")
    private String descripcion;

    // Getters y Setters
}
```

Listing 6: Entidad Room para la tabla Producto

7.3.3. Ejemplo de DAO con Room

```
@Dao
public interface ProductoDao {
    @Insert
    void insertar(Producto producto);

    @Query("SELECT * FROM productos WHERE codigo_barras = :codigo")
    Producto buscarPorCodigo(String codigo);

    @Query("SELECT * FROM productos")
    List<Producto> listarTodos();
}
```

Listing 7: DAO para Producto

7.3.4. Ejemplo de ViewModel

```
public class ProductoViewModel extends AndroidViewModel {

    private final ProductoRepository repository;
    private final LiveData<List<Producto>> productos;

    public ProductoViewModel(@NonNull Application application) {
        super(application);
        repository = new ProductoRepository(application);
        productos = repository.listarProductos();
    }

    public LiveData<List<Producto>> getProductos() {
        return productos;
    }

    public void insertar(Producto producto) {
        repository.insertar(producto);
    }
}
```

Listing 8: ViewModel con LiveData

7.4. Requisitos Funcionales (V4.0)

- RF.23 Cargar albaranes desde archivos PDF.
- RF.24 Escanear productos con cámara o lector Bluetooth.
- RF.25 Almacenar y consultar datos de forma local en SQLite.
- RF.26 Generar informes exportables (PDF/CSV).

7.5. Requisitos No Funcionales (V4.0)

- RNF.16 Uso de arquitectura MVVM para separación de responsabilidades.
- RNF.17 Persistencia implementada con Room para evitar SQL manual.
- RNF.18 Experiencia de usuario fluida y optimizada para Android.
- RNF.19 Uso eficiente de cámara y Bluetooth.

7.6. Tecnologías

- **Android Studio:** IDE para desarrollo Android.
- **Room:** Librería de persistencia para SQLite.
- **ZXing:** Biblioteca de escaneo de códigos de barras.
- **CameraX y Bluetooth API:** Integración con hardware del dispositivo.

8. Versión 5.0: App Android con IA (OpenAI) + Backend propio on-prem (Spring Boot)

8.1. Objetivo de esta versión

Desarrollar una solución en la que la app Android nativa procese **albaranes en PDF** mediante **IA (OpenAI)**, manteniendo la **persistencia y seguridad** a través de una **API propia** implementada con Spring Boot y desplegada en un **servidor físico local**.

Objetivos específicos:

- **Extracción automática** de campos del albarán (número, fecha, proveedor, líneas de productos).
- **Seguridad**: las credenciales de IA se guardan y usan *exclusivamente* en el servidor (nunca en el cliente Android).
- **Persistencia on-prem**: almacenamiento y consulta a través de la API Spring (MySQL/JPA).
- **Experiencia Android**: flujo de captura/subida de PDF, validación y revisión del resultado, y guardado.

8.2. Diseño

8.2.1. Interfaz de usuario (Android)

- **Pantalla principal**: accesos a "Cargar albarán (PDF/Foto)", "Verificar/Editar extracción", "Guardar/Consultar".
- **Flujo guiado**: (1) seleccionar/capturar PDF, (2) enviar a IA, (3) revisar/editar campos, (4) persistir.
- **Estado y feedback**: indicadores de subida, procesamiento y validación; avisos ante errores de red o formato.

8.2.2. Arquitectura

- **App Android (MVVM)**: UI (`Activity/Fragment`) + `ViewModel` + `Repository` (Retrofit/OkHttp + LiveData).
- **API Spring on-prem**: endpoints REST para **persistencia** y un endpoint **proxy de IA** que llama a OpenAI.
- **Servicio de IA**: extracción de datos (visión sobre PDF) con salida JSON estructurada.
- **BD del servidor**: MySQL con JPA/Hibernate (esquema coherente con versiones anteriores).

8.3. Implementación

8.3.1. Flujo recomendado (seguro)

1. La app Android envía el **PDF** al endpoint `/ai/extract` de la API Spring (multipart).
2. La API **sube** el PDF a IA (o lo envía en Base64) y solicita **JSON estructurado** con los campos requeridos.
3. La API devuelve a Android un **JSON** con los campos del albarán; el usuario revisa/edita.
4. La app confirma y guarda vía endpoints `/albaranes` y relacionados (`/productos`, `/verificaciones`).

```
public class AlbaranExtraidoDTO {
  public String numero;
  public String fecha;       // ISO-8601 recomendado (yyyy-MM-dd)
  public String proveedor;
  public java.util.List<Item> items;

  public static class Item {
    public String codigo;
    public String descripcion;
    public int cantidad;
  }
}
```

Listing 9: DTO de extracción (Android Java)

```
public interface AiService {
  @Multipart
  @POST("/ai/extract")
  Call<AlbaranExtraidoDTO> extract(@Part MultipartBody.Part file);
}

public interface AlbaranesService {
  @POST("/albaranes")
  Call<AlbaranDTO> guardar(@Body AlbaranDTO dto);

  @GET("/albaranes/{id}")
  Call<AlbaranDTO> obtener(@Path("id") long id);

  @GET("/albaranes")
  Call<java.util.List<AlbaranDTO>> listar();
}
```

Listing 10: Android (Java): envío de PDF al backend on-prem

```
public class NetworkModule {

  public static Retrofit buildRetrofit(String baseUrl, Supplier<String>
    tokenSupplier) {
    OkHttpClient client = new OkHttpClient.Builder()
        .callTimeout(java.time.Duration.ofSeconds(30))
        .connectTimeout(java.time.Duration.ofSeconds(15))
        .readTimeout(java.time.Duration.ofSeconds(30))
        .addInterceptor(chain -> {
          Request original = chain.request();
          Request.Builder b = original.newBuilder();
          String jwt = tokenSupplier.get();
          if (jwt != null && !jwt.isEmpty()) {
            b.header("Authorization", "Bearer " + jwt);
          }
          return chain.proceed(b.build());
        })
        .build();

    return new Retrofit.Builder()
        .baseUrl(baseUrl)
        .addConverterFactory(JacksonConverterFactory.create())
        .client(client)
        .build();
  }
}
```

Listing 11: Repository: Retrofit/OkHttp con JWT

```
public class AlbaranRepository {

  private final AiService aiService;
  private final AlbaranesService albaranesService;

  public AlbaranRepository(Retrofit retrofit) {
    this.aiService = retrofit.create(AiService.class);
    this.albaranesService = retrofit.create(AlbaranesService.class);
  }

  public void extraerDesdePdf(File pdf, Callback<AlbaranExtraidoDTO> callback) {
    RequestBody req = RequestBody.create(pdf, MediaType.parse("application/pdf"));
    MultipartBody.Part part = MultipartBody.Part.createFormData("file",
    pdf.getName(), req);
    aiService.extract(part).enqueue(callback);
  }

  public void guardarAlbaran(AlbaranDTO dto, Callback<AlbaranDTO> callback) {
    albaranesService.guardar(dto).enqueue(callback);
  }

  public void listarAlbaranes(Callback<java.util.List<AlbaranDTO>> callback) {
    albaranesService.listar().enqueue(callback);
  }
}
```

Listing 12: Repository Android (Java) con Retrofit

```
public class AlbaranViewModel extends AndroidViewModel {

  private final MutableLiveData<AlbaranExtraidoDTO> extraccion = new
    MutableLiveData<>();
  private final MutableLiveData<Boolean> loading = new MutableLiveData<>(false);
  private final MutableLiveData<String> error = new MutableLiveData<>();
  private final AlbaranRepository repository;

  public AlbaranViewModel(@NonNull Application app) {
    super(app);
    // Proveer baseUrl y token
    Supplier<String> tokenSupplier = () -> /* recuperar JWT de SharedPreferences */
    "";
    Retrofit retrofit = NetworkModule.buildRetrofit("https://tu-servidor-onprem/",
    tokenSupplier);
    repository = new AlbaranRepository(retrofit);
  }

  public LiveData<AlbaranExtraidoDTO> getExtraccion() { return extraccion; }
  public LiveData<Boolean> isLoading() { return loading; }
  public LiveData<String> getError() { return error; }

  public void subirPdfYExtraer(File pdf) {
    loading.postValue(true);
    repository.extraerDesdePdf(pdf, new Callback<AlbaranExtraidoDTO>() {
      @Override public void onResponse(Call<AlbaranExtraidoDTO> call,
    Response<AlbaranExtraidoDTO> resp) {
        loading.postValue(false);
        if (resp.isSuccessful() && resp.body() != null)
    extraccion.postValue(resp.body());
        else error.postValue("Error servidor: " + resp.code());
      }
      @Override public void onFailure(Call<AlbaranExtraidoDTO> call, Throwable t) {
        loading.postValue(false);
        error.postValue("Error de red: " + t.getMessage());
      }
    });
  }

  public void guardarAlbaran(AlbaranDTO dto, Runnable onOk, Consumer<String> onErr) {
    repository.guardarAlbaran(dto, new Callback<AlbaranDTO>() {
      @Override public void onResponse(Call<AlbaranDTO> call, Response<AlbaranDTO>
    resp) {
        if (resp.isSuccessful()) onOk.run();
        else onErr.accept("Error guardando: " + resp.code());
      }
      @Override public void onFailure(Call<AlbaranDTO> call, Throwable t) {
        onErr.accept("Error de red: " + t.getMessage());
      }
    });
  }
}
```

Listing 13: ViewModel Android (Java) con LiveData

```
@RestController
@RequestMapping("/ai")
public class AiController {

  private final AiService aiService;

  public AiController(AiService aiService) { this.aiService = aiService; }

  @PostMapping(value="/extract", consumes=MediaType.MULTIPART_FORM_DATA_VALUE)
  public ResponseEntity<Map<String,Object>> extract(@RequestPart("file")
    MultipartFile file) throws IOException {
    Map<String,Object> result = aiService.extractFromPdf(file);
    return ResponseEntity.ok(result);
  }
}
```

Listing 14: Spring: controlador para extracción IA

```
@Service
public class AiService {

  @Value("${openai.api.key}")
  private String apiKey;

  // TODO: Implementar cliente HTTP y la llamada al servicio de IA que procese PDF ->
    JSON

  public Map<String,Object> extractFromPdf(MultipartFile file) throws IOException {
    // - Subir el PDF al servicio de IA (o codificar en Base64)
    // - Solicitar salida JSON con numero, fecha, proveedor, items {codigo,
    descripcion, cantidad}
    // - Parsear y devolver como Map<String,Object> (o DTO)
    return Map.of(
      "numero", "ALB-12345",
      "fecha", "2024-06-24",
      "proveedor", "Proveedor S.A.",
      "items", java.util.List.of(
          Map.of("codigo","1234567890123","descripcion","Producto A","cantidad",10),
          Map.of("codigo","2345678901234","descripcion","Producto B","cantidad",5)
      )
    );
  }
}
```

Listing 15: Spring: servicio de extracción (esqueleto)

```
@RestController @RequestMapping("/albaranes")
class AlbaranController {
  @Autowired private AlbaranService service;

  @PostMapping public Albaran save(@RequestBody AlbaranDTO dto) { return
    service.save(dto); }
```

```
  @GetMapping("/{id}") public Albaran get(@PathVariable Long id) { return
    service.get(id); }
  @GetMapping public List<Albaran> list() { return service.list(); }
}
```

Listing 16: Spring: REST de dominio simplificado

8.4. Seguridad

Persistencia on-prem (Spring Data JPA): endpoints de dominio

- **Claves de IA en el servidor**: `OPENAI_API_KEY` vía variables de entorno; nunca en la app.
- **Autenticación de la API**: Spring Security (JWT) entre Android y el servidor on-prem.
- **Transporte**: HTTPS (reverse proxy Nginx/Traefik) y límites de tamaño de subida.
- **Privacidad**: decidir si guardar/no guardar PDFs originales; anonimizar registros si procede.

8.5. Requisitos Funcionales (V5.0)

- RF.27 Subir albaranes (PDF) desde Android a la API on-prem.
- RF.28 Extraer campos del albarán con IA y mostrar resultado editable.
- RF.29 Persistir albaranes y líneas en BD on-prem mediante endpoints REST.
- RF.30 Consultar, filtrar y exportar albaranes e informes desde la app.

8.6. Requisitos No Funcionales (V5.0)

- RNF.20 **Seguridad**: claves y llamadas de IA siempre desde servidor; cliente sin secretos.
- RNF.21 **Rendimiento**: extracción < 5 s para PDFs de hasta 2–3 páginas (objetivo).
- RNF.22 **Disponibilidad on-prem**: monitorización y *backups* del servidor físico.
- RNF.23 **Trazabilidad**: *logging*/auditoría de peticiones de extracción y cambios.

8.7. Tecnologías

- **Android**: Java, Retrofit/OkHttp, MVVM (ViewModel/LiveData).
- **Spring Boot**: Web, Security (JWT), Data JPA, MySQL.
- **Servicio de IA**: procesamiento de PDF con respuesta JSON estructurada.

- **Infra on-prem**: servidor físico (Linux), Nginx/Traefik, certificados TLS.