



**Proyecto final**

**Ingeniería de software**

**I**

**Kevin David Rodriguez Riveros**

**Frank Sebastian Pardo Amaya**

**Jorge Andrés Torres Leal**

**Departamento de Ingeniería de Sistemas e Industrial**

**Universidad Nacional de Colombia - Sede Bogotá**

**Facultad de Ingeniería**

**31 de Enero de 2025**

## 1. Verificar README

Revisar que esté correcto en el repositorio.

## 2. Levantamiento de Requerimientos

En principio con el equipo tuvimos varias ideas a desarrollar, concordamos en un problema común en pequeñas y medianas empresas con: la gestión de sus inventarios. Muchas empresas aún dependen de hojas de cálculo o registros manuales, nosotros mismos conocemos gente que lo sigue haciendo a día de hoy. Sabemos que eso puede generar errores, pérdida de información, falta de precisión en la misma y también dificultades en el control de productos. Viéndolo así, decidimos desarrollar StockEase, como sistema de gestión de inventarios que facilita el registro, control y seguimiento de productos de manera sencilla e intuitiva.

Pensamos en un sistema que permitiera registrar cada producto con detalles clave como nombre, código, precio de compra y venta, cantidad, lote y fecha de vencimiento. Además ofrece una interfaz de reportes donde los usuarios pueden filtrar productos por nombre, código o rango de fechas, asegurando un acceso rápido a la información. Hasta el momento hemos implementado la funcionalidad de registro y visualización de productos y planeamos también incluir módulos para gestión de ventas y alertas automáticas para productos próximos a vencer o con bajo stock.

En lo técnico trabajamos con Java para el desarrollo del software y definimos la estructura de la base de datos para garantizar un almacenamiento eficiente y seguro. Nuestro objetivo es que StockEase sea una solución accesible y práctica para cualquier chuzo que necesite mejorar su gestión de inventarios sin depender de sistemas complejos y caros.

## 3. Análisis de Requerimientos

- **Clasificación MoSCoW:**

Funcionalidad	Clasificación	Tiempo Estimado	Descripción	Justificación
Gestionar Perfiles y Credenciales de acceso	Must	5 días	Gestiona los usuarios y sus roles, crea las credenciales para acceder a la información en el software.	Tiene una complejidad media, nos implica validaciones de datos para seguridad y control.
Registro de productos	Must	3 días	Permite registrar productos con sus características principales.	Se necesitan formularios con validaciones y CRUD básico en la base de datos, lo cual no es complejo.
Gestión de Ventas	Must	5 días	Es para la validación de ventas de los productos y actualizarlos en el stock.	Implica lógica adicional para actualizar el inventario y guardar datos de transacciones, lo que añade

				dificultad.
Sistema de autenticación de usuario	Must	5 días	Permite la autenticación y control de acceso de los usuarios al sistema.	Se usará un sistema estándar de autenticación con hashing de contraseñas, lo que requiere validaciones adicionales.
Consultar producto	Must	8 días	Permite consultar productos del inventario, ayuda al control del stock y ver el “catálogo” de productos actual.	Requiere el diseño de la interfaz asociado a las funcionalidades de consulta, además del desarrollo de las consultas basadas en ciertos campos lo cual puede demorar un poco más, además si bien es un must deben priorizarse las más “Must”.
Informe de ventas	Must	13 días	Genera un visual general de ventas, puede ser útil para la toma de decisiones empresa basándose en ello.	Involucra consultas SQL optimizadas y diseño de reportes visuales con opciones de filtrado dinámico.
Alertas de stock bajo	Should	8 días	Mandar una notificación o alerta cuando un producto tiene bajo stock.	Implica programación de eventos automáticos y consultas constantes a la base de datos o al menos cálculos para notificar sobre niveles bajos de productos en la base de datos.
Alerta de vencimiento de productos	Should	8 días	Notifica cuando un producto dentro de la base de datos está a punto de vencer.	Similar a las alertas de stock, pero con validación de fechas en la base de datos y notificaciones automatizadas basadas en la fecha de vencimiento ingresada en cada producto.
Modo Oscuro	Could	13 días	Proporciona una interfaz visual alternativa y puede mejorar la experiencia del usuario.	Requiere de modificar cada panel de la interfaz y su configuración sin afectar la lógica del sistema.
Factura Electrónica	Won't	N/A	Darí una factura junto con los sistemas de facturación fiscal.	Requiere integración con plataformas externas y cumplimiento de regulaciones, lo que lo hace inviable en esta fase.

App Móvil	Won't	N/A	Versión mobile del sistema.	El desarrollo de una app nativa implica mayor inversión de tiempo y recursos, por lo que se deja para futuras versiones.
-----------	-------	-----	-----------------------------	--

**Nota:**

- Se priorizarán las funcionalidades Must en la primera fase del desarrollo para el MVP..
- Las funcionalidades Should y Could se agregarán en iteraciones futuras si hay tiempo y recursos.
- Las funcionalidades Won't no serán implementadas en esta versión del sistema.

#### 4. Análisis de gestión de Software

Para estimar el tiempo necesario para el desarrollo de StockEase, se utilizó la técnica MoSCoW combinada con la estimación por puntos de historia basados en la secuencia de Fibonacci (1, 2, 3, 5, 8, 13...). Esto permitió asignar tiempos de manera realista según la complejidad de cada funcionalidad.

El proceso de desarrollo se estructuró en tres fases principales, con una duración total de 10 semanas:

Fase	Actividad	Duración
Diseño y planificación.	Definición de requerimientos, herramientas y estructura del sistema.	2 semanas
Desarrollo del MVP	Implementación de funcionalidades MUST identificadas en el análisis MoSCoW.	6 semanas
Pruebas y ajustes	Testing, corrección de errores, optimización y mejoras en la interfaz.	2 semanas

La distribución del trabajo se basa en sprints de dos semanas, asegurando entregas funcionales en cada iteración.

#### Distribución del trabajo por Sprint

Cada sprint se diseñó con base en la priorización de funcionalidades críticas y la experiencia del equipo. Se organizó de la siguiente manera:

Sprint	Funcionalidad	Dificultad (Puntos)	Responsable
1	Registro y edición de productos	5	Dev 1

1	Sistema de autenticación	3	Dev 2
2	Gestión de ventas	8	Dev 3
2	Reportes de inventario	5	Dev 1
3	Alertas de stock bajo	5	Dev 2
3	Mejoras en UI/UX	3	Dev 3

Cada funcionalidad se programó estratégicamente para que su desarrollo no genere cuellos de botella en el flujo de trabajo del equipo.

La planificación del tiempo se realizó en función de la complejidad de cada tarea:

- **Tareas básicas (3 días):** Implementaciones simples como CRUD de productos o cambios en la interfaz.
- **Tareas intermedias (5 días):** Funcionalidades con lógica adicional, como autenticación de usuarios o reportes de ventas.
- **Tareas complejas (8 días):** Implementaciones que requieren procesamiento de datos avanzado o consultas optimizadas.

#### Costo: Estimación Justificada

Para definir un presupuesto realista, se investigaron precios de mercado en Glassdoor, Talent y LinkedIn, considerando los costos de recursos humanos, infraestructura y herramientas necesarias para el desarrollo de StockEase.

Recurso	Cantidad	Costo Mensual (COP)	Duración (Meses)	Total (COP)
Desarrollador Junior	2	3'000,000	2.5	15'000,000
Diseñador UX/UI	1	5'000,000	1	5'000,000
Base de datos (AWS RDS)	1 instancia	500,000	2.5	1'250,000
Servidor en la nube (AWS EC2)	1	700,000	2.5	1'750,000
Licencia y Herramientas	-	1'000,000	-	1'000,000

Costo total estimado: 24'000,000 COP

#### Justificación del Presupuesto

Se contrataron dos desarrolladores junior en lugar de un senior para reducir costos sin comprometer la calidad.

AWS se utilizará para base de datos y servidores, optimizando costos sin sacrificar escalabilidad.

El diseñador UX/UI solo estará involucrado un mes, ya que su intervención es clave solo en la fase de optimización visual.

Licencias y herramientas incluyen software de control de versiones y plataformas de prueba.

### **Alcance: Definición del MVP y Futuras Iteraciones**

Para garantizar una entrega viable en el tiempo y presupuesto estimados, el alcance del MVP (Producto Mínimo Viable) se definió con base en el análisis MoSCoW, priorizando funcionalidades esenciales y descartando aquellas que requieren mayor tiempo y recursos.

Funcionalidades Incluidas (MVP)	Funcionalidades Fuera del Alcance
Registro y edición de productos	Facturación electrónica
Sistema de autenticación	Aplicación móvil
Gestión de ventas	Integración con otras plataformas
Reportes y búsqueda avanzada	Soporte multiusuario avanzado
Alertas de stock bajo	Inteligencia artificial para predicción de stock

### **Justificación:**

-Las funcionalidades incluidas son cruciales para el funcionamiento del sistema y pueden ser desarrolladas dentro del tiempo estimado.

-Las funcionalidades excluidas requieren integraciones externas o mayores recursos técnicos y financieros, por lo que se dejaron para futuras iteraciones.

Además, se prevé que el MVP permitirá a las empresas empezar a gestionar sus inventarios de manera eficiente con las funcionalidades mínimas necesarias, dejando espacio para mejoras y expansiones futuras.

**Recursos Humanos:** El equipo de desarrollo está conformado por tres programadores que trabajarán en esta implementación de gestor de inventarios.

Elemento	Costo
Programador Junior*2	\$6'000,000 COP
Diseñador Interfaz	\$5'000,000 COP.

**Costo estimado total:** \$25,000,000 COP.

**MVP:** \$17,000,000 COP.

- Interfaz básica de búsqueda.
- Registro, visualización, edición y eliminación de productos en el inventario

con nombre, cantidad, precio entre otros detalles.

- Base de datos con almacenamiento de inventario.

## 5. Diseño y Arquitectura

En este caso usamos ChatGPT para dar un mejor diseño a la descripción del punto y el prompt que usamos fue:

“Amplia aún más la siguiente decisión sobre elegir la arquitectura MVC:

Arquitectura del Sistema: Se utilizó una arquitectura MVC (Modelo Vista Controlador)

**Modelo (Model)** Representa la lógica de negocio y la gestión de datos. Maneja la interacción con la base de datos y las reglas de negocio. No depende directamente de la interfaz gráfica. Puede notificar a la vista sobre cambios en los datos.

**Vista (View)** Se encarga de la interfaz de usuario y la presentación de los datos. No tiene lógica de negocio, solo muestra información obtenida del modelo. Puede ser una interfaz gráfica (JavaFX, Swing) o una página web (HTML, CSS, JavaScript).

**Controlador (Controller)** Actúa como intermediario entre la vista y el modelo. Recibe las acciones del usuario desde la vista y actualiza el modelo. Decide qué datos enviar a la vista y qué acciones ejecutar en el modelo.

Se eligió este modelo ya que la parte gráfica se pudo aislar a algo más low-code por medio de hacerlo con JavaFX y SceneBuilder, además lo elegimos ya que vimos en la web que se pueden realizar pruebas unitarias en el modelo sin depender de la interfaz gráfica, se pueden simular vistas o modelos en entornos de prueba sin afectar el resto del sistema, soporta múltiples vistas sin modificar la lógica central del modelo de manera que cuando lleguemos a la sección de testing esperamos que sea más fácil y fluida la realización de pruebas, esto fue una parte de las consideraciones importantes de la arquitectura”

### Resultado:

En el diseño del sistema, se optó por utilizar la arquitectura **MVC (Modelo-Vista-Controlador)** debido a su **modularidad, escalabilidad y facilidad para realizar pruebas unitarias**. A continuación, se detallan los aspectos clave de cada componente y las razones específicas por las cuales esta arquitectura fue la mejor elección para el proyecto.

---

### Descripción de la Arquitectura MVC

#### ♦ **Modelo (Model)**

- Representa la **lógica de negocio** y la gestión de datos.
- Maneja la interacción con la **base de datos**, asegurando la persistencia de la información.
- No depende directamente de la **interfaz gráfica**, lo que permite que los datos sean reutilizables en diferentes vistas.
- Puede notificar a la vista sobre **cambios en los datos** mediante mecanismos como el



## Observer Pattern o la inyección de dependencias.

- Se pueden realizar **pruebas unitarias sobre el modelo** sin necesidad de simular una interfaz de usuario.

### ♦ Vista (View)

- Se encarga de la **interfaz de usuario (UI)** y la **presentación de los datos**.
- No contiene **lógica de negocio**, solo **muestra información obtenida del modelo** y recibe interacciones del usuario.
- Puede utilizar tecnologías como **JavaFX** y **SceneBuilder** (low-code) para facilitar el diseño gráfico sin afectar la lógica del sistema.
- Soporta **múltiples interfaces**, como aplicaciones de escritorio (JavaFX, Swing) o aplicaciones web (HTML, CSS, JavaScript).
- Se pueden realizar **pruebas automatizadas en la UI** con herramientas como **TestFX** en JavaFX.

### ♦ Controlador (Controller)

- Actúa como **intermediario** entre la **vista** y el **modelo**.
- Recibe las **acciones del usuario** desde la vista y actualiza el modelo en consecuencia.
- Decide qué datos enviar a la vista y **qué acciones ejecutar en el modelo**.
- Se encarga de manejar eventos, como **clics de botones o entrada de datos**, garantizando la separación de responsabilidades.
- Puede integrarse con **patrones como Dependency Injection** para mejorar la escalabilidad y mantenimiento.

---

## Razones para Elegir MVC en este Proyecto

### 1 Separación de Responsabilidades

- **Evita la dependencia entre la lógica de negocio y la interfaz gráfica**, lo que facilita la reutilización del código.
- La interfaz gráfica puede modificarse sin alterar el funcionamiento del **modelo** o **controlador**.
- Permite que diferentes equipos trabajen de forma independiente en la **UI**, **lógica de negocio** y **gestión de datos**.

### 2 Facilidad para Implementar Pruebas Unitarias

- **El modelo puede ser probado de manera aislada**, sin necesidad de interactuar con la vista o el controlador.
- Se pueden utilizar **frameworks de pruebas como JUnit y Mockito** para validar la lógica de negocio.
- Se pueden **simular vistas o modelos** en entornos de prueba sin afectar el resto del sistema.
- Las pruebas pueden realizarse en **entornos controlados** sin la necesidad de una interfaz gráfica, lo que acelera el desarrollo y la detección de errores.

### 3 Soporte para Múltiples Vistas

- Se pueden agregar diferentes interfaces de usuario (**aplicaciones de escritorio, móviles o web**) sin modificar la lógica central del **modelo**.
- Esto facilita la escalabilidad y **posibilita la migración a otras tecnologías sin afectar el núcleo del sistema**.

### 4 Uso de JavaFX y SceneBuilder

- JavaFX permite crear **interfaces gráficas modernas y escalables** sin necesidad de escribir código extenso.
- SceneBuilder facilita el diseño visual de la UI, permitiendo un enfoque **low-code** en la parte gráfica.
- SceneBuilder genera archivos **FXML**, que se pueden vincular fácilmente con el controlador sin mezclar la lógica del negocio con la interfaz.

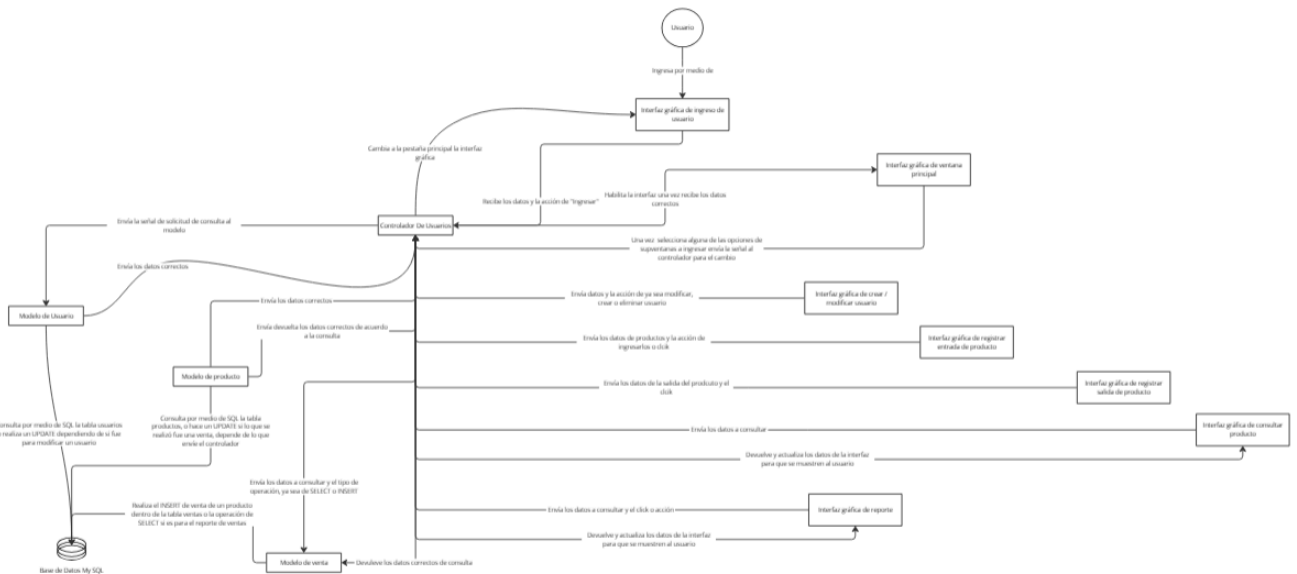
### 5 Escalabilidad y Mantenimiento

- **La modularidad de MVC** permite agregar nuevas funcionalidades sin afectar el código existente.
- Se puede mejorar el sistema a futuro agregando nuevos módulos sin alterar las estructuras fundamentales.
- La organización del código facilita la **colaboración entre desarrolladores** y mejora la **legibilidad y mantenibilidad** del proyecto.

### 6 Integración con APIs y Servicios Externos

- El modelo se puede conectar con **APIs externas, bases de datos o servicios en la nube** sin depender de la vista.
  - Esto permite **desarrollar una arquitectura distribuida o basada en microservicios** en el futuro.
-

La elección de **MVC** en este proyecto proporciona una **arquitectura sólida, escalable y modular**, asegurando una mejor **organización del código, facilidad de mantenimiento y soporte para pruebas unitarias**. Además, el uso de **JavaFX y SceneBuilder** ayuda a desacoplar la UI de la lógica de negocio, permitiendo un desarrollo más eficiente y adaptable.



NOTA: Revisar en el apartado de diagramas, la imagen se ve mejor

**Para la descripción de la elección de la base de datos usamos el siguiente prompt que contiene texto de la justificación que hicimos en la entrega preliminar:**

Mejora la siguiente explicación de porqué elegimos esa base de datos y esas entidades, además con las modificaciones que hiciste también puedes hablar de ellas:

La base de datos tiene tres tablas las cuales son productos, usuarios y ventas, básicamente la de usuarios es una tabla individual sin ningún tipo de conexión con las demás ya que solo se encarga de administrar los nombres, contraseñas y niveles de los empleados (Gerentes o empleados), luego sí existe una relación de uno a muchos entre productos y ventas ya que un producto puede tener varias ventas, sin embargo esto no se ha implementado, además se piensa añadir en un futuro una relación en la cual se refleje quién realizó x o y venta, lo cual implica relacionar usuarios y ventas, como claves primarias se tendría en productos el código junto con el lote, y como llave foránea se tendría en venta el lote y el código del producto.

Se eligió una base de datos SQL ya que es importante para futuros desarrollos el que exista una estructura bien definida de las relaciones entre tablas, incluso si aún no existe completamente.

## Elección de la Base de Datos y Diseño Relacional

Para este sistema, se optó por utilizar una **base de datos SQL**, ya que permite estructurar la información de manera clara y definir relaciones bien establecidas entre las entidades. Esto garantiza que la información sea **coherente, escalable y fácil de gestionar en el futuro**, incluso si algunas relaciones aún no están completamente implementadas.

La base de datos está compuesta por **tres tablas principales**:

- 1 Usuarios (**usuarios**)
- 2 Productos (**productos**)
- 3 Ventas (**ventas**)

Cada una de estas entidades tiene un propósito específico dentro del sistema:

♦ **Usuarios (**usuarios**)**

- Esta tabla **no estaba originalmente relacionada con las demás**, ya que su función principal es almacenar los datos de autenticación de los empleados.
- Contiene los siguientes atributos:
  - **usuario**: Identificador único (clave primaria).
  - **password**: Contraseña asociada al usuario.
  - **jefe**: Un valor **booleano (TINYINT(1))** que indica si el usuario es **Gerente (1)** o **Empleado (0)**.
- Se decidió relacionarla con la tabla **ventas**, ya que en el futuro será **necesario registrar qué usuario realizó cada venta**. Esto permite auditoría y control de ventas por empleado.

♦ **Productos (**productos**)**

- Contiene la información de los productos disponibles en el sistema.
- Se utiliza **codigo** como **clave primaria**, ya que cada producto tiene un identificador único.
- Se almacenan detalles como:
  - **precioCompra**: Para registrar el costo del producto al momento de su adquisición.
  - **precio**: Precio de venta del producto.
  - **cantidad**: Controla el stock del producto en inventario.
  - **vencimiento**: Fecha de caducidad (si aplica).
  - **lote**: Permite identificar productos provenientes de diferentes lotes de producción.

♦ **Ventas (**ventas**)**


- **Registra todas las transacciones de venta realizadas en el sistema.**
- Se estableció una **relación de uno a muchos con productos**, ya que un **producto puede estar en múltiples ventas**.
- También se implementó una **relación con usuarios** para registrar quién realizó cada venta.
- La clave primaria de esta tabla es **id** (autoincremental), y contiene las siguientes claves foráneas:
  - **codigo** → Relacionado con **productos(codigo)**, asegurando que solo se vendan productos existentes.
  - **usuario** → Relacionado con **usuarios(usuario)**, permitiendo identificar quién realizó la venta.

---

## Mejoras en la Base de Datos

Se realizaron varias mejoras con respecto al diseño inicial para garantizar una mejor integridad de datos y optimizar su estructura:

### ♦ Implementación de Claves Foráneas (**FOREIGN KEY**)

Se añadieron claves foráneas en **ventas** para establecer relaciones con **productos** y **usuarios**, garantizando que cada venta:  **Siempre esté asociada a un producto existente.**

 **Siempre tenga un usuario registrado como responsable de la transacción.**

### ♦ Inclusión de **precioCompra** y **precioVenta** en **ventas**

- Se decidió **almacenar el precio de compra y venta en la tabla **ventas****, ya que los precios pueden cambiar con el tiempo.
- Esto permite que el sistema **mantenga un historial de precios exacto al momento de cada venta**, evitando inconsistencias en reportes financieros.

### ♦ Manejo de Eliminaciones (**ON DELETE SET NULL / RESTRICT**)

- **ON DELETE RESTRICT en **codigo de productos****
  - Evita la eliminación de un producto si ya ha sido vendido.
  - Protege la integridad del historial de ventas.
- **ON DELETE SET NULL en **usuario de ventas****
  - Permite eliminar empleados sin afectar las ventas registradas.
  - Si un empleado es eliminado, sus ventas permanecerán en la base de datos, pero sin un usuario asignado.

---

## Justificación de Elección de Base de Datos SQL

Se eligió **una base de datos SQL** debido a varias razones clave:

### **1** Integridad Relacional y Estructura Definida

- SQL permite **definir relaciones entre entidades** de manera clara y asegurarse de que los datos sean **consistentes**.
- **Incluso si algunas relaciones aún no están completamente implementadas**, la estructura definida facilita futuras expansiones.

### **2** Escalabilidad y Mantenimiento

- Al utilizar claves primarias y foráneas bien estructuradas, la base de datos puede **crecer sin problemas** y mantenerse de manera eficiente.
- Permite realizar **consultas optimizadas**, reduciendo la carga en el sistema.

### 3 Seguridad y Control de Acceso

- SQL permite gestionar permisos para proteger la información, asegurando que solo **usuarios autorizados** puedan acceder o modificar ciertos datos.
- Con la relación entre **usuarios** y **ventas**, se podrá **rastrear qué usuario realizó cada venta**, lo que es útil para auditoría y control interno.

### 4 Flexibilidad para Futuros Desarrollos

- Se podrán agregar nuevas funcionalidades sin necesidad de rediseñar toda la base de datos.
- Por ejemplo, se podrá añadir una tabla de **proveedores** para gestionar el abastecimiento de productos.

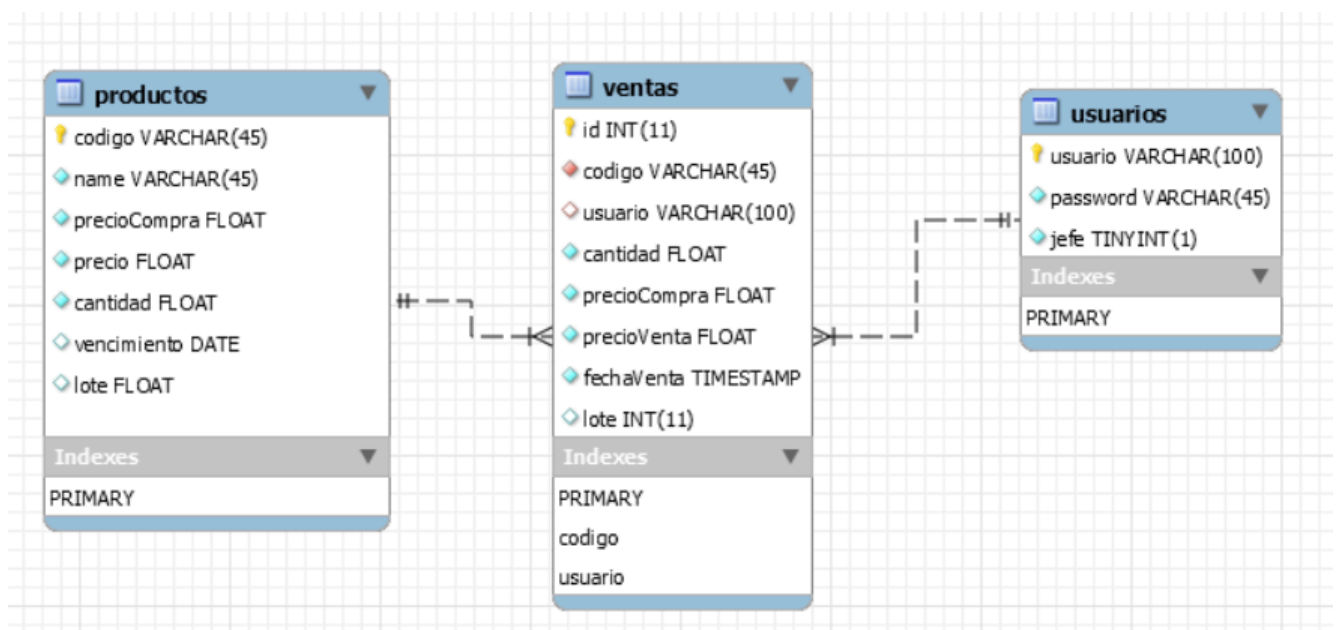
## 📌 Conclusión

El diseño de esta base de datos se basa en principios de **integridad, escalabilidad y mantenibilidad**. La elección de una base de datos SQL permite estructurar correctamente las relaciones entre **usuarios**, **productos** y **ventas**, asegurando que la información esté **organizada, protegida y lista para futuras expansiones**.

✅ **Se implementaron relaciones clave** para conectar ventas con productos y usuarios, lo que mejora el control de inventario y facilita auditorías.

✅ **Se optimizó la estructura** para manejar precios históricos y evitar pérdidas de datos por eliminación accidental.

✅ **El sistema está preparado para crecer**, permitiendo futuras mejoras sin necesidad de reestructuraciones masivas.



## 6. Patrones de diseño

Para el caso del patrón de diseño ya que ninguno había trabajado antes con uno, decidimos pedirle una recomendación a ChatGPT sobre cuál podría ser el mejor, la primera opción fue Factory Method pero nos resultó un poco “complejo” y no muy en línea con nuestras ideas, luego solicitamos otra idea y ahí fue cuando nos mostró el patrón Singleton que nos pareció buenísimo y muy en línea con lo que llevábamos desarrollado, así pues el prompt fue:

“Ahora haz lo siguiente, elige un patrón de diseño que sea bueno para la arquitectura MVC y sea simple y conocido, osea bien documentado pues: En este apartado, se evaluará la comprensión de los patrones de diseño estudiados. Documentar si aplicaron algún patrón de diseño durante el desarrollo (por ejemplo, Singleton, Factory, Observer, etc.). Justificar el uso de cada patrón, explicando: Qué problema resuelve. Por qué fue necesario en el proyecto. Cómo se implementó. Incluir un diagrama UML que ilustre cómo se aplicó cada patrón dentro del sistema (las clases involucradas, dependencias etc etc).”

Si bien parece algo improvisado, realmente nos tomamos el tiempo de analizar si sí podríamos implementarlo y estar en línea con lo que propone este patrón de diseño ya que finalmente sí lo implementamos y sabemos que es algo que se debe mostrar, así que lo tomamos muy seriamente.

### Patrón de Diseño Aplicado: Singleton en la Arquitectura MVC

El **patrón Singleton** es un patrón **creacional** que garantiza que una clase tenga **una única instancia en toda la aplicación**, proporcionando un punto de acceso global a esa instancia.

---

### ¿Qué problema resuelve Singleton?

En aplicaciones **MVC**, hay componentes que **deben ser únicos en toda la ejecución del programa**. Por ejemplo:

- ✓ **Conexión a la base de datos** → Para evitar múltiples conexiones innecesarias.
- ✓ **Gestión de configuración global** → Para almacenar variables compartidas en toda la aplicación.
- ✓ **Control de sesiones de usuario** → Para asegurar que un usuario autenticado mantenga su estado en todo el sistema.

El **Singleton** resuelve estos problemas asegurando que solo **una instancia de una clase específica** exista en la aplicación.

---

### Justificación del Uso de Singleton en este Proyecto

Durante el desarrollo del sistema, identificamos la necesidad de mantener **una única instancia** de ciertos elementos, como:

- 1 **Conexión a la base de datos** → Evita abrir y cerrar múltiples conexiones innecesarias.



② **Gestor de sesión de usuario** → Permite que un usuario autenticado mantenga su estado sin tener que autenticarse nuevamente.

✓ **Este patrón es más simple que Factory Method** y se implementa con **muy pocas líneas de código**.

---

## Implementación en el Proyecto

A continuación, se presentan **dos ejemplos** de cómo se usa Singleton en este proyecto.

---

### ♦ Ejemplo 1: Singleton para la Conexión a la Base de Datos

Java

```
public class DatabaseConnection {
    private static DatabaseConnection instance;
    private Connection connection;

    private DatabaseConnection() {
        try {
            connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/
mi_base", "usuario", "password");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }

    public Connection getConnection() {
        return connection;
    }
}
```

### ✓ Explicación:

- El constructor es `private`, lo que impide que otras clases creen instancias.
  - Se usa `getInstance()` para obtener la única instancia de `DatabaseConnection`.
  - Si la instancia aún **no existe**, se crea una nueva.
  - Si ya existe, simplemente se devuelve la misma.
  - **Evita múltiples conexiones** a la base de datos, mejorando el rendimiento.
- 

### ♦ Ejemplo 2: Singleton para el Control de Sesión del Usuario

Java

```
public class SessionManager {
    private static SessionManager instance;
    private String usuarioAutenticado;

    private SessionManager() {}

    public static SessionManager getInstance() {
        if (instance == null) {
            instance = new SessionManager();
        }
        return instance;
    }

    public void setUsuario(String usuario) {
        this.usuarioAutenticado = usuario;
    }

    public String getUsuario() {
        return usuarioAutenticado;
    }
}
```

### ✓ Explicación:

- `SessionManager` solo tendrá **una única instancia** en la aplicación.
- Se almacena el usuario autenticado en `usuarioAutenticado`.
- Se usa `getInstance()` para acceder a la única instancia de la sesión.
- Esto **permite compartir el estado del usuario** en toda la aplicación sin necesidad de crear múltiples objetos.

---

## Diagrama UML del Patrón Singleton en el Proyecto

Aquí se muestra **cómo se aplica el patrón Singleton** en la arquitectura MVC:

---

## Diagrama UML (Explicación en Texto)

Unset

```
+-----+
| DatabaseConnection |
+-----+
| - instance: DatabaseConnection (static) |
| - connection: Connection                |
+-----+
| + getInstance(): DatabaseConnection    |
| + getConnection(): Connection          |
+-----+

      ▲
      |

+-----+
|  Cualquier parte del sistema  |
+-----+
| - Usa DatabaseConnection.getInstance() |
+-----+
```

✓ La clase **DatabaseConnection** tiene una única instancia compartida en todo el sistema.

✓ Cualquier parte del código puede acceder a **DatabaseConnection.getInstance()** sin crear nuevos objetos.

---

## Beneficios de Usar Singleton

✓ **Fácil de implementar** → Se usa en pocas líneas de código.

✓ **Optimización de recursos** → Evita crear múltiples conexiones innecesarias.

✓ **Mayor seguridad** → Permite controlar el acceso a recursos críticos como la base de datos.

✓ **Útil en MVC** → Se puede aplicar en múltiples partes del sistema sin generar acoplamiento.

---

## Conclusión

El **patrón Singleton** fue una excelente elección para este sistema MVC porque:

- ✓ **Evita múltiples conexiones a la base de datos, mejorando el rendimiento.**
- ✓ **Permite gestionar sesiones de usuario sin duplicar instancias.**
- ✓ **Es más fácil de implementar y entender que otros patrones como Factory Method.**