DocumentaciÃ³n TÃ©cnica: Sistema POS para Tienda de Accesorios MÃ³viles

Autor: Manus AI
Fecha: 6 de junio de 2025
VersiÃ³n: 1.0

Tabla de Contenidos

1. [IntroducciÃ³n](#introducciÃ³n)
2. [Arquitectura del Sistema](#arquitectura-del-sistema)
3. [TecnologÃas Utilizadas](#tecnologÃas-utilizadas)
4. [Estructura del Proyecto](#estructura-del-proyecto)
5. [Backend (NestJS)](#backend-nestjs)
6. [Frontend (ReactJS + Electron)](#frontend-reactjs--electron)
7. [Base de Datos (PostgreSQL)](#base-de-datos-postgresql)
8. [Seguridad](#seguridad)
9. [API Reference](#api-reference)
10. [Despliegue](#despliegue)
11. [Mantenimiento](#mantenimiento)
12. [SoluciÃ³n de Problemas](#soluciÃ³n-de-problemas)
13. [Referencias](#referencias)

IntroducciÃ³n

El Sistema POS (Point of Sale) para Tienda de Accesorios MÃ³viles es una aplicaciÃ³n completa diseÃ±ada para gestionar todas las operaciones comerciales de una tienda de accesorios para dispositivos mÃ³viles. Este sistema permite la gestiÃ³n de ventas, inventario, productos, clientes, proveedores, empleados, gastos y reportes, proporcionando una soluciÃ³n integral para el negocio.

PropÃ³sito del Sistema

El propÃ³sito principal del Sistema POS es automatizar y optimizar los procesos de negocio de una tienda de accesorios mÃ³viles, facilitando la gestiÃ³n diaria y proporcionando informaciÃ³n valiosa para la toma de decisiones. El sistema estÃ¡ diseÃ±ado para ser utilizado por diferentes roles de usuario, como vendedores y administradores, cada uno con permisos especÃficos segÃºn sus responsabilidades.

Alcance del Sistema

El Sistema POS abarca las siguientes funcionalidades principales:

- **GestiÃ³n de ventas**: Registro de ventas con diferentes mÃ©todos de pago (efectivo, tarjeta, transferencia).
- **GestiÃ³n de productos e inventario**: Control de stock, alertas de bajo stock, y movimientos tipo Kardex.
- **GestiÃ³n de usuarios**: AutenticaciÃ³n y control de acceso basado en roles (vendedor y administrador).
- **GestiÃ³n de gastos**: Registro y categorizaciÃ³n de gastos.
- **Reportes**: GeneraciÃ³n de informes de ventas, inventario, gastos e ingresos en formatos PDF y CSV.
- **GestiÃ³n de entidades**: CreaciÃ³n y administraciÃ³n de compaÃ±Ãas, tiendas, clientes, proveedores y empleados.

Audiencia Objetivo

Esta documentaciÃ³n tÃ©cnica estÃ¡ dirigida a:

- **Desarrolladores**: Que necesiten mantener, extender o integrar el sistema con otras aplicaciones.
- **Administradores de sistemas**: Responsables de la instalaciÃ³n, configuraciÃ³n y mantenimiento del sistema.
- **Consultores tÃ©cnicos**: Que asesoren a la empresa en la implementaciÃ³n y uso del sistema.

Estructura de la DocumentaciÃ³n

Este documento estÃ¡ organizado en secciones que cubren diferentes aspectos del sistema:

1. **Arquitectura del Sistema**: VisiÃ³n general de la arquitectura y componentes del sistema.
2. **TecnologÃas Utilizadas**: DescripciÃ³n de las tecnologÃas y frameworks empleados.
3. **Estructura del Proyecto**: OrganizaciÃ³n de directorios y archivos.
4. **Backend (NestJS)**: Detalles de la implementaciÃ³n del backend.
5. **Frontend (ReactJS + Electron)**: Detalles de la implementaciÃ³n del frontend.
6. **Base de Datos (PostgreSQL)**: Estructura de la base de datos y relaciones.
7. **Seguridad**: Mecanismos de seguridad implementados.
8. **API REST**: DocumentaciÃ³n de los endpoints disponibles.
9. **Despliegue**: Instrucciones para el despliegue del sistema.
10. **Mantenimiento**: GuÃas para el mantenimiento y actualizaciÃ³n del sistema.
11. **Referencias**: Fuentes y recursos adicionales.

IntroducciÃ³n

El Sistema POS (Point of Sale) para Tienda de Accesorios MÃ³viles es una soluciÃ³n completa de gestiÃ³n diseÃ±ada especÃficamente para tiendas que comercializan accesorios para dispositivos mÃ³viles. Este sistema integra todas las funcionalidades necesarias para administrar eficientemente el negocio, desde la gestiÃ³n de inventario hasta el procesamiento de ventas y la generaciÃ³n de reportes financieros.

PropÃ³sito del Sistema

El propÃ³sito principal del Sistema POS es proporcionar una plataforma robusta y fÃ¡cil de usar que permita a los propietarios y empleados de tiendas de accesorios mÃ³viles:

- Gestionar el inventario de productos de manera eficiente
- Procesar ventas con mÃºltiples mÃ©todos de pago
- Realizar un seguimiento detallado del movimiento de productos (Kardex)
- Administrar gastos y generar reportes financieros
- Gestionar mÃºltiples tiendas bajo una misma compaÃ±Ãa
- Controlar el acceso mediante un sistema de roles y permisos

Alcance del Sistema

El Sistema POS abarca los siguientes mÃ³dulos principales:

1. **GestiÃ³n de Usuarios y Roles**: AdministraciÃ³n de usuarios con diferentes niveles de acceso (vendedor, administrador).

2. **GestiÃ³n de Productos e Inventario**: CatÃ¡logo de productos, control de stock, alertas de bajo inventario y seguimiento de movimientos tipo Kardex.

3. **GestiÃ³n de Ventas**: Procesamiento de transacciones con soporte para mÃºltiples mÃ©todos de pago (efectivo, tarjeta, transferencia).

4. **GestiÃ³n de Gastos**: Registro y categorizaciÃ³n de gastos operativos.

5. **Reportes y AnÃ¡lisis**: GeneraciÃ³n de informes financieros, de ventas e inventario en formatos PDF y CSV.

6. **AdministraciÃ³n de CompaÃ±Ãas y Tiendas**: Soporte para mÃºltiples tiendas asociadas a una compaÃ±Ãa.

Audiencia

Esta documentaciÃ³n tÃ©cnica estÃ¡ dirigida a:

- Desarrolladores que necesiten mantener o extender el sistema
- Administradores de sistemas responsables de la instalaciÃ³n y configuraciÃ³n
- Personal tÃ©cnico encargado del soporte y resoluciÃ³n de problemas
- Integradores que necesiten conectar el sistema con otras plataformas

Convenciones del Documento

En este documento se utilizan las siguientes convenciones:

- `cÃ³digo` - Fragmentos de cÃ³digo, nombres de archivos, comandos o rutas
- **Negrita** - TÃ©rminos importantes o Ã©nfasis
- *Cursiva* - Nuevos tÃ©rminos o referencias a otros documentos
- [HipervÃnculo](#) - Enlaces a otras secciones o recursos externos

Requisitos del Sistema

Requisitos de Hardware

- **Servidor**:
  - CPU: 2 nÃºcleos o mÃ¡s
  - RAM: 4GB mÃnimo (8GB recomendado)
  - Almacenamiento: 20GB mÃnimo

- **Cliente**:
  - CPU: 2 nÃºcleos o mÃ¡s
  - RAM: 4GB mÃnimo
  - Almacenamiento: 5GB mÃnimo
  - ResoluciÃ³n de pantalla: 1280x720 o superior

Requisitos de Software

- **Servidor**:
  - Sistema Operativo: Ubuntu 20.04 LTS o superior
  - Node.js 16.x o superior
  - PostgreSQL 12 o superior

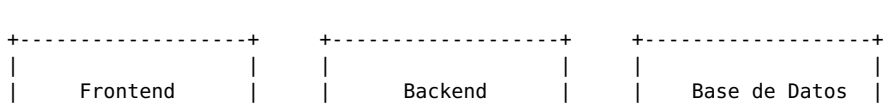
- **Cliente**:
  - Windows 10/11, macOS 10.15+, o Linux (Ubuntu 20.04+)
  - No se requiere software adicional (aplicaciÃ³n Electron autÃ³noma)

Arquitectura del Sistema

El Sistema POS para Tienda de Accesorios MÃ³viles estÃ¡ construido siguiendo una arquitectura de tres capas, con una clara separaciÃ³n entre el frontend, el backend y la base de datos. Esta arquitectura proporciona modularidad, escalabilidad y facilidad de mantenimiento.

Diagrama de Arquitectura

```
```



Componentes Principales

1. Frontend (Cliente)

El frontend estÃ¡ desarrollado con ReactJS y empaquetado como una aplicaciÃ³n de escritorio utilizando Electron. Esta capa es responsable de la interfaz de usuario y la interacciÃ³n con el usuario.

CaracterÃsticas principales:
- Interfaz de usuario moderna y responsiva con Material UI
- GestiÃ³n de estado con React Context API
- Enrutamiento con React Router
- ComunicaciÃ³n con el backend mediante Axios
- ValidaciÃ³n de formularios con React Hook Form
- ExportaciÃ³n de datos a PDF/CSV
- Tema personalizable (modo claro/oscuro)

2. Backend (Servidor)

El backend estÃ¡ desarrollado con NestJS, un framework de Node.js para construir aplicaciones del lado del servidor eficientes y escalables. Esta capa implementa la lÃ³gica de negocio y expone una API REST para la comunicaciÃ³n con el frontend.

CaracterÃsticas principales:
- Arquitectura modular basada en mÃ³dulos, controladores y servicios
- AutenticaciÃ³n y autorizaciÃ³n con JWT
- ValidaciÃ³n de datos con class-validator
- DocumentaciÃ³n de API con Swagger
- Manejo de errores centralizado
- Logging y monitoreo
- ImplementaciÃ³n de patrones de diseÃ±o (Repository, Dependency Injection, etc.)

3. Base de Datos

La base de datos utilizada es PostgreSQL, un sistema de gestiÃ³n de bases de datos relacional potente y de cÃ³digo abierto. Esta capa almacena todos los datos del sistema de manera persistente.

CaracterÃsticas principales:
- Esquema relacional normalizado
- Integridad referencial con claves forÃ¡neas
- Ãndices para optimizar consultas frecuentes
- Transacciones para garantizar la consistencia de los datos
- Triggers para automatizar ciertas operaciones (como actualizaciÃ³n de inventario)

Flujo de Datos

1. **Solicitud del Usuario**: El usuario interactÃºa con la interfaz de usuario en el frontend.
2. **Procesamiento en el Frontend**: El frontend procesa la interacciÃ³n, actualiza la UI si es necesario y envÃa una solicitud al backend a travÃ©s de la API REST.
3. **Procesamiento en el Backend**: El backend recibe la solicitud, la valida, aplica la lÃ³gica de negocio correspondiente y realiza operaciones en la base de datos si es necesario.
4. **Acceso a la Base de Datos**: El backend se comunica con la base de datos para leer o escribir datos.
5. **Respuesta al Frontend**: El backend envÃa una respuesta al frontend con los datos solicitados o la confirmaciÃ³n de la operaciÃ³n.
6. **ActualizaciÃ³n de la UI**: El frontend recibe la respuesta y actualiza la interfaz de usuario en consecuencia.

Patrones de DiseÃ±o Utilizados

1. **PatrÃ³n MVC (Model-View-Controller)**: SeparaciÃ³n clara entre los modelos de datos, la lÃ³gica de negocio y la presentaciÃ³n.
2. **PatrÃ³n Repository**: AbstracciÃ³n de la capa de acceso a datos para desacoplar la lÃ³gica de negocio de la implementaciÃ³n especÃfica de la base de datos.
3. **PatrÃ³n Dependency Injection**: InyecciÃ³n de dependencias para facilitar el testing y reducir el acoplamiento entre componentes.
4. **PatrÃ³n Observer**: Utilizado en el frontend para la gestiÃ³n de estado y la actualizaciÃ³n reactiva de la UI.
5. **PatrÃ³n Strategy**: Implementado para manejar diferentes estrategias de autenticaciÃ³n y mÃ©todos de pago.

Consideraciones de Escalabilidad

El sistema estÃ¡ diseÃ±ado para ser escalable y soportar el crecimiento del negocio:

- **Escalabilidad Horizontal**: El backend puede desplegarse en mÃºltiples instancias detrÃ¡s de un balanceador de carga.
- **Escalabilidad Vertical**: La base de datos puede configurarse para aprovechar recursos adicionales (CPU, memoria) segÃºn sea necesario.
- **Arquitectura Modular**: Facilita la adiciÃ³n de nuevas funcionalidades sin afectar las existentes.

- **CachÃ©**: ImplementaciÃ³n de estrategias de cachÃ© para mejorar el rendimiento en operaciones frecuentes.
- **OptimizaciÃ³n de Consultas**: Ãndices y consultas optimizadas para mantener un buen rendimiento incluso con grandes volÃºmenes de datos.

Arquitectura del Sistema

El Sistema POS para Tienda de Accesorios MÃ³viles estÃ¡ construido siguiendo una arquitectura moderna de aplicaciÃ³n de tres capas, con una clara separaciÃ³n entre el frontend, el backend y la base de datos. Esta arquitectura proporciona modularidad, escalabilidad y facilidad de mantenimiento.

VisiÃ³n General de la Arquitectura

La arquitectura del sistema se basa en el patrÃ³n cliente-servidor, con una aplicaciÃ³n de escritorio (Electron) que se comunica con un servidor backend a travÃ©s de una API REST. El sistema estÃ¡ diseÃ±ado para funcionar tanto en un entorno local como en un entorno distribuido.

Nota: La imagen es una representaciÃ³n conceptual de la arquitectura del sistema.

Componentes Principales

1. Frontend (Cliente)

El frontend estÃ¡ implementado como una aplicaciÃ³n de escritorio utilizando Electron, que encapsula una aplicaciÃ³n web desarrollada con ReactJS. Esta capa es responsable de:

- Presentar la interfaz de usuario
- Gestionar la interacciÃ³n del usuario
- Comunicarse con el backend a travÃ©s de la API REST
- Manejar el estado de la aplicaciÃ³n
- Proporcionar funcionalidades offline cuando sea necesario

2. Backend (Servidor)

El backend estÃ¡ implementado con NestJS, un framework para Node.js que sigue los principios de arquitectura de software modernos. Esta capa es responsable de:

- Exponer una API REST para el frontend
- Implementar la lÃ³gica de negocio
- Gestionar la autenticaciÃ³n y autorizaciÃ³n
- Comunicarse con la base de datos
- Validar y procesar los datos
- Generar reportes

3. Base de Datos

La capa de persistencia utiliza PostgreSQL, un sistema de gestiÃ³n de bases de datos relacional potente y robusto. Esta capa es responsable de:

- Almacenar todos los datos del sistema
- Garantizar la integridad de los datos
- Proporcionar mecanismos para consultas eficientes
- Mantener relaciones entre entidades

Flujo de Datos

El flujo tÃpico de datos en el sistema sigue estos pasos:

1. El usuario interactÃºa con la interfaz de usuario en la aplicaciÃ³n Electron.
2. La aplicaciÃ³n React captura la interacciÃ³n y realiza una solicitud HTTP a la API REST del backend.
3. El backend recibe la solicitud, la valida y la procesa segÃºn la lÃ³gica de negocio.
4. Si es necesario, el backend consulta o modifica datos en la base de datos PostgreSQL.
5. El backend envÃa una respuesta HTTP al frontend.
6. El frontend actualiza la interfaz de usuario segÃºn la respuesta recibida.

Patrones de DiseÃ±o

El sistema implementa varios patrones de diseÃ±o para garantizar una arquitectura limpia y mantenible:

En el Backend:

- **PatrÃ³n Repositorio**: Separa la lÃ³gica de acceso a datos de la lÃ³gica de negocio.
- **InyecciÃ³n de Dependencias**: Facilita la prueba y el mantenimiento del cÃ³digo.
- **PatrÃ³n Servicio**: Encapsula la lÃ³gica de negocio en servicios reutilizables.
- **PatrÃ³n Controlador**: Maneja las solicitudes HTTP y delega el procesamiento a los servicios.
- **PatrÃ³n Middleware**: Procesa las solicitudes antes de que lleguen a los controladores.
- **PatrÃ³n Interceptor**: Transforma las respuestas antes de enviarlas al cliente.
- **PatrÃ³n Guardia**: Controla el acceso a las rutas segÃºn los permisos del usuario.

En el Frontend:

- **PatrÃ³n Contenedor/PresentaciÃ³n**: Separa la lÃ³gica de estado de la presentaciÃ³n.
- **PatrÃ³n Context**: Gestiona el estado global de la aplicaciÃ³n.
- **PatrÃ³n Hook**: Encapsula la lÃ³gica reutilizable.
- **PatrÃ³n HOC (Higher-Order Component)**: Compone componentes con funcionalidades adicionales.
- **PatrÃ³n Render Props**: Comparte cÃ³digo entre componentes React.

Consideraciones de Escalabilidad

La arquitectura del sistema estÃ¡ diseÃ±ada para ser escalable:

- **Escalabilidad Horizontal**: El backend puede desplegarse en mÃºltiples instancias detrÃ¡s de un balanceador de carga.
- **Escalabilidad Vertical**: Tanto el backend como la base de datos pueden escalar verticalmente aumentando los recursos del servidor.
- **CachÃ©**: Se implementan estrategias de cachÃ© para mejorar el rendimiento.
- **OptimizaciÃ³n de Consultas**: Las consultas a la base de datos estÃ¡n optimizadas para manejar grandes volÃºmenes de datos.
- **Procesamiento AsÃncrono**: Las operaciones intensivas se realizan de forma asÃncrona para no bloquear el hilo principal.

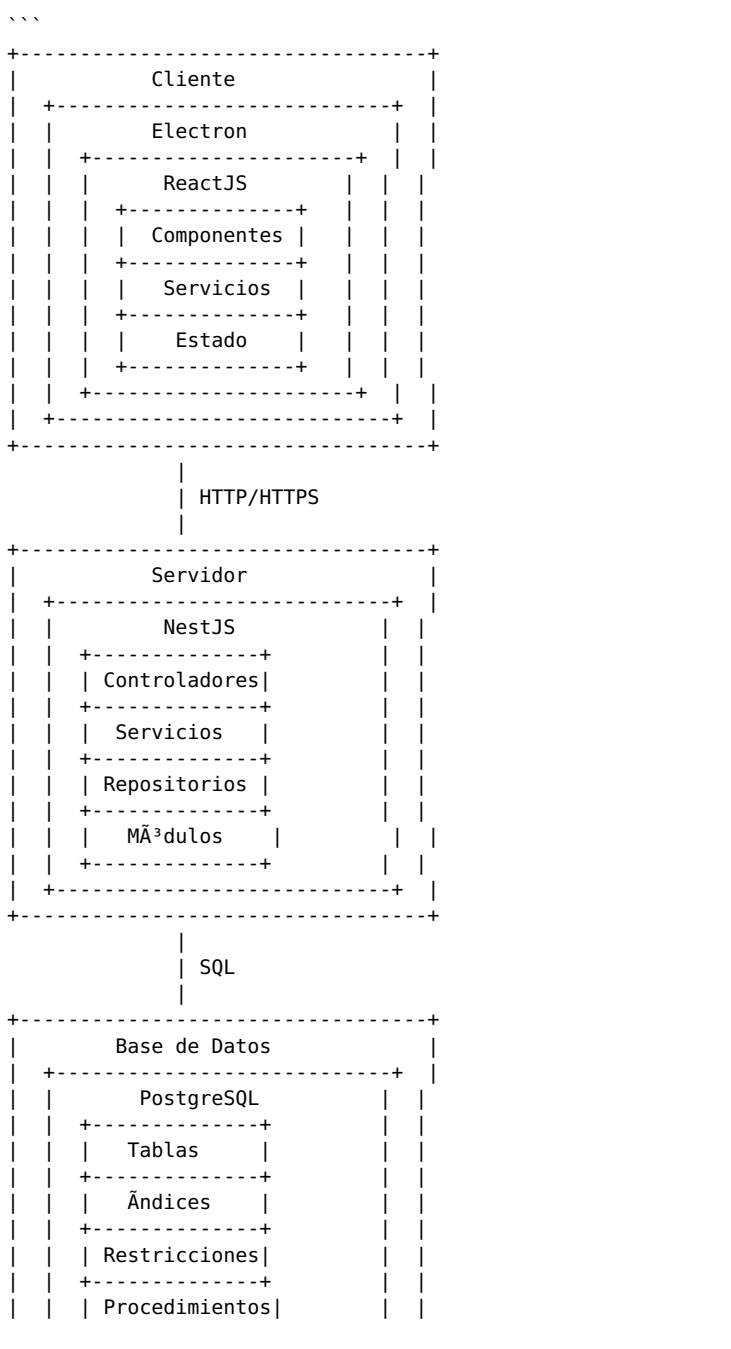
Consideraciones de Seguridad

La arquitectura incorpora varias capas de seguridad:

- **AutenticaciÃ³n**: Basada en JWT (JSON Web Tokens).
- **AutorizaciÃ³n**: Control de acceso basado en roles.
- **ValidaciÃ³n de Entrada**: Todas las entradas del usuario son validadas.
- **ProtecciÃ³n contra Ataques Comunes**: XSS, CSRF, SQL Injection, etc.
- **Cifrado**: Datos sensibles cifrados en la base de datos.
- **HTTPS**: ComunicaciÃ³n cifrada entre el cliente y el servidor.

Diagrama de Componentes

A continuaciÃ³n se muestra un diagrama detallado de los componentes del sistema:



| | +--------------+ | |
| +---------------------------+ |
+---------------------------------+
```

Este diagrama muestra la separaciÃ³n clara entre las tres capas principales del sistema y los componentes dentro de cada capa.


## Arquitectura del Sistema

El Sistema POS para Tienda de Accesorios MÃ³viles estÃ¡ construido siguiendo una arquitectura de tres capas con un enfoque modular y orientado a servicios. Esta arquitectura proporciona una clara separaciÃ³n de responsabilidades, facilita el mantenimiento y permite la escalabilidad del sistema.

### VisiÃ³n General de la Arquitectura

El sistema se compone de tres capas principales:

1. **Capa de PresentaciÃ³n (Frontend)**: Implementada con ReactJS y empaquetada como aplicaciÃ³n de escritorio mediante Electron.
2. **Capa de LÃ³gica de Negocio (Backend)**: Desarrollada con NestJS, proporciona una API RESTful.
3. **Capa de Datos**: Gestionada por PostgreSQL como sistema de gestiÃ³n de base de datos relacional.

La siguiente figura muestra la arquitectura general del sistema:

```
â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
â”‚                                 â”‚
â”‚    Frontend (ReactJS+Electron)  â”‚
â”‚                                 â”‚
â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¬â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
                â”‚
                â–¼
â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
â”‚                                 â”‚
â”‚      Backend API (NestJS)       â”‚
â”‚                                 â”‚
â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¬â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
                â”‚
                â–¼
â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
â”‚                                 â”‚
â”‚     Base de Datos (PostgreSQL)  â”‚
â”‚                                 â”‚
â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
```

Componentes Principales

Frontend (ReactJS + Electron)

El frontend estÃ¡ desarrollado como una Single Page Application (SPA) utilizando ReactJS y se empaqueta como una aplicaciÃ³n de escritorio mediante Electron. Esta capa se encarga de:

- Presentar la interfaz de usuario
- Gestionar la interacciÃ³n del usuario
- Comunicarse con el backend a travÃ©s de peticiones HTTP
- Manejar el estado de la aplicaciÃ³n mediante contextos de React
- Proporcionar funcionalidades offline bÃ¡sicas

La aplicaciÃ³n Electron permite que el sistema funcione como una aplicaciÃ³n de escritorio nativa en diferentes sistemas operativos (Windows, macOS, Linux).

Backend (NestJS)

El backend estÃ¡ implementado con NestJS, un framework para Node.js que sigue los principios de arquitectura de Angular. Esta capa proporciona:

- API RESTful para la comunicaciÃ³n con el frontend
- ImplementaciÃ³n de la lÃ³gica de negocio
- ValidaciÃ³n de datos y manejo de errores
- AutenticaciÃ³n y autorizaciÃ³n mediante JWT
- Acceso a la base de datos mediante TypeORM

El backend sigue una arquitectura modular, donde cada funcionalidad del sistema (usuarios, productos, ventas, etc.) estÃ¡ implementada como un mÃ³dulo independiente.

Base de Datos (PostgreSQL)

PostgreSQL se utiliza como sistema de gestiÃ³n de base de datos relacional. La estructura de la base de datos estÃ¡ diseÃ±ada para:
```

- Almacenar todos los datos del sistema de manera persistente
- Garantizar la integridad referencial mediante relaciones y restricciones
- Optimizar las consultas mediante Ãndices
- Proporcionar transacciones ACID para operaciones crÃticas

Patrones de DiseÃ±o Utilizados

El sistema implementa varios patrones de diseÃ±o para mejorar la calidad del cÃ³digo y facilitar el mantenimiento:

1. **PatrÃ³n Repositorio**: Utilizado en el backend para abstraer el acceso a la base de datos.
2. **PatrÃ³n Servicio**: Implementado para encapsular la lÃ³gica de negocio.
3. **PatrÃ³n Controlador**: Utilizado para manejar las peticiones HTTP y delegar el procesamiento a los servicios.
4. **PatrÃ³n InyecciÃ³n de Dependencias**: Implementado mediante el sistema de inyecciÃ³n de dependencias de NestJS.
5. **PatrÃ³n Observador**: Utilizado en el frontend para la gestiÃ³n del estado y la reactividad.
6. **PatrÃ³n Decorador**: Empleado para aÃ±adir funcionalidades como validaciÃ³n, autenticaciÃ³n y autorizaciÃ³n.

Flujo de Datos

El flujo tÃpico de datos en el sistema sigue estos pasos:

1. El usuario interactÃºa con la interfaz de usuario en el frontend.
2. El frontend envÃa una peticiÃ³n HTTP a la API del backend.
3. El backend valida la peticiÃ³n y la autenticaciÃ³n/autorizaciÃ³n del usuario.
4. El backend procesa la peticiÃ³n, aplicando la lÃ³gica de negocio necesaria.
5. El backend accede a la base de datos para leer o escribir datos.
6. El backend devuelve una respuesta al frontend.
7. El frontend actualiza la interfaz de usuario con los datos recibidos.

Escalabilidad y Extensibilidad

La arquitectura del sistema estÃ¡ diseÃ±ada para ser escalable y extensible:

- **Escalabilidad Horizontal**: El backend puede desplegarse en mÃºltiples instancias detrÃ¡s de un balanceador de carga.
- **Escalabilidad Vertical**: Tanto el backend como la base de datos pueden escalar verticalmente aumentando los recursos del servidor.
- **Extensibilidad**: La arquitectura modular permite aÃ±adir nuevas funcionalidades sin afectar a las existentes.
- **Configurabilidad**: El sistema utiliza archivos de configuraciÃ³n y variables de entorno para adaptarse a diferentes entornos.

Consideraciones de Seguridad en la Arquitectura

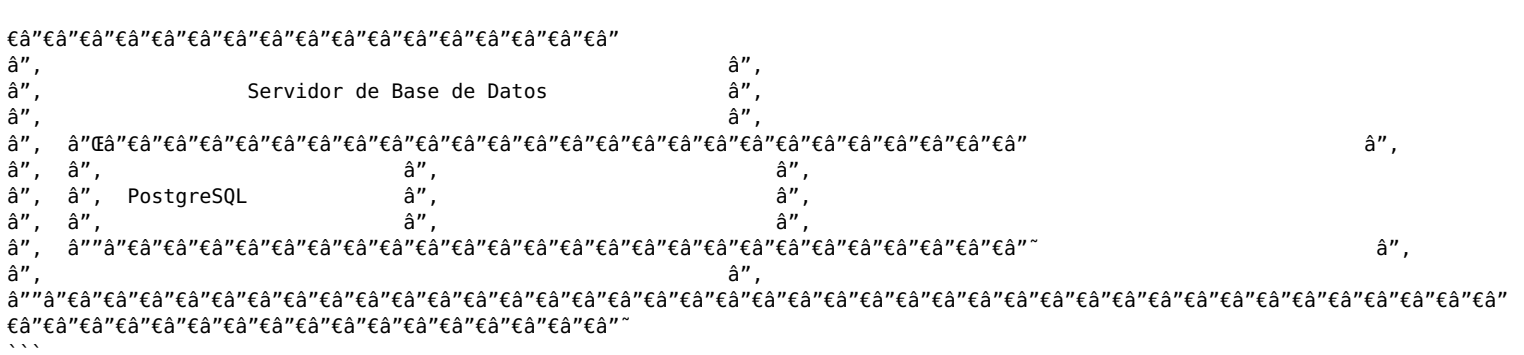
La seguridad estÃ¡ integrada en todos los niveles de la arquitectura:

- **Frontend**: ValidaciÃ³n de datos de entrada, protecciÃ³n contra XSS, rutas protegidas.
- **Backend**: AutenticaciÃ³n JWT, autorizaciÃ³n basada en roles, validaciÃ³n de datos, protecciÃ³n contra inyecciÃ³n SQL.
- **Base de Datos**: Acceso restringido, cifrado de datos sensibles, backups regulares.
- **ComunicaciÃ³n**: HTTPS para la comunicaciÃ³n entre frontend y backend.

Diagrama de Despliegue

El siguiente diagrama muestra la configuraciÃ³n de despliegue recomendada para el sistema:

```
â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
â”‚                         â”‚     â”‚                         â”‚
â”‚  Cliente (AplicaciÃ³n    â”‚     â”‚  Cliente (AplicaciÃ³n    â”‚
â”‚  Electron)              â”‚     â”‚  Electron)              â”‚
â”‚                         â”‚     â”‚                         â”‚
â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¬â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¬â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
            â”‚                                  â”‚
            â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¬â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
                        â”‚
                        â–¼
â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
â”‚                                                         â”‚
â”‚                  Servidor Backend                       â”‚
â”‚                                                         â”‚
â”‚  â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”   â”‚
â”‚  â”‚                         â”‚  â”‚                     â”‚   â”‚
â”‚  â”‚  API NestJS             â”‚  â”‚  Servidor Web       â”‚   â”‚
â”‚  â”‚                         â”‚  â”‚  (Nginx/Apache)     â”‚   â”‚
â”‚  â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¬â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜   â”‚
â”‚                 â”‚                                         â”‚
â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¼â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
                 â”‚
                 â–¼
â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
```



Este diagrama representa una configuraciÃ³n de despliegue tÃpica, donde:

- Los clientes ejecutan la aplicaciÃ³n Electron en sus equipos locales.
- El servidor backend aloja la API NestJS y puede incluir un servidor web para servir archivos estÃ¡ticos.
- El servidor de base de datos aloja PostgreSQL y puede estar en la misma mÃ¡quina que el backend o en una mÃ¡quina separada para mayor escalabilidad.

TecnologÃas Utilizadas

El Sistema POS para Tienda de Accesorios MÃ³viles utiliza un conjunto de tecnologÃas modernas y robustas para garantizar un rendimiento Ã³ptimo, seguridad y facilidad de mantenimiento. A continuaciÃ³n, se detallan las principales tecnologÃas empleadas en cada capa del sistema.

Backend

NestJS

NestJS es un framework para construir aplicaciones del lado del servidor en Node.js, inspirado en Angular. Proporciona una arquitectura que favorece la modularidad, testabilidad y mantenibilidad del cÃ³digo.

- **VersiÃ³n**: 9.x
- **CaracterÃsticas utilizadas**:
  - MÃ³dulos para organizar el cÃ³digo
  - Controladores para manejar las solicitudes HTTP
  - Servicios para implementar la lÃ³gica de negocio
  - Guards para proteger rutas
  - Interceptores para transformar respuestas
  - Pipes para validaciÃ³n de datos
  - Decoradores para metadatos y configuraciÃ³n

TypeORM

TypeORM es un ORM (Object-Relational Mapping) para TypeScript y JavaScript que facilita la interacciÃ³n con bases de datos relacionales.

- **VersiÃ³n**: 0.3.x
- **CaracterÃsticas utilizadas**:
  - Entidades para mapear tablas de la base de datos
  - Repositorios para operaciones CRUD
  - Relaciones entre entidades (one-to-one, one-to-many, many-to-many)
  - Migraciones para gestionar cambios en el esquema
  - Transacciones para operaciones atÃ³micas
  - Consultas personalizadas para operaciones complejas

Passport.js

Passport.js es un middleware de autenticaciÃ³n para Node.js que proporciona estrategias de autenticaciÃ³n flexibles.

- **VersiÃ³n**: 0.6.x
- **CaracterÃsticas utilizadas**:
  - Estrategia JWT para autenticaciÃ³n basada en tokens
  - Estrategia Local para autenticaciÃ³n con credenciales

Otras Bibliotecas Backend

- **class-validator**: Para validaciÃ³n de datos basada en decoradores
- **class-transformer**: Para transformaciÃ³n de objetos
- **bcrypt**: Para hash de contraseÃ±as
- **jsonwebtoken**: Para generaciÃ³n y verificaciÃ³n de tokens JWT
- **swagger-ui-express**: Para documentaciÃ³n de API
- **helmet**: Para seguridad HTTP
- **compression**: Para compresiÃ³n de respuestas HTTP
- **winston**: Para logging

Frontend

ReactJS

ReactJS es una biblioteca de JavaScript para construir interfaces de usuario, desarrollada por Facebook.

- **VersiÃ³n**: 18.x
- **CaracterÃsticas utilizadas**:
  - Componentes funcionales
  - Hooks (useState, useEffect, useContext, useReducer, etc.)
  - Context API para gestiÃ³n de estado global
  - React Router para enrutamiento
  - React Hook Form para manejo de formularios
  - Error Boundaries para manejo de errores

Electron

Electron es un framework para crear aplicaciones de escritorio multiplataforma utilizando tecnologÃas web.

- **VersiÃ³n**: 24.x
- **CaracterÃsticas utilizadas**:
  - IPC (Inter-Process Communication) para comunicaciÃ³n entre procesos
  - Acceso al sistema de archivos
  - IntegraciÃ³n con impresoras
  - Notificaciones nativas
  - Auto-actualizaciÃ³n

Material UI

Material UI es una biblioteca de componentes de React que implementa el diseÃ±o Material Design de Google.

- **VersiÃ³n**: 5.x
- **CaracterÃsticas utilizadas**:
  - Componentes de UI (Button, TextField, Table, etc.)
  - Sistema de Grid para layouts responsivos
  - Temas personalizables
  - Iconos
  - Componentes de navegaciÃ³n (Drawer, AppBar, etc.)

Otras Bibliotecas Frontend

- **Axios**: Para peticiones HTTP
- **date-fns**: Para manipulaciÃ³n de fechas
- **recharts**: Para visualizaciÃ³n de datos
- **jspdf**: Para generaciÃ³n de PDF
- **xlsx**: Para exportaciÃ³n a Excel
- **electron-store**: Para almacenamiento persistente
- **electron-builder**: Para empaquetado y distribuciÃ³n

Base de Datos

PostgreSQL

PostgreSQL es un sistema de gestiÃ³n de bases de datos relacional de cÃ³digo abierto, conocido por su robustez, escalabilidad y cumplimiento de estÃ¡ndares.

- **VersiÃ³n**: 14.x
- **CaracterÃsticas utilizadas**:
  - Tablas con relaciones
  - Ãndices para optimizaciÃ³n de consultas
  - Triggers para automatizaciÃ³n
  - Funciones almacenadas
  - Transacciones ACID
  - Constraints para integridad de datos

Herramientas de Desarrollo

- **TypeScript**: Superset de JavaScript que aÃ±ade tipado estÃ¡tico
- **ESLint**: Para anÃ¡lisis estÃ¡tico de cÃ³digo
- **Prettier**: Para formateo de cÃ³digo
- **Jest**: Para testing unitario
- **Supertest**: Para testing de API
- **Docker**: Para contenerizaciÃ³n (opcional)
- **Git**: Para control de versiones

Entorno de EjecuciÃ³n

- **Node.js**: Entorno de ejecuciÃ³n para JavaScript del lado del servidor
- **npm**: Gestor de paquetes para Node.js

Esta combinaciÃ³n de tecnologÃas proporciona un equilibrio entre rendimiento, facilidad de desarrollo, mantenibilidad y experiencia de usuario, permitiendo crear un sistema POS robusto y escalable.

TecnologÃas Utilizadas

El Sistema POS para Tienda de Accesorios MÃ³viles ha sido desarrollado utilizando un stack tecnolÃ³gico moderno y robusto,

seleccionado para garantizar rendimiento, escalabilidad, mantenibilidad y una experiencia de usuario Ã³ptima. A continuaciÃ³n, se detallan las principales tecnologÃas empleadas en cada capa del sistema.

Backend

NestJS

NestJS es un framework para construir aplicaciones del lado del servidor en Node.js, inspirado en Angular. Proporciona una arquitectura que favorece la modularidad, la testabilidad y la escalabilidad.

- **VersiÃ³n**: 9.x
- **CaracterÃsticas utilizadas**:
  - MÃ³dulos para organizar el cÃ³digo
  - Controladores para manejar las solicitudes HTTP
  - Servicios para encapsular la lÃ³gica de negocio
  - Pipes para validaciÃ³n de datos
  - Guards para protecciÃ³n de rutas
  - Interceptors para transformaciÃ³n de respuestas
  - Decoradores para metadatos y configuraciÃ³n

TypeORM

TypeORM es un ORM (Object-Relational Mapping) para TypeScript y JavaScript que facilita la interacciÃ³n con bases de datos relacionales.

- **VersiÃ³n**: 0.3.x
- **CaracterÃsticas utilizadas**:
  - Entidades para mapear tablas de la base de datos
  - Repositorios para operaciones CRUD
  - Relaciones entre entidades (one-to-one, one-to-many, many-to-many)
  - Migraciones para gestionar cambios en el esquema
  - Transacciones para operaciones atÃ³micas
  - CachÃ© para mejorar el rendimiento

Passport.js

Passport.js es un middleware de autenticaciÃ³n para Node.js que proporciona estrategias de autenticaciÃ³n flexibles.

- **VersiÃ³n**: 0.6.x
- **CaracterÃsticas utilizadas**:
  - Estrategia JWT para autenticaciÃ³n basada en tokens
  - Estrategia Local para autenticaciÃ³n con credenciales

Otras Bibliotecas Backend

- **class-validator**: Para validaciÃ³n de datos basada en decoradores
- **class-transformer**: Para transformaciÃ³n de objetos
- **bcrypt**: Para hash de contraseÃ±as
- **jsonwebtoken**: Para generaciÃ³n y verificaciÃ³n de JWT
- **swagger-ui-express**: Para documentaciÃ³n de API
- **helmet**: Para seguridad HTTP
- **compression**: Para compresiÃ³n de respuestas HTTP
- **winston**: Para logging

Frontend

ReactJS

ReactJS es una biblioteca JavaScript para construir interfaces de usuario, desarrollada por Facebook.

- **VersiÃ³n**: 18.x
- **CaracterÃsticas utilizadas**:
  - Componentes funcionales
  - Hooks (useState, useEffect, useContext, useReducer, etc.)
  - Context API para gestiÃ³n de estado global
  - React Router para enrutamiento
  - Lazy loading para carga diferida de componentes
  - Error boundaries para manejo de errores

Electron

Electron es un framework para crear aplicaciones de escritorio multiplataforma utilizando tecnologÃas web.

- **VersiÃ³n**: 24.x
- **CaracterÃsticas utilizadas**:
  - IPC (Inter-Process Communication) para comunicaciÃ³n entre procesos
  - APIs nativas para acceso al sistema de archivos
  - Empaquetado y distribuciÃ³n de aplicaciones
  - Auto-actualizaciÃ³n

Material-UI

Material-UI es una biblioteca de componentes React que implementa el diseÃ±o Material Design de Google.

- **VersiÃ³n**: 5.x
- **CaracterÃsticas utilizadas**:
  - Componentes de UI (Button, TextField, Table, etc.)
  - Sistema de temas para personalizaciÃ³n
  - Grid system para layouts responsivos
  - Iconos
  - Componentes de navegaciÃ³n (Drawer, AppBar, etc.)

Otras Bibliotecas Frontend

- **axios**: Para peticiones HTTP
- **react-hook-form**: Para manejo de formularios
- **jspdf**: Para generaciÃ³n de PDFs
- **xlsx**: Para exportaciÃ³n a Excel
- **recharts**: Para visualizaciÃ³n de datos
- **date-fns**: Para manipulaciÃ³n de fechas
- **i18next**: Para internacionalizaciÃ³n

Base de Datos

PostgreSQL

PostgreSQL es un sistema de gestiÃ³n de bases de datos relacional de cÃ³digo abierto, conocido por su robustez, escalabilidad y cumplimiento de estÃ¡ndares.

- **VersiÃ³n**: 14.x
- **CaracterÃsticas utilizadas**:
  - Tipos de datos avanzados
  - Ãndices para optimizaciÃ³n de consultas
  - Restricciones para integridad de datos
  - Triggers para automatizaciÃ³n
  - Funciones y procedimientos almacenados
  - Vistas para consultas complejas
  - Particionamiento para tablas grandes

Herramientas de Desarrollo

- **TypeScript**: Superset de JavaScript que aÃ±ade tipado estÃ¡tico
- **ESLint**: Para anÃ¡lisis estÃ¡tico de cÃ³digo
- **Prettier**: Para formateo de cÃ³digo
- **Jest**: Para testing unitario
- **Supertest**: Para testing de API
- **Webpack**: Para bundling de cÃ³digo frontend
- **npm**: Para gestiÃ³n de dependencias
- **Git**: Para control de versiones

Herramientas de Despliegue

- **Docker**: Para contenerizaciÃ³n
- **Docker Compose**: Para orquestaciÃ³n de contenedores
- **Electron Builder**: Para empaquetado de aplicaciones Electron
- **PM2**: Para gestiÃ³n de procesos Node.js en producciÃ³n

Tabla Comparativa de TecnologÃas

CategorÃa	TecnologÃa	VersiÃ³n	Alternativas Consideradas	RazÃ³n de ElecciÃ³n
Backend Framework	NestJS	9.x	Express, Koa, Fastify	Arquitectura modular, soporte para TypeScript, documentaciÃ³n robusta
ORM	TypeORM	0.3.x	Sequelize, Prisma, Knex	IntegraciÃ³n con TypeScript, soporte para mÃºltiples bases de datos
AutenticaciÃ³n	Passport.js	0.6.x	Auth0, Firebase Auth	Flexibilidad, integraciÃ³n con NestJS
Frontend Library	ReactJS	18.x	Angular, Vue.js	Flexibilidad, rendimiento, ecosistema
Desktop Framework	Electron	24.x	NW.js, Tauri	Madurez, documentaciÃ³n, comunidad
UI Library	Material-UI	5.x	Ant Design, Chakra UI	Componentes completos, personalizaciÃ³n, accesibilidad
Base de Datos	PostgreSQL	14.x	MySQL, SQLite, MongoDB	Robustez, caracterÃsticas avanzadas, escalabilidad
Lenguaje	TypeScript	4.9.x	JavaScript	Tipado estÃ¡tico, mejor tooling, menos errores

Esta combinaciÃ³n de tecnologÃas proporciona un equilibrio entre rendimiento, facilidad de desarrollo, mantenibilidad y experiencia de usuario, permitiendo crear un sistema POS robusto y escalable.

TecnologÃas Utilizadas

El Sistema POS para Tienda de Accesorios MÃ³viles ha sido desarrollado utilizando un conjunto de tecnologÃas modernas y robustas que garantizan un alto rendimiento, seguridad y escalabilidad. A continuaciÃ³n, se detallan las principales tecnologÃas empleadas en cada capa del sistema.

Backend

NestJS

NestJS es un framework para construir aplicaciones del lado del servidor en Node.js de manera eficiente y escalable. EstÃ¡

inspirado en Angular y proporciona una arquitectura que favorece la modularidad, la inyecciÃ³n de dependencias y la separaciÃ³n de responsabilidades.

VersiÃ³n utilizada: 9.0.0

CaracterÃsticas principales:
- Arquitectura modular
- Soporte para TypeScript
- InyecciÃ³n de dependencias
- Decoradores para metadatos
- Middleware, interceptores, filtros y pipes
- Soporte para WebSockets y microservicios
- IntegraciÃ³n con mÃºltiples bases de datos

TypeORM

TypeORM es un ORM (Object-Relational Mapping) que facilita la interacciÃ³n con bases de datos relacionales desde TypeScript y JavaScript.

VersiÃ³n utilizada: 0.3.10

CaracterÃsticas principales:
- Soporte para mÃºltiples bases de datos
- Modelado de entidades mediante decoradores
- Relaciones entre entidades
- Migraciones de base de datos
- Transacciones
- CachÃ© de consultas
- Eventos de entidades

Passport.js

Passport.js es un middleware de autenticaciÃ³n para Node.js que proporciona estrategias de autenticaciÃ³n flexibles y modulares.

VersiÃ³n utilizada: 0.6.0

CaracterÃsticas principales:
- MÃºltiples estrategias de autenticaciÃ³n
- IntegraciÃ³n con JWT
- Sesiones
- AutenticaciÃ³n social (OAuth)

JWT (JSON Web Tokens)

JWT es un estÃ¡ndar abierto (RFC 7519) que define una forma compacta y autÃ³noma de transmitir informaciÃ³n de forma segura entre partes como un objeto JSON.

VersiÃ³n utilizada: 9.0.0 (jsonwebtoken)

CaracterÃsticas principales:
- Tokens firmados digitalmente
- InformaciÃ³n verificable
- ExpiraciÃ³n configurable
- Payload personalizable

Class Validator

Class Validator es una biblioteca que permite validar objetos JavaScript/TypeScript utilizando decoradores.

VersiÃ³n utilizada: 0.13.2

CaracterÃsticas principales:
- ValidaciÃ³n basada en decoradores
- Mensajes de error personalizables
- ValidaciÃ³n condicional
- ValidaciÃ³n personalizada

Swagger

Swagger es un conjunto de herramientas para diseÃ±ar, construir, documentar y consumir APIs RESTful.

VersiÃ³n utilizada: 6.1.0 (@nestjs/swagger)

CaracterÃsticas principales:
- DocumentaciÃ³n automÃ¡tica de API
- Interfaz interactiva para probar endpoints
- GeneraciÃ³n de clientes API
- EspecificaciÃ³n OpenAPI

Frontend

ReactJS

ReactJS es una biblioteca JavaScript para construir interfaces de usuario, especialmente para aplicaciones de una sola pÃ¡gina (SPA).

VersiÃ³n utilizada: 18.2.0

CaracterÃsticas principales:
- Componentes reutilizables
- Virtual DOM
- Renderizado declarativo
- Flujo de datos unidireccional
- Hooks para gestiÃ³n de estado y efectos
- Context API para estado global

Electron

Electron es un framework que permite desarrollar aplicaciones de escritorio multiplataforma utilizando tecnologÃas web como JavaScript, HTML y CSS.

VersiÃ³n utilizada: 22.0.0

CaracterÃsticas principales:
- Aplicaciones multiplataforma (Windows, macOS, Linux)
- Acceso a APIs nativas del sistema
- Actualizaciones automÃ¡ticas
- Empaquetado y distribuciÃ³n simplificados
- IntegraciÃ³n con Node.js

Material-UI

Material-UI es una biblioteca de componentes React que implementa el diseÃ±o Material Design de Google.

VersiÃ³n utilizada: 5.11.0

CaracterÃsticas principales:
- Componentes prediseÃ±ados
- PersonalizaciÃ³n de temas
- Responsive design
- Sistema de grid
- Iconos y tipografÃa
- Animaciones y transiciones

React Router

React Router es una biblioteca de enrutamiento para React que permite la navegaciÃ³n entre diferentes componentes.

VersiÃ³n utilizada: 6.6.1

CaracterÃsticas principales:
- Enrutamiento declarativo
- NavegaciÃ³n programÃ¡tica
- Rutas anidadas
- ParÃ¡metros de ruta
- Redirecciones
- ProtecciÃ³n de rutas

Axios

Axios es un cliente HTTP basado en promesas para el navegador y Node.js.

VersiÃ³n utilizada: 1.2.1

CaracterÃsticas principales:
- Peticiones HTTP basadas en promesas
- Interceptores de peticiones y respuestas
- TransformaciÃ³n automÃ¡tica de datos JSON
- CancelaciÃ³n de peticiones
- ProtecciÃ³n contra XSRF

React Hook Form

React Hook Form es una biblioteca para gestionar formularios en React con un enfoque en el rendimiento y la experiencia del desarrollador.

VersiÃ³n utilizada: 7.41.0

CaracterÃsticas principales:
- ValidaciÃ³n de formularios
- Manejo de errores
- Rendimiento optimizado
- IntegraciÃ³n con bibliotecas UI
- Campos controlados y no controlados

Base de Datos

PostgreSQL

PostgreSQL es un sistema de gestiÃ³n de bases de datos relacional orientado a objetos y de cÃ³digo abierto.

VersiÃ³n utilizada: 14.0

CaracterÃsticas principales:
- ACID compliant
- Soporte para JSON y JSONB
- Ãndices avanzados (B-tree, Hash, GiST, SP-GiST, GIN, BRIN)
- Triggers y procedimientos almacenados
- Extensibilidad
- ReplicaciÃ³n
- Particionamiento de tablas

Herramientas de Desarrollo

TypeScript

TypeScript es un superconjunto tipado de JavaScript que compila a JavaScript plano.

VersiÃ³n utilizada: 4.9.4

CaracterÃsticas principales:
- Tipado estÃ¡tico
- Interfaces y tipos
- Decoradores
- GenÃ©ricos
- MÃ³dulos
- Namespaces

ESLint

ESLint es una herramienta de anÃ¡lisis de cÃ³digo estÃ¡tico para identificar patrones problemÃ¡ticos en el cÃ³digo JavaScript.

VersiÃ³n utilizada: 8.30.0

CaracterÃsticas principales:
- Reglas configurables
- IntegraciÃ³n con editores
- Autofix
- Plugins personalizados
- Soporte para TypeScript

Jest

Jest es un framework de testing para JavaScript con un enfoque en la simplicidad.

VersiÃ³n utilizada: 29.3.1

CaracterÃsticas principales:
- Zero config
- Snapshots
- Mocks y spies
- Cobertura de cÃ³digo
- ParalelizaciÃ³n de tests

Prettier

Prettier es un formateador de cÃ³digo opinado que soporta mÃºltiples lenguajes.

VersiÃ³n utilizada: 2.8.1

CaracterÃsticas principales:
- Formateo consistente
- IntegraciÃ³n con editores
- ConfiguraciÃ³n mÃnima
- Soporte para mÃºltiples lenguajes

Herramientas de ConstrucciÃ³n y Despliegue

npm

npm es el gestor de paquetes por defecto para Node.js.

VersiÃ³n utilizada: 8.19.2

CaracterÃsticas principales:
- GestiÃ³n de dependencias
- Scripts

- Versionado semÃ¡ntico
- PublicaciÃ³n de paquetes
- Workspaces

Webpack

Webpack es un empaquetador de mÃ³dulos para aplicaciones JavaScript modernas.

VersiÃ³n utilizada: 5.75.0

CaracterÃsticas principales:
- Empaquetado de mÃ³dulos
- Code splitting
- Loaders para diferentes tipos de archivos
- Plugins para optimizaciÃ³n
- Hot Module Replacement

electron-builder

electron-builder es una soluciÃ³n completa para empaquetar y construir aplicaciones Electron listas para distribuciÃ³n.

VersiÃ³n utilizada: 23.6.0

CaracterÃsticas principales:
- Empaquetado para mÃºltiples plataformas
- Auto-actualizaciÃ³n
- Code signing
- GeneraciÃ³n de instaladores
- PublicaciÃ³n en tiendas de aplicaciones

Tabla Resumen de TecnologÃas

CategorÃa	TecnologÃa	VersiÃ³n	PropÃ³sito
Backend	NestJS	9.0.0	Framework de servidor
	TypeORM	0.3.10	ORM para base de datos
	Passport.js	0.6.0	AutenticaciÃ³n
	JWT	9.0.0	Tokens de autenticaciÃ³n
	Class Validator	0.13.2	ValidaciÃ³n de datos
	Swagger	6.1.0	DocumentaciÃ³n de API
Frontend	ReactJS	18.2.0	Biblioteca UI
	Electron	22.0.0	Framework de escritorio
	Material-UI	5.11.0	Componentes UI
	React Router	6.6.1	Enrutamiento
	Axios	1.2.1	Cliente HTTP
	React Hook Form	7.41.0	GestiÃ³n de formularios
Base de Datos	PostgreSQL	14.0	Sistema de gestiÃ³n de BD
Desarrollo	TypeScript	4.9.4	Lenguaje de programaciÃ³n
	ESLint	8.30.0	Linting de cÃ³digo
	Jest	29.3.1	Testing
	Prettier	2.8.1	Formateo de cÃ³digo
ConstrucciÃ³n	npm	8.19.2	Gestor de paquetes
	Webpack	5.75.0	Empaquetador
	electron-builder	23.6.0	ConstrucciÃ³n de aplicaciÃ³n

Esta combinaciÃ³n de tecnologÃas proporciona una base sÃ³lida para el desarrollo, mantenimiento y escalabilidad del Sistema POS para Tienda de Accesorios MÃ³viles.

Estructura del Proyecto

La estructura del proyecto estÃ¡ organizada de manera modular para facilitar el desarrollo, mantenimiento y escalabilidad del sistema. A continuaciÃ³n, se detalla la estructura de directorios tanto para el backend como para el frontend.

Estructura General

```
pos-system/
â”œâ”€â”€ backend/
â”‚   â””â”€â”€ pos-backend/       # Proyecto NestJS
â”œâ”€â”€ frontend/
â”‚   â””â”€â”€ pos-frontend/      # Proyecto React + Electron
â”œâ”€â”€ docs/                  # DocumentaciÃ³n
â”‚   â”œâ”€â”€ tecnica/           # DocumentaciÃ³n tÃ©cnica
â”‚   â””â”€â”€ usuario/           # Manual de usuario
â””â”€â”€ scripts/               # Scripts de utilidad
```

Estructura del Backend (NestJS)

```
pos-backend/
â”œâ”€â”€ src/
```

```
â”‚   â”œâ”€â”€ app.module.ts      # MÃ³dulo principal de la aplicaciÃ³n
â”‚   â”œâ”€â”€ app.controller.ts  # Controlador principal
â”‚   â”œâ”€â”€ app.service.ts     # Servicio principal
â”‚   â”œâ”€â”€ main.ts            # Punto de entrada de la aplicaciÃ³n
â”‚   â”œâ”€â”€ common/            # CÃ³digo compartido entre mÃ³dulos
â”‚   â”‚   â”œâ”€â”€ decorators/    # Decoradores personalizados
â”‚   â”‚   â”œâ”€â”€ dto/           # DTOs compartidos
â”‚   â”‚   â”œâ”€â”€ enums/         # Enumeraciones
â”‚   â”‚   â”œâ”€â”€ filters/       # Filtros de excepciÃ³n
â”‚   â”‚   â”œâ”€â”€ guards/        # Guards compartidos
â”‚   â”‚   â”œâ”€â”€ interceptors/  # Interceptores
â”‚   â”‚   â”œâ”€â”€ interfaces/    # Interfaces compartidas
â”‚   â”‚   â”œâ”€â”€ middleware/    # Middleware
â”‚   â”‚   â””â”€â”€ pipes/         # Pipes de validaciÃ³n
â”‚   â”œâ”€â”€ config/            # ConfiguraciÃ³n de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ database.config.ts  # ConfiguraciÃ³n de la base de datos
â”‚   â”‚   â”œâ”€â”€ jwt.config.ts       # ConfiguraciÃ³n de JWT
â”‚   â”‚   â””â”€â”€ cors.config.ts      # ConfiguraciÃ³n de CORS
â”‚   â”œâ”€â”€ modules/           # MÃ³dulos funcionales
â”‚   â”‚   â”œâ”€â”€ auth/          # MÃ³dulo de autenticaciÃ³n
â”‚   â”‚   â”œâ”€â”€ usuarios/      # MÃ³dulo de usuarios
â”‚   â”‚   â”œâ”€â”€ roles/         # MÃ³dulo de roles
â”‚   â”‚   â”œâ”€â”€ personas/      # MÃ³dulo de personas
â”‚   â”‚   â”œâ”€â”€ companias/     # MÃ³dulo de compaÃ±Ãas
â”‚   â”‚   â”œâ”€â”€ tiendas/       # MÃ³dulo de tiendas
â”‚   â”‚   â”œâ”€â”€ productos/     # MÃ³dulo de productos
â”‚   â”‚   â”œâ”€â”€ inventario/    # MÃ³dulo de inventario
â”‚   â”‚   â”œâ”€â”€ kardex/        # MÃ³dulo de kardex
â”‚   â”‚   â”œâ”€â”€ ventas/        # MÃ³dulo de ventas
â”‚   â”‚   â”œâ”€â”€ gastos/        # MÃ³dulo de gastos
â”‚   â”‚   â”œâ”€â”€ metodos-pago/  # MÃ³dulo de mÃ©todos de pago
â”‚   â”‚   â”œâ”€â”€ proveedores/   # MÃ³dulo de proveedores
â”‚   â”‚   â”œâ”€â”€ reportes/      # MÃ³dulo de reportes
â”‚   â”‚   â””â”€â”€ alertas/       # MÃ³dulo de alertas
â”œâ”€â”€ test/                  # Pruebas
â”‚   â”œâ”€â”€ app.e2e-spec.ts    # Pruebas end-to-end
â”‚   â””â”€â”€ jest-e2e.json      # ConfiguraciÃ³n de Jest para pruebas e2e
â”œâ”€â”€ dist/                  # CÃ³digo compilado
â”œâ”€â”€ node_modules/          # Dependencias
â”œâ”€â”€ .env                   # Variables de entorno
â”œâ”€â”€ .env.development       # Variables de entorno para desarrollo
â”œâ”€â”€ .env.production        # Variables de entorno para producciÃ³n
â”œâ”€â”€ .eslintrc.js           # ConfiguraciÃ³n de ESLint
â”œâ”€â”€ .prettierrc            # ConfiguraciÃ³n de Prettier
â”œâ”€â”€ nest-cli.json          # ConfiguraciÃ³n de NestJS CLI
â”œâ”€â”€ package.json           # Dependencias y scripts
â”œâ”€â”€ tsconfig.json          # ConfiguraciÃ³n de TypeScript
â””â”€â”€ README.md              # DocumentaciÃ³n bÃ¡sica
```

Estructura de un MÃ³dulo TÃpico

Cada mÃ³dulo funcional sigue una estructura similar:

```
modulo/
â”œâ”€â”€ controllers/           # Controladores del mÃ³dulo
â”‚   â””â”€â”€ modulo.controller.ts
â”œâ”€â”€ dto/                   # Data Transfer Objects
â”‚   â”œâ”€â”€ create-modulo.dto.ts
â”‚   â””â”€â”€ update-modulo.dto.ts
â”œâ”€â”€ entities/              # Entidades de base de datos
â”‚   â””â”€â”€ modulo.entity.ts
â”œâ”€â”€ interfaces/            # Interfaces especÃficas del mÃ³dulo
â”‚   â””â”€â”€ modulo.interface.ts
â”œâ”€â”€ services/              # Servicios del mÃ³dulo
â”‚   â””â”€â”€ modulo.service.ts
â””â”€â”€ modulo.module.ts       # DefiniciÃ³n del mÃ³dulo
```

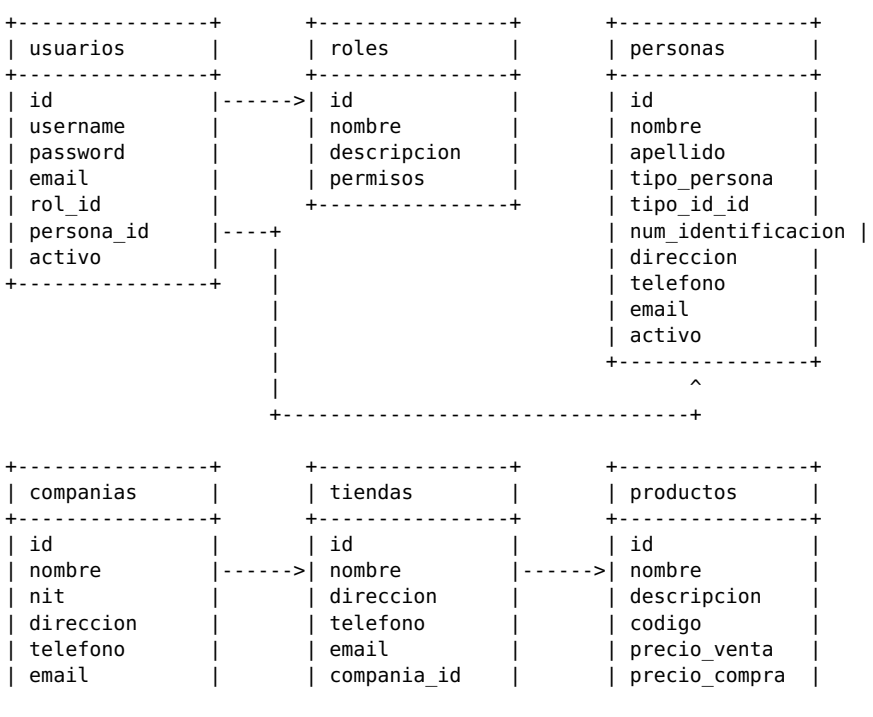
Estructura del Frontend (React + Electron)

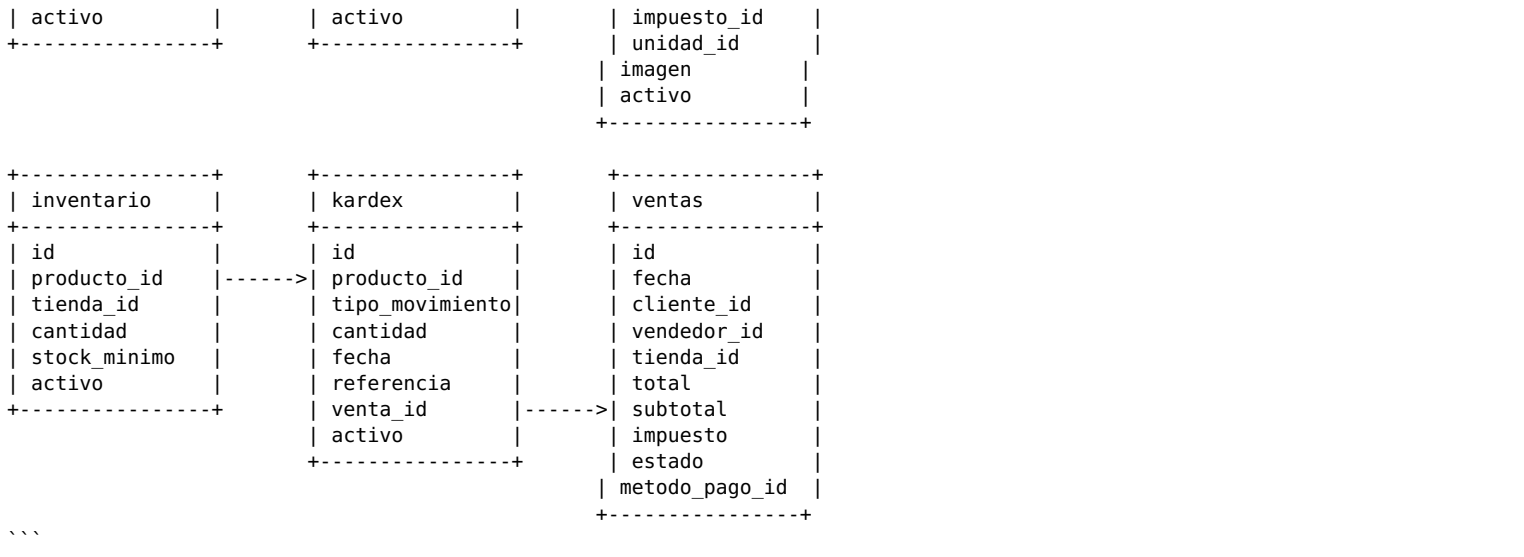
```
pos-frontend/
â”œâ”€â”€ public/                # Archivos pÃºblicos
â”‚   â”œâ”€â”€ index.html         # Plantilla HTML
â”‚   â”œâ”€â”€ favicon.ico        # Favicon
â”‚   â””â”€â”€ manifest.json      # Manifest para PWA
â”œâ”€â”€ src/                   # CÃ³digo fuente
â”‚   â”œâ”€â”€ assets/            # Recursos estÃ¡ticos
â”‚   â”‚   â”œâ”€â”€ images/        # ImÃ¡genes
â”‚   â”‚   â”œâ”€â”€ styles/        # Estilos globales
â”‚   â”‚   â””â”€â”€ fonts/         # Fuentes
```

```
â”‚   â”œâ”€â”€ components/        # Componentes reutilizables
â”‚   â”‚   â”œâ”€â”€ common/        # Componentes comunes
â”‚   â”‚   â”œâ”€â”€ forms/         # Componentes de formularios
â”‚   â”‚   â”œâ”€â”€ layout/        # Componentes de layout
â”‚   â”‚   â””â”€â”€ ui/            # Componentes de UI
â”‚   â”œâ”€â”€ contexts/          # Contextos de React
â”‚   â”‚   â”œâ”€â”€ AuthContext.tsx  # Contexto de autenticaciÃ³n
â”‚   â”‚   â””â”€â”€ ThemeContext.tsx # Contexto de tema
â”‚   â”œâ”€â”€ hooks/             # Hooks personalizados
â”‚   â”‚   â”œâ”€â”€ useForm.ts     # Hook para formularios
â”‚   â”‚   â””â”€â”€ useApi.ts      # Hook para llamadas a API
â”‚   â”œâ”€â”€ layouts/           # Layouts de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ MainLayout.tsx # Layout principal
â”‚   â”‚   â””â”€â”€ AuthLayout.tsx # Layout de autenticaciÃ³n
â”‚   â”œâ”€â”€ pages/             # PÃ¡ginas de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ Login.tsx      # PÃ¡gina de login
â”‚   â”‚   â”œâ”€â”€ Dashboard.tsx  # Dashboard principal
â”‚   â”‚   â”œâ”€â”€ Productos.tsx  # GestiÃ³n de productos
â”‚   â”‚   â”œâ”€â”€ Ventas.tsx     # GestiÃ³n de ventas
â”‚   â”‚   â”œâ”€â”€ Gastos.tsx     # GestiÃ³n de gastos
â”‚   â”‚   â”œâ”€â”€ Reportes.tsx   # Reportes
â”‚   â”‚   â””â”€â”€ Administracion.tsx # AdministraciÃ³n
â”‚   â”œâ”€â”€ services/          # Servicios
â”‚   â”‚   â”œâ”€â”€ api.ts         # Cliente API
â”‚   â”‚   â””â”€â”€ auth.ts        # Servicio de autenticaciÃ³n
â”‚   â”œâ”€â”€ utils/             # Utilidades
â”‚   â”‚   â”œâ”€â”€ formatters.ts  # Formateadores
â”‚   â”‚   â”œâ”€â”€ validators.ts  # Validadores
â”‚   â”‚   â””â”€â”€ helpers.ts     # Funciones auxiliares
â”‚   â”œâ”€â”€ theme/             # ConfiguraciÃ³n de tema
â”‚   â”‚   â””â”€â”€ index.ts       # Tema principal
â”‚   â”œâ”€â”€ App.tsx            # Componente principal
â”‚   â”œâ”€â”€ index.tsx          # Punto de entrada
â”‚   â””â”€â”€ react-app-env.d.ts # Tipos para CRA
â”œâ”€â”€ electron/              # CÃ³digo de Electron
â”‚   â”œâ”€â”€ main.js            # Proceso principal
â”‚   â””â”€â”€ preload.js         # Script de precarga
â”œâ”€â”€ build/                 # CÃ³digo compilado para web
â”œâ”€â”€ dist/                  # DistribuciÃ³n de Electron
â”œâ”€â”€ node_modules/          # Dependencias
â”œâ”€â”€ .env                   # Variables de entorno
â”œâ”€â”€ .env.development       # Variables para desarrollo
â”œâ”€â”€ .env.production        # Variables para producciÃ³n
â”œâ”€â”€ .eslintrc.js           # ConfiguraciÃ³n de ESLint
â”œâ”€â”€ .prettierrc            # ConfiguraciÃ³n de Prettier
â”œâ”€â”€ electron-builder.json  # ConfiguraciÃ³n de electron-builder
â”œâ”€â”€ package.json           # Dependencias y scripts
â”œâ”€â”€ tsconfig.json          # ConfiguraciÃ³n de TypeScript
â””â”€â”€ README.md              # DocumentaciÃ³n bÃ¡sica
```

Estructura de la Base de Datos

La estructura de la base de datos estÃ¡ diseÃ±ada siguiendo principios de normalizaciÃ³n y optimizaciÃ³n para el rendimiento. A continuaciÃ³n, se presenta un diagrama simplificado de las principales tablas y sus relaciones:






```

Esta estructura de proyecto proporciona una organizaciÃ³n clara y modular, facilitando el desarrollo colaborativo y el mantenimiento a largo plazo del sistema.


## Estructura del Proyecto

El Sistema POS para Tienda de Accesorios MÃ³viles estÃ¡ organizado en una estructura de directorios clara y modular que facilita el desarrollo, mantenimiento y escalabilidad del cÃ³digo. La estructura sigue las mejores prÃ¡cticas para proyectos NestJS (backend) y ReactJS con Electron (frontend).

### Estructura General

El proyecto estÃ¡ dividido en dos directorios principales:

```
/pos-system/
â”œâ”€â”€ backend/   # CÃ³digo del backend (NestJS)
â”‚   â””â”€â”€ pos-backend/   # Proyecto NestJS
â”œâ”€â”€ frontend/   # CÃ³digo del frontend (ReactJS + Electron)
â”‚   â””â”€â”€ pos-frontend/   # Proyecto React con Electron
â”œâ”€â”€ docs/   # DocumentaciÃ³n del proyecto
â”‚   â”œâ”€â”€ tecnica/   # DocumentaciÃ³n tÃ©cnica
â”‚   â””â”€â”€ usuario/   # Manual de usuario
â”œâ”€â”€ dev-start.sh   # Script para iniciar el sistema en modo desarrollo
â”œâ”€â”€ run-integration-tests.sh # Script para ejecutar pruebas de integraciÃ³n
â””â”€â”€ optimize.sh   # Script para optimizaciÃ³n y correcciÃ³n de errores
```

### Estructura del Backend

El backend sigue la estructura de directorios recomendada por NestJS, con algunas personalizaciones para adaptarse a las necesidades especÃficas del proyecto:

```
/backend/pos-backend/
â”œâ”€â”€ src/   # CÃ³digo fuente
â”‚   â”œâ”€â”€ app.module.ts   # MÃ³dulo principal de la aplicaciÃ³n
â”‚   â”œâ”€â”€ app.controller.ts # Controlador principal
â”‚   â”œâ”€â”€ app.service.ts   # Servicio principal
â”‚   â”œâ”€â”€ main.ts   # Punto de entrada de la aplicaciÃ³n
â”‚   â”œâ”€â”€ common/   # CÃ³digo compartido entre mÃ³dulos
â”‚   â”‚   â”œâ”€â”€ decorators/ # Decoradores personalizados
â”‚   â”‚   â”œâ”€â”€ dto/   # DTOs compartidos
â”‚   â”‚   â”œâ”€â”€ enums/   # Enumeraciones
â”‚   â”‚   â”œâ”€â”€ filters/   # Filtros de excepciÃ³n
â”‚   â”‚   â”œâ”€â”€ guards/   # Guards compartidos
â”‚   â”‚   â”œâ”€â”€ interceptors/ # Interceptores
â”‚   â”‚   â”œâ”€â”€ interfaces/ # Interfaces compartidas
â”‚   â”‚   â”œâ”€â”€ middleware/ # Middleware
â”‚   â”‚   â””â”€â”€ utils/   # Utilidades
â”‚   â”œâ”€â”€ config/   # ConfiguraciÃ³n de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ database.config.ts # ConfiguraciÃ³n de la base de datos
â”‚   â”‚   â”œâ”€â”€ jwt.config.ts # ConfiguraciÃ³n de JWT
â”‚   â”‚   â””â”€â”€ cors.config.ts # ConfiguraciÃ³n de CORS
â”‚   â””â”€â”€ modules/   # MÃ³dulos de la aplicaciÃ³n
â”‚   â”œâ”€â”€ auth/   # MÃ³dulo de autenticaciÃ³n
â”‚   â”œâ”€â”€ usuarios/   # MÃ³dulo de usuarios
â”‚   â”œâ”€â”€ roles/   # MÃ³dulo de roles
â”‚   â”œâ”€â”€ personas/   # MÃ³dulo de personas
â”‚   â”œâ”€â”€ companias/   # MÃ³dulo de compaÃ±Ãas
â”‚   â”œâ”€â”€ tiendas/   # MÃ³dulo de tiendas
```
```

```
â”‚       â”œâ”€â”€ productos/   # MÃ³dulo de productos
â”‚       â”œâ”€â”€ inventario/  # MÃ³dulo de inventario
â”‚       â”œâ”€â”€ kardex/      # MÃ³dulo de kardex
â”‚       â”œâ”€â”€ ventas/      # MÃ³dulo de ventas
â”‚       â”œâ”€â”€ gastos/      # MÃ³dulo de gastos
â”‚       â”œâ”€â”€ metodos-pago/ # MÃ³dulo de mÃ©todos de pago
â”‚       â”œâ”€â”€ proveedores/ # MÃ³dulo de proveedores
â”‚       â”œâ”€â”€ reportes/    # MÃ³dulo de reportes
â”‚       â””â”€â”€ alertas/     # MÃ³dulo de alertas
â”œâ”€â”€ test/                # Pruebas
â”‚   â”œâ”€â”€ app.e2e-spec.ts  # Pruebas end-to-end
â”‚   â””â”€â”€ integration/     # Pruebas de integraciÃ³n
â”œâ”€â”€ dist/                # CÃ³digo compilado
â”œâ”€â”€ node_modules/        # Dependencias
â”œâ”€â”€ .env                 # Variables de entorno
â”œâ”€â”€ .env.development     # Variables de entorno para desarrollo
â”œâ”€â”€ .env.production      # Variables de entorno para producciÃ³n
â”œâ”€â”€ .eslintrc.js         # ConfiguraciÃ³n de ESLint
â”œâ”€â”€ .prettierrc          # ConfiguraciÃ³n de Prettier
â”œâ”€â”€ nest-cli.json        # ConfiguraciÃ³n de NestJS CLI
â”œâ”€â”€ package.json         # Dependencias y scripts
â”œâ”€â”€ tsconfig.json        # ConfiguraciÃ³n de TypeScript
â””â”€â”€ README.md            # DocumentaciÃ³n bÃ¡sica
```

Estructura de un MÃ³dulo

Cada mÃ³dulo en el backend sigue una estructura similar:

```
/modules/ejemplo/
â”œâ”€â”€ controllers/       # Controladores del mÃ³dulo
â”œâ”€â”€ dto/               # DTOs especÃficos del mÃ³dulo
â”œâ”€â”€ entities/          # Entidades de base de datos
â”œâ”€â”€ interfaces/        # Interfaces especÃficas del mÃ³dulo
â”œâ”€â”€ services/          # Servicios del mÃ³dulo
â”œâ”€â”€ ejemplo.module.ts  # DefiniciÃ³n del mÃ³dulo
â”œâ”€â”€ ejemplo.controller.ts # Controlador principal
â””â”€â”€ ejemplo.service.ts # Servicio principal
```

Estructura del Frontend

El frontend sigue una estructura de directorios organizada por funcionalidad, con separaciÃ³n clara entre componentes, pÃ¡ginas y servicios:

```
/frontend/pos-frontend/
â”œâ”€â”€ public/             # Archivos pÃºblicos
â”‚   â”œâ”€â”€ index.html      # Plantilla HTML principal
â”‚   â”œâ”€â”€ favicon.ico     # Favicon
â”‚   â””â”€â”€ manifest.json   # Manifest para PWA
â”œâ”€â”€ src/                # CÃ³digo fuente
â”‚   â”œâ”€â”€ assets/         # Recursos estÃ¡ticos (imÃ¡genes, fuentes, etc.)
â”‚   â”œâ”€â”€ components/     # Componentes reutilizables
â”‚   â”‚   â”œâ”€â”€ common/     # Componentes comunes
â”‚   â”‚   â”œâ”€â”€ forms/      # Componentes de formularios
â”‚   â”‚   â”œâ”€â”€ layout/     # Componentes de layout
â”‚   â”‚   â””â”€â”€ ui/         # Componentes de UI
â”‚   â”œâ”€â”€ contexts/       # Contextos de React
â”‚   â”‚   â”œâ”€â”€ AuthContext.tsx # Contexto de autenticaciÃ³n
â”‚   â”‚   â””â”€â”€ ThemeContext.tsx # Contexto de tema
â”‚   â”œâ”€â”€ hooks/          # Hooks personalizados
â”‚   â”œâ”€â”€ layouts/        # Layouts de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ MainLayout.tsx # Layout principal
â”‚   â”‚   â””â”€â”€ AuthLayout.tsx # Layout de autenticaciÃ³n
â”‚   â”œâ”€â”€ pages/          # PÃ¡ginas de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ Login.tsx   # PÃ¡gina de login
â”‚   â”‚   â”œâ”€â”€ Dashboard.tsx # Dashboard principal
â”‚   â”‚   â”œâ”€â”€ Productos.tsx # GestiÃ³n de productos
â”‚   â”‚   â”œâ”€â”€ Ventas.tsx  # GestiÃ³n de ventas
â”‚   â”‚   â”œâ”€â”€ Gastos.tsx  # GestiÃ³n de gastos
â”‚   â”‚   â”œâ”€â”€ Reportes.tsx # Reportes
â”‚   â”‚   â””â”€â”€ Administracion.tsx # AdministraciÃ³n
â”‚   â”œâ”€â”€ services/       # Servicios para comunicaciÃ³n con el backend
â”‚   â”‚   â”œâ”€â”€ api.ts      # Cliente API base
â”‚   â”‚   â”œâ”€â”€ auth.service.ts # Servicio de autenticaciÃ³n
â”‚   â”‚   â”œâ”€â”€ productos.service.ts # Servicio de productos
â”‚   â”‚   â””â”€â”€ ...         # Otros servicios
â”‚   â”œâ”€â”€ theme/          # ConfiguraciÃ³n de tema
â”‚   â”œâ”€â”€ utils/          # Utilidades
â”‚   â”‚   â”œâ”€â”€ validations.ts # Utilidades de validaciÃ³n
â”‚   â”‚   â”œâ”€â”€ formatters.ts # Formateadores
```

```
â”‚   â”‚   â”œâ”€â”€ helpers.ts  # Funciones auxiliares
â”‚   â”œâ”€â”€ App.tsx         # Componente principal
â”‚   â”œâ”€â”€ index.tsx       # Punto de entrada
â”‚   â””â”€â”€ react-app-env.d.ts # Declaraciones de tipos
â”œâ”€â”€ electron/           # CÃ³digo especÃfico de Electron
â”‚   â”œâ”€â”€ main.js         # Proceso principal de Electron
â”‚   â””â”€â”€ preload.js      # Script de precarga
â”œâ”€â”€ node_modules/       # Dependencias
â”œâ”€â”€ .env                # Variables de entorno
â”œâ”€â”€ .env.development    # Variables de entorno para desarrollo
â”œâ”€â”€ .env.production     # Variables de entorno para producciÃ³n
â”œâ”€â”€ .eslintrc.js        # ConfiguraciÃ³n de ESLint
â”œâ”€â”€ .prettierrc         # ConfiguraciÃ³n de Prettier
â”œâ”€â”€ electron-builder.json # ConfiguraciÃ³n de Electron Builder
â”œâ”€â”€ package.json        # Dependencias y scripts
â”œâ”€â”€ tsconfig.json       # ConfiguraciÃ³n de TypeScript
â””â”€â”€ README.md           # DocumentaciÃ³n bÃ¡sica
```

Estructura de la Base de Datos

La estructura de la base de datos PostgreSQL refleja el modelo de dominio del sistema, con tablas para cada entidad principal y relaciones entre ellas. Las principales tablas son:

- `usuarios`: Almacena informaciÃ³n de los usuarios del sistema
- `roles`: Define los roles disponibles en el sistema
- `personas`: Almacena informaciÃ³n de personas (clientes, empleados, proveedores)
- `companias`: Almacena informaciÃ³n de las compaÃ±Ãas
- `tiendas`: Almacena informaciÃ³n de las tiendas asociadas a una compaÃ±Ãa
- `productos`: CatÃ¡logo de productos
- `inventario`: Estado actual del inventario por producto y tienda
- `kardex`: Registro de movimientos de inventario
- `ventas`: Registro de ventas
- `detalle_ventas`: Detalles de productos en cada venta
- `gastos`: Registro de gastos
- `detalle_gastos`: Detalles de cada gasto
- `metodos_pago`: MÃ©todos de pago disponibles
- `proveedores_productos`: RelaciÃ³n entre proveedores y productos

Convenciones de Nomenclatura

El proyecto sigue convenciones de nomenclatura consistentes para facilitar la comprensiÃ³n y mantenimiento del cÃ³digo:

Backend

- **Archivos**: kebab-case (ej. `auth-service.ts`)
- **Clases**: PascalCase (ej. `AuthService`)
- **MÃ©todos y variables**: camelCase (ej. `getUserById`)
- **Constantes**: UPPER_SNAKE_CASE (ej. `MAX_LOGIN_ATTEMPTS`)
- **Interfaces**: PascalCase con prefijo I (ej. `IUser`)
- **Enums**: PascalCase (ej. `UserRole`)
- **Decoradores**: camelCase (ej. `@isPublic()`)

Frontend

- **Archivos de componentes**: PascalCase (ej. `LoginForm.tsx`)
- **Archivos de utilidades**: camelCase (ej. `formatCurrency.ts`)
- **Componentes**: PascalCase (ej. `<ProductCard />`)
- **Hooks personalizados**: camelCase con prefijo use (ej. `useAuth`)
- **Contextos**: PascalCase con sufijo Context (ej. `AuthContext`)
- **Estilos**: camelCase (ej. `loginContainer`)

Base de Datos

- **Tablas**: snake_case en plural (ej. `usuarios`)
- **Columnas**: snake_case (ej. `fecha_creacion`)
- **Claves primarias**: Prefijo `id_` (ej. `id_usuario`)
- **Claves forÃ¡neas**: Nombre de la tabla referenciada en singular con prefijo `id_` (ej. `id_rol`)
- **Ãndices**: Prefijo `idx_` (ej. `idx_usuarios_email`)

Flujo de Trabajo de Desarrollo

El proyecto estÃ¡ configurado para soportar un flujo de trabajo de desarrollo eficiente:

1. **Desarrollo local**: Utilizando el script `dev-start.sh` que inicia tanto el backend como el frontend en modo desarrollo con recarga en caliente.
2. **Pruebas**: EjecuciÃ³n de pruebas unitarias y de integraciÃ³n mediante scripts npm.
3. **ConstrucciÃ³n**: GeneraciÃ³n de versiones optimizadas para producciÃ³n.
4. **Despliegue**: Instrucciones y scripts para desplegar el sistema en diferentes entornos.

Esta estructura de proyecto proporciona una base sÃ³lida para el desarrollo y mantenimiento del Sistema POS, facilitando la colaboraciÃ³n entre desarrolladores y la extensiÃ³n del sistema con nuevas funcionalidades.

Estructura del Proyecto

El Sistema POS para Tienda de Accesorios MÃ³viles estÃ¡ organizado en una estructura de directorios clara y modular que facilita el desarrollo, mantenimiento y escalabilidad del cÃ³digo. La estructura sigue las mejores prÃ¡cticas para proyectos NestJS (backend) y ReactJS/Electron (frontend).

Estructura General

El proyecto estÃ¡ dividido en dos directorios principales:

```
/pos-system/
â”œâ”€â”€ backend/           # CÃ³digo del servidor NestJS
â”œâ”€â”€ frontend/          # CÃ³digo de la aplicaciÃ³n ReactJS/Electron
â”œâ”€â”€ docs/              # DocumentaciÃ³n del proyecto
â”œâ”€â”€ scripts/           # Scripts de utilidad
â””â”€â”€ .env               # Variables de entorno globales
```

Estructura del Backend

El backend sigue la estructura de directorios recomendada por NestJS, con una organizaciÃ³n modular basada en las funcionalidades del sistema:

```
/backend/
â”œâ”€â”€ dist/              # CÃ³digo compilado
â”œâ”€â”€ node_modules/      # Dependencias
â”œâ”€â”€ src/
â”‚   â”œâ”€â”€ common/        # CÃ³digo compartido entre mÃ³dulos
â”‚   â”‚   â”œâ”€â”€ decorators/    # Decoradores personalizados
â”‚   â”‚   â”œâ”€â”€ dto/           # DTOs compartidos
â”‚   â”‚   â”œâ”€â”€ enums/         # Enumeraciones
â”‚   â”‚   â”œâ”€â”€ exceptions/    # Excepciones personalizadas
â”‚   â”‚   â”œâ”€â”€ filters/       # Filtros de excepciÃ³n
â”‚   â”‚   â”œâ”€â”€ guards/        # Guards compartidos
â”‚   â”‚   â”œâ”€â”€ interceptors/  # Interceptores
â”‚   â”‚   â”œâ”€â”€ interfaces/    # Interfaces compartidas
â”‚   â”‚   â”œâ”€â”€ middleware/    # Middleware
â”‚   â”‚   â””â”€â”€ utils/         # Utilidades
â”‚   â”œâ”€â”€ config/        # ConfiguraciÃ³n de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ database.config.ts
â”‚   â”‚   â”œâ”€â”€ jwt.config.ts
â”‚   â”‚   â””â”€â”€ app.config.ts
â”‚   â”œâ”€â”€ modules/       # MÃ³dulos funcionales
â”‚   â”‚   â”œâ”€â”€ auth/          # AutenticaciÃ³n y autorizaciÃ³n
â”‚   â”‚   â”œâ”€â”€ usuarios/      # GestiÃ³n de usuarios
â”‚   â”‚   â”œâ”€â”€ roles/         # GestiÃ³n de roles
â”‚   â”‚   â”œâ”€â”€ personas/      # GestiÃ³n de personas (clientes, empleados, proveedores)
â”‚   â”‚   â”œâ”€â”€ companias/     # GestiÃ³n de compaÃ±Ãas
â”‚   â”‚   â”œâ”€â”€ tiendas/       # GestiÃ³n de tiendas
â”‚   â”‚   â”œâ”€â”€ productos/     # GestiÃ³n de productos
â”‚   â”‚   â”œâ”€â”€ inventario/    # GestiÃ³n de inventario
â”‚   â”‚   â”œâ”€â”€ kardex/        # Movimientos de inventario
â”‚   â”‚   â”œâ”€â”€ ventas/        # GestiÃ³n de ventas
â”‚   â”‚   â”œâ”€â”€ gastos/        # GestiÃ³n de gastos
â”‚   â”‚   â”œâ”€â”€ metodos-pago/  # MÃ©todos de pago
â”‚   â”‚   â”œâ”€â”€ proveedores/   # GestiÃ³n de proveedores
â”‚   â”‚   â”œâ”€â”€ reportes/      # GeneraciÃ³n de reportes
â”‚   â”‚   â””â”€â”€ alertas/       # Sistema de alertas
â”‚   â”œâ”€â”€ app.module.ts  # MÃ³dulo principal
â”‚   â”œâ”€â”€ app.controller.ts  # Controlador principal
â”‚   â”œâ”€â”€ app.service.ts     # Servicio principal
â”‚   â””â”€â”€ main.ts        # Punto de entrada
â”œâ”€â”€ test/              # Tests
â”‚   â”œâ”€â”€ unit/          # Tests unitarios
â”‚   â”œâ”€â”€ integration/  # Tests de integraciÃ³n
â”‚   â””â”€â”€ e2e/           # Tests end-to-end
â”œâ”€â”€ .env               # Variables de entorno
â”œâ”€â”€ .env.development  # Variables de entorno para desarrollo
â”œâ”€â”€ .env.production   # Variables de entorno para producciÃ³n
â”œâ”€â”€ .eslintrc.js      # ConfiguraciÃ³n de ESLint
â”œâ”€â”€ .prettierrc       # ConfiguraciÃ³n de Prettier
â”œâ”€â”€ nest-cli.json     # ConfiguraciÃ³n de NestJS CLI
â”œâ”€â”€ package.json      # Dependencias y scripts
â”œâ”€â”€ tsconfig.json     # ConfiguraciÃ³n de TypeScript
â””â”€â”€ README.md         # DocumentaciÃ³n bÃ¡sica
```

Estructura de un MÃ³dulo

Cada mÃ³dulo en el backend sigue una estructura consistente:

```
/modules/ejemplo/
â”œâ”€â”€ controllers/      # Controladores del mÃ³dulo
â”‚   â””â”€â”€ ejemplo.controller.ts
â”œâ”€â”€ dto/              # Data Transfer Objects
â”‚   â”œâ”€â”€ create-ejemplo.dto.ts
â”‚   â””â”€â”€ update-ejemplo.dto.ts
â”œâ”€â”€ entities/         # Entidades de base de datos
â”‚   â””â”€â”€ ejemplo.entity.ts
â”œâ”€â”€ services/         # Servicios con lÃ³gica de negocio
â”‚   â””â”€â”€ ejemplo.service.ts
â”œâ”€â”€ repositories/     # Repositorios personalizados (opcional)
â”‚   â””â”€â”€ ejemplo.repository.ts
â”œâ”€â”€ interfaces/       # Interfaces especÃficas del mÃ³dulo
â”‚   â””â”€â”€ ejemplo.interface.ts
â”œâ”€â”€ ejemplo.module.ts # DefiniciÃ³n del mÃ³dulo
â””â”€â”€ ejemplo.constants.ts  # Constantes especÃficas del mÃ³dulo
```

Estructura del Frontend

El frontend sigue una estructura organizada por caracterÃsticas y componentes:

```
/frontend/
â”œâ”€â”€ build/            # CÃ³digo compilado para producciÃ³n
â”œâ”€â”€ node_modules/     # Dependencias
â”œâ”€â”€ public/           # Archivos estÃ¡ticos
â”‚   â”œâ”€â”€ index.html    # Plantilla HTML principal
â”‚   â”œâ”€â”€ favicon.ico   # Favicon
â”‚   â””â”€â”€ assets/       # Recursos estÃ¡ticos
â”œâ”€â”€ electron/         # ConfiguraciÃ³n de Electron
â”‚   â”œâ”€â”€ main.js       # Proceso principal de Electron
â”‚   â””â”€â”€ preload.js    # Script de precarga
â”œâ”€â”€ src/
â”‚   â”œâ”€â”€ assets/       # Recursos de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ images/   # ImÃ¡genes
â”‚   â”‚   â”œâ”€â”€ styles/   # Estilos globales
â”‚   â”‚   â””â”€â”€ fonts/    # Fuentes
â”‚   â”œâ”€â”€ components/   # Componentes reutilizables
â”‚   â”‚   â”œâ”€â”€ common/   # Componentes comunes
â”‚   â”‚   â”œâ”€â”€ forms/    # Componentes de formularios
â”‚   â”‚   â”œâ”€â”€ layout/   # Componentes de layout
â”‚   â”‚   â””â”€â”€ ui/       # Componentes de UI
â”‚   â”œâ”€â”€ contexts/     # Contextos de React
â”‚   â”‚   â”œâ”€â”€ AuthContext.tsx
â”‚   â”‚   â””â”€â”€ ThemeContext.tsx
â”‚   â”œâ”€â”€ hooks/        # Hooks personalizados
â”‚   â”‚   â”œâ”€â”€ useForm.ts
â”‚   â”‚   â””â”€â”€ useApi.ts
â”‚   â”œâ”€â”€ layouts/      # Layouts de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ MainLayout.tsx
â”‚   â”‚   â””â”€â”€ AuthLayout.tsx
â”‚   â”œâ”€â”€ pages/        # PÃ¡ginas de la aplicaciÃ³n
â”‚   â”‚   â”œâ”€â”€ Login.tsx
â”‚   â”‚   â”œâ”€â”€ Dashboard.tsx
â”‚   â”‚   â”œâ”€â”€ Productos.tsx
â”‚   â”‚   â”œâ”€â”€ Ventas.tsx
â”‚   â”‚   â”œâ”€â”€ Gastos.tsx
â”‚   â”‚   â”œâ”€â”€ Reportes.tsx
â”‚   â”‚   â””â”€â”€ Administracion.tsx
â”‚   â”œâ”€â”€ services/     # Servicios para comunicaciÃ³n con API
â”‚   â”‚   â”œâ”€â”€ api.ts
â”‚   â”‚   â”œâ”€â”€ auth.service.ts
â”‚   â”‚   â”œâ”€â”€ productos.service.ts
â”‚   â”‚   â””â”€â”€ ventas.service.ts
â”‚   â”œâ”€â”€ utils/        # Utilidades
â”‚   â”‚   â”œâ”€â”€ validations.ts
â”‚   â”‚   â”œâ”€â”€ formatters.ts
â”‚   â”‚   â””â”€â”€ helpers.ts
â”‚   â”œâ”€â”€ theme/        # ConfiguraciÃ³n de temas
â”‚   â”‚   â””â”€â”€ index.ts
â”‚   â”œâ”€â”€ App.tsx       # Componente principal
â”‚   â””â”€â”€ index.tsx     # Punto de entrada
â”œâ”€â”€ .env              # Variables de entorno
â”œâ”€â”€ .env.development  # Variables de entorno para desarrollo
â”œâ”€â”€ .env.production   # Variables de entorno para producciÃ³n
â”œâ”€â”€ .eslintrc.js      # ConfiguraciÃ³n de ESLint
â”œâ”€â”€ .prettierrc       # ConfiguraciÃ³n de Prettier
â”œâ”€â”€ electron-builder.json  # ConfiguraciÃ³n de electron-builder
â”œâ”€â”€ package.json      # Dependencias y scripts
â”œâ”€â”€ tsconfig.json     # ConfiguraciÃ³n de TypeScript
```

â””â”€â”€ README.md   # DocumentaciÃ³n bÃ¡sica
```

### Estructura de la Base de Datos

La estructura de la base de datos PostgreSQL refleja el modelo de dominio del sistema:

```
pos_db
â”œâ”€â”€ public
â”‚   â”œâ”€â”€ compania                # Tabla de compaÃ±Ãas
â”‚   â”œâ”€â”€ tienda                  # Tabla de tiendas
â”‚   â”œâ”€â”€ tipo_identificacion     # Tipos de identificaciÃ³n
â”‚   â”œâ”€â”€ persona                 # Tabla de personas (clientes, empleados, proveedores)
â”‚   â”œâ”€â”€ rol                     # Tabla de roles
â”‚   â”œâ”€â”€ usuario                 # Tabla de usuarios
â”‚   â”œâ”€â”€ unidad                  # Unidades de medida
â”‚   â”œâ”€â”€ impuesto                # Tabla de impuestos
â”‚   â”œâ”€â”€ producto                # Tabla de productos
â”‚   â”œâ”€â”€ inventario              # Tabla de inventario
â”‚   â”œâ”€â”€ kardex                  # Tabla de movimientos de inventario
â”‚   â”œâ”€â”€ metodo_pago             # Tabla de mÃ©todos de pago
â”‚   â”œâ”€â”€ venta                   # Tabla de ventas
â”‚   â”œâ”€â”€ detalle_venta           # Detalles de ventas
â”‚   â”œâ”€â”€ gasto                   # Tabla de gastos
â”‚   â”œâ”€â”€ detalle_gasto           # Detalles de gastos
â”‚   â””â”€â”€ proveedor_producto      # RelaciÃ³n entre proveedores y productos
```

Convenciones de Nomenclatura

El proyecto sigue convenciones de nomenclatura consistentes para facilitar la comprensiÃ³n y mantenimiento del cÃ³digo:

Backend

- **Archivos**: kebab-case (ej. `user-profile.service.ts`)
- **Clases**: PascalCase (ej. `UserProfileService`)
- **MÃ©todos y propiedades**: camelCase (ej. `getUserProfile()`)
- **Constantes**: SNAKE_CASE_MAYÃšSCULAS (ej. `MAX_LOGIN_ATTEMPTS`)
- **Interfaces**: PascalCase con prefijo I (ej. `IUserProfile`)
- **Enumeraciones**: PascalCase (ej. `UserRole`)
- **MÃ³dulos**: kebab-case (ej. `user-profile`)

Frontend

- **Archivos de componentes**: PascalCase (ej. `UserProfile.tsx`)
- **Archivos de utilidades**: camelCase (ej. `formatDate.ts`)
- **Componentes**: PascalCase (ej. `UserProfile`)
- **Props**: camelCase (ej. `userName`)
- **Hooks personalizados**: camelCase con prefijo use (ej. `useFormValidation`)
- **Contextos**: PascalCase con sufijo Context (ej. `UserContext`)

Base de Datos

- **Tablas**: snake_case en singular (ej. `usuario`)
- **Columnas**: snake_case (ej. `fecha_creacion`)
- **Claves primarias**: id o tabla_id (ej. `usuario_id`)
- **Claves forÃ¡neas**: tabla_referenciada_id (ej. `rol_id`)
- **Ãndices**: idx_tabla_columna (ej. `idx_usuario_email`)

GestiÃ³n de Dependencias

Las dependencias del proyecto se gestionan mediante npm y estÃ¡n definidas en los archivos `package.json` de los directorios backend y frontend. Las dependencias estÃ¡n organizadas en:

- **dependencies**: Dependencias necesarias para la ejecuciÃ³n del proyecto en producciÃ³n.
- **devDependencies**: Dependencias necesarias solo para el desarrollo y testing.

Scripts de Utilidad

El proyecto incluye varios scripts para facilitar el desarrollo, testing y despliegue:

Backend

- `npm run start`: Inicia el servidor en modo desarrollo.
- `npm run start:dev`: Inicia el servidor en modo desarrollo con hot reload.
- `npm run start:prod`: Inicia el servidor en modo producciÃ³n.
- `npm run build`: Compila el proyecto.
- `npm run test`: Ejecuta los tests unitarios.
- `npm run test:e2e`: Ejecuta los tests end-to-end.
- `npm run lint`: Ejecuta el linter.

Frontend
```

- `npm run start`: Inicia la aplicaciÃ³n React en modo desarrollo.
- `npm run build`: Compila la aplicaciÃ³n React para producciÃ³n.
- `npm run electron:dev`: Inicia la aplicaciÃ³n Electron en modo desarrollo.
- `npm run electron:build`: Compila la aplicaciÃ³n Electron para producciÃ³n.
- `npm run test`: Ejecuta los tests.
- `npm run lint`: Ejecuta el linter.

Flujo de Desarrollo

El flujo de desarrollo recomendado para el proyecto es:

1. Clonar el repositorio.
2. Instalar las dependencias con `npm install` en los directorios backend y frontend.
3. Configurar las variables de entorno en los archivos `.env`.
4. Iniciar la base de datos PostgreSQL.
5. Ejecutar las migraciones de la base de datos.
6. Iniciar el backend con `npm run start:dev`.
7. Iniciar el frontend con `npm run electron:dev`.
8. Desarrollar siguiendo las convenciones del proyecto.
9. Ejecutar los tests antes de hacer commit.
10. Construir la aplicaciÃ³n para producciÃ³n cuando sea necesario.

Esta estructura de proyecto proporciona una base sÃ³lida para el desarrollo y mantenimiento del Sistema POS para Tienda de Accesorios MÃ³viles, facilitando la colaboraciÃ³n entre desarrolladores y la escalabilidad del sistema.

Estructura del Proyecto

El Sistema POS para Tienda de Accesorios MÃ³viles estÃ¡ organizado siguiendo una estructura modular y clara que facilita el desarrollo, mantenimiento y escalabilidad. La organizaciÃ³n de archivos y directorios sigue las mejores prÃ¡cticas para proyectos NestJS (backend) y React+Electron (frontend).

Estructura General del Proyecto

```
pos-system/
â”œâ”€â”€ backend/                # CÃ³digo del backend
â”‚   â””â”€â”€ pos-backend/        # Proyecto NestJS
â”œâ”€â”€ frontend/               # CÃ³digo del frontend
â”‚   â””â”€â”€ pos-frontend/       # Proyecto React+Electron
â”œâ”€â”€ docs/                   # DocumentaciÃ³n
â”‚   â”œâ”€â”€ tecnica/            # DocumentaciÃ³n tÃ©cnica
â”‚   â””â”€â”€ usuario/            # Manual de usuario
â”œâ”€â”€ scripts/                # Scripts de utilidad
â”œâ”€â”€ dev-start.sh            # Script para iniciar el entorno de desarrollo
â”œâ”€â”€ run-integration-tests.sh # Script para ejecutar pruebas de integraciÃ³n
â””â”€â”€ optimize.sh             # Script para optimizaciÃ³n y correcciÃ³n de errores
```

Estructura del Backend (NestJS)

El backend estÃ¡ organizado siguiendo la estructura recomendada por NestJS, con una clara separaciÃ³n de responsabilidades y un enfoque modular.

```
pos-backend/
â”œâ”€â”€ src/
â”‚   â”œâ”€â”€ main.ts               # Punto de entrada de la aplicaciÃ³n
â”‚   â”œâ”€â”€ app.module.ts         # MÃ³dulo principal
â”‚   â”œâ”€â”€ app.controller.ts     # Controlador principal
â”‚   â”œâ”€â”€ app.service.ts        # Servicio principal
â”‚   â”œâ”€â”€ common/               # CÃ³digo compartido
â”‚   â”‚   â”œâ”€â”€ decorators/       # Decoradores personalizados
â”‚   â”‚   â”œâ”€â”€ dto/              # DTOs compartidos
â”‚   â”‚   â”œâ”€â”€ enums/            # Enumeraciones
â”‚   â”‚   â”‚   â”œâ”€â”€ tipo-movimiento.enum.ts
â”‚   â”‚   â”‚   â””â”€â”€ tipo-persona.enum.ts
â”‚   â”‚   â”œâ”€â”€ filters/          # Filtros de excepciÃ³n
â”‚   â”‚   â”‚   â”œâ”€â”€ http-exception.filter.ts
â”‚   â”‚   â”‚   â””â”€â”€ all-exceptions.filter.ts
â”‚   â”‚   â”œâ”€â”€ interceptors/     # Interceptores
â”‚   â”‚   â”‚   â””â”€â”€ transform.interceptor.ts
â”‚   â”‚   â””â”€â”€ middleware/       # Middleware
â”‚   â”‚       â””â”€â”€ logger.middleware.ts
â”‚   â”œâ”€â”€ config/               # ConfiguraciÃ³n
â”‚   â”‚   â”œâ”€â”€ database.config.ts # ConfiguraciÃ³n de base de datos
â”‚   â”‚   â”œâ”€â”€ jwt.config.ts     # ConfiguraciÃ³n de JWT
â”‚   â”‚   â””â”€â”€ cors.config.ts    # ConfiguraciÃ³n de CORS
â”‚   â””â”€â”€ modules/              # MÃ³dulos funcionales
â”‚       â”œâ”€â”€ auth/             # AutenticaciÃ³n
â”‚       â”‚   â”œâ”€â”€ auth.controller.ts
â”‚       â”‚   â”œâ”€â”€ auth.module.ts
```

```
â”‚       â”‚   â”œâ”€â”€ auth.service.ts
â”‚       â”‚   â”œâ”€â”€ dto/
â”‚       â”‚   â”‚   â”œâ”€â”€ login.dto.ts
â”‚       â”‚   â”‚   â””â”€â”€ register.dto.ts
â”‚       â”‚   â”œâ”€â”€ guards/
â”‚       â”‚   â”‚   â”œâ”€â”€ jwt-auth.guard.ts
â”‚       â”‚   â”‚   â””â”€â”€ roles.guard.ts
â”‚       â”‚   â”œâ”€â”€ decorators/
â”‚       â”‚   â”‚   â”œâ”€â”€ public.decorator.ts
â”‚       â”‚   â”‚   â””â”€â”€ roles.decorator.ts
â”‚       â”‚   â””â”€â”€ strategies/
â”‚       â”‚       â””â”€â”€ jwt.strategy.ts
â”‚       â”œâ”€â”€ usuarios/          # Usuarios
â”‚       â”‚   â”œâ”€â”€ usuarios.controller.ts
â”‚       â”‚   â”œâ”€â”€ usuarios.module.ts
â”‚       â”‚   â”œâ”€â”€ usuarios.service.ts
â”‚       â”‚   â”œâ”€â”€ dto/
â”‚       â”‚   â”‚   â”œâ”€â”€ create-usuario.dto.ts
â”‚       â”‚   â”‚   â””â”€â”€ update-usuario.dto.ts
â”‚       â”‚   â”œâ”€â”€ entities/
â”‚       â”‚       â””â”€â”€ usuario.entity.ts
â”‚       â”œâ”€â”€ roles/             # Roles
â”‚       â”œâ”€â”€ personas/          # Personas
â”‚       â”œâ”€â”€ companias/         # CompaÃ±Ãas
â”‚       â”œâ”€â”€ tiendas/           # Tiendas
â”‚       â”œâ”€â”€ productos/         # Productos
â”‚       â”‚   â”œâ”€â”€ entities/
â”‚       â”‚   â”‚   â”œâ”€â”€ producto.entity.ts
â”‚       â”‚   â”‚   â”œâ”€â”€ unidad.entity.ts
â”‚       â”‚   â”‚   â””â”€â”€ impuesto.entity.ts
â”‚       â”‚   â””â”€â”€ ...
â”‚       â”œâ”€â”€ inventario/        # Inventario
â”‚       â”œâ”€â”€ kardex/            # Kardex
â”‚       â”œâ”€â”€ ventas/            # Ventas
â”‚       â”‚   â”œâ”€â”€ entities/
â”‚       â”‚   â”‚   â”œâ”€â”€ venta.entity.ts
â”‚       â”‚   â”‚   â””â”€â”€ detalle-venta.entity.ts
â”‚       â”‚   â””â”€â”€ ...
â”‚       â”œâ”€â”€ gastos/            # Gastos
â”‚       â”‚   â”œâ”€â”€ entities/
â”‚       â”‚   â”‚   â”œâ”€â”€ gasto.entity.ts
â”‚       â”‚   â”‚   â””â”€â”€ detalle-gasto.entity.ts
â”‚       â”‚   â””â”€â”€ ...
â”‚       â”œâ”€â”€ metodos-pago/      # MÃ©todos de pago
â”‚       â”œâ”€â”€ proveedores/       # Proveedores
â”‚       â”œâ”€â”€ reportes/          # Reportes
â”‚       â””â”€â”€ alertas/           # Alertas
â”œâ”€â”€ test/                     # Pruebas
â”‚   â”œâ”€â”€ app.e2e-spec.ts
â”‚   â”œâ”€â”€ jest-e2e.json
â”‚   â””â”€â”€ integration/           # Pruebas de integraciÃ³n
â”‚       â”œâ”€â”€ auth.integration.spec.ts
â”‚       â”œâ”€â”€ productos.integration.spec.ts
â”‚       â””â”€â”€ ventas-gastos.integration.spec.ts
â”œâ”€â”€ .env                      # Variables de entorno
â”œâ”€â”€ .env.development          # Variables para desarrollo
â”œâ”€â”€ .env.production           # Variables para producciÃ³n
â”œâ”€â”€ nest-cli.json             # ConfiguraciÃ³n de NestJS CLI
â”œâ”€â”€ package.json              # Dependencias y scripts
â”œâ”€â”€ tsconfig.json             # ConfiguraciÃ³n de TypeScript
â””â”€â”€ README.md                 # DocumentaciÃ³n bÃ¡sica
```

Estructura de un MÃ³dulo TÃpico

Cada mÃ³dulo funcional sigue una estructura similar para mantener la consistencia en todo el proyecto:

```
modulo/
â”œâ”€â”€ modulo.controller.ts    # Controlador: maneja las peticiones HTTP
â”œâ”€â”€ modulo.module.ts        # MÃ³dulo: configura el mÃ³dulo y sus dependencias
â”œâ”€â”€ modulo.service.ts       # Servicio: implementa la lÃ³gica de negocio
â”œâ”€â”€ dto/                    # Data Transfer Objects
â”‚   â”œâ”€â”€ create-modulo.dto.ts  # DTO para creaciÃ³n
â”‚   â””â”€â”€ update-modulo.dto.ts  # DTO para actualizaciÃ³n
â””â”€â”€ entities/               # Entidades de base de datos
    â””â”€â”€ modulo.entity.ts    # DefiniciÃ³n de la entidad
```

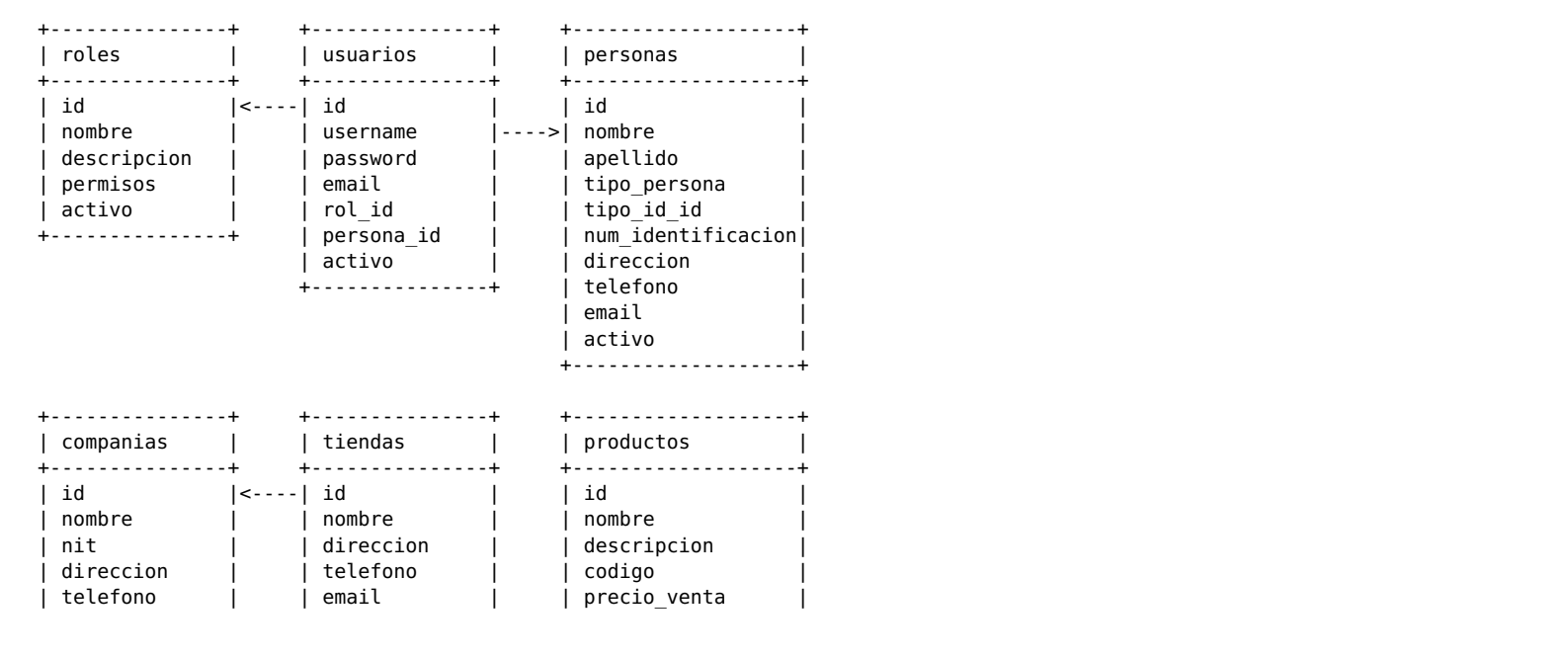
Estructura del Frontend (React + Electron)

El frontend estÃ¡ organizado siguiendo las mejores prÃ¡cticas para aplicaciones React, con una estructura que facilita la separaciÃ³n de responsabilidades y la reutilizaciÃ³n de componentes.

```
pos-frontend/
â”œâ”€â”€ public/                # Archivos pÃºblicos
â”‚   â”œâ”€â”€ index.html         # Plantilla HTML
â”‚   â”œâ”€â”€ favicon.ico        # Favicon
â”‚   â””â”€â”€ manifest.json      # Web App Manifest
â”œâ”€â”€ src/                   # CÃ³digo fuente
â”‚   â”œâ”€â”€ index.tsx          # Punto de entrada
â”‚   â”œâ”€â”€ App.tsx            # Componente principal
â”‚   â”œâ”€â”€ assets/            # Recursos estÃ¡ticos
â”‚   â”‚   â”œâ”€â”€ images/        # ImÃ¡genes
â”‚   â”‚   â””â”€â”€ styles/        # Estilos globales
â”‚   â”œâ”€â”€ components/        # Componentes reutilizables
â”‚   â”‚   â”œâ”€â”€ FormField.tsx   # Campo de formulario
â”‚   â”‚   â”œâ”€â”€ ProtectedRoute.tsx # Ruta protegida
â”‚   â”‚   â””â”€â”€ ThemeToggle.tsx # Cambio de tema
â”‚   â”œâ”€â”€ contexts/          # Contextos de React
â”‚   â”‚   â”œâ”€â”€ AuthContext.tsx # Contexto de autenticaciÃ³n
â”‚   â”‚   â””â”€â”€ ThemeContext.tsx # Contexto de tema
â”‚   â”œâ”€â”€ hooks/             # Hooks personalizados
â”‚   â”‚   â””â”€â”€ useForm.ts      # Hook para formularios
â”‚   â”œâ”€â”€ layouts/           # Layouts
â”‚   â”‚   â”œâ”€â”€ MainLayout.tsx  # Layout principal
â”‚   â”‚   â””â”€â”€ AuthLayout.tsx  # Layout de autenticaciÃ³n
â”‚   â”œâ”€â”€ pages/             # PÃ¡ginas
â”‚   â”‚   â”œâ”€â”€ Login.tsx       # PÃ¡gina de login
â”‚   â”‚   â”œâ”€â”€ Dashboard.tsx   # Dashboard
â”‚   â”‚   â”œâ”€â”€ Productos.tsx   # GestiÃ³n de productos
â”‚   â”‚   â”œâ”€â”€ Ventas.tsx      # GestiÃ³n de ventas
â”‚   â”‚   â”œâ”€â”€ Gastos.tsx      # GestiÃ³n de gastos
â”‚   â”‚   â”œâ”€â”€ Reportes.tsx    # Reportes
â”‚   â”‚   â”œâ”€â”€ Administracion.tsx # AdministraciÃ³n
â”‚   â”‚   â”œâ”€â”€ NotFound.tsx    # PÃ¡gina 404
â”‚   â”‚   â””â”€â”€ Unauthorized.tsx # Acceso no autorizado
â”‚   â”œâ”€â”€ services/          # Servicios
â”‚   â”‚   â””â”€â”€ api.ts          # Cliente API
â”‚   â”œâ”€â”€ theme/             # ConfiguraciÃ³n de tema
â”‚   â”‚   â””â”€â”€ index.ts        # Tema personalizado
â”‚   â””â”€â”€ utils/             # Utilidades
â”‚       â””â”€â”€ validations.ts  # Validaciones
â”œâ”€â”€ electron/              # CÃ³digo de Electron
â”‚   â”œâ”€â”€ main.js            # Proceso principal
â”‚   â””â”€â”€ preload.js         # Script de precarga
â”œâ”€â”€ scripts/               # Scripts
â”‚   â””â”€â”€ release.js         # Script de release
â”œâ”€â”€ .env.development       # Variables para desarrollo
â”œâ”€â”€ .env.production        # Variables para producciÃ³n
â”œâ”€â”€ electron-builder.json   # ConfiguraciÃ³n de electron-builder
â”œâ”€â”€ package.json           # Dependencias y scripts
â”œâ”€â”€ tsconfig.json          # ConfiguraciÃ³n de TypeScript
â””â”€â”€ README.md              # DocumentaciÃ³n bÃ¡sica
```

Estructura de la Base de Datos

La base de datos PostgreSQL estÃ¡ diseÃ±ada siguiendo principios de normalizaciÃ³n y optimizaciÃ³n para el rendimiento. A continuaciÃ³n se muestra un diagrama simplificado de las principales tablas y sus relaciones:



```
| email         |     | compania_id   |     | precio_compra     |
| activo        |     | activo        |     | impuesto_id       |
+---------------+     +---------------+     | unidad_id         |
                                            | imagen            |
                                            | activo            |
                                            +-------------------+

+---------------+     +---------------+     +-------------------+
| inventario    |     | kardex        |     | ventas            |
+---------------+     +---------------+     +-------------------+
| id            |     | id            |     | id                |
| producto_id   |     | producto_id   |     | fecha             |
| tienda_id     |     | tipo_movimiento|    | cliente_id        |
| cantidad      |     | cantidad      |     | vendedor_id       |
| stock_minimo  |     | fecha         |     | tienda_id         |
| activo        |     | referencia    |     | total             |
+---------------+     | venta_id      |     | subtotal          |
                      | activo        |     | impuesto          |
                      +---------------+     | estado            |
                                            | metodo_pago_id    |
                                            +-------------------+

+---------------+     +---------------+     +-------------------+
| gastos        |     | detalle_gasto |     | detalle_venta     |
+---------------+     +---------------+     +-------------------+
| id            |<----| id            |     | id                |
| fecha         |     | gasto_id      |     | venta_id          |
| descripcion   |     | descripcion   |     | producto_id       |
| total         |     | monto         |     | cantidad          |
| tienda_id     |     | activo        |     | precio            |
| usuario_id    |     +---------------+     | subtotal          |
| activo        |                           | impuesto          |
+---------------+                           | activo            |
                                            +-------------------+
```

Convenciones de Nomenclatura

Para mantener la consistencia en todo el proyecto, se siguen las siguientes convenciones de nomenclatura:

1. **Archivos y Directorios**:
   - Nombres en minÃºsculas con guiones para separar palabras (kebab-case)
   - Extensiones segÃºn el tipo de archivo (.ts, .tsx, .js, .jsx, .md)

2. **Clases**:
   - PascalCase (primera letra de cada palabra en mayÃºscula)
   - Sufijos segÃºn su funciÃ³n (Controller, Service, Entity, etc.)

3. **Interfaces y Tipos**:
   - PascalCase
   - Prefijo "I" para interfaces (opcional)

4. **Variables y Funciones**:
   - camelCase (primera letra en minÃºscula, resto de palabras con primera letra en mayÃºscula)

5. **Constantes**:
   - UPPER_SNAKE_CASE (mayÃºsculas con guiones bajos)

6. **Componentes React**:
   - PascalCase
   - Un componente por archivo

7. **Tablas de Base de Datos**:
   - snake_case (minÃºsculas con guiones bajos)
   - Nombres en plural

8. **Columnas de Base de Datos**:
   - snake_case
   - Claves primarias: "id"
   - Claves forÃ¡neas: "entidad_id"

Esta estructura de proyecto proporciona una organizaciÃ³n clara y coherente que facilita el desarrollo, mantenimiento y escalabilidad del sistema.

Funcionalidades del Sistema

El Sistema POS para Tienda de Accesorios MÃ³viles ofrece un conjunto completo de funcionalidades diseÃ±adas para cubrir todas las necesidades operativas de una tienda de accesorios mÃ³viles, desde la gestiÃ³n de inventario hasta el procesamiento de ventas y la generaciÃ³n de reportes. A continuaciÃ³n, se detallan las principales funcionalidades del sistema.

GestiÃ³n de Usuarios y AutenticaciÃ³n

AutenticaciÃ³n y Control de Acceso

- **Inicio de sesiÃ³n seguro**: AutenticaciÃ³n mediante credenciales (usuario/contraseÃ±a) con encriptaciÃ³n de contraseÃ±as mediante bcrypt.
- **AutenticaciÃ³n basada en tokens**: ImplementaciÃ³n de JWT (JSON Web Tokens) para mantener sesiones seguras.
- **Control de acceso basado en roles**: Dos roles principales (vendedor y administrador) con diferentes niveles de acceso a funcionalidades.
- **ProtecciÃ³n de rutas**: Middleware de autenticaciÃ³n para proteger rutas sensibles.
- **GestiÃ³n de sesiones**: Manejo de sesiones con expiraciÃ³n configurable.
- **Bloqueo de cuentas**: Bloqueo temporal de cuentas despuÃ©s de mÃºltiples intentos fallidos de inicio de sesiÃ³n.

GestiÃ³n de Usuarios

- **Registro de usuarios**: CreaciÃ³n de nuevos usuarios con asignaciÃ³n de roles.
- **EdiciÃ³n de perfiles**: ActualizaciÃ³n de informaciÃ³n personal y credenciales.
- **GestiÃ³n de roles**: AsignaciÃ³n y modificaciÃ³n de roles para usuarios.
- **ActivaciÃ³n/desactivaciÃ³n de usuarios**: Control de acceso al sistema mediante estados de usuario.
- **RecuperaciÃ³n de contraseÃ±as**: Mecanismo seguro para restablecer contraseÃ±as olvidadas.
- **Historial de actividad**: Registro de acciones realizadas por cada usuario.

GestiÃ³n de Productos e Inventario

CatÃ¡logo de Productos

- **Registro de productos**: CreaciÃ³n y mantenimiento de productos con informaciÃ³n detallada.
- **CategorizaciÃ³n**: OrganizaciÃ³n de productos por categorÃas y subcategorÃas.
- **GestiÃ³n de precios**: ConfiguraciÃ³n de precios de venta y compra.
- **GestiÃ³n de impuestos**: AsignaciÃ³n de tasas de impuestos a productos.
- **CÃ³digos de barras**: GeneraciÃ³n y lectura de cÃ³digos de barras.
- **ImÃ¡genes de productos**: Carga y visualizaciÃ³n de imÃ¡genes de productos.
- **BÃºsqueda avanzada**: Filtros y bÃºsqueda por mÃºltiples criterios.

Control de Inventario

- **Seguimiento de stock**: Monitoreo en tiempo real de niveles de inventario.
- **MÃºltiples ubicaciones**: GestiÃ³n de inventario en diferentes tiendas.
- **Alertas de stock bajo**: Notificaciones automÃ¡ticas cuando el inventario alcanza niveles crÃticos.
- **Ajustes de inventario**: CorrecciÃ³n manual de discrepancias en el inventario.
- **Reserva de productos**: Capacidad para reservar productos para pedidos especÃficos.
- **Transferencias entre tiendas**: Movimiento de productos entre diferentes ubicaciones.

Sistema Kardex

- **Registro de movimientos**: DocumentaciÃ³n detallada de todas las entradas y salidas de inventario.
- **Tipos de movimientos**: CategorizaciÃ³n de movimientos (compra, venta, ajuste, transferencia).
- **Historial completo**: Trazabilidad completa de cada producto en el inventario.
- **ValoraciÃ³n de inventario**: CÃ¡lculo del valor del inventario basado en diferentes mÃ©todos (FIFO, LIFO, promedio ponderado).
- **Reportes de movimientos**: GeneraciÃ³n de informes detallados de movimientos de inventario.

GestiÃ³n de Ventas

Proceso de Venta

- **Interfaz de punto de venta**: Interfaz intuitiva y rÃ¡pida para procesar ventas.
- **BÃºsqueda rÃ¡pida de productos**: BÃºsqueda por cÃ³digo, nombre o escaneo de cÃ³digo de barras.
- **CÃ¡lculo automÃ¡tico**: CÃ¡lculo de subtotales, impuestos y total.
- **Descuentos**: AplicaciÃ³n de descuentos por producto o al total de la venta.
- **MÃºltiples mÃ©todos de pago**: Soporte para efectivo, tarjeta, transferencia y pagos mixtos.
- **EmisiÃ³n de comprobantes**: GeneraciÃ³n de facturas, boletas y tickets.
- **Ventas a crÃ©dito**: Registro de ventas con pago diferido.
- **Devoluciones**: Procesamiento de devoluciones y reembolsos.

GestiÃ³n de Clientes

- **Registro de clientes**: Mantenimiento de base de datos de clientes.
- **Historial de compras**: Seguimiento de compras por cliente.
- **CrÃ©dito y saldo**: Control de crÃ©dito disponible y saldo pendiente.
- **CategorizaciÃ³n de clientes**: ClasificaciÃ³n de clientes por volumen de compras o frecuencia.
- **Notificaciones**: EnvÃo de notificaciones a clientes sobre promociones o saldos pendientes.

GestiÃ³n de Gastos

- **Registro de gastos**: DocumentaciÃ³n de gastos operativos y administrativos.
- **CategorizaciÃ³n**: ClasificaciÃ³n de gastos por tipo y concepto.
- **Comprobantes**: Adjuntar comprobantes digitales a los registros de gastos.
- **Aprobaciones**: Flujo de aprobaciÃ³n para gastos que superan ciertos montos.
- **Gastos recurrentes**: ConfiguraciÃ³n de gastos periÃ³dicos.
- **Presupuestos**: ComparaciÃ³n de gastos contra presupuestos establecidos.

Reportes y AnÃ¡lisis

Reportes Financieros

- **Balance de ingresos y gastos**: Resumen mensual de ingresos y gastos.
- **Estado de resultados**: Reporte de ganancias y pÃ©rdidas.
- **Flujo de caja**: Seguimiento de entradas y salidas de efectivo.
- **AnÃ¡lisis de rentabilidad**: CÃ¡lculo de mÃ¡rgenes de ganancia por producto y categorÃa.
- **Proyecciones**: Estimaciones de ventas y gastos futuros basados en datos histÃ³ricos.

Reportes de Ventas

- **Ventas por perÃodo**: AnÃ¡lisis de ventas por dÃa, semana, mes o aÃ±o.
- **Ventas por vendedor**: DesempeÃ±o de cada vendedor.
- **Ventas por producto**: Productos mÃ¡s y menos vendidos.
- **Ventas por categorÃa**: AnÃ¡lisis por categorÃa de producto.
- **Ventas por mÃ©todo de pago**: DistribuciÃ³n de ventas segÃºn forma de pago.
- **Ventas por cliente**: AnÃ¡lisis de compras por cliente.
- **Tendencias de ventas**: GrÃ¡ficos y anÃ¡lisis de tendencias a lo largo del tiempo.

Reportes de Inventario

- **ValoraciÃ³n de inventario**: Valor total del inventario actual.
- **RotaciÃ³n de inventario**: AnÃ¡lisis de la velocidad de rotaciÃ³n de productos.
- **Productos sin movimiento**: IdentificaciÃ³n de productos estancados.
- **Productos mÃ¡s vendidos**: Ranking de productos por volumen de ventas.
- **Productos con stock bajo**: Lista de productos que requieren reposiciÃ³n.
- **PÃ©rdidas y mermas**: Registro y anÃ¡lisis de pÃ©rdidas de inventario.

AdministraciÃ³n del Sistema

GestiÃ³n de CompaÃ±Ãas y Tiendas

- **Registro de compaÃ±Ãas**: CreaciÃ³n y configuraciÃ³n de compaÃ±Ãas.
- **GestiÃ³n de tiendas**: AdministraciÃ³n de mÃºltiples tiendas asociadas a una compaÃ±Ãa.
- **ConfiguraciÃ³n por tienda**: ParÃ¡metros especÃficos para cada tienda.
- **Permisos por tienda**: Control de acceso de usuarios a tiendas especÃficas.

GestiÃ³n de Proveedores

- **Registro de proveedores**: Mantenimiento de base de datos de proveedores.
- **CatÃ¡logo por proveedor**: AsociaciÃ³n de productos con proveedores.
- **Historial de compras**: Seguimiento de compras realizadas a cada proveedor.
- **EvaluaciÃ³n de proveedores**: CalificaciÃ³n basada en cumplimiento y calidad.
- **Contactos**: GestiÃ³n de informaciÃ³n de contacto de proveedores.

ConfiguraciÃ³n del Sistema

- **ParÃ¡metros generales**: ConfiguraciÃ³n de parÃ¡metros operativos del sistema.
- **PersonalizaciÃ³n de interfaz**: AdaptaciÃ³n de la interfaz a necesidades especÃficas.
- **GestiÃ³n de impuestos**: ConfiguraciÃ³n de tasas y reglas de impuestos.
- **Formatos de documentos**: PersonalizaciÃ³n de facturas, boletas y reportes.
- **Copias de seguridad**: ProgramaciÃ³n y ejecuciÃ³n de respaldos de datos.
- **Registro de auditorÃa**: Seguimiento de cambios y acciones en el sistema.

CaracterÃsticas TÃ©cnicas Avanzadas

Seguridad

- **EncriptaciÃ³n de datos sensibles**: ProtecciÃ³n de informaciÃ³n confidencial.
- **ValidaciÃ³n de datos**: VerificaciÃ³n de integridad y formato de datos ingresados.
- **ProtecciÃ³n contra ataques comunes**: ImplementaciÃ³n de medidas contra SQL injection, XSS, CSRF, etc.
- **Registro de actividad**: Log detallado de acciones realizadas en el sistema.
- **PolÃticas de contraseÃ±as**: Reglas para garantizar contraseÃ±as seguras.

Rendimiento y Escalabilidad

- **OptimizaciÃ³n de consultas**: Consultas SQL eficientes y uso de Ãndices.
- **CachÃ©**: ImplementaciÃ³n de estrategias de cachÃ© para mejorar el rendimiento.
- **PaginaciÃ³n**: Carga eficiente de grandes conjuntos de datos.
- **Arquitectura modular**: DiseÃ±o que facilita la escalabilidad horizontal y vertical.
- **Procesamiento asÃncrono**: Manejo de operaciones intensivas sin bloquear la interfaz de usuario.

IntegraciÃ³n y Extensibilidad

- **API RESTful**: Interfaz de programaciÃ³n para integraciÃ³n con otros sistemas.
- **ExportaciÃ³n de datos**: GeneraciÃ³n de archivos en formatos estÃ¡ndar (CSV, Excel, PDF).
- **ImportaciÃ³n de datos**: Capacidad para importar datos desde archivos externos.
- **Arquitectura extensible**: DiseÃ±o que permite aÃ±adir nuevas funcionalidades sin modificar el cÃ³digo existente.
- **Webhooks**: Notificaciones en tiempo real de eventos importantes.

Esta amplia gama de funcionalidades hace del Sistema POS una soluciÃ³n completa y versÃ¡til para la gestiÃ³n de tiendas de accesorios mÃ³viles, adaptable a diferentes tamaÃ±os de negocio y necesidades especÃficas.

Backend (NestJS)

El backend del Sistema POS estÃ¡ desarrollado con NestJS, un framework progresivo para Node.js que proporciona una arquitectura de aplicaciÃ³n elegante y robusta. Esta secciÃ³n detalla la implementaciÃ³n del backend, sus componentes principales y patrones de diseÃ±o utilizados.

Arquitectura del Backend

El backend sigue una arquitectura modular basada en los principios de NestJS, que a su vez se inspira en Angular. Esta arquitectura facilita la organizaciÃ³n del cÃ³digo, la inyecciÃ³n de dependencias y la separaciÃ³n de responsabilidades.

Componentes Principales

1. **MÃ³dulos**: Unidades organizativas que encapsulan un conjunto de funcionalidades relacionadas.
2. **Controladores**: Manejan las solicitudes HTTP y delegan el procesamiento a los servicios.
3. **Servicios**: Implementan la lÃ³gica de negocio y se comunican con los repositorios.
4. **Repositorios**: Abstraen el acceso a la base de datos y proporcionan mÃ©todos para operaciones CRUD.
5. **Entidades**: Representan las tablas de la base de datos y definen la estructura de los datos.
6. **DTOs (Data Transfer Objects)**: Definen la estructura de los datos que se transfieren entre el cliente y el servidor.
7. **Guards**: Protegen las rutas segÃºn criterios especÃficos, como la autenticaciÃ³n y autorizaciÃ³n.
8. **Interceptores**: Transforman las respuestas antes de enviarlas al cliente.
9. **Pipes**: Validan y transforman los datos de entrada.
10. **Filtros**: Manejan las excepciones de manera centralizada.

MÃ³dulos Implementados

El backend estÃ¡ organizado en mÃ³dulos funcionales que encapsulan caracterÃsticas especÃficas del sistema:

MÃ³dulo de AutenticaciÃ³n (auth)

Este mÃ³dulo maneja la autenticaciÃ³n de usuarios y la generaciÃ³n de tokens JWT.

```typescript
// auth.module.ts
@Module({
  imports: [
    PassportModule.register({ defaultStrategy: 'jwt' }),
    JwtModule.registerAsync({
      imports: [ConfigModule],
      useFactory: async (configService: ConfigService) => ({
        secret: configService.get<string>('JWT_SECRET'),
        signOptions: {
          expiresIn: configService.get<string>('JWT_EXPIRES_IN', '1d'),
        },
      }),
      inject: [ConfigService],
    }),
    UsuariosModule,
  ],
  controllers: [AuthController],
  providers: [AuthService, JwtStrategy],
  exports: [JwtStrategy, PassportModule],
})
export class AuthModule {}
```

CaracterÃsticas principales:
- AutenticaciÃ³n basada en JWT
- Estrategia de autenticaciÃ³n local para login
- GeneraciÃ³n y validaciÃ³n de tokens
- ProtecciÃ³n de rutas mediante guards

MÃ³dulo de Usuarios (usuarios)

Este mÃ³dulo gestiona los usuarios del sistema, incluyendo su creaciÃ³n, actualizaciÃ³n y eliminaciÃ³n.

```typescript
// usuarios.module.ts
@Module({
  imports: [
    TypeOrmModule.forFeature([Usuario]),
    RolesModule,
    PersonasModule,
  ],
  controllers: [UsuariosController],
  providers: [UsuariosService],
  exports: [UsuariosService],
})
export class UsuariosModule {}
```

CaracterÃsticas principales:
- CRUD completo para usuarios
- ValidaciÃ³n de datos de entrada

- EncriptaciÃ³n de contraseÃ±as
- Relaciones con roles y personas

MÃ³dulo de Roles (roles)

Este mÃ³dulo gestiona los roles y permisos del sistema.

```typescript
// roles.module.ts
@Module({
  imports: [
    TypeOrmModule.forFeature([Rol]),
  ],
  controllers: [RolesController],
  providers: [RolesService],
  exports: [RolesService],
})
export class RolesModule {}
```

CaracterÃsticas principales:
- DefiniciÃ³n de roles (administrador, vendedor)
- AsignaciÃ³n de permisos a roles
- VerificaciÃ³n de permisos

MÃ³dulo de Productos (productos)

Este mÃ³dulo gestiona el catÃ¡logo de productos del sistema.

```typescript
// productos.module.ts
@Module({
  imports: [
    TypeOrmModule.forFeature([Producto, Unidad, Impuesto]),
  ],
  controllers: [ProductosController],
  providers: [ProductosService],
  exports: [ProductosService],
})
export class ProductosModule {}
```

CaracterÃsticas principales:
- CRUD completo para productos
- GestiÃ³n de unidades de medida
- GestiÃ³n de impuestos
- ValidaciÃ³n de datos de productos

MÃ³dulo de Inventario (inventario)

Este mÃ³dulo gestiona el inventario de productos por tienda.

```typescript
// inventario.module.ts
@Module({
  imports: [
    TypeOrmModule.forFeature([Inventario]),
    ProductosModule,
    TiendasModule,
  ],
  controllers: [InventarioController],
  providers: [InventarioService],
  exports: [InventarioService],
})
export class InventarioModule {}
```

CaracterÃsticas principales:
- Control de stock por producto y tienda
- Alertas de bajo stock
- ActualizaciÃ³n automÃ¡tica de inventario

MÃ³dulo de Kardex (kardex)

Este mÃ³dulo registra los movimientos de inventario (entradas, salidas, ventas).

```typescript
// kardex.module.ts
@Module({
  imports: [
    TypeOrmModule.forFeature([Kardex]),
    ProductosModule,
    InventarioModule,
```

```
  ],
  controllers: [KardexController],
  providers: [KardexService],
  exports: [KardexService],
})
export class KardexModule {}
```

CaracterÃsticas principales:
- Registro de movimientos de inventario
- ClasificaciÃ³n por tipo de movimiento
- Trazabilidad de productos

MÃ³dulo de Ventas (ventas)

Este mÃ³dulo gestiona las ventas y sus detalles.

```typescript
// ventas.module.ts
@Module({
  imports: [
    TypeOrmModule.forFeature([Venta, DetalleVenta]),
    ProductosModule,
    InventarioModule,
    KardexModule,
    PersonasModule,
    MetodosPagoModule,
    TiendasModule,
  ],
  controllers: [VentasController],
  providers: [VentasService],
  exports: [VentasService],
})
export class VentasModule {}
```

CaracterÃsticas principales:
- Registro de ventas con mÃºltiples productos
- CÃ¡lculo automÃ¡tico de totales e impuestos
- IntegraciÃ³n con diferentes mÃ©todos de pago
- ActualizaciÃ³n automÃ¡tica de inventario y kardex

MÃ³dulo de Gastos (gastos)

Este mÃ³dulo gestiona los gastos del negocio.

```typescript
// gastos.module.ts
@Module({
  imports: [
    TypeOrmModule.forFeature([Gasto, DetalleGasto]),
    TiendasModule,
  ],
  controllers: [GastosController],
  providers: [GastosService],
  exports: [GastosService],
})
export class GastosModule {}
```

CaracterÃsticas principales:
- Registro de gastos con categorizaciÃ³n
- AsignaciÃ³n de gastos a tiendas
- CÃ¡lculo de totales

MÃ³dulo de Reportes (reportes)

Este mÃ³dulo genera reportes y estadÃsticas del sistema.

```typescript
// reportes.module.ts
@Module({
  imports: [
    VentasModule,
    GastosModule,
    ProductosModule,
    InventarioModule,
  ],
  controllers: [ReportesController],
  providers: [ReportesService],
})
export class ReportesModule {}
```

CaracterÃsticas principales:
- Reportes de ventas por perÃodo
- Reportes de gastos por perÃodo
- Balance de ingresos y gastos
- Reportes de inventario
- ExportaciÃ³n a PDF y CSV

AutenticaciÃ³n y AutorizaciÃ³n

El sistema implementa un robusto mecanismo de autenticaciÃ³n y autorizaciÃ³n basado en JWT (JSON Web Tokens).

Flujo de AutenticaciÃ³n

1. El usuario envÃa sus credenciales (nombre de usuario y contraseÃ±a) al endpoint `/auth/login`.
2. El servidor valida las credenciales contra la base de datos.
3. Si las credenciales son vÃ¡lidas, el servidor genera un token JWT que contiene el ID del usuario, su rol y otros datos relevantes.
4. El token se envÃa al cliente, que lo almacena (generalmente en localStorage).
5. Para las solicitudes posteriores, el cliente incluye el token en el encabezado `Authorization` con el formato `Bearer {token}`.
6. El servidor valida el token en cada solicitud protegida y extrae la informaciÃ³n del usuario.

```typescript
// auth.service.ts
@Injectable()
export class AuthService {
  constructor(
    private usuariosService: UsuariosService,
    private jwtService: JwtService,
  ) {}

  async validateUser(username: string, password: string): Promise<any> {
    const user = await this.usuariosService.findByUsername(username);
    if (user && await bcrypt.compare(password, user.password)) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }

  async login(user: any) {
    const payload = {
      sub: user.id,
      username: user.username,
      rol: user.rol.nombre
    };
    return {
      access_token: this.jwtService.sign(payload),
      user: {
        id: user.id,
        username: user.username,
        rol: user.rol.nombre,
      },
    };
  }
}
```

AutorizaciÃ³n Basada en Roles

El sistema implementa un control de acceso basado en roles (RBAC) mediante guards personalizados:

```typescript
// roles.guard.ts
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<string[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.some((role) => user.rol === role);
  }
}
```

Este guard se utiliza junto con un decorador personalizado para proteger rutas especÃficas:

```typescript
// roles.decorator.ts
export const ROLES_KEY = 'roles';
export const Roles = (...roles: string[]) => SetMetadata(ROLES_KEY, roles);
```

Ejemplo de uso en un controlador:

```typescript
// productos.controller.ts
@Controller('productos')
export class ProductosController {
  constructor(private productosService: ProductosService) {}

  @Get()
  findAll() {
    return this.productosService.findAll();
  }

  @Post()
  @Roles('administrador')
  @UseGuards(JwtAuthGuard, RolesGuard)
  create(@Body() createProductoDto: CreateProductoDto) {
    return this.productosService.create(createProductoDto);
  }
}
```

ValidaciÃ³n de Datos

El sistema utiliza class-validator y class-transformer para validar y transformar los datos de entrada:

```typescript
// create-producto.dto.ts
export class CreateProductoDto {
  @ApiProperty({ description: 'Nombre del producto' })
  @IsString()
  @IsNotEmpty()
  @MaxLength(100)
  nombre: string;

  @ApiProperty({ description: 'DescripciÃ³n del producto' })
  @IsString()
  @IsOptional()
  @MaxLength(500)
  descripcion?: string;

  @ApiProperty({ description: 'CÃ³digo del producto' })
  @IsString()
  @IsNotEmpty()
  @MaxLength(50)
  codigo: string;

  @ApiProperty({ description: 'Precio de venta del producto' })
  @IsNumber()
  @IsPositive()
  precio_venta: number;

  @ApiProperty({ description: 'Precio de compra del producto' })
  @IsNumber()
  @IsPositive()
  precio_compra: number;

  @ApiProperty({ description: 'ID del impuesto aplicable' })
  @IsNumber()
  @IsPositive()
  impuesto_id: number;

  @ApiProperty({ description: 'ID de la unidad de medida' })
  @IsNumber()
  @IsPositive()
  unidad_id: number;

  @ApiProperty({ description: 'URL de la imagen del producto' })
  @IsString()
  @IsOptional()
  imagen?: string;
}
```

Manejo de Errores

El sistema implementa un manejo centralizado de errores mediante filtros de excepciÃ³n:

```typescript
// http-exception.filter.ts
@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
    const status = exception.getStatus();
    const exceptionResponse = exception.getResponse();

    const errorResponse = {
      statusCode: status,
      timestamp: new Date().toISOString(),
      path: request.url,
      method: request.method,
      message: typeof exceptionResponse === 'object' && 'message' in exceptionResponse
        ? exceptionResponse['message']
        : exception.message,
    };

    response.status(status).json(errorResponse);
  }
}
```

DocumentaciÃ³n de API

El backend utiliza Swagger para documentar la API REST:

```typescript
// main.ts
async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // ConfiguraciÃ³n de Swagger
  const config = new DocumentBuilder()
    .setTitle('Sistema POS API')
    .setDescription('API para el Sistema POS de Tienda de Accesorios MÃ³viles')
    .setVersion('1.0')
    .addBearerAuth()
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api/docs', app, document);

  // Otras configuraciones...

  await app.listen(3001);
}
bootstrap();
```

Transacciones

El sistema utiliza transacciones para garantizar la integridad de los datos en operaciones crÃticas:

```typescript
// ventas.service.ts
@Injectable()
export class VentasService {
  constructor(
    @InjectRepository(Venta)
    private ventasRepository: Repository<Venta>,
    @InjectRepository(DetalleVenta)
    private detalleVentasRepository: Repository<DetalleVenta>,
    private inventarioService: InventarioService,
    private kardexService: KardexService,
    private connection: Connection,
  ) {}

  async create(createVentaDto: CreateVentaDto) {
    const queryRunner = this.connection.createQueryRunner();
    await queryRunner.connect();
    await queryRunner.startTransaction();

    try {
      // Crear la venta
      const venta = this.ventasRepository.create({
        fecha: new Date(),
        cliente_id: createVentaDto.cliente_id,
```

```
      vendedor_id: createVentaDto.vendedor_id,
      tienda_id: createVentaDto.tienda_id,
      metodo_pago_id: createVentaDto.metodo_pago_id,
      subtotal: 0,
      impuesto: 0,
      total: 0,
      estado: 'completada',
    });

    const ventaGuardada = await queryRunner.manager.save(venta);

    let subtotal = 0;
    let impuestoTotal = 0;

    // Procesar cada detalle de venta
    for (const detalle of createVentaDto.detalles) {
      // Verificar stock
      await this.inventarioService.verificarStock(
        detalle.producto_id,
        createVentaDto.tienda_id,
        detalle.cantidad,
      );

      // Obtener producto con su impuesto
      const producto = await this.productosService.findOne(detalle.producto_id);

      // Calcular valores
      const precioUnitario = detalle.precio || producto.precio_venta;
      const subtotalDetalle = precioUnitario * detalle.cantidad;
      const impuestoDetalle = subtotalDetalle * (producto.impuesto.porcentaje / 100);

      // Crear detalle de venta
      const detalleVenta = this.detalleVentasRepository.create({
        venta_id: ventaGuardada.id,
        producto_id: detalle.producto_id,
        cantidad: detalle.cantidad,
        precio_unitario: precioUnitario,
        subtotal: subtotalDetalle,
        impuesto: impuestoDetalle,
      });

      await queryRunner.manager.save(detalleVenta);

      // Actualizar inventario
      await this.inventarioService.reducirStock(
        queryRunner.manager,
        detalle.producto_id,
        createVentaDto.tienda_id,
        detalle.cantidad,
      );

      // Registrar movimiento en kardex
      await this.kardexService.registrarMovimiento(
        queryRunner.manager,
        {
          producto_id: detalle.producto_id,
          tipo_movimiento: TipoMovimiento.VENTA,
          cantidad: detalle.cantidad,
          referencia: `Venta #${ventaGuardada.id}`,
          venta_id: ventaGuardada.id,
        },
      );

      subtotal += subtotalDetalle;
      impuestoTotal += impuestoDetalle;
    }

    // Actualizar totales de la venta
    ventaGuardada.subtotal = subtotal;
    ventaGuardada.impuesto = impuestoTotal;
    ventaGuardada.total = subtotal + impuestoTotal;

    await queryRunner.manager.save(ventaGuardada);

    await queryRunner.commitTransaction();

    return ventaGuardada;
  } catch (error) {
    await queryRunner.rollbackTransaction();
    throw new Error(`Error al crear la venta: ${error.message}`);
  } finally {
    await queryRunner.release();
  }
```

```
  }
}
```

OptimizaciÃ³n de Rendimiento

El backend implementa varias estrategias para optimizar el rendimiento:

1. **CachÃ©**: Utiliza cachÃ© para operaciones frecuentes y costosas.

```typescript
// app.module.ts
@Module({
  imports: [
    CacheModule.register({
      ttl: 60, // segundos
      max: 100, // mÃ¡ximo nÃºmero de items en cachÃ©
    }),
    // otros mÃ³dulos...
  ],
})
export class AppModule {}
```

2. **Lazy Loading**: Carga diferida de mÃ³dulos para reducir el tiempo de inicio.

3. **Consultas Optimizadas**: Uso de consultas SQL optimizadas y joins eficientes.

```typescript
// productos.service.ts
@Injectable()
export class ProductosService {
  constructor(
    @InjectRepository(Producto)
    private productosRepository: Repository<Producto>,
  ) {}

  async findAll(): Promise<Producto[]> {
    return this.productosRepository.createQueryBuilder('producto')
      .leftJoinAndSelect('producto.impuesto', 'impuesto')
      .leftJoinAndSelect('producto.unidad', 'unidad')
      .select([
        'producto.id',
        'producto.nombre',
        'producto.codigo',
        'producto.precio_venta',
        'producto.precio_compra',
        'producto.imagen',
        'impuesto.id',
        'impuesto.nombre',
        'impuesto.porcentaje',
        'unidad.id',
        'unidad.nombre',
        'unidad.simbolo',
      ])
      .where('producto.activo = :activo', { activo: true })
      .getMany();
  }
}
```

4. **CompresiÃ³n**: CompresiÃ³n de respuestas HTTP para reducir el tamaÃ±o de los datos transferidos.

```typescript
// main.ts
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(compression());
  // otras configuraciones...
  await app.listen(3001);
}
bootstrap();
```

Seguridad

El backend implementa varias medidas de seguridad:

1. **Helmet**: ProtecciÃ³n contra vulnerabilidades web comunes mediante el establecimiento de cabeceras HTTP seguras.

```typescript
// main.ts
async function bootstrap() {
```

```
  const app = await NestFactory.create(AppModule);
  app.use(helmet());
  // otras configuraciones...
  await app.listen(3001);
}
bootstrap();
```

2. **Rate Limiting**: LimitaciÃ³n de tasa para prevenir ataques de fuerza bruta y DDoS.

```typescript
// main.ts
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(
    rateLimit({
      windowMs: 15 * 60 * 1000, // 15 minutos
      max: 100, // lÃmite de 100 solicitudes por ventana
    }),
  );
  // otras configuraciones...
  await app.listen(3001);
}
bootstrap();
```

3. **ValidaciÃ³n de Entrada**: ValidaciÃ³n estricta de todos los datos de entrada para prevenir inyecciones y otros ataques.

4. **CORS**: ConfiguraciÃ³n adecuada de CORS para controlar quÃ© dominios pueden acceder a la API.

```typescript
// main.ts
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors({
    origin: process.env.CORS_ORIGIN || 'http://localhost:3000',
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
    credentials: true,
  });
  // otras configuraciones...
  await app.listen(3001);
}
bootstrap();
```

5. **EncriptaciÃ³n de ContraseÃ±as**: Las contraseÃ±as se almacenan encriptadas utilizando bcrypt.

```typescript
// usuarios.service.ts
@Injectable()
export class UsuariosService {
  // ...

  async create(createUsuarioDto: CreateUsuarioDto): Promise<Usuario> {
    const { password, ...userData } = createUsuarioDto;

    // Verificar si el usuario ya existe
    const existingUser = await this.findByUsername(createUsuarioDto.username);
    if (existingUser) {
      throw new ConflictException('El nombre de usuario ya estÃ¡ en uso');
    }

    // Encriptar la contraseÃ±a
    const hashedPassword = await bcrypt.hash(password, 10);

    // Crear y guardar el nuevo usuario
    const newUser = this.usuariosRepository.create({
      ...userData,
      password: hashedPassword,
    });

    return this.usuariosRepository.save(newUser);
  }
}
```

Logging

El sistema implementa un sistema de logging robusto utilizando Winston:

```typescript
// logger.middleware.ts
@Injectable()
```

```
export class LoggerMiddleware implements NestMiddleware {
  private logger = new Logger('HTTP');

  use(req: Request, res: Response, next: Function) {
    const { ip, method, originalUrl } = req;
    const userAgent = req.get('user-agent') || '';

    res.on('finish', () => {
      const { statusCode } = res;
      const contentLength = res.get('content-length');

      this.logger.log(
        `${method} ${originalUrl} ${statusCode} ${contentLength} - ${userAgent} ${ip}`,
      );
    });

    next();
  }
}
```

ConclusiÃ³n

El backend del Sistema POS proporciona una base sÃ³lida y robusta para la aplicaciÃ³n, implementando las mejores prÃ¡cticas de desarrollo con NestJS. La arquitectura modular, la separaciÃ³n de responsabilidades y los patrones de diseÃ±o utilizados facilitan el mantenimiento y la extensiÃ³n del sistema.

Funcionalidades del Sistema

El Sistema POS para Tienda de Accesorios MÃ³viles ofrece un conjunto completo de funcionalidades diseÃ±adas para satisfacer las necesidades de gestiÃ³n de una tienda de accesorios mÃ³viles. A continuaciÃ³n, se detallan las principales funcionalidades organizadas por mÃ³dulos.

1. AutenticaciÃ³n y Control de Acceso

1.1. AutenticaciÃ³n de Usuarios

- **Inicio de sesiÃ³n**: AutenticaciÃ³n mediante credenciales (usuario y contraseÃ±a)
- **GeneraciÃ³n de tokens JWT**: Para mantener la sesiÃ³n del usuario
- **RenovaciÃ³n automÃ¡tica de tokens**: Para mantener la sesiÃ³n activa
- **Cierre de sesiÃ³n**: InvalidaciÃ³n de tokens

1.2. Control de Acceso Basado en Roles

- **Roles predefinidos**: Administrador y Vendedor
- **Permisos granulares**: Acceso especÃfico a funcionalidades segÃºn el rol
- **ProtecciÃ³n de rutas**: VerificaciÃ³n de permisos en cada endpoint
- **Interfaz adaptativa**: Elementos de UI visibles segÃºn permisos del usuario

2. GestiÃ³n de Usuarios y Roles

2.1. GestiÃ³n de Usuarios

- **CreaciÃ³n de usuarios**: Registro de nuevos usuarios con datos bÃ¡sicos
- **AsignaciÃ³n de roles**: DefiniciÃ³n del nivel de acceso
- **EdiciÃ³n de usuarios**: ModificaciÃ³n de datos y permisos
- **ActivaciÃ³n/desactivaciÃ³n**: Control del estado de los usuarios
- **Restablecimiento de contraseÃ±as**: Mecanismo seguro para recuperaciÃ³n

2.2. GestiÃ³n de Roles

- **DefiniciÃ³n de roles**: CreaciÃ³n y configuraciÃ³n de roles
- **AsignaciÃ³n de permisos**: ConfiguraciÃ³n de accesos por rol
- **EdiciÃ³n de roles**: ModificaciÃ³n de permisos y descripciÃ³n
- **VisualizaciÃ³n de usuarios por rol**: Listado filtrado

3. GestiÃ³n de Productos e Inventario

3.1. CatÃ¡logo de Productos

- **Registro de productos**: CreaciÃ³n con datos completos (nombre, descripciÃ³n, cÃ³digos, precios)
- **CategorizaciÃ³n**: OrganizaciÃ³n por categorÃas y subcategorÃas
- **GestiÃ³n de imÃ¡genes**: Carga y visualizaciÃ³n de imÃ¡genes de productos
- **BÃºsqueda avanzada**: Filtros por mÃºltiples criterios
- **ExportaciÃ³n de catÃ¡logo**: GeneraciÃ³n de reportes en PDF/CSV

3.2. Control de Inventario

- **Registro de stock**: Seguimiento de existencias por producto y tienda
- **Alertas de stock mÃnimo**: Notificaciones automÃ¡ticas
- **Ajustes de inventario**: Correcciones y actualizaciones manuales
- **VisualizaciÃ³n de estado**: Dashboard con indicadores de inventario

- **Historial de cambios**: Registro de todas las modificaciones

3.3. Sistema Kardex

- **Registro de movimientos**: Entradas, salidas y ventas
- **Trazabilidad completa**: Seguimiento del origen y destino de cada movimiento
- **CÃ¡lculo automÃ¡tico de saldos**: ActualizaciÃ³n en tiempo real
- **Reportes detallados**: Movimientos por producto, fecha o tipo
- **ExportaciÃ³n de datos**: GeneraciÃ³n de informes en PDF/CSV

4. GestiÃ³n de Ventas

4.1. Proceso de Venta

- **Interfaz de punto de venta**: DiseÃ±o intuitivo para operaciones rÃ¡pidas
- **BÃºsqueda de productos**: Por cÃ³digo, nombre o escaneo
- **SelecciÃ³n de cantidades**: Ajuste de unidades a vender
- **AplicaciÃ³n de descuentos**: A nivel de producto o venta total
- **CÃ¡lculo automÃ¡tico**: Subtotales, impuestos y total
- **SelecciÃ³n de cliente**: AsociaciÃ³n de venta a cliente registrado o genÃ©rico
- **MÃºltiples mÃ©todos de pago**: Efectivo, tarjeta, transferencia

4.2. GestiÃ³n de Ventas Realizadas

- **Historial de ventas**: Registro completo con filtros
- **Detalles de venta**: VisualizaciÃ³n de productos, cantidades y precios
- **AnulaciÃ³n de ventas**: Con registro de motivo y usuario
- **ReimpresiÃ³n de comprobantes**: GeneraciÃ³n de duplicados
- **EstadÃsticas de ventas**: Por perÃodo, vendedor, producto o cliente

5. GestiÃ³n de Gastos

5.1. Registro de Gastos

- **CategorizaciÃ³n de gastos**: OrganizaciÃ³n por tipos
- **Registro detallado**: DescripciÃ³n, monto, fecha y responsable
- **Adjuntos digitales**: Posibilidad de asociar comprobantes
- **AprobaciÃ³n de gastos**: Flujo configurable segÃºn monto

5.2. Reportes de Gastos

- **Gastos por perÃodo**: Filtrado por rangos de fechas
- **Gastos por categorÃa**: DistribuciÃ³n por tipos
- **Gastos por tienda**: Comparativa entre sucursales
- **ExportaciÃ³n de reportes**: GeneraciÃ³n en PDF/CSV

6. Reportes y EstadÃsticas

6.1. Reportes Financieros

- **Balance mensual**: Ingresos vs gastos
- **Flujo de caja**: Movimientos de efectivo
- **Rentabilidad**: AnÃ¡lisis de mÃ¡rgenes por producto
- **Proyecciones**: Estimaciones basadas en histÃ³ricos

6.2. Reportes de Ventas

- **Ventas por perÃodo**: Diarias, semanales, mensuales, anuales
- **Ventas por producto**: Ranking de productos mÃ¡s vendidos
- **Ventas por vendedor**: DesempeÃ±o del equipo
- **Ventas por cliente**: FidelizaciÃ³n y recurrencia
- **Ventas por mÃ©todo de pago**: DistribuciÃ³n de formas de pago

6.3. Reportes de Inventario

- **Stock actual**: Estado del inventario en tiempo real
- **RotaciÃ³n de productos**: Velocidad de venta
- **Productos sin movimiento**: IdentificaciÃ³n de inventario estancado
- **ValorizaciÃ³n de inventario**: Costo total del stock

7. GestiÃ³n de CompaÃ±Ãas y Tiendas

7.1. GestiÃ³n de CompaÃ±Ãas

- **Registro de compaÃ±Ãas**: Datos fiscales y de contacto
- **ConfiguraciÃ³n de parÃ¡metros**: Ajustes especÃficos por compaÃ±Ãa
- **GestiÃ³n de logos e identidad**: PersonalizaciÃ³n visual

7.2. GestiÃ³n de Tiendas

- **MÃºltiples sucursales**: AdministraciÃ³n centralizada
- **ConfiguraciÃ³n por tienda**: ParÃ¡metros especÃficos
- **AsignaciÃ³n de empleados**: Personal por sucursal

- **Inventarios independientes**: Control separado por tienda

8. GestiÃ³n de Clientes y Proveedores

8.1. GestiÃ³n de Clientes

- **Registro de clientes**: Datos personales y de contacto
- **Historial de compras**: Seguimiento de transacciones
- **SegmentaciÃ³n**: CategorizaciÃ³n por frecuencia o volumen
- **BÃºsqueda avanzada**: Filtros por mÃºltiples criterios

8.2. GestiÃ³n de Proveedores

- **Registro de proveedores**: Datos comerciales y de contacto
- **CatÃ¡logo de productos por proveedor**: AsociaciÃ³n producto-proveedor
- **Historial de compras**: Seguimiento de adquisiciones
- **EvaluaciÃ³n de proveedores**: CalificaciÃ³n por cumplimiento

9. ConfiguraciÃ³n del Sistema

9.1. ParÃ¡metros Generales

- **Datos de la empresa**: InformaciÃ³n fiscal y de contacto
- **ConfiguraciÃ³n de impuestos**: Tasas aplicables
- **Unidades de medida**: DefiniciÃ³n de unidades para productos
- **MÃ©todos de pago**: ConfiguraciÃ³n de formas de pago aceptadas

9.2. PersonalizaciÃ³n

- **Temas visuales**: Modo claro/oscuro
- **DiseÃ±o de comprobantes**: PersonalizaciÃ³n de tickets y facturas
- **Accesos directos**: ConfiguraciÃ³n de atajos de teclado
- **Dashboard personalizable**: Widgets configurables

10. Seguridad y AuditorÃa

10.1. Seguridad

- **EncriptaciÃ³n de datos sensibles**: ProtecciÃ³n de informaciÃ³n crÃtica
- **ValidaciÃ³n de entradas**: PrevenciÃ³n de inyecciones y ataques
- **Control de sesiones**: GestiÃ³n de tiempo de inactividad
- **PolÃticas de contraseÃ±as**: Requisitos de complejidad y caducidad

10.2. AuditorÃa

- **Registro de actividades**: Log de acciones de usuarios
- **Trazabilidad de cambios**: Historial de modificaciones
- **Alertas de seguridad**: NotificaciÃ³n de actividades sospechosas
- **Reportes de auditorÃa**: GeneraciÃ³n de informes de actividad

11. IntegraciÃ³n y ExportaciÃ³n

11.1. IntegraciÃ³n con Sistemas Externos

- **API RESTful**: Endpoints documentados para integraciÃ³n
- **Webhooks**: Notificaciones en tiempo real de eventos
- **ImportaciÃ³n masiva**: Carga de datos desde archivos CSV/Excel

11.2. ExportaciÃ³n de Datos

- **Formatos mÃºltiples**: PDF, CSV, Excel
- **PersonalizaciÃ³n de reportes**: ConfiguraciÃ³n de campos y filtros
- **ProgramaciÃ³n de exportaciones**: GeneraciÃ³n automÃ¡tica periÃ³dica

12. AplicaciÃ³n de Escritorio

12.1. CaracterÃsticas de la AplicaciÃ³n Electron

- **Interfaz nativa**: Experiencia de usuario de aplicaciÃ³n de escritorio
- **Funcionamiento offline**: OperaciÃ³n sin conexiÃ³n con sincronizaciÃ³n posterior
- **ImpresiÃ³n directa**: ConexiÃ³n con impresoras locales
- **Actualizaciones automÃ¡ticas**: DistribuciÃ³n de nuevas versiones

12.2. OptimizaciÃ³n para Punto de Venta

- **Modo pantalla completa**: MaximizaciÃ³n del Ã¡rea de trabajo
- **Soporte para perifÃ©ricos**: Lectores de cÃ³digos, impresoras tÃ©rmicas, cajones
- **Atajos de teclado**: OperaciÃ³n rÃ¡pida sin ratÃ³n
- **Modo de venta rÃ¡pida**: Interfaz simplificada para operaciones frecuentes

Esta amplia gama de funcionalidades hace del Sistema POS una soluciÃ³n completa para la gestiÃ³n integral de tiendas de accesorios mÃ³viles, cubriendo todos los aspectos operativos del negocio desde la administraciÃ³n de inventario hasta la generaciÃ³n de reportes financieros.

InstalaciÃ³n y ConfiguraciÃ³n

Esta secciÃ³n proporciona instrucciones detalladas para instalar, configurar y desplegar el Sistema POS para Tienda de Accesorios MÃ³viles en diferentes entornos.

Requisitos Previos

Requisitos de Hardware

- **Servidor Backend**:
  - CPU: 2 nÃºcleos o mÃ¡s
  - RAM: 4 GB mÃnimo (8 GB recomendado)
  - Almacenamiento: 20 GB mÃnimo (SSD recomendado)
  - ConexiÃ³n a Internet: 10 Mbps mÃnimo

- **Estaciones de Trabajo (Frontend)**:
  - CPU: 2 nÃºcleos o mÃ¡s
  - RAM: 4 GB mÃnimo
  - Almacenamiento: 10 GB mÃnimo
  - Pantalla: ResoluciÃ³n mÃnima de 1366x768

Requisitos de Software

- **Sistema Operativo**:
  - Servidor: Ubuntu 20.04 LTS o superior, Windows Server 2019 o superior
  - Estaciones de Trabajo: Windows 10/11, macOS 11 o superior, Ubuntu 20.04 o superior

- **Software Base**:
  - Node.js 16.x o superior
  - npm 8.x o superior
  - PostgreSQL 14.x o superior
  - Git (para instalaciÃ³n desde repositorio)

- **Navegadores Compatibles** (para versiÃ³n web):
  - Google Chrome 90 o superior
  - Mozilla Firefox 88 o superior
  - Microsoft Edge 90 o superior
  - Safari 14 o superior

InstalaciÃ³n del Backend

InstalaciÃ³n desde Repositorio

1. **Clonar el repositorio**:
   ```bash
   git clone https://github.com/empresa/pos-system.git
   cd pos-system/backend/pos-backend
   ```

2. **Instalar dependencias**:
   ```bash
   npm install
   ```

3. **Configurar variables de entorno**:
   ```bash
   cp .env.example .env
   ```
   Editar el archivo `.env` con los valores apropiados para el entorno:
   ```
   # ConfiguraciÃ³n de la base de datos
   DATABASE_HOST=localhost
   DATABASE_PORT=5432
   DATABASE_NAME=pos_db
   DATABASE_USER=posadmin
   DATABASE_PASSWORD=posadmin123

   # ConfiguraciÃ³n de JWT
   JWT_SECRET=your_jwt_secret_key
   JWT_EXPIRATION=1d

   # ConfiguraciÃ³n del servidor
   PORT=3001
   NODE_ENV=production
   ```

4. **Configurar la base de datos**:
   ```bash
   # Crear la base de datos
   sudo -u postgres psql -c "CREATE USER posadmin WITH PASSWORD 'posadmin123';"
   sudo -u postgres psql -c "CREATE DATABASE pos_db;"
```

```
   sudo -u postgres psql -c "GRANT ALL PRIVILEGES ON DATABASE pos_db TO posadmin;"

   # Importar el esquema inicial
   sudo -u postgres psql -d pos_db -f /path/to/esquema_pos.sql
   ```

5. **Compilar el proyecto**:
   ```bash
   npm run build
   ```

6. **Iniciar el servidor**:
   ```bash
   npm run start:prod
   ```

InstalaciÃ³n con Docker

1. **Crear archivo docker-compose.yml**:
   ```yaml
   version: '3.8'

   services:
     postgres:
       image: postgres:14
       container_name: pos-postgres
       environment:
         POSTGRES_USER: posadmin
         POSTGRES_PASSWORD: posadmin123
         POSTGRES_DB: pos_db
       volumes:
         - postgres_data:/var/lib/postgresql/data
         - ./esquema_pos.sql:/docker-entrypoint-initdb.d/esquema_pos.sql
       ports:
         - "5432:5432"
       networks:
         - pos-network

     backend:
       build:
         context: ./backend/pos-backend
         dockerfile: Dockerfile
       container_name: pos-backend
       environment:
         DATABASE_HOST: postgres
         DATABASE_PORT: 5432
         DATABASE_NAME: pos_db
         DATABASE_USER: posadmin
         DATABASE_PASSWORD: posadmin123
         JWT_SECRET: your_jwt_secret_key
         JWT_EXPIRATION: 1d
         PORT: 3001
         NODE_ENV: production
       ports:
         - "3001:3001"
       depends_on:
         - postgres
       networks:
         - pos-network

   networks:
     pos-network:
       driver: bridge

   volumes:
     postgres_data:
   ```

2. **Crear Dockerfile para el backend**:
   ```dockerfile
   FROM node:16-alpine

   WORKDIR /app

   COPY package*.json ./

   RUN npm install

   COPY . .

   RUN npm run build

   EXPOSE 3001
```

   CMD ["npm", "run", "start:prod"]
   ```

3. **Iniciar los contenedores**:
   ```bash
   docker-compose up -d
   ```

InstalaciÃ³n del Frontend

InstalaciÃ³n de la AplicaciÃ³n de Escritorio

1. **Clonar el repositorio** (si no se ha hecho ya):
   ```bash
   git clone https://github.com/empresa/pos-system.git
   cd pos-system/frontend/pos-frontend
   ```

2. **Instalar dependencias**:
   ```bash
   npm install
   ```

3. **Configurar variables de entorno**:
   ```bash
   cp .env.example .env
   ```
   Editar el archivo `.env` con los valores apropiados:
   ```
   REACT_APP_API_URL=http://localhost:3001/api
   REACT_APP_ENV=production
   ```

4. **Compilar la aplicaciÃ³n para producciÃ³n**:
   ```bash
   npm run electron:build
   ```

5. **Distribuir la aplicaciÃ³n**:
   Los archivos de instalaciÃ³n se generarÃ¡n en la carpeta `dist` y estarÃ¡n listos para ser distribuidos a los usuarios finales.

InstalaciÃ³n de la VersiÃ³n Web

1. **Clonar el repositorio** (si no se ha hecho ya):
   ```bash
   git clone https://github.com/empresa/pos-system.git
   cd pos-system/frontend/pos-frontend
   ```

2. **Instalar dependencias**:
   ```bash
   npm install
   ```

3. **Configurar variables de entorno**:
   ```bash
   cp .env.example .env
   ```
   Editar el archivo `.env` con los valores apropiados:
   ```
   REACT_APP_API_URL=http://localhost:3001/api
   REACT_APP_ENV=production
   ```

4. **Compilar la aplicaciÃ³n para producciÃ³n**:
   ```bash
   npm run build
   ```

5. **Desplegar en un servidor web**:
   Los archivos estÃ¡ticos generados en la carpeta `build` pueden ser servidos por cualquier servidor web como Nginx o Apache.

   Ejemplo de configuraciÃ³n para Nginx:
   ```nginx
   server {
       listen 80;
       server_name pos.example.com;

       root /path/to/pos-frontend/build;
       index index.html;

```
        location / {
            try_files $uri $uri/ /index.html;
        }

        location /api {
            proxy_pass http://localhost:3001;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
        }
    }
    ```

ConfiguraciÃ³n del Sistema

ConfiguraciÃ³n de la Base de Datos

1. **OptimizaciÃ³n de PostgreSQL**:
   Editar el archivo `postgresql.conf` para optimizar el rendimiento:
   ```
   # Memoria
   shared_buffers = 1GB                  # 25% de la RAM disponible
   work_mem = 32MB                       # Para operaciones complejas
   maintenance_work_mem = 256MB          # Para mantenimiento

   # Escritura en disco
   wal_buffers = 16MB                    # Buffer para logs de transacciones
   checkpoint_completion_target = 0.9    # Distribuir escrituras de checkpoint

   # Planificador de consultas
   random_page_cost = 1.1                # Para SSD
   effective_cache_size = 3GB            # 75% de la RAM disponible

   # Paralelismo
   max_worker_processes = 8              # NÃºmero de nÃºcleos disponibles
   max_parallel_workers_per_gather = 4   # Mitad de max_worker_processes
   max_parallel_workers = 8              # Igual a max_worker_processes
   ```

2. **Ãndices recomendados**:
   ```sql
   -- Ãndices para bÃºsquedas frecuentes
   CREATE INDEX idx_productos_nombre ON productos(nombre);
   CREATE INDEX idx_productos_codigo ON productos(codigo);
   CREATE INDEX idx_ventas_fecha ON ventas(fecha);
   CREATE INDEX idx_kardex_fecha ON kardex(fecha);
   CREATE INDEX idx_kardex_producto_id ON kardex(producto_id);
   CREATE INDEX idx_inventario_producto_tienda ON inventario(producto_id, tienda_id);

   -- Ãndices para relaciones
   CREATE INDEX idx_usuarios_rol_id ON usuarios(rol_id);
   CREATE INDEX idx_tiendas_compania_id ON tiendas(compania_id);
   CREATE INDEX idx_ventas_cliente_id ON ventas(cliente_id);
   CREATE INDEX idx_ventas_vendedor_id ON ventas(vendedor_id);
   ```

ConfiguraciÃ³n de Seguridad

1. **ConfiguraciÃ³n de CORS**:
   Editar el archivo `src/config/cors.config.ts` para definir los orÃgenes permitidos:
   ```typescript
   export const corsOptions = {
     origin: [
       'http://localhost:3000',
       'https://pos.example.com'
     ],
     methods: ['GET', 'POST', 'PUT', 'DELETE', 'PATCH'],
     allowedHeaders: ['Content-Type', 'Authorization'],
     credentials: true,
     maxAge: 86400
   };
   ```

2. **ConfiguraciÃ³n de JWT**:
   Editar el archivo `src/config/jwt.config.ts` para ajustar la configuraciÃ³n de tokens:
   ```typescript
   export const jwtConfig = {
     secret: process.env.JWT_SECRET || 'your_jwt_secret_key',
     expiresIn: process.env.JWT_EXPIRATION || '1d',
     refreshExpiresIn: '7d',
```

```
    issuer: 'pos-system',
    audience: 'pos-users'
  };
  ```

3. **ConfiguraciÃ³n de Helmet**:
  Editar el archivo `src/main.ts` para ajustar las cabeceras de seguridad:
  ```typescript
  app.use(helmet({
    contentSecurityPolicy: {
      directives: {
        defaultSrc: ["'self'"],
        scriptSrc: ["'self'", "'unsafe-inline'", "'unsafe-eval'"],
        styleSrc: ["'self'", "'unsafe-inline'", 'https://fonts.googleapis.com'],
        imgSrc: ["'self'", 'data:', 'https://storage.example.com'],
        connectSrc: ["'self'", 'https://api.example.com'],
        fontSrc: ["'self'", 'https://fonts.gstatic.com'],
        objectSrc: ["'none'"],
        mediaSrc: ["'self'"],
        frameSrc: ["'none'"],
      },
    },
    xssFilter: true,
    noSniff: true,
    referrerPolicy: { policy: 'same-origin' }
  }));
  ```

ConfiguraciÃ³n para ProducciÃ³n

1. **ConfiguraciÃ³n de PM2** (para gestiÃ³n de procesos Node.js):
  Crear archivo `ecosystem.config.js`:
  ```javascript
  module.exports = {
    apps: [{
      name: 'pos-backend',
      script: 'dist/main.js',
      instances: 'max',
      exec_mode: 'cluster',
      autorestart: true,
      watch: false,
      max_memory_restart: '1G',
      env: {
        NODE_ENV: 'production',
        PORT: 3001
      }
    }]
  };
  ```

2. **Iniciar con PM2**:
  ```bash
  pm2 start ecosystem.config.js
  ```

3. **Configurar inicio automÃ¡tico**:
  ```bash
  pm2 startup
  pm2 save
  ```

ActualizaciÃ³n del Sistema

ActualizaciÃ³n del Backend

1. **Respaldo de datos**:
  ```bash
  pg_dump -U posadmin -d pos_db > backup_$(date +%Y%m%d).sql
  ```

2. **Actualizar cÃ³digo**:
  ```bash
  cd /path/to/pos-system
  git pull origin main
  cd backend/pos-backend
  npm install
  npm run build
  ```

3. **Reiniciar servicios**:
  ```bash
  pm2 restart pos-backend
  ```
```
```

ActualizaciÃ³n del Frontend

1. **Actualizar cÃ³digo**:
   ```bash
   cd /path/to/pos-system
   git pull origin main
   cd frontend/pos-frontend
   npm install
   ```

2. **Compilar nueva versiÃ³n**:
   ```bash
   # Para aplicaciÃ³n web
   npm run build

   # Para aplicaciÃ³n de escritorio
   npm run electron:build
   ```

3. **Distribuir nueva versiÃ³n**:
   - Para la versiÃ³n web, actualizar los archivos en el servidor web.
   - Para la aplicaciÃ³n de escritorio, distribuir los nuevos instaladores a los usuarios.

SoluciÃ³n de Problemas Comunes

Problemas de ConexiÃ³n a la Base de Datos

1. **Verificar estado del servicio PostgreSQL**:
   ```bash
   sudo systemctl status postgresql
   ```

2. **Verificar configuraciÃ³n de conexiÃ³n**:
   ```bash
   cat /etc/postgresql/14/main/pg_hba.conf
   ```

3. **Verificar logs de PostgreSQL**:
   ```bash
   sudo tail -n 100 /var/log/postgresql/postgresql-14-main.log
   ```

Problemas de AutenticaciÃ³n

1. **Verificar configuraciÃ³n de JWT**:
   ```bash
   cat /path/to/pos-system/backend/pos-backend/.env | grep JWT
   ```

2. **Regenerar secreto JWT**:
   ```bash
   node -e "console.log(require('crypto').randomBytes(32).toString('hex'))"
   ```
   Actualizar el valor en el archivo `.env`.

Problemas de Rendimiento

1. **Verificar uso de recursos**:
   ```bash
   top
   htop
   ```

2. **Verificar logs de la aplicaciÃ³n**:
   ```bash
   pm2 logs pos-backend
   ```

3. **Analizar consultas lentas en PostgreSQL**:
   ```bash
   sudo -u postgres psql -d pos_db -c "SELECT * FROM pg_stat_activity WHERE state = 'active';"
   ```

Esta guÃa de instalaciÃ³n y configuraciÃ³n proporciona los pasos necesarios para desplegar el Sistema POS en diferentes entornos, desde desarrollo hasta producciÃ³n, y ofrece soluciones para los problemas mÃ¡s comunes que pueden surgir durante la operaciÃ³n del sistema.

Frontend (ReactJS + Electron)

El frontend del Sistema POS estÃ¡ desarrollado con ReactJS y Electron, proporcionando una interfaz de usuario moderna, responsiva y fÃ¡cil de usar. Esta secciÃ³n detalla la implementaciÃ³n del frontend, sus componentes principales y patrones de

diseÃ±o utilizados.

Arquitectura del Frontend

El frontend sigue una arquitectura basada en componentes, utilizando React como biblioteca principal para la construcciÃ³n de la interfaz de usuario. La aplicaciÃ³n se integra con Electron para proporcionar una experiencia de escritorio nativa.

Componentes Principales

1. **Componentes**: Unidades reutilizables que encapsulan la interfaz de usuario y su comportamiento.
2. **PÃ¡ginas**: Componentes de nivel superior que representan las diferentes vistas de la aplicaciÃ³n.
3. **Layouts**: Componentes que definen la estructura general de las pÃ¡ginas.
4. **Contextos**: Proporcionan estado global a la aplicaciÃ³n y facilitan la comunicaciÃ³n entre componentes.
5. **Hooks**: Encapsulan la lÃ³gica reutilizable y el estado.
6. **Servicios**: Manejan la comunicaciÃ³n con el backend y otras operaciones asÃncronas.
7. **Utilidades**: Funciones auxiliares para tareas comunes.

Estructura de Componentes

El frontend estÃ¡ organizado en componentes reutilizables que siguen el principio de responsabilidad Ãºnica:

Componentes de Interfaz de Usuario

```jsx
// src/components/FormField.tsx
import React from 'react';
import { TextField, FormControl, FormHelperText, InputLabel, MenuItem, Select } from '@mui/material';

interface FormFieldProps {
  id: string;
  label: string;
  value: any;
  onChange: (e: React.ChangeEvent<HTMLInputElement | { name?: string; value: unknown }>) => void;
  error?: string;
  type?: string;
  required?: boolean;
  options?: { value: string | number; label: string }[];
  fullWidth?: boolean;
  disabled?: boolean;
}

const FormField: React.FC<FormFieldProps> = ({
  id,
  label,
  value,
  onChange,
  error,
  type = 'text',
  required = false,
  options,
  fullWidth = true,
  disabled = false,
}) => {
  if (type === 'select' && options) {
    return (
      <FormControl fullWidth={fullWidth} error={!!error} required={required} disabled={disabled}>
        <InputLabel id={`${id}-label`}>{label}</InputLabel>
        <Select
          labelId={`${id}-label`}
          id={id}
          value={value}
          label={label}
          onChange={onChange}
        >
          {options.map((option) => (
            <MenuItem key={option.value} value={option.value}>
              {option.label}
            </MenuItem>
          ))}
        </Select>
        {error && <FormHelperText>{error}</FormHelperText>}
      </FormControl>
    );
  }

  return (
    <TextField
      id={id}
      label={label}
      value={value}
      onChange={onChange}
      error={!!error}
      helperText={error}
```

```
      type={type}
      required={required}
      fullWidth={fullWidth}
      disabled={disabled}
      variant="outlined"
      margin="normal"
    />
  );
};

export default FormField;
```

Componentes de Layout

```jsx
// src/layouts/MainLayout.tsx
import React, { useState, useEffect } from 'react';
import { styled, useTheme } from '@mui/material/styles';
import {
  Box,
  Drawer,
  AppBar,
  Toolbar,
  List,
  Typography,
  Divider,
  IconButton,
  ListItem,
  ListItemIcon,
  ListItemText,
  CssBaseline,
  useMediaQuery,
} from '@mui/material';
import {
  Menu as MenuIcon,
  ChevronLeft as ChevronLeftIcon,
  ChevronRight as ChevronRightIcon,
  Dashboard as DashboardIcon,
  ShoppingCart as ShoppingCartIcon,
  Inventory as InventoryIcon,
  Receipt as ReceiptIcon,
  BarChart as BarChartIcon,
  Settings as SettingsIcon,
  ExitToApp as ExitToAppIcon,
} from '@mui/icons-material';
import { useNavigate, useLocation } from 'react-router-dom';
import { useAuth } from '../contexts/AuthContext';
import ThemeToggle from '../components/ThemeToggle';

const drawerWidth = 240;

const Main = styled('main', { shouldForwardProp: (prop) => prop !== 'open' })<{
  open?: boolean;
}>(({ theme, open }) => ({
  flexGrow: 1,
  padding: theme.spacing(3),
  transition: theme.transitions.create('margin', {
    easing: theme.transitions.easing.sharp,
    duration: theme.transitions.duration.leavingScreen,
  }),
  marginLeft: `-${drawerWidth}px`,
  ...(open && {
    transition: theme.transitions.create('margin', {
      easing: theme.transitions.easing.easeOut,
      duration: theme.transitions.duration.enteringScreen,
    }),
    marginLeft: 0,
  }),
}));

const AppBarStyled = styled(AppBar, {
  shouldForwardProp: (prop) => prop !== 'open',
})<{
  open?: boolean;
}>(({ theme, open }) => ({
  transition: theme.transitions.create(['margin', 'width'], {
    easing: theme.transitions.easing.sharp,
    duration: theme.transitions.duration.leavingScreen,
  }),
  ...(open && {
    width: `calc(100% - ${drawerWidth}px)`,
    marginLeft: `${drawerWidth}px`,
```

```
    transition: theme.transitions.create(['margin', 'width'], {
      easing: theme.transitions.easing.easeOut,
      duration: theme.transitions.duration.enteringScreen,
    }),
  }),
}));

const DrawerHeader = styled('div')(({ theme }) => ({
  display: 'flex',
  alignItems: 'center',
  padding: theme.spacing(0, 1),
  ...theme.mixins.toolbar,
  justifyContent: 'flex-end',
}));

interface MainLayoutProps {
  children: React.ReactNode;
}

const MainLayout: React.FC<MainLayoutProps> = ({ children }) => {
  const theme = useTheme();
  const navigate = useNavigate();
  const location = useLocation();
  const { user, logout } = useAuth();
  const [open, setOpen] = useState(true);
  const isMobile = useMediaQuery(theme.breakpoints.down('md'));

  useEffect(() => {
    if (isMobile) {
      setOpen(false);
    }
  }, [isMobile]);

  const handleDrawerOpen = () => {
    setOpen(true);
  };

  const handleDrawerClose = () => {
    setOpen(false);
  };

  const menuItems = [
    { text: 'Dashboard', icon: <DashboardIcon />, path: '/dashboard' },
    { text: 'Ventas', icon: <ShoppingCartIcon />, path: '/ventas' },
    { text: 'Productos', icon: <InventoryIcon />, path: '/productos' },
    { text: 'Gastos', icon: <ReceiptIcon />, path: '/gastos' },
    { text: 'Reportes', icon: <BarChartIcon />, path: '/reportes' },
  ];

  // Solo mostrar administraciÃ³n si el usuario es administrador
  if (user?.rol === 'administrador') {
    menuItems.push({ text: 'AdministraciÃ³n', icon: <SettingsIcon />, path: '/administracion' });
  }

  return (
    <Box sx={{ display: 'flex' }}>
      <CssBaseline />
      <AppBarStyled position="fixed" open={open}>
        <Toolbar>
          <IconButton
            color="inherit"
            aria-label="open drawer"
            onClick={handleDrawerOpen}
            edge="start"
            sx={{ mr: 2, ...(open && { display: 'none' }) }}
          >
            <MenuIcon />
          </IconButton>
          <Typography variant="h6" noWrap component="div" sx={{ flexGrow: 1 }}>
            Sistema POS - Tienda de Accesorios MÃ³viles
          </Typography>
          <ThemeToggle />
        </Toolbar>
      </AppBarStyled>
      <Drawer
        sx={{
          width: drawerWidth,
          flexShrink: 0,
          '& .MuiDrawer-paper': {
            width: drawerWidth,
            boxSizing: 'border-box',
          },
        }}
```

```
        variant={isMobile ? 'temporary' : 'persistent'}
        anchor="left"
        open={open}
        onClose={handleDrawerClose}
      >
        <DrawerHeader>
          <IconButton onClick={handleDrawerClose}>
            {theme.direction === 'ltr' ? <ChevronLeftIcon /> : <ChevronRightIcon />}
          </IconButton>
        </DrawerHeader>
        <Divider />
        <List>
          {menuItems.map((item) => (
            <ListItem
              button
              key={item.text}
              onClick={() => navigate(item.path)}
              selected={location.pathname === item.path}
            >
              <ListItemIcon>{item.icon}</ListItemIcon>
              <ListItemText primary={item.text} />
            </ListItem>
          ))}
        </List>
        <Divider />
        <List>
          <ListItem button onClick={logout}>
            <ListItemIcon>
              <ExitToAppIcon />
            </ListItemIcon>
            <ListItemText primary="Cerrar SesiÃ³n" />
          </ListItem>
        </List>
      </Drawer>
      <Main open={open}>
        <DrawerHeader />
        {children}
      </Main>
    </Box>
  );
};

export default MainLayout;
```

GestiÃ³n de Estado

El frontend utiliza varios mecanismos para la gestiÃ³n de estado:

Contexto de AutenticaciÃ³n

```jsx
// src/contexts/AuthContext.tsx
import React, { createContext, useContext, useState, useEffect } from 'react';
import api from '../services/api';

interface User {
  id: number;
  username: string;
  rol: string;
}

interface AuthContextType {
  user: User | null;
  token: string | null;
  loading: boolean;
  error: string | null;
  login: (username: string, password: string) => Promise<void>;
  logout: () => void;
  isAuthenticated: () => boolean;
}

const AuthContext = createContext<AuthContextType | undefined>(undefined);

export const useAuth = () => {
  const context = useContext(AuthContext);
  if (!context) {
    throw new Error('useAuth debe ser usado dentro de un AuthProvider');
  }
  return context;
};

export const AuthProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {
```

```
  const [user, setUser] = useState<User | null>(null);
  const [token, setToken] = useState<string | null>(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    // Verificar si hay un token almacenado
    const storedToken = localStorage.getItem('token');
    const storedUser = localStorage.getItem('user');

    if (storedToken && storedUser) {
      setToken(storedToken);
      setUser(JSON.parse(storedUser));
      api.setAuthToken(storedToken);
    }

    setLoading(false);
  }, []);

  const login = async (username: string, password: string) => {
    try {
      setLoading(true);
      setError(null);

      const response = await api.post('/auth/login', { username, password });
      const { access_token, user } = response.data;

      localStorage.setItem('token', access_token);
      localStorage.setItem('user', JSON.stringify(user));

      setToken(access_token);
      setUser(user);
      api.setAuthToken(access_token);
    } catch (err: any) {
      setError(err.response?.data?.message || 'Error al iniciar sesiÃ³n');
      throw err;
    } finally {
      setLoading(false);
    }
  };

  const logout = () => {
    localStorage.removeItem('token');
    localStorage.removeItem('user');
    setToken(null);
    setUser(null);
    api.setAuthToken(null);
  };

  const isAuthenticated = () => {
    return !!token;
  };

  const value = {
    user,
    token,
    loading,
    error,
    login,
    logout,
    isAuthenticated,
  };

  return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
};
```

Contexto de Tema

```jsx
// src/contexts/ThemeContext.tsx
import React, { createContext, useContext, useState, useEffect } from 'react';
import { ThemeProvider as MuiThemeProvider, createTheme, Theme } from '@mui/material/styles';
import { PaletteMode } from '@mui/material';

interface ThemeContextType {
  mode: PaletteMode;
  toggleTheme: () => void;
}

const ThemeContext = createContext<ThemeContextType | undefined>(undefined);

export const useThemeContext = () => {
```

```
  const context = useContext(ThemeContext);
  if (!context) {
    throw new Error('useThemeContext debe ser usado dentro de un ThemeProvider');
  }
  return context;
};

export const ThemeProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {
  const [mode, setMode] = useState<PaletteMode>('light');

  useEffect(() => {
    const storedMode = localStorage.getItem('themeMode') as PaletteMode | null;
    if (storedMode) {
      setMode(storedMode);
    }
  }, []);

  const theme = React.useMemo(
    () =>
      createTheme({
        palette: {
          mode,
          primary: {
            main: '#1976d2',
          },
          secondary: {
            main: '#dc004e',
          },
        },
        typography: {
          fontFamily: '"Roboto", "Helvetica", "Arial", sans-serif',
          h1: {
            fontSize: '2.5rem',
            fontWeight: 500,
          },
          h2: {
            fontSize: '2rem',
            fontWeight: 500,
          },
          h3: {
            fontSize: '1.75rem',
            fontWeight: 500,
          },
          h4: {
            fontSize: '1.5rem',
            fontWeight: 500,
          },
          h5: {
            fontSize: '1.25rem',
            fontWeight: 500,
          },
          h6: {
            fontSize: '1rem',
            fontWeight: 500,
          },
        },
        components: {
          MuiButton: {
            styleOverrides: {
              root: {
                textTransform: 'none',
              },
            },
          },
        },
      }),
    [mode],
  );

  const toggleTheme = () => {
    const newMode = mode === 'light' ? 'dark' : 'light';
    setMode(newMode);
    localStorage.setItem('themeMode', newMode);
  };

  return (
    <ThemeContext.Provider value={{ mode, toggleTheme }}>
      <MuiThemeProvider theme={theme}>{children}</MuiThemeProvider>
    </ThemeContext.Provider>
  );
};
```

ComunicaciÃ³n con el Backend

El frontend utiliza Axios para la comunicaciÃ³n con el backend:

```typescript
// src/services/api.ts
import axios, { AxiosInstance, AxiosRequestConfig, AxiosResponse } from 'axios';

class Api {
  private api: AxiosInstance;

  constructor() {
    this.api = axios.create({
      baseURL: process.env.REACT_APP_API_URL || 'http://localhost:3001/api',
      headers: {
        'Content-Type': 'application/json',
      },
    });

    // Interceptor para manejar errores
    this.api.interceptors.response.use(
      (response) => response,
      (error) => {
        // Manejar errores de autenticaciÃ³n
        if (error.response && error.response.status === 401) {
          // Limpiar token y redirigir a login
          localStorage.removeItem('token');
          localStorage.removeItem('user');
          window.location.href = '/login';
        }
        return Promise.reject(error);
      }
    );
  }

  // Configurar token de autenticaciÃ³n
  setAuthToken(token: string | null): void {
    if (token) {
      this.api.defaults.headers.common['Authorization'] = `Bearer ${token}`;
    } else {
      delete this.api.defaults.headers.common['Authorization'];
    }
  }

  // MÃ©todos HTTP
  async get<T = any>(url: string, config?: AxiosRequestConfig): Promise<AxiosResponse<T>> {
    return this.api.get<T>(url, config);
  }

  async post<T = any>(url: string, data?: any, config?: AxiosRequestConfig): Promise<AxiosResponse<T>> {
    return this.api.post<T>(url, data, config);
  }

  async put<T = any>(url: string, data?: any, config?: AxiosRequestConfig): Promise<AxiosResponse<T>> {
    return this.api.put<T>(url, data, config);
  }

  async delete<T = any>(url: string, config?: AxiosRequestConfig): Promise<AxiosResponse<T>> {
    return this.api.delete<T>(url, config);
  }

  async patch<T = any>(url: string, data?: any, config?: AxiosRequestConfig): Promise<AxiosResponse<T>> {
    return this.api.patch<T>(url, data, config);
  }
}

const api = new Api();
export default api;
```

Rutas y NavegaciÃ³n

El frontend utiliza React Router para la gestiÃ³n de rutas:

```jsx
// src/App.tsx
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-router-dom';
import { useAuth } from './contexts/AuthContext';
import MainLayout from './layouts/MainLayout';
import AuthLayout from './layouts/AuthLayout';
import Login from './pages/Login';
import Dashboard from './pages/Dashboard';
```

```
import Productos from './pages/Productos';
import Ventas from './pages/Ventas';
import Gastos from './pages/Gastos';
import Reportes from './pages/Reportes';
import Administracion from './pages/Administracion';
import NotFound from './pages/NotFound';
import Unauthorized from './pages/Unauthorized';
import ProtectedRoute from './components/ProtectedRoute';

const App: React.FC = () => {
  return (
    <Router>
      <Routes>
        {/* Rutas pÃºblicas */}
        <Route path="/login" element={<AuthLayout><Login /></AuthLayout>} />

        {/* Rutas protegidas */}
        <Route path="/dashboard" element={
          <ProtectedRoute>
            <MainLayout><Dashboard /></MainLayout>
          </ProtectedRoute>
        } />

        <Route path="/productos" element={
          <ProtectedRoute>
            <MainLayout><Productos /></MainLayout>
          </ProtectedRoute>
        } />

        <Route path="/ventas" element={
          <ProtectedRoute>
            <MainLayout><Ventas /></MainLayout>
          </ProtectedRoute>
        } />

        <Route path="/gastos" element={
          <ProtectedRoute>
            <MainLayout><Gastos /></MainLayout>
          </ProtectedRoute>
        } />

        <Route path="/reportes" element={
          <ProtectedRoute>
            <MainLayout><Reportes /></MainLayout>
          </ProtectedRoute>
        } />

        <Route path="/administracion" element={
          <ProtectedRoute requiredRole="administrador">
            <MainLayout><Administracion /></MainLayout>
          </ProtectedRoute>
        } />

        {/* Rutas de error */}
        <Route path="/unauthorized" element={<AuthLayout><Unauthorized /></AuthLayout>} />
        <Route path="/404" element={<AuthLayout><NotFound /></AuthLayout>} />

        {/* Redirecciones */}
        <Route path="/" element={<Navigate to="/dashboard" replace />} />
        <Route path="*" element={<Navigate to="/404" replace />} />
      </Routes>
    </Router>
  );
};

export default App;
```

Componente de Ruta Protegida

```jsx
// src/components/ProtectedRoute.tsx
import React from 'react';
import { Navigate } from 'react-router-dom';
import { useAuth } from '../contexts/AuthContext';

interface ProtectedRouteProps {
  children: React.ReactNode;
  requiredRole?: string;
}

const ProtectedRoute: React.FC<ProtectedRouteProps> = ({ children, requiredRole }) => {
  const { isAuthenticated, user, loading } = useAuth();
```

```
  if (loading) {
    return <div>Cargando...</div>;
  }

  if (!isAuthenticated()) {
    return <Navigate to="/login" replace />;
  }

  if (requiredRole && user?.rol !== requiredRole) {
    return <Navigate to="/unauthorized" replace />;
  }

  return <>{children}</>;
};

export default ProtectedRoute;
```

Formularios y ValidaciÃ³n

El frontend utiliza un hook personalizado para la gestiÃ³n de formularios:

```typescript
// src/hooks/useForm.ts
import { useState, useCallback } from 'react';

interface ValidationRules {
  [key: string]: (value: any) => string | undefined;
}

interface FormState {
  [key: string]: any;
}

interface FormErrors {
  [key: string]: string;
}

const useForm = <T extends FormState>(initialState: T, validationRules?: ValidationRules) => {
  const [values, setValues] = useState<T>(initialState);
  const [errors, setErrors] = useState<FormErrors>({});
  const [touched, setTouched] = useState<{ [key: string]: boolean }>({});

  const handleChange = useCallback(
    (e: React.ChangeEvent<HTMLInputElement | { name?: string; value: unknown }>) => {
      const { name, value } = e.target;
      if (!name) return;

      setValues((prevValues) => ({
        ...prevValues,
        [name]: value,
      }));

      setTouched((prevTouched) => ({
        ...prevTouched,
        [name]: true,
      }));

      if (validationRules && validationRules[name]) {
        const error = validationRules[name](value);
        setErrors((prevErrors) => ({
          ...prevErrors,
          [name]: error || '',
        }));
      }
    },
    [validationRules]
  );

  const handleBlur = useCallback(
    (e: React.FocusEvent<HTMLInputElement>) => {
      const { name } = e.target;
      if (!name) return;

      setTouched((prevTouched) => ({
        ...prevTouched,
        [name]: true,
      }));

      if (validationRules && validationRules[name]) {
        const error = validationRules[name](values[name]);
        setErrors((prevErrors) => ({
```

```
          ...prevErrors,
          [name]: error || '',
        }));
      }
    },
    [validationRules, values]
  );

  const validateForm = useCallback(() => {
    if (!validationRules) return true;

    const newErrors: FormErrors = {};
    let isValid = true;

    Object.keys(validationRules).forEach((key) => {
      const error = validationRules[key](values[key]);
      if (error) {
        newErrors[key] = error;
        isValid = false;
      }
    });

    setErrors(newErrors);
    return isValid;
  }, [validationRules, values]);

  const resetForm = useCallback(() => {
    setValues(initialState);
    setErrors({});
    setTouched({});
  }, [initialState]);

  const setFieldValue = useCallback((name: string, value: any) => {
    setValues((prevValues) => ({
      ...prevValues,
      [name]: value,
    }));

    if (validationRules && validationRules[name]) {
      const error = validationRules[name](value);
      setErrors((prevErrors) => ({
        ...prevErrors,
        [name]: error || '',
      }));
    }
  }, [validationRules]);

  return {
    values,
    errors,
    touched,
    handleChange,
    handleBlur,
    validateForm,
    resetForm,
    setFieldValue,
  };
};

export default useForm;
```

IntegraciÃ³n con Electron

El frontend se integra con Electron para proporcionar una experiencia de escritorio nativa:

```javascript
// electron/main.js
const { app, BrowserWindow, ipcMain } = require('electron');
const path = require('path');
const isDev = require('electron-is-dev');
const Store = require('electron-store');

// ConfiguraciÃ³n de almacenamiento persistente
const store = new Store();

// Mantener una referencia global del objeto window para evitar que la ventana se cierre automÃ¡ticamente
// cuando el objeto JavaScript es recogido por el recolector de basura.
let mainWindow;

function createWindow() {
  // Crear la ventana del navegador.
  mainWindow = new BrowserWindow({
```

```
    width: 1200,
    height: 800,
    minWidth: 800,
    minHeight: 600,
    webPreferences: {
      nodeIntegration: false,
      contextIsolation: true,
      preload: path.join(__dirname, 'preload.js'),
    },
    icon: path.join(__dirname, '../assets/icon.png'),
  });

  // Cargar la aplicaciÃ³n React.
  mainWindow.loadURL(
    isDev
      ? 'http://localhost:3000'
      : `file://${path.join(__dirname, '../build/index.html')}`
  );

  // Abrir las herramientas de desarrollo en modo desarrollo.
  if (isDev) {
    mainWindow.webContents.openDevTools();
  }

  // Emitido cuando la ventana es cerrada.
  mainWindow.on('closed', () => {
    // Eliminar la referencia al objeto window, normalmente guardarÃas las ventanas
    // en un array si tu aplicaciÃ³n soporta mÃºltiples ventanas, este es el momento
    // en que deberÃas eliminar el elemento correspondiente.
    mainWindow = null;
  });
}

// Este mÃ©todo serÃ¡ llamado cuando Electron haya terminado
// la inicializaciÃ³n y estÃ© listo para crear ventanas del navegador.
// Algunas APIs pueden usarse sÃ³lo despuÃ©s de que este evento ocurra.
app.whenReady().then(createWindow);

// Salir cuando todas las ventanas estÃ©n cerradas, excepto en macOS.
app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit();
  }
});

app.on('activate', () => {
  // En macOS es comÃºn volver a crear una ventana en la aplicaciÃ³n cuando el
  // icono del dock es clicado y no hay otras ventanas abiertas.
  if (mainWindow === null) {
    createWindow();
  }
});

// ConfiguraciÃ³n de IPC (comunicaciÃ³n entre procesos)
ipcMain.handle('get-store-value', (event, key) => {
  return store.get(key);
});

ipcMain.handle('set-store-value', (event, key, value) => {
  store.set(key, value);
  return true;
});

ipcMain.handle('print-pdf', async (event, pdfPath) => {
  const pdfWindow = new BrowserWindow({
    width: 800,
    height: 600,
    show: false,
  });

  await pdfWindow.loadURL(`file://${pdfPath}`);
  const data = await pdfWindow.webContents.printToPDF({});
  pdfWindow.close();

  return data;
});
```

```javascript
// electron/preload.js
const { contextBridge, ipcRenderer } = require('electron');

// Exponer APIs protegidas a la ventana del navegador.
```

```
contextBridge.exposeInMainWorld('electron', {
  store: {
    get: (key) => ipcRenderer.invoke('get-store-value', key),
    set: (key, value) => ipcRenderer.invoke('set-store-value', key, value),
  },
  printer: {
    printPDF: (pdfPath) => ipcRenderer.invoke('print-pdf', pdfPath),
  },
  appInfo: {
    getVersion: () => process.env.npm_package_version,
  },
});
```

Empaquetado y DistribuciÃ³n

El frontend utiliza electron-builder para empaquetar la aplicaciÃ³n para diferentes plataformas:

```json
// electron-builder.json
{
  "appId": "com.posystem.app",
  "productName": "Sistema POS",
  "copyright": "Copyright Â© 2025",
  "directories": {
    "output": "dist",
    "buildResources": "assets"
  },
  "files": [
    "build/**/*",
    "electron/**/*",
    "assets/**/*"
  ],
  "win": {
    "target": [
      "nsis"
    ],
    "icon": "assets/icon.png"
  },
  "mac": {
    "target": [
      "dmg"
    ],
    "icon": "assets/icon.png"
  },
  "linux": {
    "target": [
      "AppImage",
      "deb"
    ],
    "icon": "assets/icon.png",
    "category": "Office"
  },
  "nsis": {
    "oneClick": false,
    "allowToChangeInstallationDirectory": true,
    "createDesktopShortcut": true,
    "createStartMenuShortcut": true
  }
}
```

OptimizaciÃ³n de Rendimiento

El frontend implementa varias estrategias para optimizar el rendimiento:

1. **MemoizaciÃ³n**: Uso de React.memo, useMemo y useCallback para evitar renderizados innecesarios.

```jsx
// Ejemplo de uso de React.memo
const ProductItem = React.memo(({ product, onSelect }) => {
  return (
    <ListItem button onClick={() => onSelect(product)}>
      <ListItemText primary={product.nombre} secondary={`$${product.precio_venta}`} />
    </ListItem>
  );
});
```

2. **Lazy Loading**: Carga diferida de componentes para reducir el tamaÃ±o del bundle inicial.

```jsx
// src/App.tsx
```

```
import React, { lazy, Suspense } from 'react';
import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-router-dom';
import MainLayout from './layouts/MainLayout';
import AuthLayout from './layouts/AuthLayout';
import Login from './pages/Login';
import ProtectedRoute from './components/ProtectedRoute';

// Lazy loading de componentes
const Dashboard = lazy(() => import('./pages/Dashboard'));
const Productos = lazy(() => import('./pages/Productos'));
const Ventas = lazy(() => import('./pages/Ventas'));
const Gastos = lazy(() => import('./pages/Gastos'));
const Reportes = lazy(() => import('./pages/Reportes'));
const Administracion = lazy(() => import('./pages/Administracion'));
const NotFound = lazy(() => import('./pages/NotFound'));
const Unauthorized = lazy(() => import('./pages/Unauthorized'));

const App: React.FC = () => {
  return (
    <Router>
      <Suspense fallback={<div>Cargando...</div>}>
        <Routes>
          {/* Rutas... */}
        </Routes>
      </Suspense>
    </Router>
  );
};
```

3. **VirtualizaciÃ³n**: Uso de react-window para renderizar listas largas de manera eficiente.

```jsx
// Ejemplo de uso de react-window
import { FixedSizeList as List } from 'react-window';

const ProductList = ({ products, onSelectProduct }) => {
  const Row = ({ index, style }) => (
    <div style={style}>
      <ProductItem product={products[index]} onSelect={onSelectProduct} />
    </div>
  );

  return (
    <List
      height={400}
      width="100%"
      itemCount={products.length}
      itemSize={72}
    >
      {Row}
    </List>
  );
};
```

4. **Code Splitting**: DivisiÃ³n del cÃ³digo en chunks mÃ¡s pequeÃ±os para mejorar los tiempos de carga.

```jsx
// webpack.config.js (configuraciÃ³n en react-scripts)
module.exports = {
  // ...
  optimization: {
    splitChunks: {
      chunks: 'all',
      name: false,
    },
  },
};
```

Seguridad

El frontend implementa varias medidas de seguridad:

1. **SanitizaciÃ³n de Datos**: Limpieza de datos de entrada para prevenir ataques XSS.

```jsx
// Ejemplo de sanitizaciÃ³n de datos
import DOMPurify from 'dompurify';

const sanitizeInput = (input) => {
  return DOMPurify.sanitize(input);
```

};
```

2. **ProtecciÃ³n de Rutas**: Control de acceso basado en autenticaciÃ³n y roles.

3. **Almacenamiento Seguro**: Uso de electron-store para almacenar datos sensibles de manera segura.

4. **ValidaciÃ³n de Datos**: ValidaciÃ³n estricta de todos los datos de entrada.

```typescript
// src/utils/validations.ts
export const required = (value: any): string | undefined => {
  return value ? undefined : 'Este campo es obligatorio';
};

export const email = (value: string): string | undefined => {
  return /^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(value)
    ? undefined
    : 'DirecciÃ³n de correo electrÃ³nico invÃ¡lida';
};

export const minLength = (min: number) => (value: string): string | undefined => {
  return value && value.length >= min
    ? undefined
    : `Este campo debe tener al menos ${min} caracteres`;
};

export const maxLength = (max: number) => (value: string): string | undefined => {
  return value && value.length <= max
    ? undefined
    : `Este campo debe tener como mÃ¡ximo ${max} caracteres`;
};

export const numeric = (value: string): string | undefined => {
  return /^\d+$/.test(value) ? undefined : 'Este campo debe contener solo nÃºmeros';
};

export const decimal = (value: string): string | undefined => {
  return /^\d+(\.\d{1,2})?$/.test(value)
    ? undefined
    : 'Este campo debe ser un nÃºmero decimal vÃ¡lido';
};

export const composeValidators = (...validators: Array<(value: any) => string | undefined>) => (
  value: any
): string | undefined => {
  for (const validator of validators) {
    const error = validator(value);
    if (error) {
      return error;
    }
  }
  return undefined;
};
```

ConclusiÃ³n

El frontend del Sistema POS proporciona una interfaz de usuario moderna, responsiva y fÃ¡cil de usar, implementando las mejores prÃ¡cticas de desarrollo con ReactJS y Electron. La arquitectura basada en componentes, la separaciÃ³n de responsabilidades y los patrones de diseÃ±o utilizados facilitan el mantenimiento y la extensiÃ³n del sistema.

Frontend (ReactJS + Electron)

El frontend del Sistema POS estÃ¡ desarrollado con ReactJS y Electron, proporcionando una interfaz de usuario moderna, responsiva y adaptada para aplicaciones de escritorio. Esta secciÃ³n detalla la implementaciÃ³n del frontend, sus componentes principales y patrones de diseÃ±o utilizados.

Arquitectura del Frontend

El frontend sigue una arquitectura basada en componentes, utilizando los patrones y prÃ¡cticas recomendadas de React. La aplicaciÃ³n estÃ¡ estructurada para maximizar la reutilizaciÃ³n de cÃ³digo, mantener un estado global coherente y proporcionar una experiencia de usuario fluida.

Componentes Principales

1. **Componentes**: Unidades visuales reutilizables que encapsulan la interfaz de usuario y la lÃ³gica de presentaciÃ³n.
2. **PÃ¡ginas**: Componentes de nivel superior que representan rutas completas en la aplicaciÃ³n.
3. **Layouts**: Estructuras de diseÃ±o que definen la disposiciÃ³n de los elementos en la pantalla.
4. **Contextos**: Gestores de estado global que proporcionan datos y funcionalidades a travÃ©s del Ã¡rbol de componentes.
5. **Hooks**: Funciones personalizadas que encapsulan lÃ³gica reutilizable.
6. **Servicios**: MÃ³dulos que manejan la comunicaciÃ³n con el backend y otras operaciones asÃncronas.
```

7. **Utilidades**: Funciones auxiliares para tareas comunes.

IntegraciÃ³n con Electron

La integraciÃ³n con Electron permite que la aplicaciÃ³n React funcione como una aplicaciÃ³n de escritorio nativa, con acceso a funcionalidades del sistema operativo y una experiencia de usuario mejorada.

ConfiguraciÃ³n de Electron

El archivo principal de Electron (`main.js`) configura la ventana de la aplicaciÃ³n y establece la comunicaciÃ³n entre los procesos principal y de renderizado:

```javascript
// electron/main.js
const { app, BrowserWindow, ipcMain } = require('electron');
const path = require('path');
const isDev = require('electron-is-dev');
const Store = require('electron-store');

// ConfiguraciÃ³n del almacenamiento persistente
const store = new Store();

// Mantener una referencia global del objeto window
let mainWindow;

function createWindow() {
  // Crear la ventana del navegador
  mainWindow = new BrowserWindow({
    width: 1200,
    height: 800,
    webPreferences: {
      nodeIntegration: false,
      contextIsolation: true,
      enableRemoteModule: false,
      preload: path.join(__dirname, 'preload.js')
    },
    icon: path.join(__dirname, '../assets/icon.png')
  });

  // Cargar la URL de la aplicaciÃ³n
  mainWindow.loadURL(
    isDev
      ? 'http://localhost:3000'
      : `file://${path.join(__dirname, '../build/index.html')}`
  );

  // Abrir DevTools en modo desarrollo
  if (isDev) {
    mainWindow.webContents.openDevTools();
  }

  // Maximizar la ventana al inicio
  mainWindow.maximize();

  // Evento cuando la ventana es cerrada
  mainWindow.on('closed', () => {
    mainWindow = null;
  });
}

// Crear ventana cuando Electron haya terminado de inicializarse
app.whenReady().then(createWindow);

// Salir cuando todas las ventanas estÃ©n cerradas
app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit();
  }
});

app.on('activate', () => {
  if (mainWindow === null) {
    createWindow();
  }
});

// ComunicaciÃ³n IPC para operaciones del sistema
ipcMain.handle('get-store-value', (event, key) => {
  return store.get(key);
});

ipcMain.handle('set-store-value', (event, key, value) => {
  store.set(key, value);
```

```
  return true;
});

// ComunicaciÃ³n IPC para impresiÃ³n
ipcMain.handle('print-content', async (event, options) => {
  const result = await mainWindow.webContents.print(options);
  return result;
});

// ComunicaciÃ³n IPC para acceso al sistema de archivos
ipcMain.handle('save-file', async (event, content, defaultPath) => {
  const { dialog } = require('electron');
  const fs = require('fs').promises;

  const { filePath } = await dialog.showSaveDialog({
    defaultPath
  });

  if (filePath) {
    await fs.writeFile(filePath, content);
    return filePath;
  }

  return null;
});
```

El archivo de precarga (`preload.js`) expone las funciones IPC al proceso de renderizado de manera segura:

```javascript
// electron/preload.js
const { contextBridge, ipcRenderer } = require('electron');

// Exponer API segura a travÃ©s del puente de contexto
contextBridge.exposeInMainWorld('electron', {
  store: {
    get: (key) => ipcRenderer.invoke('get-store-value', key),
    set: (key, value) => ipcRenderer.invoke('set-store-value', key, value),
  },
  print: (options) => ipcRenderer.invoke('print-content', options),
  saveFile: (content, defaultPath) => ipcRenderer.invoke('save-file', content, defaultPath),
  appVersion: process.env.npm_package_version,
});
```

GestiÃ³n de Estado

El frontend utiliza varios mecanismos para la gestiÃ³n de estado, adaptados a diferentes necesidades:

Contexto de AutenticaciÃ³n

El contexto de autenticaciÃ³n gestiona el estado de autenticaciÃ³n del usuario y proporciona funciones para iniciar y cerrar sesiÃ³n:

```jsx
// src/contexts/AuthContext.tsx
import React, { createContext, useContext, useState, useEffect } from 'react';
import api from '../services/api';

interface User {
  id: number;
  username: string;
  rol: string;
}

interface AuthContextType {
  user: User | null;
  token: string | null;
  loading: boolean;
  login: (username: string, password: string) => Promise<void>;
  logout: () => void;
  isAuthenticated: boolean;
  isAdmin: boolean;
}

const AuthContext = createContext<AuthContextType | undefined>(undefined);

export const AuthProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {
  const [user, setUser] = useState<User | null>(null);
  const [token, setToken] = useState<string | null>(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
```

```
    // Cargar usuario y token desde localStorage al iniciar
    const storedUser = localStorage.getItem('user');
    const storedToken = localStorage.getItem('token');

    if (storedUser && storedToken) {
      setUser(JSON.parse(storedUser));
      setToken(storedToken);
      api.setAuthToken(storedToken);
    }

    setLoading(false);
  }, []);

  const login = async (username: string, password: string) => {
    try {
      setLoading(true);
      const response = await api.post('/auth/login', { username, password });
      const { access_token, user: userData } = response.data;

      // Guardar en estado y localStorage
      setUser(userData);
      setToken(access_token);
      localStorage.setItem('user', JSON.stringify(userData));
      localStorage.setItem('token', access_token);
      api.setAuthToken(access_token);
    } catch (error) {
      console.error('Error de autenticaciÃ³n:', error);
      throw error;
    } finally {
      setLoading(false);
    }
  };

  const logout = () => {
    // Limpiar estado y localStorage
    setUser(null);
    setToken(null);
    localStorage.removeItem('user');
    localStorage.removeItem('token');
    api.removeAuthToken();
  };

  const isAuthenticated = !!user && !!token;
  const isAdmin = user?.rol === 'administrador';

  const value = {
    user,
    token,
    loading,
    login,
    logout,
    isAuthenticated,
    isAdmin,
  };

  return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
};

export const useAuth = () => {
  const context = useContext(AuthContext);
  if (context === undefined) {
    throw new Error('useAuth debe ser usado dentro de un AuthProvider');
  }
  return context;
};
```

Contexto de Tema

El contexto de tema gestiona la apariencia visual de la aplicaciÃ³n, permitiendo cambiar entre modo claro y oscuro:

```jsx
// src/contexts/ThemeContext.tsx
import React, { createContext, useContext, useState, useEffect } from 'react';
import { ThemeProvider as MuiThemeProvider } from '@mui/material/styles';
import { createTheme } from '../theme';

type ThemeMode = 'light' | 'dark';

interface ThemeContextType {
  mode: ThemeMode;
  toggleTheme: () => void;
  setMode: (mode: ThemeMode) => void;
```

```
}

const ThemeContext = createContext<ThemeContextType | undefined>(undefined);

export const ThemeProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {
  // Obtener preferencia guardada o usar preferencia del sistema
  const getInitialMode = (): ThemeMode => {
    const savedMode = localStorage.getItem('themeMode') as ThemeMode;
    if (savedMode) return savedMode;

    const prefersDark = window.matchMedia('(prefers-color-scheme: dark)').matches;
    return prefersDark ? 'dark' : 'light';
  };

  const [mode, setMode] = useState<ThemeMode>(getInitialMode);

  // Crear tema basado en el modo
  const theme = React.useMemo(() => createTheme(mode), [mode]);

  // Guardar preferencia cuando cambia
  useEffect(() => {
    localStorage.setItem('themeMode', mode);
    // TambiÃ©n guardar en electron store si estÃ¡ disponible
    if (window.electron?.store) {
      window.electron.store.set('themeMode', mode);
    }
  }, [mode]);

  const toggleTheme = () => {
    setMode(prevMode => (prevMode === 'light' ? 'dark' : 'light'));
  };

  const value = {
    mode,
    toggleTheme,
    setMode,
  };

  return (
    <ThemeContext.Provider value={value}>
      <MuiThemeProvider theme={theme}>{children}</MuiThemeProvider>
    </ThemeContext.Provider>
  );
};

export const useTheme = () => {
  const context = useContext(ThemeContext);
  if (context === undefined) {
    throw new Error('useTheme debe ser usado dentro de un ThemeProvider');
  }
  return context;
};
```

ComunicaciÃ³n con el Backend

El frontend se comunica con el backend a travÃ©s de un servicio API centralizado que utiliza Axios:

```typescript
// src/services/api.ts
import axios, { AxiosInstance, AxiosRequestConfig, AxiosResponse } from 'axios';

class ApiService {
  private api: AxiosInstance;
  private baseURL: string;

  constructor() {
    this.baseURL = process.env.REACT_APP_API_URL || 'http://localhost:3001/api';

    this.api = axios.create({
      baseURL: this.baseURL,
      headers: {
        'Content-Type': 'application/json',
      },
    });

    // Interceptor para manejar errores
    this.api.interceptors.response.use(
      (response) => response,
      (error) => {
        // Manejar errores de autenticaciÃ³n
        if (error.response && error.response.status === 401) {
          // Limpiar token y redirigir a login
```

```
          localStorage.removeItem('token');
          localStorage.removeItem('user');
          window.location.href = '/login';
        }
        return Promise.reject(error);
      }
    );

    // Cargar token si existe
    const token = localStorage.getItem('token');
    if (token) {
      this.setAuthToken(token);
    }
  }

  // Configurar token de autenticaciÃ³n
  setAuthToken(token: string): void {
    this.api.defaults.headers.common['Authorization'] = `Bearer ${token}`;
  }

  // Eliminar token de autenticaciÃ³n
  removeAuthToken(): void {
    delete this.api.defaults.headers.common['Authorization'];
  }

  // MÃ©todos para realizar peticiones HTTP
  async get<T = any>(url: string, config?: AxiosRequestConfig): Promise<T> {
    const response: AxiosResponse<T> = await this.api.get(url, config);
    return response.data;
  }

  async post<T = any>(url: string, data?: any, config?: AxiosRequestConfig): Promise<T> {
    const response: AxiosResponse<T> = await this.api.post(url, data, config);
    return response.data;
  }

  async put<T = any>(url: string, data?: any, config?: AxiosRequestConfig): Promise<T> {
    const response: AxiosResponse<T> = await this.api.put(url, data, config);
    return response.data;
  }

  async patch<T = any>(url: string, data?: any, config?: AxiosRequestConfig): Promise<T> {
    const response: AxiosResponse<T> = await this.api.patch(url, data, config);
    return response.data;
  }

  async delete<T = any>(url: string, config?: AxiosRequestConfig): Promise<T> {
    const response: AxiosResponse<T> = await this.api.delete(url, config);
    return response.data;
  }
}

const api = new ApiService();
export default api;
```

Rutas y NavegaciÃ³n

La navegaciÃ³n en la aplicaciÃ³n se gestiona mediante React Router, con protecciÃ³n de rutas basada en autenticaciÃ³n y roles:

```jsx
// src/App.tsx
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-router-dom';
import { AuthProvider, useAuth } from './contexts/AuthContext';
import { ThemeProvider } from './contexts/ThemeContext';

// Layouts
import MainLayout from './layouts/MainLayout';
import AuthLayout from './layouts/AuthLayout';

// PÃ¡ginas
import Login from './pages/Login';
import Dashboard from './pages/Dashboard';
import Productos from './pages/Productos';
import Ventas from './pages/Ventas';
import Gastos from './pages/Gastos';
import Reportes from './pages/Reportes';
import Administracion from './pages/Administracion';
import NotFound from './pages/NotFound';
import Unauthorized from './pages/Unauthorized';
```

```
// Componente de ruta protegida
const ProtectedRoute = ({ children, requiredRole }) => {
  const { isAuthenticated, user, loading } = useAuth();

  if (loading) {
    return <div>Cargando...</div>;
  }

  if (!isAuthenticated) {
    return <Navigate to="/login" />;
  }

  if (requiredRole && user.rol !== requiredRole) {
    return <Navigate to="/unauthorized" />;
  }

  return children;
};

const App = () => {
  return (
    <ThemeProvider>
      <AuthProvider>
        <Router>
          <Routes>
            {/* Rutas pÃºblicas */}
            <Route path="/login" element={
              <AuthLayout>
                <Login />
              </AuthLayout>
            } />

            {/* Rutas protegidas */}
            <Route path="/" element={
              <ProtectedRoute>
                <MainLayout>
                  <Dashboard />
                </MainLayout>
              </ProtectedRoute>
            } />

            <Route path="/productos" element={
              <ProtectedRoute>
                <MainLayout>
                  <Productos />
                </MainLayout>
              </ProtectedRoute>
            } />

            <Route path="/ventas" element={
              <ProtectedRoute>
                <MainLayout>
                  <Ventas />
                </MainLayout>
              </ProtectedRoute>
            } />

            <Route path="/gastos" element={
              <ProtectedRoute>
                <MainLayout>
                  <Gastos />
                </MainLayout>
              </ProtectedRoute>
            } />

            <Route path="/reportes" element={
              <ProtectedRoute>
                <MainLayout>
                  <Reportes />
                </MainLayout>
              </ProtectedRoute>
            } />

            {/* Rutas solo para administradores */}
            <Route path="/administracion" element={
              <ProtectedRoute requiredRole="administrador">
                <MainLayout>
                  <Administracion />
                </MainLayout>
              </ProtectedRoute>
            } />

            {/* Rutas de error */}
```

```
            <Route path="/unauthorized" element={
              <AuthLayout>
                <Unauthorized />
              </AuthLayout>
            } />

            <Route path="*" element={
              <AuthLayout>
                <NotFound />
              </AuthLayout>
            } />
          </Routes>
        </Router>
      </AuthProvider>
    </ThemeProvider>
  );
};

export default App;
```

Componentes Principales

Layout Principal

El layout principal define la estructura general de la aplicaciÃ³n para usuarios autenticados:

```jsx
// src/layouts/MainLayout.tsx
import React, { useState, useEffect } from 'react';
import { useNavigate, useLocation } from 'react-router-dom';
import { styled } from '@mui/material/styles';
import {
  Box,
  Drawer,
  AppBar,
  Toolbar,
  List,
  Typography,
  Divider,
  IconButton,
  ListItem,
  ListItemIcon,
  ListItemText,
  Menu,
  MenuItem,
  Avatar,
} from '@mui/material';
import {
  Menu as MenuIcon,
  ChevronLeft as ChevronLeftIcon,
  Dashboard as DashboardIcon,
  Inventory as InventoryIcon,
  ShoppingCart as ShoppingCartIcon,
  Receipt as ReceiptIcon,
  BarChart as BarChartIcon,
  Settings as SettingsIcon,
  AccountCircle,
  Logout,
} from '@mui/icons-material';
import { useAuth } from '../contexts/AuthContext';
import ThemeToggle from '../components/ThemeToggle';

const drawerWidth = 240;

const Main = styled('main', { shouldForwardProp: (prop) => prop !== 'open' })<{
  open?: boolean;
}>(({ theme, open }) => ({
  flexGrow: 1,
  padding: theme.spacing(3),
  transition: theme.transitions.create('margin', {
    easing: theme.transitions.easing.sharp,
    duration: theme.transitions.duration.leavingScreen,
  }),
  marginLeft: `-${drawerWidth}px`,
  ...(open && {
    transition: theme.transitions.create('margin', {
      easing: theme.transitions.easing.easeOut,
      duration: theme.transitions.duration.enteringScreen,
    }),
    marginLeft: 0,
  }),
}));
```

```
const AppBarStyled = styled(AppBar, {
  shouldForwardProp: (prop) => prop !== 'open',
})<{ open?: boolean }>(({ theme, open }) => ({
  transition: theme.transitions.create(['margin', 'width'], {
    easing: theme.transitions.easing.sharp,
    duration: theme.transitions.duration.leavingScreen,
  }),
  ...(open && {
    width: `calc(100% - ${drawerWidth}px)`,
    marginLeft: `${drawerWidth}px`,
    transition: theme.transitions.create(['margin', 'width'], {
      easing: theme.transitions.easing.easeOut,
      duration: theme.transitions.duration.enteringScreen,
    }),
  }),
}));

const DrawerHeader = styled('div')(({ theme }) => ({
  display: 'flex',
  alignItems: 'center',
  padding: theme.spacing(0, 1),
  ...theme.mixins.toolbar,
  justifyContent: 'flex-end',
}));

const MainLayout: React.FC<{ children: React.ReactNode }> = ({ children }) => {
  const [open, setOpen] = useState(true);
  const [anchorEl, setAnchorEl] = useState<null | HTMLElement>(null);
  const { user, logout, isAdmin } = useAuth();
  const navigate = useNavigate();
  const location = useLocation();

  const handleDrawerOpen = () => {
    setOpen(true);
  };

  const handleDrawerClose = () => {
    setOpen(false);
  };

  const handleMenu = (event: React.MouseEvent<HTMLElement>) => {
    setAnchorEl(event.currentTarget);
  };

  const handleClose = () => {
    setAnchorEl(null);
  };

  const handleLogout = () => {
    handleClose();
    logout();
    navigate('/login');
  };

  const menuItems = [
    { text: 'Dashboard', icon: <DashboardIcon />, path: '/' },
    { text: 'Productos', icon: <InventoryIcon />, path: '/productos' },
    { text: 'Ventas', icon: <ShoppingCartIcon />, path: '/ventas' },
    { text: 'Gastos', icon: <ReceiptIcon />, path: '/gastos' },
    { text: 'Reportes', icon: <BarChartIcon />, path: '/reportes' },
  ];

  // AÃ±adir opciÃ³n de administraciÃ³n solo para administradores
  if (isAdmin) {
    menuItems.push({
      text: 'AdministraciÃ³n',
      icon: <SettingsIcon />,
      path: '/administracion',
    });
  }

  return (
    <Box sx={{ display: 'flex' }}>
      <AppBarStyled position="fixed" open={open}>
        <Toolbar>
          <IconButton
            color="inherit"
            aria-label="open drawer"
            onClick={handleDrawerOpen}
            edge="start"
            sx={{ mr: 2, ...(open && { display: 'none' }) }}
          >
```

```
      <MenuIcon />
    </IconButton>
    <Typography variant="h6" noWrap component="div" sx={{ flexGrow: 1 }}>
      Sistema POS - Tienda de Accesorios MÃ³viles
    </Typography>
    <ThemeToggle />
    <div>
      <IconButton
        size="large"
        aria-label="account of current user"
        aria-controls="menu-appbar"
        aria-haspopup="true"
        onClick={handleMenu}
        color="inherit"
      >
        <Avatar sx={{ width: 32, height: 32 }}>
          {user?.username?.charAt(0).toUpperCase()}
        </Avatar>
      </IconButton>
      <Menu
        id="menu-appbar"
        anchorEl={anchorEl}
        anchorOrigin={{
          vertical: 'bottom',
          horizontal: 'right',
        }}
        keepMounted
        transformOrigin={{
          vertical: 'top',
          horizontal: 'right',
        }}
        open={Boolean(anchorEl)}
        onClose={handleClose}
      >
        <MenuItem disabled>
          <Typography variant="body2">
            {user?.username} ({user?.rol})
          </Typography>
        </MenuItem>
        <Divider />
        <MenuItem onClick={handleLogout}>
          <ListItemIcon>
            <Logout fontSize="small" />
          </ListItemIcon>
          <ListItemText>Cerrar sesiÃ³n</ListItemText>
        </MenuItem>
      </Menu>
    </div>
  </Toolbar>
</AppBarStyled>
<Drawer
  sx={{
    width: drawerWidth,
    flexShrink: 0,
    '& .MuiDrawer-paper': {
      width: drawerWidth,
      boxSizing: 'border-box',
    },
  }}
  variant="persistent"
  anchor="left"
  open={open}
>
  <DrawerHeader>
    <IconButton onClick={handleDrawerClose}>
      <ChevronLeftIcon />
    </IconButton>
  </DrawerHeader>
  <Divider />
  <List>
    {menuItems.map((item) => (
      <ListItem
        button
        key={item.text}
        onClick={() => navigate(item.path)}
        selected={location.pathname === item.path}
      >
        <ListItemIcon>{item.icon}</ListItemIcon>
        <ListItemText primary={item.text} />
      </ListItem>
    ))}
  </List>
</Drawer>
```

```
      <Main open={open}>
        <DrawerHeader />
        {children}
      </Main>
    </Box>
  );
};

export default MainLayout;
```

Formulario de Login

El componente de login maneja la autenticaciÃ³n de usuarios:

```jsx
// src/pages/Login.tsx
import React, { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import {
  Avatar,
  Button,
  TextField,
  Paper,
  Box,
  Grid,
  Typography,
  Alert,
} from '@mui/material';
import { LockOutlined } from '@mui/icons-material';
import { useAuth } from '../contexts/AuthContext';

const Login = () => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');
  const [loading, setLoading] = useState(false);
  const { login } = useAuth();
  const navigate = useNavigate();

  const handleSubmit = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();

    if (!username || !password) {
      setError('Por favor ingrese usuario y contraseÃ±a');
      return;
    }

    try {
      setError('');
      setLoading(true);
      await login(username, password);
      navigate('/');
    } catch (err) {
      setError('Usuario o contraseÃ±a incorrectos');
    } finally {
      setLoading(false);
    }
  };

  return (
    <Grid container component="main" sx={{ height: '100vh' }}>
      <Grid
        item
        xs={false}
        sm={4}
        md={7}
        sx={{
          backgroundImage: 'url(/images/login-bg.jpg)',
          backgroundRepeat: 'no-repeat',
          backgroundSize: 'cover',
          backgroundPosition: 'center',
        }}
      />
      <Grid item xs={12} sm={8} md={5} component={Paper} elevation={6} square>
        <Box
          sx={{
            my: 8,
            mx: 4,
            display: 'flex',
            flexDirection: 'column',
            alignItems: 'center',
          }}
```

```
        >
          <Avatar sx={{ m: 1, bgcolor: 'primary.main' }}>
            <LockOutlined />
          </Avatar>
          <Typography component="h1" variant="h5">
            Iniciar SesiÃ³n
          </Typography>
          {error && (
            <Alert severity="error" sx={{ mt: 2, width: '100%' }}>
              {error}
            </Alert>
          )}
          <Box component="form" noValidate onSubmit={handleSubmit} sx={{ mt: 1 }}>
            <TextField
              margin="normal"
              required
              fullWidth
              id="username"
              label="Nombre de Usuario"
              name="username"
              autoComplete="username"
              autoFocus
              value={username}
              onChange={(e) => setUsername(e.target.value)}
              disabled={loading}
            />
            <TextField
              margin="normal"
              required
              fullWidth
              name="password"
              label="ContraseÃ±a"
              type="password"
              id="password"
              autoComplete="current-password"
              value={password}
              onChange={(e) => setPassword(e.target.value)}
              disabled={loading}
            />
            <Button
              type="submit"
              fullWidth
              variant="contained"
              sx={{ mt: 3, mb: 2 }}
              disabled={loading}
            >
              {loading ? 'Iniciando sesiÃ³n...' : 'Iniciar SesiÃ³n'}
            </Button>
            <Typography variant="body2" color="text.secondary" align="center">
              Sistema POS para Tienda de Accesorios MÃ³viles
            </Typography>
          </Box>
        </Box>
      </Grid>
    </Grid>
  );
};

export default Login;
```

Tema y Estilos

La aplicaciÃ³n utiliza Material-UI con un tema personalizado que soporta modo claro y oscuro:

```typescript
// src/theme/index.ts
import { createTheme, responsiveFontSizes } from '@mui/material/styles';

// Crear tema personalizado
export const createAppTheme = (mode: 'light' | 'dark') => {
  let theme = createTheme({
    palette: {
      mode,
      primary: {
        main: '#1976d2',
        light: '#42a5f5',
        dark: '#1565c0',
      },
      secondary: {
        main: '#9c27b0',
        light: '#ba68c8',
        dark: '#7b1fa2',
```

```
  },
  ...(mode === 'light'
    ? {
        // Paleta para modo claro
        background: {
          default: '#f5f5f5',
          paper: '#ffffff',
        },
        text: {
          primary: 'rgba(0, 0, 0, 0.87)',
          secondary: 'rgba(0, 0, 0, 0.6)',
        },
      }
    : {
        // Paleta para modo oscuro
        background: {
          default: '#121212',
          paper: '#1e1e1e',
        },
        text: {
          primary: '#ffffff',
          secondary: 'rgba(255, 255, 255, 0.7)',
        },
      }),
},
typography: {
  fontFamily: [
    'Roboto',
    '"Helvetica Neue"',
    'Arial',
    'sans-serif',
  ].join(','),
  h1: {
    fontSize: '2.5rem',
    fontWeight: 500,
  },
  h2: {
    fontSize: '2rem',
    fontWeight: 500,
  },
  h3: {
    fontSize: '1.75rem',
    fontWeight: 500,
  },
  h4: {
    fontSize: '1.5rem',
    fontWeight: 500,
  },
  h5: {
    fontSize: '1.25rem',
    fontWeight: 500,
  },
  h6: {
    fontSize: '1rem',
    fontWeight: 500,
  },
},
components: {
  MuiButton: {
    styleOverrides: {
      root: {
        textTransform: 'none',
        borderRadius: 8,
      },
    },
  },
  MuiPaper: {
    styleOverrides: {
      root: {
        borderRadius: 8,
      },
    },
  },
  MuiCard: {
    styleOverrides: {
      root: {
        borderRadius: 8,
        boxShadow: mode === 'light'
          ? '0px 2px 4px rgba(0, 0, 0, 0.1)'
          : '0px 2px 4px rgba(0, 0, 0, 0.3)',
      },
    },
  },
```

```
      MuiTableCell: {
        styleOverrides: {
          head: {
            fontWeight: 'bold',
          },
        },
      },
    },
  });

  // Hacer las fuentes responsivas
  theme = responsiveFontSizes(theme);

  return theme;
};

export default createAppTheme;
```

ValidaciÃ³n de Formularios

La aplicaciÃ³n utiliza React Hook Form para la validaciÃ³n de formularios:

```typescript
// src/hooks/useForm.ts
import { useState, useCallback } from 'react';

interface ValidationRules {
  required?: boolean;
  minLength?: number;
  maxLength?: number;
  pattern?: RegExp;
  validate?: (value: any) => boolean | string;
}

interface FieldErrors {
  [key: string]: string;
}

export const useForm = <T extends Record<string, any>>(initialValues: T) => {
  const [values, setValues] = useState<T>(initialValues);
  const [errors, setErrors] = useState<FieldErrors>({});
  const [touched, setTouched] = useState<Record<string, boolean>>({});

  const handleChange = useCallback(
    (e: React.ChangeEvent<HTMLInputElement | HTMLTextAreaElement | HTMLSelectElement>) => {
      const { name, value } = e.target;
      setValues((prev) => ({ ...prev, [name]: value }));
      setTouched((prev) => ({ ...prev, [name]: true }));
    },
    []
  );

  const handleBlur = useCallback((e: React.FocusEvent<HTMLInputElement | HTMLTextAreaElement | HTMLSelectElement>) => {
    const { name } = e.target;
    setTouched((prev) => ({ ...prev, [name]: true }));
  }, []);

  const validate = useCallback(
    (validationRules: Record<keyof T, ValidationRules>) => {
      const newErrors: FieldErrors = {};
      let isValid = true;

      Object.keys(validationRules).forEach((key) => {
        const value = values[key];
        const rules = validationRules[key as keyof T];

        if (rules.required && !value) {
          newErrors[key] = 'Este campo es obligatorio';
          isValid = false;
        } else if (rules.minLength && value.length < rules.minLength) {
          newErrors[key] = `Debe tener al menos ${rules.minLength} caracteres`;
          isValid = false;
        } else if (rules.maxLength && value.length > rules.maxLength) {
          newErrors[key] = `Debe tener como mÃ¡ximo ${rules.maxLength} caracteres`;
          isValid = false;
        } else if (rules.pattern && !rules.pattern.test(value)) {
          newErrors[key] = 'Formato invÃ¡lido';
          isValid = false;
        } else if (rules.validate) {
          const result = rules.validate(value);
          if (typeof result === 'string') {
            newErrors[key] = result;
```

```
            isValid = false;
          } else if (!result) {
            newErrors[key] = 'Valor invÃ¡lido';
            isValid = false;
          }
        }
      });

      setErrors(newErrors);
      return isValid;
    },
    [values]
  );

  const resetForm = useCallback(() => {
    setValues(initialValues);
    setErrors({});
    setTouched({});
  }, [initialValues]);

  return {
    values,
    errors,
    touched,
    handleChange,
    handleBlur,
    validate,
    resetForm,
    setValues,
  };
};
```

Componentes Reutilizables

La aplicaciÃ³n incluye varios componentes reutilizables para mejorar la consistencia y reducir la duplicaciÃ³n de cÃ³digo:

```jsx
// src/components/FormField.tsx
import React from 'react';
import {
  TextField,
  FormControl,
  InputLabel,
  Select,
  MenuItem,
  FormHelperText,
  TextFieldProps,
  SelectProps,
} from '@mui/material';

interface FormFieldProps extends Omit<TextFieldProps, 'select' | 'SelectProps'> {
  name: string;
  label: string;
  value: string | number;
  onChange: (e: React.ChangeEvent<HTMLInputElement | HTMLTextAreaElement | HTMLSelectElement>) => void;
  onBlur?: (e: React.FocusEvent<HTMLInputElement | HTMLTextAreaElement | HTMLSelectElement>) => void;
  error?: string;
  touched?: boolean;
  select?: boolean;
  options?: Array<{ value: string | number; label: string }>;
  selectProps?: Omit<SelectProps, 'value' | 'onChange' | 'error'>;
}

const FormField: React.FC<FormFieldProps> = ({
  name,
  label,
  value,
  onChange,
  onBlur,
  error,
  touched,
  select,
  options,
  selectProps,
  ...rest
}) => {
  const showError = touched && !!error;

  if (select) {
    return (
      <FormControl
        fullWidth
```

```
        error={showError}
        margin="normal"
        {...rest}
      >
        <InputLabel id={`${name}-label`}>{label}</InputLabel>
        <Select
          labelId={`${name}-label`}
          id={name}
          name={name}
          value={value}
          label={label}
          onChange={onChange}
          onBlur={onBlur}
          {...selectProps}
        >
          {options?.map((option) => (
            <MenuItem key={option.value} value={option.value}>
              {option.label}
            </MenuItem>
          ))}
        </Select>
        {showError && <FormHelperText>{error}</FormHelperText>}
      </FormControl>
    );
  }

  return (
    <TextField
      fullWidth
      id={name}
      name={name}
      label={label}
      value={value}
      onChange={onChange}
      onBlur={onBlur}
      error={showError}
      helperText={showError ? error : ''}
      margin="normal"
      {...rest}
    />
  );
};

export default FormField;
```

Empaquetado y DistribuciÃ³n

La aplicaciÃ³n utiliza electron-builder para empaquetar y distribuir la aplicaciÃ³n:

```json
// electron-builder.json
{
  "appId": "com.posystem.app",
  "productName": "Sistema POS",
  "copyright": "Copyright Â© 2025",
  "directories": {
    "output": "dist",
    "buildResources": "assets"
  },
  "files": [
    "build/**/*",
    "electron/**/*",
    "assets/**/*"
  ],
  "mac": {
    "category": "public.app-category.business",
    "target": ["dmg", "zip"],
    "icon": "assets/icon.png"
  },
  "win": {
    "target": ["nsis"],
    "icon": "assets/icon.png"
  },
  "linux": {
    "target": ["AppImage", "deb"],
    "category": "Office",
    "icon": "assets/icon.png"
  },
  "nsis": {
    "oneClick": false,
    "allowToChangeInstallationDirectory": true,
    "createDesktopShortcut": true,
```

```
    "createStartMenuShortcut": true
  },
  "publish": {
    "provider": "github",
    "releaseType": "release"
  }
}
```

OptimizaciÃ³n de Rendimiento

El frontend implementa varias estrategias para optimizar el rendimiento:

1. **MemoizaciÃ³n**: Uso de React.memo, useMemo y useCallback para evitar renderizados innecesarios.

```jsx
// Ejemplo de uso de useMemo y useCallback
const MemoizedComponent = React.memo(({ data, onAction }) => {
  // Componente que solo se renderiza cuando data u onAction cambian
  return (
    <div>
      {data.map(item => (
        <div key={item.id} onClick={() => onAction(item.id)}>
          {item.name}
        </div>
      ))}
    </div>
  );
});

const ParentComponent = () => {
  const [items, setItems] = useState([]);

  // Memoizar datos procesados
  const processedData = useMemo(() => {
    return items.map(item => ({
      ...item,
      fullName: `${item.firstName} ${item.lastName}`
    }));
  }, [items]);

  // Memoizar funciÃ³n de callback
  const handleAction = useCallback((id) => {
    console.log(`Action on item ${id}`);
  }, []);

  return <MemoizedComponent data={processedData} onAction={handleAction} />;
};
```

2. **Code Splitting**: DivisiÃ³n del cÃ³digo en chunks mÃ¡s pequeÃ±os que se cargan bajo demanda.

```jsx
// src/App.tsx con lazy loading
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import LoadingScreen from './components/LoadingScreen';

// Importaciones lazy para cargar componentes bajo demanda
const Login = lazy(() => import('./pages/Login'));
const Dashboard = lazy(() => import('./pages/Dashboard'));
const Productos = lazy(() => import('./pages/Productos'));
const Ventas = lazy(() => import('./pages/Ventas'));
// ... mÃ¡s importaciones lazy

const App = () => {
  return (
    <Router>
      <Suspense fallback={<LoadingScreen />}>
        <Routes>
          <Route path="/login" element={<Login />} />
          <Route path="/" element={<Dashboard />} />
          <Route path="/productos" element={<Productos />} />
          <Route path="/ventas" element={<Ventas />} />
          {/* ... mÃ¡s rutas */}
        </Routes>
      </Suspense>
    </Router>
  );
};
```

3. **VirtualizaciÃ³n**: Renderizado eficiente de listas largas.

```jsx
// Ejemplo de virtualizaciÃ³n con react-window
import { FixedSizeList } from 'react-window';

const VirtualizedList = ({ items }) => {
  const Row = ({ index, style }) => (
    <div style={style}>
      Item {items[index].name}
    </div>
  );

  return (
    <FixedSizeList
      height={400}
      width="100%"
      itemCount={items.length}
      itemSize={50}
    >
      {Row}
    </FixedSizeList>
  );
};
```

Seguridad

El frontend implementa varias medidas de seguridad:

1. **SanitizaciÃ³n de Entradas**: ValidaciÃ³n y limpieza de datos de entrada para prevenir ataques XSS.

2. **ProtecciÃ³n de Rutas**: Control de acceso basado en autenticaciÃ³n y roles.

3. **Almacenamiento Seguro**: Uso de localStorage para tokens con tiempo de expiraciÃ³n.

4. **Manejo de Errores**: Captura y manejo adecuado de errores para evitar fugas de informaciÃ³n.

5. **ComunicaciÃ³n Segura**: Uso de HTTPS para todas las comunicaciones con el backend.

ConclusiÃ³n

El frontend del Sistema POS proporciona una interfaz de usuario moderna, intuitiva y responsiva, implementando las mejores prÃ¡cticas de desarrollo con ReactJS y Electron. La arquitectura basada en componentes, la gestiÃ³n de estado centralizada y los patrones de diseÃ±o utilizados facilitan el mantenimiento y la extensiÃ³n del sistema, mientras que la integraciÃ³n con Electron permite una experiencia de usuario nativa en aplicaciones de escritorio.

Sistema de Notificaciones y Alertas

El Sistema POS incluye un completo sistema de notificaciones y alertas que permite mantener informados a los usuarios sobre eventos importantes del sistema, como alertas de bajo stock, ventas realizadas, gastos registrados y otros eventos relevantes. Este sistema mejora la experiencia del usuario y facilita la toma de decisiones rÃ¡pidas.

Arquitectura del Sistema de Notificaciones

El sistema de notificaciones sigue una arquitectura basada en contextos de React y servicios especializados, con integraciÃ³n tanto en el backend como en el frontend.

Componentes Principales

1. **Backend (NestJS)**:
   - **AlertasService**: Servicio encargado de generar alertas basadas en condiciones del sistema (bajo stock, stock cero, productos sin movimiento).
   - **AlertasController**: Expone endpoints REST para consultar las diferentes alertas del sistema.
   - **IntegraciÃ³n con otros mÃ³dulos**: Los mÃ³dulos de ventas, gastos e inventario generan eventos que pueden convertirse en notificaciones.

2. **Frontend (React)**:
   - **NotificationContext**: Contexto de React que gestiona el estado global de las notificaciones.
   - **NotificationCenter**: Componente que muestra las notificaciones en la interfaz de usuario.
   - **ToastManager**: Componente que gestiona las notificaciones emergentes (toast).
   - **NotificationService**: Servicio que se comunica con el backend para obtener y procesar las notificaciones.

Tipos de Notificaciones

El sistema soporta cuatro tipos principales de notificaciones:

1. **Informativas (info)**: Notificaciones generales sobre eventos del sistema.
2. **Advertencias (warning)**: Alertas sobre situaciones que requieren atenciÃ³n pero no son crÃticas.
3. **Errores (error)**: Notificaciones sobre problemas que requieren atenciÃ³n inmediata.
4. **Ã‰xito (success)**: Confirmaciones de operaciones completadas con Ã©xito.

Flujo de Notificaciones

El flujo tÃpico de una notificaciÃ³n en el sistema es el siguiente:

1. **GeneraciÃ³n**: Un evento en el sistema (backend o frontend) genera una notificaciÃ³n.
2. **Procesamiento**: La notificaciÃ³n se procesa y se clasifica segÃºn su tipo e importancia.
3. **Almacenamiento**: La notificaciÃ³n se almacena en el estado global gestionado por el NotificationContext.
4. **VisualizaciÃ³n**: La notificaciÃ³n se muestra al usuario a travÃ©s del NotificationCenter y/o como una notificaciÃ³n emergente (toast).
5. **InteracciÃ³n**: El usuario puede interactuar con la notificaciÃ³n (marcarla como leÃda, eliminarla, hacer clic para ver mÃ¡s detalles).
6. **ActualizaciÃ³n**: El estado de la notificaciÃ³n se actualiza segÃºn la interacciÃ³n del usuario.

ImplementaciÃ³n del NotificationContext

El `NotificationContext` es un componente fundamental del sistema de notificaciones, ya que gestiona el estado global de las notificaciones y proporciona mÃ©todos para interactuar con ellas.

```typescript
// DefiniciÃ³n de la interfaz Notification
export interface Notification {
  id: number | string;
  title: string;
  message: string;
  type: 'info' | 'warning' | 'error' | 'success';
  read: boolean;
  createdAt: string;
  link?: string;
}

// Contexto de notificaciones
const NotificationContext = createContext<NotificationContextType | undefined>(undefined);

// Proveedor del contexto
export const NotificationProvider: React.FC<{ children: ReactNode }> = ({ children }) => {
  const [notifications, setNotifications] = useState<Notification[]>([]);
  const { user } = useAuth();

  // Calcular el nÃºmero de notificaciones no leÃdas
  const unreadCount = notifications.filter(notification => !notification.read).length;

  // Cargar notificaciones al iniciar y cuando cambia el usuario
  useEffect(() => {
    if (user) {
      fetchNotifications();

      // Configurar un intervalo para verificar nuevas notificaciones cada 5 minutos
      const intervalId = setInterval(() => {
        fetchNotifications();
      }, 5 * 60 * 1000);

      return () => clearInterval(intervalId);
    }
  }, [user]);

  // FunciÃ³n para obtener notificaciones del servidor
  const fetchNotifications = async () => {
    try {
      if (!user) return;

      // Obtener la tienda del usuario actual (si estÃ¡ disponible)
      const tiendaId = user.tienda_id;

      // Obtener todas las notificaciones
      const allNotifications = await notificationService.getAllNotifications(tiendaId);

      setNotifications(allNotifications);
    } catch (error) {
      console.error('Error al obtener notificaciones:', error);
    }
  };

  // FunciÃ³n para agregar una nueva notificaciÃ³n
  const addNotification = (notification: Omit<Notification, 'id' | 'createdAt' | 'read'>) => {
    const newNotification: Notification = {
      ...notification,
      id: Date.now(),
      read: false,
      createdAt: new Date().toISOString()
    };

    setNotifications(prev => [newNotification, ...prev]);
```

```
    // Mostrar notificaciÃ³n nativa del sistema si estÃ¡ soportado
    if ('Notification' in window && Notification.permission === 'granted') {
      new Notification(notification.title, {
        body: notification.message,
        icon: '/logo192.png'
      });
    }
  };

  // Otras funciones para gestionar notificaciones...

  // Valor del contexto
  const value = {
    notifications,
    unreadCount,
    addNotification,
    markAsRead,
    markAllAsRead,
    removeNotification,
    clearNotifications,
    fetchNotifications
  };

  return <NotificationContext.Provider value={value}>{children}</NotificationContext.Provider>;
};
```

ImplementaciÃ³n del NotificationCenter

El `NotificationCenter` es un componente de interfaz de usuario que muestra las notificaciones al usuario y le permite interactuar con ellas.

```typescript
const NotificationCenter: React.FC = () => {
  const theme = useTheme();
  const navigate = useNavigate();
  const {
    notifications,
    unreadCount,
    markAsRead,
    markAllAsRead,
    removeNotification,
    clearNotifications
  } = useNotifications();
  const [anchorEl, setAnchorEl] = useState<null | HTMLElement>(null);

  // Funciones para gestionar la apertura y cierre del menÃº de notificaciones
  const handleOpenMenu = (event: React.MouseEvent<HTMLElement>) => {
    setAnchorEl(event.currentTarget);
  };

  const handleCloseMenu = () => {
    setAnchorEl(null);
  };

  // FunciÃ³n para manejar el clic en una notificaciÃ³n
  const handleNotificationClick = (notification: Notification) => {
    markAsRead(notification.id);
    if (notification.link) {
      navigate(notification.link);
    }
    handleCloseMenu();
  };

  // Renderizado del componente
  return (
    <>
      <Tooltip title="Notificaciones">
        <IconButton color="inherit" onClick={handleOpenMenu}>
          <Badge badgeContent={unreadCount} color="error">
            <NotificationsIcon />
          </Badge>
        </IconButton>
      </Tooltip>

      <Menu
        anchorEl={anchorEl}
        open={Boolean(anchorEl)}
        onClose={handleCloseMenu}
        // Otras propiedades del menÃº...
      >
        {/* Contenido del menÃº de notificaciones */}
      </Menu>
```

```
    </>
  );
};
```

ImplementaciÃ³n del ToastManager

El `ToastManager` es un componente que gestiona las notificaciones emergentes (toast) que aparecen temporalmente en la pantalla.

```typescript
const ToastManager: React.FC = () => {
  const { notifications, markAsRead } = useNotifications();
  const [activeToasts, setActiveToasts] = useState<Notification[]>([]);
  const [queue, setQueue] = useState<Notification[]>([]);

  // MÃ¡ximo nÃºmero de toasts visibles simultÃ¡neamente
  const MAX_TOASTS = 3;

  // Procesar nuevas notificaciones
  useEffect(() => {
    // Filtrar solo notificaciones no leÃdas y que no estÃ©n ya en la cola o activas
    const newNotifications = notifications.filter(
      notification =>
        !notification.read &&
        !queue.some(q => q.id === notification.id) &&
        !activeToasts.some(t => t.id === notification.id)
    );

    if (newNotifications.length > 0) {
      setQueue(prev => [...prev, ...newNotifications]);
    }
  }, [notifications]);

  // Procesar la cola de notificaciones
  useEffect(() => {
    if (queue.length > 0 && activeToasts.length < MAX_TOASTS) {
      // Tomar la primera notificaciÃ³n de la cola
      const nextToast = queue[0];

      // Eliminarla de la cola
      setQueue(prev => prev.slice(1));

      // AÃ±adirla a los toasts activos
      setActiveToasts(prev => [...prev, nextToast]);
    }
  }, [queue, activeToasts]);

  // Manejar el cierre de un toast
  const handleCloseToast = (id: number | string) => {
    // Marcar la notificaciÃ³n como leÃda
    markAsRead(id);

    // Eliminar de los toasts activos
    setActiveToasts(prev => prev.filter(toast => toast.id !== id));
  };

  // Renderizado del componente
  return (
    <>
      {activeToasts.map(toast => (
        <NotificationToast
          key={toast.id}
          notification={toast}
          onClose={() => handleCloseToast(toast.id)}
          autoHideDuration={6000}
        />
      ))}
    </>
  );
};
```

IntegraciÃ³n con el Sistema de Alertas del Backend

El sistema de notificaciones se integra con el mÃ³dulo de alertas del backend para mostrar notificaciones sobre productos con bajo stock, productos sin movimiento y otras alertas importantes.

```typescript
// Servicio de notificaciones
class NotificationService {
  // Obtener alertas de bajo stock y convertirlas en notificaciones
  async getLowStockNotifications(tiendaId?: number): Promise<Notification[]> {
```

```
    try {
      let response;
      if (tiendaId) {
        response = await alertService.getLowStock();
      } else {
        // Si no se proporciona ID de tienda, obtener alertas de todas las tiendas
        response = await alertService.get('/alertas/bajo-stock');
      }

      const alerts = response.data.data;

      return alerts.map((alert: any) => ({
        id: alert.producto_id,
        title: 'Alerta de Stock',
        message: `El producto ${alert.nombre_producto} tiene un stock bajo (${alert.stock_actual} unidades) en la tienda
${alert.nombre_tienda}.`,
        type: 'warning' as const,
        read: false,
        createdAt: new Date().toISOString(),
        link: '/productos'
      }));
    } catch (error) {
      console.error('Error al obtener notificaciones de bajo stock:', error);
      return [];
    }
  }

  // Otras funciones para obtener diferentes tipos de alertas...
}
```

Notificaciones en Tiempo Real

El sistema incluye soporte para notificaciones en tiempo real cuando se realizan operaciones importantes como ventas o
gastos:

```typescript
// En el servicio de ventas
async createSale(saleData: any) {
  try {
    const response = await this.post('/ventas', saleData);

    // Crear notificaciÃ³n para la venta
    const notification = notificationService.createSaleNotification(response.data.data);

    // AÃ±adir la notificaciÃ³n al sistema
    // (esto se hace a travÃ©s del contexto de notificaciones)

    return response;
  } catch (error) {
    throw error;
  }
}
```

Notificaciones Nativas del Sistema Operativo

El sistema tambiÃ©n puede mostrar notificaciones nativas del sistema operativo cuando estÃ¡ disponible:

```typescript
// Solicitar permiso para notificaciones nativas al cargar
useEffect(() => {
  if ('Notification' in window && Notification.permission !== 'denied') {
    Notification.requestPermission();
  }
}, []);

// Mostrar notificaciÃ³n nativa
if ('Notification' in window && Notification.permission === 'granted') {
  new Notification(notification.title, {
    body: notification.message,
    icon: '/logo192.png'
  });
}
```

Pruebas del Sistema de Notificaciones

Se han implementado pruebas automatizadas para verificar el correcto funcionamiento del sistema de notificaciones:

```javascript
// Pruebas para el sistema de notificaciones
describe('Sistema de Notificaciones', () => {
```

```
  test('Debe mostrar el centro de notificaciones', () => {
    render(
      <TestWrapper>
        <NotificationCenter />
      </TestWrapper>
    );

    // Verificar que el botÃ³n de notificaciones existe
    const notificationButton = screen.getByRole('button');
    expect(notificationButton).toBeInTheDocument();
  });

  test('Debe aÃ±adir una notificaciÃ³n y actualizar el contador', async () => {
    render(
      <TestWrapper>
        <TestComponent />
      </TestWrapper>
    );

    // Verificar que inicialmente no hay notificaciones
    expect(screen.getByText('Notificaciones no leÃdas: 0')).toBeInTheDocument();

    // AÃ±adir una notificaciÃ³n
    fireEvent.click(screen.getByText('AÃ±adir notificaciÃ³n info'));

    // Verificar que el contador se actualiza
    await waitFor(() => {
      expect(screen.getByText('Notificaciones no leÃdas: 1')).toBeInTheDocument();
    });
  });

  // MÃ¡s pruebas...
});
```

OptimizaciÃ³n del Sistema de Notificaciones

Para garantizar un rendimiento Ã³ptimo, el sistema de notificaciones incluye varias optimizaciones:

1. **MemorizaciÃ³n de componentes**: Los componentes de notificaciones utilizan React.memo para evitar renderizados innecesarios.
2. **Throttling de actualizaciones**: Las actualizaciones de notificaciones se limitan para evitar sobrecargar el sistema.
3. **Carga bajo demanda**: Las notificaciones se cargan solo cuando son necesarias.
4. **Limpieza automÃ¡tica**: Las notificaciones antiguas se eliminan automÃ¡ticamente despuÃ©s de un perÃodo de tiempo.

ConclusiÃ³n

El sistema de notificaciones y alertas proporciona una forma eficiente de mantener informados a los usuarios sobre eventos importantes del sistema. Su arquitectura modular y su integraciÃ³n con el resto del sistema permiten una experiencia de usuario fluida y coherente.