**Plan del Proyecto — Sistema de Ventas (Full-Stack)**

**Justificación**

El plan de trabajo propuesto para la prueba técnica de un Sistema de Ventas Locatel está comprendido en tres ejes principales: alineación con los objetivos de negocio, uso de una pila tecnológica robusta y sostenible, y minimización de riesgos mediante fases claras e iterativas.

**1. Alineación con los objetivos de negocio**

* El sistema responde a una necesidad básica de cualquier organización comercial: gestionar clientes, productos y ventas con trazabilidad, seguridad y facilidad de uso.
* La implementación de un MVP (Producto Mínimo Viable) con módulos de autenticación, clientes, productos y ventas garantiza que se entregue valor rápidamente y se puedan mostrar resultados en poco tiempo.
* La arquitectura modular permite escalar hacia funcionalidades futuras (inventarios, reportes avanzados, multiusuario) sin necesidad de reescribir el sistema.

**2. Pila técnica**

* **Backend con Python y Django REST Framework (DRF):**
  + Aporta rapidez de desarrollo gracias a su ORM, serializers y viewsets.
  + Escalable y mantenible gracias a la separación en aplicaciones (customers, products, sales).
  + Integra fácilmente JWT (SimpleJWT) para autenticación moderna, estandarizada y segura.
  + Generación automática de documentación con Swagger/OpenAPI, lo cual mejora la trazabilidad y facilita integraciones.
* **Frontend con React (Vite + TypeScript):**
  + Permite construir una SPA rápida y responsiva con experiencia de usuario fluida.
  + El uso de TypeScript asegura tipado estático, lo que reduce errores y aumenta la mantenibilidad.
  + Arquitectura basada en componentes que facilita la extensión y reutilización del código.
  + Axios + interceptores resuelve de forma elegante el manejo de tokens expuestos por el backend.
* **Base de datos PostgreSQL:**
  + Estándar en entornos empresariales, soporta transacciones seguras, tipos avanzados y extensiones.
  + Escalable horizontal y verticalmente, ideal para una futura explotación analítica.
* **Docker & Docker Compose:**
  + Garantiza entornos reproducibles tanto en desarrollo como en producción.
  + Asegura que todos los desarrolladores y entornos de despliegue trabajen bajo las mismas condiciones.
  + Facilita la migración a entornos cloud o CI/CD sin cambios de configuración.

**3. Seguridad**

* La autenticación con JWT (access + refresh) protege el sistema y permite una experiencia moderna sin sesiones tradicionales.
* Se implementan buenas prácticas: variables de entorno, CORS restringido, separación de secretos, HTTPS en producción.
* El cálculo de totales se realiza en el servidor, evitando manipulación fraudulenta de precios en el cliente.

**5. Escalabilidad y mantenibilidad**

* La separación frontend / backend / DB permite escalar cada componente de manera independiente.
* El uso de serializers y viewsets DRF facilita mantener la API limpia, versionada y extensible.
* La modularidad en React garantiza que la interfaz pueda crecer en nuevas vistas o funcionalidades sin romper el código existente.
* Con Docker, el despliegue en entornos de nube o CI/CD es inmediato y consistente.

**6. Riesgos mitigados**

* Errores por diferencias de entorno → mitigados con Docker.
* Expiración de tokens → mitigada con interceptores y refresh tokens.
* Datos inconsistentes en totales de ventas → mitigados con validaciones backend y cálculos centralizados.
* Complejidad futura → mitigada con la elección de tecnologías estándar, maduras y documentadas.

**Prácticas Implementadas**

Aplicación web para gestionar **Clientes**, **Productos** y **Ventas** (con ítems), con autenticación mediante **JWT**.  
Arquitectura **SPA + API REST**: **React (Vite + TypeScript)** en el front y **Phyton** y **Django REST Framework** en el back, **PostgreSQL** como base de datos y **Docker Compose** para orquestación local y despliegue.

**Objetivos**

* CRUD de **Clientes** y **Productos** con validaciones.
* Registro de **Ventas** con detalles (producto, cantidad, precio capturado, cálculo de totales e IVA).
* Seguridad con **JWT access/refresh** y configuración de **CORS**.
* Documentación de API con **OpenAPI/Swagger**.
* Entorno reproducible con **Docker** y **.env**.

**Alcance Producto Mínimo Viable**

* Módulos: Autenticación, Clientes, Productos, Ventas.
* Roles (MVP): **admin** autenticado para todas las operaciones.

**Pila tecnológica**

**Frontend**

* Vite + React + TypeScript
* Axios, React Router
* CSS modular/Tailwind opcional (tema verde corporativo)

**Backend**

* Python 3.11, Django 4.x
* Django REST Framework
* SimpleJWT (tokens access + refresh)
* drf-spectacular (OpenAPI/Swagger)
* django-cors-headers
* gunicorn (prod)

**Base de datos**

* PostgreSQL 14+

**Infraestructura**

* Docker & Docker Compose
* Nginx (opcional en prod como reverse proxy)
* .env por entorno (dev, staging, prod)

**5) Arquitectura y componentes**

**Diagrama

El contenido generado por IA puede ser incorrecto.**

**Diagrama Arquitectura**

frontend/ (Vite React)

src/ pages/ (Customers, Products, Sales)

src/ lib/api.ts (axios + interceptores, JWT)

.env (VITE\_API\_URL, VITE\_API\_AUTH)

backend/ (Django)

app/backend/config/settings.py

app/customers, app/products, app/sales (apps DRF)

serializers, viewsets, urls, tests

/api/v1/... (routers DRF)

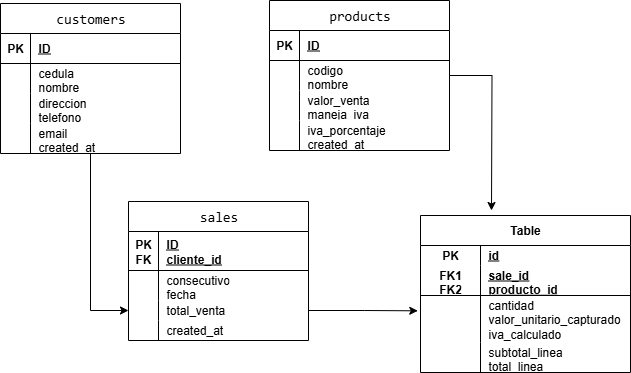
OpenAPI en /api/schema y /api/docs

db/ PostgreSQL

docker-compose.yml (web, db, front, nginx opcional)

Comunicación:  
SPA → Axios → http://localhost:8000/api/v1/\*  
Autorización: Authorization: Bearer <access\_token>

**6) Modelo de datos**

****

**Diagrama Base de Datos**

**customers**

* id (PK), cedula (unique), nombre, direccion, telefono, email (nullable), created\_at

**products**

* id (PK), codigo (unique), nombre, valor\_venta (decimal), maneja\_iva (bool), iva\_porcentaje (nullable), created\_at

**sales**

* id (PK), consecutivo (string o int incremental), cliente (FK customers), fecha (date), total\_venta (decimal), created\_at

**sale\_items**

* id (PK), venta (FK sales), producto (FK products), cantidad (int), valor\_unitario\_capturado (decimal)
* calculados: subtotal, iva\_valor, total\_linea

El **total de la venta** se calcula server-side sumando los ítems (y el IVA según maneja\_iva/iva\_porcentaje del producto).

**API REST (endpoints principales)**

* POST /api/v1/auth/login/ → { username, password } → { access, refresh }
* POST /api/v1/auth/refresh/ → { refresh } → { access }
* GET|POST /api/v1/customers/
* GET|PUT|DELETE /api/v1/customers/{id}/
* GET|POST /api/v1/products/
* GET|PUT|DELETE /api/v1/products/{id}/
* GET|POST /api/v1/sales/
  + **POST body esperado**:
  + {
  + "cliente": 1,
  + "fecha": "2025-08-23",
  + "items": [
  + { "producto": 6, "cantidad": 2, "valor\_unitario\_capturado": 5200 }
  + ]
  + }
* GET /api/v1/sales/{id}/
* **Docs**: /api/docs (Swagger UI), /api/schema (OpenAPI JSON)

**Autenticación, autorización y sesión**

* **SimpleJWT** con tiempos (configurables en settings.py):
  + ACCESS\_TOKEN\_LIFETIME = 30 min
  + REFRESH\_TOKEN\_LIFETIME = 7 días
* El front guarda access + refresh (localStorage) y usa **interceptor de Axios**:
  + En 401 → intenta refresh (si falla, redirige a login).
* CORS configurado para http://localhost:5173 (dev) y dominio real (prod).

**Servicios:**

* Backend → <http://localhost:8000/admin/>
* API Docs → <http://localhost:8000/api/schema/swagger-ui>
* Frontend → <http://localhost:5173>

**Frontend (UX/Flujo)**

* Navbar: **Clientes | Productos | Ventas**.
* **Clientes/Productos**:
  + Formulario de alta/edición.
  + **Tabla** con acciones ✏️ editar / 🗑️ borrar.
* **Ventas**:
  + select de **Cliente** (cargado de /customers/).
  + Lista dinámica de **Ítems**:
    - select de Producto (de /products/)
    - al elegir producto, autocompletar valor\_unitario\_capturado con valor\_venta
    - el usuario ingresa **cantidad**.
  + Botón **+ Producto** (agrega fila).
  + Submit: POST /sales/ y refresco de lista.
* Estilos: tema verde (similar Locatel) + estado de carga/errores.

**Validaciones clave**

**Frontend**

* Cliente seleccionado obligatorio.
* En ítems: producto obligatorio, cantidad ≥ 1.
* Si maneja\_iva = false → enviar iva\_porcentaje = null.

**Backend**

* Serializers DRF con validaciones:
  + producto existe, cantidad > 0.
  + Si maneja\_iva es false → iva\_porcentaje debe ser null.
  + Cálculo de totales protegido en el servidor.

**Seguridad**

* JWT en header Authorization.
* HTTPS en producción (Nginx/Cloud provider).
* **SECRET\_KEY** y credenciales **solo por .env**.
* CORS restrictivo por entorno.
* Deshabilitar DEBUG en prod.
* Migraciones versionadas.
* (Opcional) Rate-limit o throttling DRF.

**Observabilidad**

* Logging estructurado (Gunicorn + Django).
* Health check: /api/health (opcional).
* Métricas: contadores de requests (Nginx) y errores 4xx/5xx.

**Pruebas**

**Backend (pytest/DRF)**

* Unit tests de serializers (clientes, productos, ventas y cálculo de totales).
* API tests (200/201/400/401/404).
* Autenticación (login/refresh).

**Frontend (Vitest/RTL)**

* Render de páginas y flujos CRUD.
* Interceptores de Axios (401 → refresh).

**Entornos & variables**

**Backend .env**

DJANGO\_SECRET\_KEY=...

DJANGO\_DEBUG=True

DB\_HOST=db

DB\_NAME=ventas

DB\_USER=ventas

DB\_PASSWORD=ventas

DB\_PORT=5432

CORS\_ALLOWED\_ORIGINS=http://localhost:5173

# SIMPLE\_JWT override (opcional via settings)

**Frontend .env**

\*VITE\_API\_URL=http://localhost:8000/api/v1

VITE\_API\_AUTH=http://localhost:8000/api/v1/auth/login

**16) Puesta en marcha (local)**

**Con Docker**

docker compose up -d --build

# backend: http://localhost:8000

# frontend: http://localhost:5173

# swagger: http://localhost:8000/api/docs

**Sin Docker (dev)**

# backend

python -m venv .venv && source .venv/bin/activate

pip install -r requirements.txt

python manage.py migrate

python manage.py createsuperuser

python manage.py runserver 0.0.0.0:8000

# frontend

cd frontend

npm install

npm run dev # http://localhost:5173

**Despliegue (prod)**

* Build de front (npm run build) servido por Nginx.
* Backend con Gunicorn detrás de Nginx (reverse proxy + TLS).
* PostgreSQL gestionado (RDS/CloudSQL) o contenedor aislado.
* .env de prod con DEBUG=False, ALLOWED\_HOSTS, CORS de dominio, tiempos JWT.

**CI/CD**

* GitHub Actions:
  + Lint + tests (frontend y backend).
  + Build imágenes Docker y push a registry.
  + Deploy a VPS/Cloud con compose o a ECS/Kubernetes (según presupuesto).

**Cronograma de alto nivel**

* **D1 (1 Dia)**: base Django, modelos, serializers, auth JWT, CRUD customers/products, Swagger, front scaffolding + login + tablas.
* **D2 (1 Dia)**: ventas + items, cálculos totales/IVA, validaciones, estilo UI, interceptores refresh token.
* **D3 (1 Dia)**: pruebas, dockerización, documentación, hardening.

**Riesgos y mitigación**

* **CORS/Preflight**: validar CORS\_ALLOWED\_ORIGINS, CSRF\_TRUSTED\_ORIGINS y encabezados.
* **Expiración de JWT**: interceptores con refresh bien probado; tiempos ajustables.
* **Integridad de totales**: cálculos server-side y tests de bordes (IVA, cantidades).
* **Datos nulos**: normalizar iva\_porcentaje cuando maneja\_iva=false.

**Criterios de aceptación**

* Autenticación operativa (login + refresh).
* CRUD clientes/productos con edición/eliminación desde tabla.
* Crear venta seleccionando cliente y items; totales correctos.
* Swagger disponible y actualizado.
* Proyecto levanta con docker compose up sin pasos manuales extra.

**Entregables**

* Repositorio con **frontend**, **backend**, **docker-compose**, **README, Docs** de instalación, uso y documentacion. Link proyecto: <https://github.com/angelicadr/prueba-ventas.git>
* Archivo **Postman** con colecciones de endpoints[. ventas.postman\_collection.json](../ventas.postman_collection.json)
* **OpenAPI** exportado (JSON) + enlace a Swagger. <http://localhost:8000/api/schema/swagger-ui>

**Detalles prácticos**

**Cálculo de totales (server-side, idea general)**

total = 0

for item in items:

precio = item.valor\_unitario\_capturado

sub = precio \* item.cantidad

if item.producto.maneja\_iva:

iva = sub \* (item.producto.iva\_porcentaje / 100)

else:

iva = 0

total += sub + iva

venta.total\_venta = total

**Interceptor de Axios (refresh en 401)**

* Si 401 y existe refresh, llamar /auth/refresh/ y reintentar.
* Si el refresh falla → logout.

**Beneficios Obtenidos**

**Beneficios de negocio**

* **Gestión centralizada de clientes, productos y ventas**  
  Evita duplicidad de datos, mejora la trazabilidad y permite tomar decisiones basadas en información confiable.
* **Reducción de errores operativos**  
  Validaciones en frontend y backend aseguran que no se registren ventas con datos incompletos o inconsistentes.
* **Mayor productividad**  
  Interfaz ágil y sencilla permite a los usuarios registrar clientes, productos y ventas de manera rápida, reduciendo tiempos de operación.
* **Base sólida para reportes financieros**  
  Al tener totales de ventas y cálculo de impuestos automatizados, se facilita la generación de reportes contables y proyecciones.
* **Escalabilidad funcional**  
  El sistema se diseñó modular, lo que permite agregar nuevas funcionalidades (inventario, multiusuario, facturación electrónica) sin reescribir el núcleo.

**Beneficios técnicos**

* **Entorno reproducible con Docker**  
  Todos los desarrolladores y servidores de despliegue usan la misma configuración, eliminando errores por diferencias de entorno.
* **Arquitectura desacoplada (frontend + backend + DB)**  
  Facilita el mantenimiento, pruebas, escalabilidad y posibilidad de reemplazar componentes en el futuro.
* **Seguridad mediante JWT**  
  Sesiones modernas y seguras, con control de expiración y refresco de tokens, reduciendo riesgos de accesos no autorizados.
* **Estandarización con DRF y OpenAPI**  
  API documentada automáticamente, lista para integrarse con otros sistemas sin esfuerzos adicionales.
* **Mantenibilidad y reducción de errores**  
  Uso de TypeScript en el frontend y DRF en el backend asegura tipado fuerte y validaciones, lo que reduce fallos en producción.

**Beneficios para el usuario final**

* **Experiencia de usuario fluida**  
  El frontend con React permite navegar y registrar datos sin recargas de página, lo que mejora la usabilidad.
* **Interfaz consistente y moderna**  
  Diseño basado en tablas, formularios validados y colores corporativos facilita la adopción por parte de los usuarios.
* **Acceso confiable a la información**  
  Los usuarios siempre verán datos actualizados en tiempo real, sin depender de procesos manuales.

**Beneficios estratégicos**

* **Adaptabilidad al crecimiento**  
  Al estar basado en tecnologías estándar (Django, React, PostgreSQL), el sistema puede crecer en complejidad y usuarios sin comprometer el rendimiento.
* **Reducción de costos a futuro**  
  Uso de tecnologías open source y despliegue en contenedores minimiza licenciamiento y facilita migraciones a la nube.
* **Soporte para transformación digital**  
  La plataforma sienta las bases para automatización de procesos, analítica avanzada e integración con otros servicios (ERP, facturación electrónica, BI).