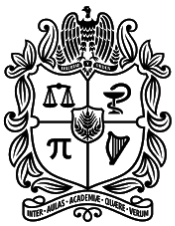

UNIVERSIDAD
NACIONAL
DE COLOMBIA

Manual De Backend

GestionApp

Integrantes:

Rafael Nicolas Ramirez Hernandez

Profesor: Nestor German Bolivar Pulgarin

Asignatura: Programación Orientada A Objetos (POO)

1. Instalación

- Python 3.9 o superior instalado en el sistema (3.13 recomendado)

EN UNA VENTANA DE CMD INGRESAR: (LOS COMANDOS EN VERDE :D)

- Pip actualizado usando `python -m ensurepip --upgrade`
- Firebase-admin instalado con `pip install firebase-admin`
- Supabase instalado con `pip install supabase`
- Customtkinter instalado con `pip install customtkinter`
- Sdl2 instalado con `pip install PySDL2 pysdl2-dll==2.26.5`
- Tkinter (ya incluido en la instalación estándar de Python, no requiere pip)
- OpenAI instalado con `pip install openai`
- PIL (Pillow) instalado con `pip install Pillow`

- **API:**

Declarar la variable de entorno en PowerShell:
setx OPENAI_API_KEY "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"

NOTA: Si va a usar otro API después para otro proyecto, asegurese de eliminar la variable de entorno para evitar problemas, en PowerShell escribir: setx OPENAI_API_KEY ""

2. Core

ThemeManager.py – Clase ThemeManager

- **Descripción**

Administra el tema visual de la aplicación (claro u oscuro) y permite que otros componentes actualicen su interfaz cuando el tema cambia. Implementa un patrón Singleton para asegurar una única instancia.

- **Métodos**

`__new__`
Garantiza una sola instancia del gestor de temas.

`__init__`
Establece el tema actual según CustomTkinter.

`subscribe(callback)`
Registra funciones que deben actualizarse al cambiar el tema.

`_notify()`
Notifica a todos los suscriptores.

`toggle_theme()`
Alterna entre modo claro y oscuro.

`set_theme(mode)`
Define el tema solicitado y actualiza a los suscriptores.

`get_theme()`
Retorna el tema actual.

`get_color(key)`
Devuelve un color del tema activo según la clave indicada.

CORE – Archivos complementarios

Core incluye módulos auxiliares y easter eggs que no afectan la lógica principal.

DoomCore.py
Muestra una ventana secundaria y ejecuta el juego DOOM mediante Chocolate Doom.

halo_detector.py
Detecta una secuencia específica de acciones para activar un evento oculto.

konami_detector.py
Detecta el “Konami Code” y ejecuta Chocolate Doom.

sdl_loader.py
Gestiona la carga de bibliotecas SDL utilizadas para funciones multimedia.

SDL2.dll, SDL3.dll, SDL2_mixer.dll
Bibliotecas necesarias para audio y otros efectos multimedia.

3. Servicios

ChatService.py

- **Descripción**

Servicio que gestiona la comunicación con la API de OpenAI. Genera respuestas basadas en el historial de conversación y en datos obtenidos desde Firebase sobre tiendas, productos y ventas del usuario.

- **Métodos**

`__init__(api_key)`
Crea el cliente de OpenAI y el lector de Firebase.

`_construir_contexto(user_id)`
Arma un texto contextual con los datos del usuario almacenados en Firebase.

`send(history, user_id, ...)`
Construye los mensajes, consulta al modelo y devuelve la respuesta generada.

FireBServ.py

- **Descripción**

Clase encargada de centralizar la comunicación con Firebase (Realtime Database y Storage). Gestiona usuarios, tiendas, productos y facturación mediante métodos CRUD y abstrae toda la lógica de acceso a datos.

- **Métodos**

`__init__`
Inicializa Firebase si aún no existe una instancia previa (`firebase_admin._apps`). Carga las credenciales desde *FBKey.json* y define la URL del Realtime Database.

- **USUARIOS**

`guardar_usuario`
Inserta un nuevo usuario en `/usuarios`. Convierte el objeto usuario a diccionario (nombre, correo, contraseña) y lo guarda con `ref.push()`.

- **TIENDAS**

`guardar_tienda`
Guarda una tienda en `/tiendas` incluyendo nombre, categoría, dirección y propietario.

`obtener_tiendas_por_usuario`
Filtra todas las tiendas y devuelve solo aquellas cuyo campo `propietario` coincide con el usuario especificado.

`actualizar_tienda`
Actualiza únicamente los campos enviados dentro de la tienda indicada usando `ref.update()`.

`eliminar_tienda`
Elimina completamente la tienda seleccionada mediante `ref.delete()`.

- **PRODUCTOS**

`agregar_producto`
Agrega un producto dentro de `/tiendas/{id_tienda}/productos` con nombre, precio, stock, ubicación e imagen.

`obtener_productos`
Obtiene todos los productos de una tienda. Retorna un diccionario o `{}` si no existen.

`actualizar_producto`
Actualiza un producto sin sobrescribir campos no incluidos en la solicitud.

`eliminar_producto`
Elimina un producto específico de la tienda.

- **FACTURACIÓN**

`registrar_factura`
Registra una venta en `/tiendas/{id_tienda}/ventas` y retorna el ID generado.

`obtener_historial_ventas`
Devuelve todas las ventas de la tienda o `{}` si no hay registros.

`obtener_venta`
Obtiene la información completa de una venta mediante su ID.

`eliminar_venta`
Elimina la venta seleccionada de Firebase.

`descontar_stocks`
Recorre los ítems de una factura y descuenta la cantidad correspondiente a cada producto:

- Obtiene el producto real
- Calcula `nuevo_stock = stock_actual - cantidad`
- Evita valores negativos con `max()`
- Actualiza el stock final

FBReadService.py

- **Descripción**

Servicio dedicado únicamente a lecturas en Firebase. Permite obtener tiendas, productos y ventas asociadas a un usuario, realizando búsquedas filtradas y devolviendo estructuras listas para usar en la lógica del backend.

- **Métodos**

`__init__`
Inicializa Firebase si aún no existe una instancia previa. Carga *FBKey.json* y define la URL del Realtime Database.

- **TIENDAS**

`obtener_tiendas_por_usuario`
Devuelve todas las tiendas cuyo campo `propietario` coincide con el usuario indicado. Si no existen tiendas, retorna `{}`.

`obtener_tienda_por_nombre`
Busca dentro de las tiendas del usuario aquella cuyo nombre coincida (ignorando mayúsculas/minúsculas).
Retorna un diccionario que incluye el ID real de la tienda o `None` si no hay coincidencias.

- **PRODUCTOS**

`obtener_productos_de_tienda`
Verifica que la tienda pertenezca al usuario y luego obtiene todos los productos almacenados en `/tiendas/{id_tienda}/productos`.
Retorna un diccionario o `{}` si no existen productos.

`buscar_producto`
Busca coincidencias en el nombre de productos dentro **todas** las tiendas del usuario.
Devuelve una lista de resultados con:

- ID de tienda
- Nombre de tienda
- ID del producto
- Información completa del producto

- **VENTAS**

`obtener_ventas_por_usuario`
Obtiene todas las ventas de todas las tiendas pertenecientes al usuario.
Construye un diccionario con la forma:

```
{id_tienda: {id_venta: datos_venta}}
```

ImageStorage.py

- **Descripción**

Servicio encargado de manejar imágenes mediante Supabase Storage. Permite subir archivos locales, obtener URLs públicas y eliminar imágenes almacenadas en un bucket específico.

- **Métodos**

`__init__`
Inicializa el cliente de Supabase usando la URL y clave proporcionadas.
Define el bucket donde se almacenarán las imágenes (por defecto `"Bucket"`).

- **IMÁGENES**

`upload_image`
Sube una imagen desde una ruta local a Supabase Storage.

- Verifica que el archivo exista
  Genera un nombre único usando `uuid4()`
- Construye la ruta remota respetando el folder si se envía
  Sube el archivo al bucket configurado
  Retorna la URL pública generada.

`get_public_url`
Genera la URL pública de un archivo almacenado dentro del bucket.
Usa la función nativa de Supabase Storage para construirla.

`delete_image`
Elimina una imagen específica del bucket usando su ruta dentro del almacenamiento.
Retorna la respuesta del servidor Supabase.

4. Carpeta Models

Usuario.py

- **Descripción**

Modelo que representa un usuario de la aplicación, almacenando su nombre, correo y contraseña. Permite acceder a información básica del usuario de forma sencilla.

- **Métodos**

`__init__`
Inicializa un nuevo usuario asignando nombre, correo y contraseña.

`mostrar_info`
Devuelve un string con la información principal del usuario (nombre y correo) para visualización o depuración.

Tienda.py

- **Descripción**

Modelo que representa una tienda registrada por un usuario. Almacena información básica como nombre, categoría, dirección y propietario (objeto `Usuario`).

- **Métodos**

`__init__`
Inicializa una nueva tienda asignando nombre, categoría, dirección y propietario (instancia de `Usuario`).

`mostrar_info`
Devuelve un string con la información completa de la tienda, incluyendo nombre, categoría, dirección y nombre del propietario.

Producto.py

- **Descripción**

Modelo que representa un producto dentro de una tienda. Contiene información básica como nombre, precio, stock, ubicación y opcionalmente la URL de una imagen.

- **Métodos**

`__init__`
Inicializa un nuevo producto asignando nombre, precio, stock, ubicación y una URL de imagen opcional.

`to_dict`
Convierte la instancia del producto en un diccionario, listo para guardarse en bases de datos o enviarse a APIs.

`__str__`
Devuelve un string representativo del producto mostrando nombre, precio, stock y ubicación para impresión o debug.

FacturaModel.py

- **Descripción**

Modelo que representa una factura y sus ítems en la aplicación. Almacena información de la tienda, los productos vendidos, total, fecha y un ID único de venta.

- **Clases**

`FacturaItem`

- Representa un ítem dentro de la factura.
- Atributos: `id_producto`, `nombre`, `cantidad`, `precio_unitario` y `subtotal`.

`FacturaModel`

- Representa la factura completa.
- Atributos: `id_tienda`, `items` (lista de `FacturaItem`), `total`, `fecha` (ISO string) e `id_venta` (UUID generado automáticamente).

5. Carpeta ViewModels

FacturaViewModel.py

- **Descripción**

Gestiona toda la lógica de facturación: carga tiendas y productos desde Firebase, administra el carrito, valida stock, genera facturas completas, actualiza existencias y permite consultar/eliminar ventas.

- **Métodos**

`__init__`
Inicializa la conexión a Firebase, guarda el usuario, crea caches de tiendas/productos y un carrito vacío. También prepara el ID de la tienda activa.

`cargar_tiendas_usuario`
Consulta en Firebase todas las tiendas del usuario, las transforma a un mapa `nombre → id` y lo guarda en cache.

`obtener_id_tienda_por_nombre`
Busca en el cache y devuelve el ID real Firebase de la tienda seleccionada.

`cargar_productos`
Solicita los productos de la tienda en Firebase, los guarda en un cache local y fija la tienda actual para futuras operaciones.

`obtener_producto`
Devuelve desde el cache un producto específico usando su ID.

`agregar_item_al_carrito`
Verifica existencia del producto y cantidad válida, suma unidades si ya está en el carrito o crea un `FacturaItem` nuevo calculando su subtotal.

`eliminar_item_del_carrito`
Filtra el carrito para remover únicamente el producto con el ID indicado.

`vaciar_carrito`
Limpia completamente la lista del carrito.

`obtener_total`
Suma los subtotales de todos los `FacturaItem` almacenados.

`obtener_resumen_carrito`
Transforma cada ítem del carrito en un diccionario fácil de renderizar (nombre, precio, cantidad, subtotal).

`stock_suficiente_para_carrito`
Revisa cada ítem del carrito contra el stock real en Firebase y devuelve si todo está disponible o un mensaje de error.

`guardar_factura_completa`
Verifica tienda activa y carrito no vacío, valida stock, arma la factura con fecha, ítems y total, la registra en Firebase, descuenta stock producto por producto y vacía el carrito.

`listar_ventas`
Obtiene desde Firebase el historial completo de facturas de la tienda.

`obtener_detalle_venta`
Trae una factura específica (su contenido completo) usando su ID.

`eliminar_factura`
Elimina la factura seleccionada en Firebase y retorna confirmación.

ProductViewModel.py

- **Descripción**

Gestiona todos los productos de una tienda: creación, lectura, actualización y eliminación. Usa `FBServ` como servicio de base de datos y el modelo `Producto` para estructurar los datos.

- **Métodos**

`__init__`
Guarda el ID de la tienda y crea una instancia de `FBServ` para interactuar con Firebase.

`agregar_producto`
Crea un objeto `Producto` con nombre, precio, stock, ubicación e imagen. Luego llama a `db.agregar_producto()` para enviarlo a Firebase.

`listar_productos`
Solicita los productos desde Firebase con `db.obtener_productos()`, imprime un listado en consola y retorna el diccionario completo.

`editar_producto`
Carga la lista de productos, permite seleccionar uno, captura nuevos valores desde consola, arma un `patch` con solo los campos modificados y lo envía a Firebase mediante `db.actualizar_producto()`.

`eliminar_producto`
Muestra los productos disponibles, permite escoger uno por índice, obtiene su ID real y lo elimina llamando a `db.eliminar_producto()`.

TiendaViewModel.py

- **Descripción**

Administra todas las operaciones CRUD de tiendas: registro, listado, edición y eliminación, usando Firebase como backend y asociando cada tienda al usuario dueño.

- **Métodos**

`__init__`
Recibe el usuario activo, inicializa el servicio Firebase y deja listo el ViewModel para operar sobre sus tiendas.

`registrar_tienda`
Crea un objeto `Tienda` con nombre, categoría, dirección y usuario dueño.
Lo envía a Firebase mediante `guardar_tienda`.

`listar_tiendas`
Consulta a Firebase todas las tiendas asociadas al usuario.
Imprime cada una y construye una lista en forma `(id_tienda, datos)` que se devuelve para usar en la UI u otras operaciones.

`editar_tienda`
Pide nuevos valores por consola, arma un `patch` solo con los campos escritos y actualiza la tienda en Firebase usando `actualizar_tienda`.

`eliminar_tienda`
Pide confirmación por consola y, si es afirmativa, elimina la tienda completa en Firebase junto con sus productos mediante `eliminar_tienda`.

UserViewModel.py

- **Descripción**
  Maneja el registro, autenticación y consulta de usuarios utilizando Firebase.
  Convierte los datos obtenidos en instancias del modelo `Usuario` cuando aplica.
- **Métodos**

`__init__`
Inicializa el servicio Firebase para interactuar con la base de datos.

`registrar_usuario`
Crea un objeto `Usuario` con nombre, correo y contraseña, lo guarda en Firebase mediante `guardar_usuario` y devuelve la instancia creada.

`iniciar_sesion`
Lee todos los usuarios desde Firebase, compara correo y contraseña, y si encuentra coincidencia devuelve un objeto `Usuario`; de lo contrario, retorna `None`.

`obtener_usuario_por_correo`
Busca en Firebase un usuario cuyo correo coincida.
Si lo encuentra, retorna un diccionario con su id y datos completos; si no, devuelve `None`.

ChatViewModel.py

- **Descripción**

Administra la instancia de chat para cada usuario (patrón singleton por `user_id`), guarda el historial de mensajes y gestiona el envío asíncrono de mensajes al servicio de IA.

- **Métodos**

`__new__`
Implementa un singleton por usuario: si ya existe una instancia para ese `user_id`, la reutiliza; si no, crea una nueva.

`__init__`
Evita reinicializar una instancia existente. Configura el `user_id`, el servicio de chat, el historial y marca la instancia como inicializada.

`send_message`
Agrega el mensaje del usuario al historial y lanza un hilo en segundo plano para enviar la conversación al modelo de IA.
Cuando llega la respuesta, la agrega al historial y ejecuta `on_success`.
Si ocurre un error, llama `on_error` o retorna el mensaje de error por `on_success`.

`clear_history`
Reinicia completamente el historial de la conversación.

6. Carpeta View

BasedView.py

- **Descripción**

`BaseView` es la ventana principal de la aplicación. Administra el diseño general, la navegación entre vistas, la aplicación de tema visual, la barra lateral, el encabezado, el manejo de teclas y los distintos easter eggs. Sirve como base para todas las ventanas de usuario y centraliza la lógica de interacción general.

- **Métodos de la clase**

`__init__(titulo, ancho, alto, usuario)`
Configura la ventana principal, carga el usuario, inicializa el gestor de temas, crea el menú lateral, el encabezado y el contenedor principal. También activa los detectores de teclas para easter eggs.

- **Métodos de tema**

`cambiar_tema()`
Alterna entre tema claro y oscuro y actualiza los colores de la interfaz.

`aplicar_tema()`
Aplica los colores del tema activo a todos los elementos principales de la vista.

- **Métodos de navegación**

`toggle_menu()`
Muestra u oculta el menú lateral.

`mostrar_vista(vista_class, **kwargs)`
Carga dinámicamente una vista dentro del contenedor principal.

`ir_inicio()`
Carga la vista principal.

`ir_tiendas()`
Carga la vista de tiendas.

`ir_configuracion()`
Carga la vista de configuración y pasa los callbacks correspondientes.

`LogOut()`
Cierra la sesión actual y vuelve a la vista de inicio de sesión.

`salir()`
Cierra la aplicación previa confirmación del usuario.

- **Teclas y easter eggs**

`detectar_tecla(event)`
Detecta cada tecla presionada y la envía a los detectores de Konami y Halo Code.

`activar_doom()`
Ejecuta el easter egg de DOOM.

`efecto_alerta_doom()`
Aplica un efecto visual temporal en el encabezado para acompañar la activación del easter egg.

`activar_easter_egg()`
Activa un easter egg visual dependiente de SDL.

- **Audio**

`reproducir_halo_music()`
Reproduce el tema de Halo usando la biblioteca SDL_mixer si está disponible.

- **Chat**

`ir_chat()`
Inicializa el ViewModel del chat si es necesario y carga la vista correspondiente.

- **Facturación**

`ir_facturar()`
Carga la vista de facturación.

ChatView.py

- **Descripción**

`ChatView` es la interfaz gráfica del módulo de chat. Muestra el historial de conversación, permite enviar mensajes, ajustar la temperatura del modelo y visualizar respuestas del asistente. Gestiona el renderizado del chat, los controles, la entrada del usuario y la aplicación del tema visual.

- **Métodos de la clase**

`__init__(parent, usuario, view_model, …)`
Inicializa la vista, aplica el tema, construye la interfaz y carga el historial del chat.

- **Aplicación de tema**

`_apply_theme()`
Obtiene los colores desde `ThemeManager` y los aplica a la vista principal.

- **Interfaz gráfica**

`_build_ui()`
Construye todos los elementos de la ventana: encabezado, área de chat, entrada de texto, botones, controles y slider de temperatura.

- **Auxiliares**

`_append_chat(prefix, text, prefix_color)`
Inserta una línea en el área de chat con formato diferenciado entre usuario y bot.

`_set_status(text, color)`
Actualiza el texto de estado mostrado debajo del chat.

`_set_controls_state(enabled)`
Activa o desactiva los botones y la entrada de mensaje.

- **Eventos**

`_on_send_clicked(event)`
Obtiene el texto del usuario, lo muestra en el chat y envía el mensaje al ViewModel.

`_on_response(msg)`
Muestra la respuesta del modelo en el chat y habilita nuevamente los controles.

`_on_error(error_msg)`
Muestra un mensaje de error en el chat y reactiva los controles.

`_on_clear()`
Limpia el historial del chat tanto visual como internamente.

- **Actualización visual**

`_on_theme_change()`
Reaplica todos los colores a la interfaz cuando el tema cambia.

- **Historial**

`render_chat_history()`
Vuelve a dibujar el historial completo del chat en el área de texto.

- **Envío directo**

`send_message()`
Envía un mensaje utilizando el texto actual del campo de entrada y actualiza el historial luego de la respuesta.

FacturarView.py

- **Descripción**

`FacturaView` es la vista encargada de gestionar todo el módulo de facturación. Permite crear nuevas facturas, listar ventas por tienda, mostrar detalles, eliminar ventas y actualizar el historial. Utiliza un ViewModel para manejar la lógica y ThemeManager para aplicar colores.

- **Métodos de la clase**

`__init__(parent, usuario, volver_callback)`
Inicializa la vista, crea el ViewModel, aplica los colores del tema y construye la interfaz.

`apply_theme_colors()`
Obtiene los colores actuales del tema y los guarda para su uso en la interfaz.

`render_ui()`
Genera toda la estructura visual: encabezado, botones principales, área desplazable de historial y carga inicial de ventas.

`mostrar_historial()`
Lista todas las tiendas del usuario y muestra sus ventas asociadas. Si no hay datos, muestra mensajes informativos.

- **Facturación**

`nueva_factura()`
Abre una ventana para crear una factura nueva. Permite seleccionar tienda, producto, cantidad, agregar/quitar ítems al carrito, calcular totales y guardar la factura.

- **Acciones internas de nueva factura (definidas dentro de `nueva_factura`)**

- `cargar_productos_para_tienda()`
  Carga productos de la tienda seleccionada y actualiza los menús desplegables.

- `actualizar_stock()`
  Muestra el stock disponible según el producto seleccionado.

- `agregar_item_ui()`
  Añade un producto al carrito validando stock y cantidad.

- `quitar_item_ui()`
  Elimina un producto del carrito y actualiza el total.

- **`facturar_ui()`**
  Guarda la factura completa y recarga el historial.

- **Detalles y eliminación**

`ver_detalle(id_tienda, id_venta)`
Muestra una ventana con la información completa de una venta: fecha, total e ítems.

`eliminar_venta(id_tienda, id_venta)`
Elimina una venta tras confirmación del usuario y actualiza la lista de historial.

LoginView.py

- **Descripción**

`LoginView` es la ventana principal de inicio de sesión. Permite al usuario ingresar con correo y contraseña, mostrar/ocultar la clave.

- **Métodos de la clase**

`__init__()`
Configura la ventana de inicio de sesión, crea el ViewModel de usuario, centra la ventana y construye los campos de correo, contraseña y botones de acción.

`centrar_ventana(ancho, alto)`
Posiciona la ventana en el centro de la pantalla.

`toggle_password()`
Activa o desactiva la visibilidad de la contraseña en el campo correspondiente.

`abrir_registro()`
Cierra la ventana actual y abre la vista de registro de usuario.

`login()`
Obtiene los datos ingresados, valida que no estén vacíos, solicita autenticación al ViewModel y, si es exitosa, abre la vista principal (`MainView`).

ResgisterView.py

- **UI**

Título “Crear una cuenta nueva”
Campos:
Nombre completo

Correo electrónico
Contraseña
Confirmar contraseña
Checkbox “Mostrar contraseñas”
Botón “Registrar”
Botón “Volver al inicio de sesión”

- **Métodos**

`__init__`
Crea ventana, configura colores, instancia ViewModel, construye campos del formulario y botones.

`centrar_ventana`
Centra la ventana en la pantalla según ancho y alto.

`toggle_password`
Alterna entre mostrar/ocultar caracteres en los campos de contraseña.

`registrar_usuario`
Valida campos y formato del correo, verifica seguridad de la contraseña, registra usuario mediante ViewModel y redirige al login si es exitoso.

`volver_login`
Cierra la vista actual y abre la pantalla de inicio de sesión.

ProductView.py

- **Imports clave**
  - **customtkinter**
  - **PIL / requests**
  - **filedialog**
  - **ImageStorage (Supabase)**
  - **sdl2 / sdlmixer**
  - **ProductoViewModel**
  - **ThemeManager**.

- **UI**
  - **Título de la vista**
  - **Botón “Agregar Producto”**
  - **ScrollableFrame** donde se renderizan todos los productos.
  - **Botón “Volver”**
    Cada producto se muestra con:

    - Imagen descargada desde URL.
    - Nombre, precio, stock y ubicación.
    - Botón “Editar”
    - Botón de eliminar (icono de basura)
    - Evento de sonido (solo para el producto demo)

- Métodos

`__init__`

Inicializa tema, almacenamiento, ViewModel y dibuja la interfaz.

`render_ui`

Construye toda la estructura visual de la página.

`play_demo_sound`

Reproduce un archivo WAV al hacer hover sobre el producto demo.

`cargar_productos`

Carga los productos, crea sus tarjetas visuales e inserta el producto demo “test”.

`agregar_producto`

Abre ventana para agregar un producto con nombre, precio, stock, ubicación e imagen opcional. Sube imagen a Supabase.

`editar_producto(id_prod)`

Abre ventana para editar valores del producto y cambiar su imagen (elimina la anterior si se reemplaza).

`eliminar_producto(id_prod)`

Verifica confirmación y elimina el producto en la base de datos.

TiendaView.py

- **Imports clave**

    - **customtkinter**
    - **messagebox**
    - **TiendaViewModel**
    - **ProductoView**
    - **ThemeManager**

- **UI**

  Título de la vista

  Botón “+ Nueva Tienda”

  ScrollableFrame donde se listan todas las tiendas

  Cada tienda se muestra con:

    - Nombre, categoría y dirección
    - Botón de opciones “⋮ ” que despliega:
        - **Editar**
        - **Eliminar**
        - **Ver productos**

- **Métodos**

`__init__`

Inicializa ViewModel, tema y dibuja la interfaz.

`apply_theme_colors`

Carga los colores actuales del tema.

`render_ui`

Construye la estructura visual principal y coloca el botón de nueva tienda y el contenedor scrollable.

`mostrar_tiendas`

Carga las tiendas del usuario y genera sus tarjetas visuales con botón de opciones.

`menu_opciones(id_tienda, tienda_data)`

Muestra un menú modal con opciones para editar, eliminar o ver productos de la tienda.

`agregar_tienda`

Abre ventana para registrar una nueva tienda y guarda los datos en Firebase.

`editar_tienda(id_tienda, tienda_data)`

Abre ventana para editar nombre, categoría y dirección de la tienda.

`eliminar_tienda(id_tienda, tienda_data)`

Pide confirmación y elimina la tienda.

`ver_productos(id_tienda)`

Oculta la vista actual y abre la vista de productos de la tienda seleccionada.

MainView.py

- **UI**
  - Texto de bienvenida con el nombre del usuario.
  - Mensaje adicional motivacional.
  - Carga del logo desde `assets/logo_light.png` y `assets/logo_dark.png` (si faltan, muestra texto alternativo).
  - Texto descriptivo explicando que esta es la pantalla principal de la aplicación.
- **Métodos**

`__init__(self, usuario)`

  - Inicializa BaseView con el usuario.
  - Aplica colores según ThemeManager.
  - Construye la tarjeta central y todos sus elementos: bienvenida, logo, descripción.

MainFrameView.py

- **Descripción**

Es la misma interfaz que MainView, pero diseñada como un CTkFrame en lugar de una ventana completa. Se usa para mostrar la pantalla de inicio dentro del contenedor principal sin abrir nuevas ventanas.

ConfigView.py

- **Descripción**

Vista simple de configuración incluida como CTkFrame. Permite dos acciones básicas: cambiar el tema de la aplicación y cerrar sesión.

- **Elementos principales**
  • Título “Configuración” en la parte superior
  • Botón “Cambiar tema” que ejecuta la función recibida
  • Botón “Cerrar sesión” que ejecuta la función recibida
- **Métodos**

  `__init__`

  Crea el frame, guarda callbacks recibidos y construye la interfaz con los botones de configuración.

SplashView.py

- **Métodos**

`__init__`

Configura la ventana del splash, aplica estilo, genera la interfaz base y programa el inicio de la animación.

`center_window`

Calcula y ubica la ventana en el centro exacto de la pantalla.

`create_ui`

Construye todos los elementos visibles (título, subtítulo, barra de progreso y etiqueta del porcentaje).

`animate_loading`

Crea un hilo que aumenta el progreso paso a paso usando `time.sleep()`; actualiza la barra y el porcentaje sin bloquear la UI. Cuando completa la animación, solicita la apertura del login.

`GoToLogin`

Cierra la pantalla splash y lanza la vista de inicio de sesión (`LoginView().mainloop()`).