INSTITUTO POLITÉCTICO NACIONAL

Arquitectura de SW

Aplicación móvil

SOLEYA

Una aplicación para la búsqueda y compra de zapatos dentro de su gran catálogo, disponible para toda la comunidad que busque el calzado que se ajuste a sus necesidades como consumidor.

Elaborado por el Equipo 1: *Mewtwo*

Integrantes:

- **Gómez Pliego Carlos Uriel**
- **Gonzales Sabas Luis Ever**
- **Jiménez Rogel Sergio**
- **Mondragón Mejía Ángel Amed**
- **Paz Alonso Gabriel**
- **Pérez Galeana Jemmy Alondra**

Profesor: Gustavo Martínez Vázquez

Fecha: 17 de junio de 2025

INDICE

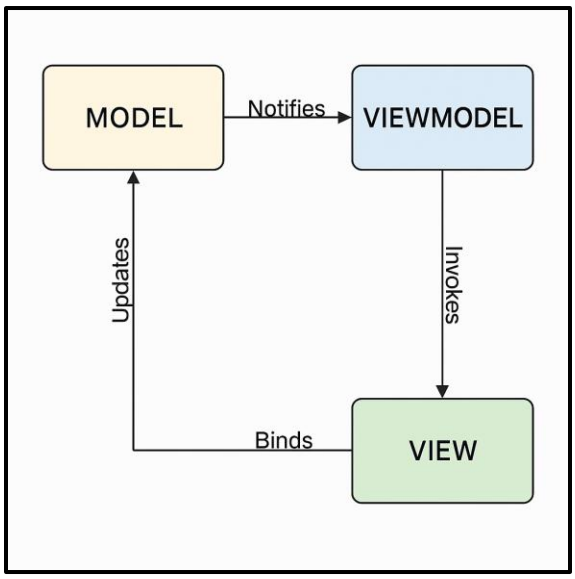
1 INTRODUCCION

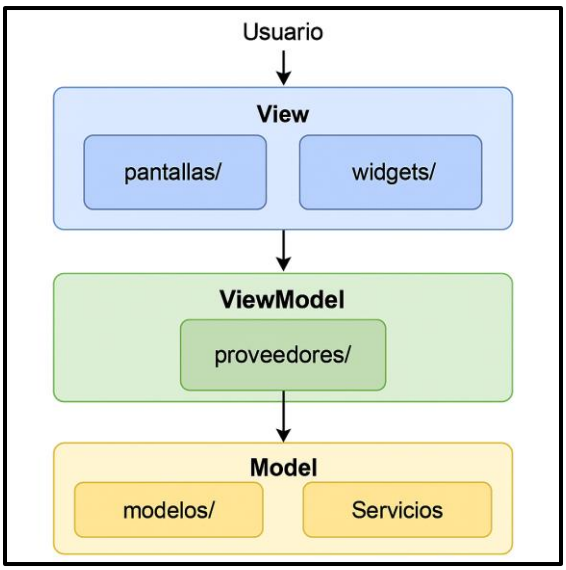
Este documento detalla la arquitectura de software de una aplicación móvil desarrollada en Flutter, enfocada en ofrecer una experiencia fluida, moderna e interactiva al usuario. La app está orientada a la gestión de productos, favoritos, mapas y bases de datos híbridas (local y en la nube), usando tecnologías como Firebase, Google Maps, SQLite y múltiples componentes personalizados.

El diseño sigue principios de modularidad, reutilización y escalabilidad usando el patrón arquitectónico MVVM (Model-View-ViewModel).

2 ARQUITECTURA MVVM (MODEL, VIEW, MODEL-VIEW)

La aplicación "Soleya" fue diseñada bajo una arquitectura modular basada en el patrón **MVVM (Model-View-ViewModel)**. Esta arquitectura permite una separación clara entre la lógica de negocio, la interfaz gráfica y el manejo del estado, facilitando el mantenimiento, escalabilidad y pruebas del sistema.





2.1 Funcionamiento

El modelo **MVVM (Model-View-ViewModel)** organiza el código de la aplicación en tres capas principales que trabajan juntas pero están desacopladas:

1. **Model (Modelo):**
   Maneja la estructura de los datos y su persistencia (Firebase, SQLite).
   En esta aplicación, incluye archivos de la carpeta modelos, Servicios y db_local, encargados de representar productos, usuarios y gestionar operaciones de lectura y escritura.
2. **ViewModel (Vista-Modelo):**
   Gestiona la lógica de negocio y de presentación.
   Es responsable de exponer los datos listos para usar por la interfaz de usuario y de manejar eventos del usuario.
   Se implementa en proveedores/ con clases como CartProvider, FavoritosModel, o ProductProvider.
3. **View (Vista):**
   Representa la interfaz visual. Consume los datos expuestos por el ViewModel y escucha sus cambios.
   Las pantallas (pantallas/) y widgets (widgets/) presentan estos datos al usuario, y responden a interacciones como toques o desplazamientos.

Cada capa se comunica solo con la siguiente adyacente, lo cual promueve una estructura limpia y mantenible.

2.2 Propósito

El patrón MVVM en esta aplicación tiene como objetivos:

- **Separar responsabilidades:** cada capa se enfoca en su tarea específica (UI, lógica, datos).
- **Mejorar el mantenimiento del código:** es más fácil corregir errores o extender funcionalidades sin afectar otras partes.
- **Facilitar pruebas:** al estar separadas, se pueden realizar pruebas unitarias en los ViewModel sin depender de la UI.
- **Permitir una UI reactiva:** con Provider, las vistas se actualizan automáticamente cuando el estado cambia.

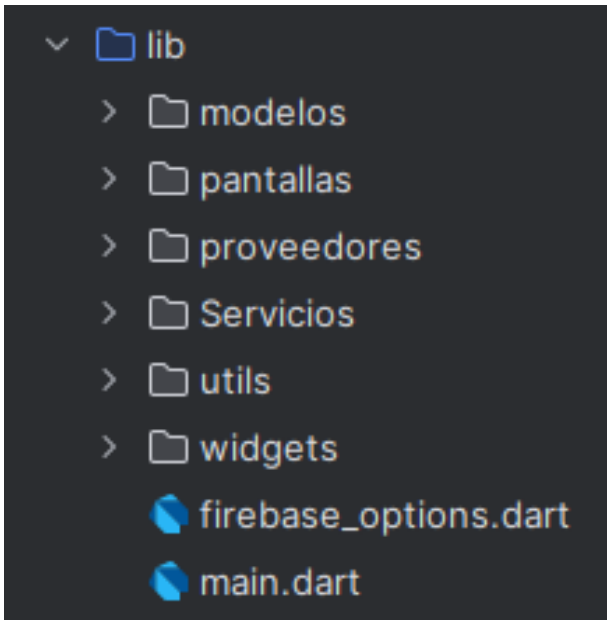
2.3 Escalabilidad

El uso de MVVM permite que esta app escale en complejidad y funcionalidad sin volverse difícil de mantener. Algunos beneficios de escalabilidad:

- **Nuevas funcionalidades** (por ejemplo, métodos de pago, historial de compras) se pueden agregar implementando nuevos modelos, servicios y ViewModels, sin alterar pantallas existentes.
- **Soporte multiplataforma** (por ejemplo, web, desktop) se facilita gracias a la separación de lógica y presentación.
- **Múltiples desarrolladores** pueden trabajar de forma simultánea: uno en la interfaz (View), otro en la lógica (ViewModel), y otro en el backend (Model), sin interferirse.

Esta arquitectura modular también permite migrar partes del sistema a otras tecnologías en el futuro si fuera necesario (por ejemplo, migrar Firebase por otro backend).

3 ESTRUCTURA DEL PROYECTO



4 DESCRIPCION DE LOS ARCHIVOS EN LA ARQUITECTURA

4.1 Vista (View)

Estas carpetas contienen las interfaces gráficas y componentes visuales de la aplicación.

4.1.1 lib/screens/

Contiene las pantallas principales que el usuario ve y con las que interactúa. Cada archivo representa una vista completa:

- **welcome_screen.dart**: Pantalla inicial de bienvenida.
- **login_screen.dart y registro_screen.dart**: Interfaces para autenticación.
- **inicio/ (inicio_content.dart, product_detail_screen.dart)**: Página de inicio y detalles de producto.
- **busquedascreen.dart**: Pantalla con buscador de productos usando autocompletado.
- **favoritos_screen.dart**: Muestra la lista de productos marcados como favoritos.
- **cart_screen.dart, envio_screen.dart, pago_screen.dart, resumen_screen.dart**: Flujo de carrito, envío, pago y resumen de compra.
- **perfil_screen.dart, mis_compras_screen.dart**: Pantallas relacionadas al perfil del usuario y su historial de compras.

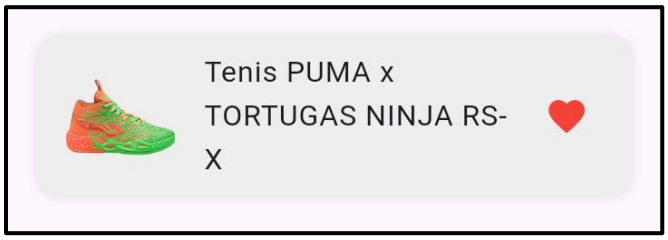
4.1.2 lib/widgets/

Componentes visuales reutilizables que forman parte de las vistas:

- **custom_bottom_nav_bar.dart**: Barra de navegación inferior con iconos e indicador de productos en carrito.
- **animated_favorite_icon.dart**: Icono animado de favorito.
- **custom_app_bar.dart**: Barra superior personalizada.
- **loading_indicator.dart**: Indicador de carga.
- **product_card.dart, suggestion_card.dart, size_selector.dart**: Tarjetas e interfaces específicas para mostrar productos.

Ejemplo con la lista de productos en favoritos (Código y representación visible)

```
child: ListView.builder(
  physics: const BouncingScrollPhysics(),
  shrinkWrap: true,
  itemCount: _items.length,
  itemBuilder: (context, index) {
    return SlideFadeInFromBottom(
      delay: Duration(milliseconds: 100 * (index + 1)),
      child: _buildFavoriteTile(_items[index]),
    );  // SlideFadeInFromBottom
  },
),  // ListView.builder
```



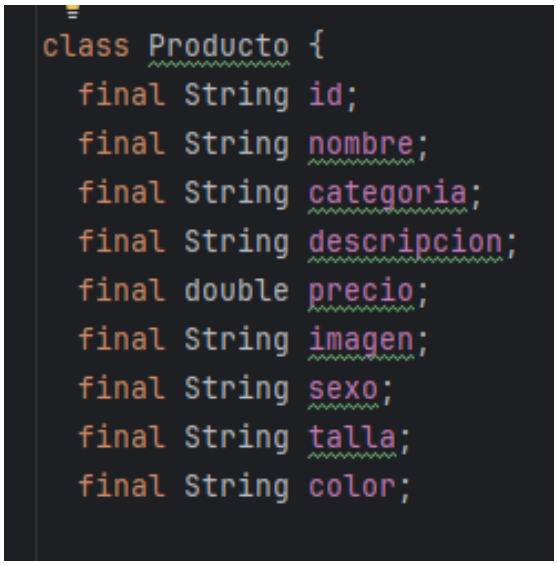
4.2 Modelo (Model)

Contiene clases que definen la estructura de los datos utilizados en la app. Se comunican con las bases de datos (Firestore o SQLite).

- **producto_model.dart**: Modelo para los productos (nombre, precio, imagen, etc.).
- **usuario_model.dart**: Modelo de datos del usuario.
- **favorito_model.dart**: Modelo para representar productos marcados como favoritos.
- **carrito_model.dart**: Modelo para los productos en el carrito de compras.
- **talla_model.dart**: Modelo para representar las tallas disponibles de productos.

Estas clases encapsulan los datos y ofrecen métodos como fromMap() y toMap() para integrarse con Firebase y SQLite.

Ejemplo: Modelo para definir la estructura de datos de un producto, incluye funciones para convertir desde/hacia mapas (Map<String, dynamic>) para Firestore o SQLite.



```dart
class Producto {
  final String id;
  final String nombre;
  final String categoria;
  final String descripcion;
  final double precio;
  final String imagen;
  final String sexo;
  final String talla;
  final String color;
```

```dart
Map<String, dynamic> toMap() {
  return {
    'Nombre': nombre,
    'Categoria': categoria,
    'Descripcion': descripcion,
    'Precio': precio,
    'Imagen': imagen,
    'Sexo': sexo,
    'Talla': talla,
    'Color': color,
  };
}
```

4.3 Vista-Modelo (ViewModel)

Encargados de gestionar el estado de la aplicación y hacer la conexión entre las vistas y los modelos.

- **favoritos_model.dart**: Mantiene la lista de productos favoritos. Permite agregar, quitar y verificar si un producto está marcado.
- **cart_provider.dart**: Controla los productos en el carrito. Permite agregar, eliminar y actualizar cantidades. Expone items, total y funciones como agregarProducto() o limpiarCarrito().
- **product_provider.dart**: Provee acceso a productos, sus detalles y lógica relacionada como búsqueda, sugerencias y selección de tallas.

Todos estos archivos usan ChangeNotifier para notificar cambios en el estado a las vistas mediante Provider.

Ejemplo: Encapsula la lógica de agregar y quitar favoritos, y notifica a la vista cuando cambia el estado.

La vista reacciona automáticamente gracias a ChangeNotifier.

```dart
class FavoritosModel extends ChangeNotifier {
  // Lista de productos favoritos
  List<Producto> _favoritos = [];

  // Devuelve los productos favoritos
  List<Producto> get favoritos => _favoritos;

  // Verifica si un producto es favorito
  void agregarFavorito(Producto producto) {
    if (!_favoritos.contains(producto)) {
      operaciones_db().addFavorito(producto);
      _favoritos.add(producto);
      //Subir a la db en la nube
      //Base local

      notifyListeners(); // Notifica a los consu
    }
  }
```

4.4 Conexión con datos y servicios

4.4.1 lib/Servicios/

Contiene lógica de negocio y acceso a servicios externos:

- firestore_service.dart: Funciones para interactuar con Firestore (consultas, escritura, inicio de sesión y registro).

4.4.2 db_local.dart

Base de datos local con SQLite. Implementa funciones para:

- Leer y escribir favoritos.
- Guardar historial de productos vistos.
- Persistencia local del carrito.

Ejemplo: Obtención e inserción de datos dentro de la base de datos local de SQLite

```dart
Future<List<Map<String, dynamic>>> mostrarUsuarios() async{
  final local = await sqlite_db;
  return await local.query('usuario', columns: ['user_uid', 'nombre', 'correo', 'avatar', 'password']);
}

Future<void> setUsuarioLocal(Map<String, dynamic>? data, String password) async{
  //final user_id = _auth.currentUser?.uid;
  final local = await sqlite_db;
  final existencia = await local.query('usuario',
      columns: ['user_uid'],
      where: 'user_uid = ?',
      whereArgs: [user_id]
  );
  //print('Existe? $existencia');

  if(existencia.isEmpty){
    await local.insert(
        'usuario',
        {
          'user_uid': user_id,
          'nombre': data?['nombre'],
          'apellido': data?['apellido'],
          'correo': data?['correo'],
          'avatar': null,
          'password': password
        },
        conflictAlgorithm: ConflictAlgorithm.replace);
  }
}
```