# 1. Introducción

La empresa MASS, dedicada a la venta de productos de consumo masivo, actualmente opera a través de su página web. Con el objetivo de mejorar la accesibilidad y brindar una experiencia más cómoda a sus usuarios, han solicitado el desarrollo de un aplicativo móvil que permita realizar compras, acceder a promociones y gestionar pedidos de manera efectiva.

## Objetivos del Proyecto

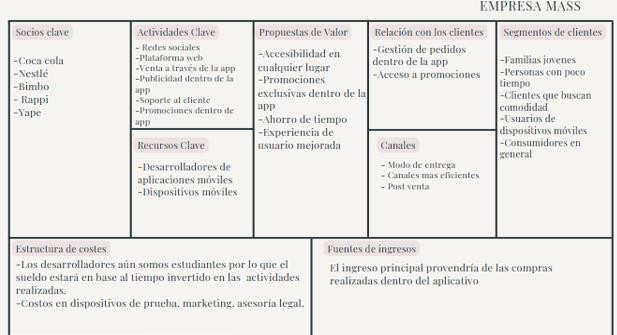
Desarrollar un aplicativo móvil para la empresa mediante esta se podrá realizar compras, promociones y gestionar pedidos de manera eficiente, complementando la experiencia ofrecida por la plataforma web.

1.1. Objetivos específicos  
  
Diseñar una interfaz intuitiva y adaptable que facilite la navegación y el proceso de compra desde dispositivos móviles.

Implementar funcionalidades clave como registro de usuario, catálogo de productos, carrito de compras, historial de pedidos y promociones.

Optimizar el rendimiento de la aplicación para dispositivos Android, asegurando una experiencia fluida.

## Modelo de Negocio en tiendas MASS

****

## Alcance del proyecto:

### Arquitectura del Sistema

La aplicación ha sido desarrollada siguiendo la arquitectura Clean Architecture con el patrón MVVM (Model-View-ViewModel), lo que permite una clara separación de responsabilidades y facilita el mantenimiento y escalabilidad del código. Esta arquitectura divide la aplicación en capas claramente definidas, cada una con responsabilidades específicas y con dependencias que fluyen desde las capas externas hacia las internas. La implementación de esta arquitectura ha permitido un desarrollo modular y testeable, facilitando la incorporación de nuevas funcionalidades y la corrección de errores.

### Creación de Base de Datos Web del Proyecto

Para la implementación de la base de datos web, se ha establecido una robusta conexión con servicios REST que interactúan con el backend. La configuración de red se ha realizado utilizando Retrofit para las llamadas a la API, OkHttp para la gestión de las conexiones HTTP y Gson para la serialización y deserialización de datos JSON. Adicionalmente, se ha implementado un interceptor personalizado (AuthInterceptor) que se encarga de añadir los tokens de autenticación a las solicitudes que lo requieren, mejorando así la seguridad de las comunicaciones.

El módulo de red está configurado con tiempos de espera adecuados para garantizar una experiencia de usuario óptima incluso en condiciones de red desfavorables. Se han implementado mecanismos de manejo de errores que proporcionan mensajes claros y útiles para el usuario final, diferenciando entre errores de conexión, errores del servidor y errores de autenticación.

La configuración de red incluye un sistema de logging que facilita la depuración durante el desarrollo, registrando tanto las solicitudes como las respuestas, lo que ha sido fundamental para identificar y resolver problemas durante la fase de implementación.

## 3.3 Creación de la API para el Servicio Web

La interfaz de API se ha implementado de manera exhaustiva, cubriendo todos los endpoints necesarios para la funcionalidad completa de la aplicación. La API proporciona métodos para autenticación (registro y login), gestión de productos, categorías, carrito de compras y órdenes. Cada endpoint está cuidadosamente documentado y tipado, lo que facilita su uso y reduce la posibilidad de errores.

La API implementa un sistema de autenticación basado en tokens JWT, donde el token se almacena de forma segura utilizando DataStore con encriptación. Los endpoints protegidos verifican automáticamente la validez del token y manejan adecuadamente los casos de tokens expirados, redirigiendo al usuario a la pantalla de login cuando es necesario.

Se han implementado modelos de solicitud (Request) y respuesta (Response) específicos para cada endpoint, siguiendo las mejores prácticas de diseño de API REST. Estos modelos se convierten a modelos de dominio antes de ser utilizados por la aplicación, lo que proporciona una capa adicional de abstracción y seguridad.  
  
3.4 Código de Consumo de la API en la App

El consumo de la API se ha implementado siguiendo el patrón Repository, que proporciona una abstracción sobre las fuentes de datos. Cada repositorio se encarga de obtener datos de la API, convertirlos al formato adecuado para la aplicación y manejar los posibles errores.

La implementación utiliza corrutinas de Kotlin para realizar operaciones asíncronas de manera eficiente, evitando bloquear el hilo principal y proporcionando una experiencia de usuario fluida. Los resultados de las operaciones se devuelven utilizando la clase Result de Kotlin, que encapsula tanto el resultado exitoso como los posibles errores, facilitando el manejo de errores en las capas superiores.

Los datos obtenidos de la API se almacenan temporalmente en la aplicación utilizando diferentes mecanismos según sus características y requisitos de persistencia. Para datos de sesión y preferencias de usuario se utiliza DataStore, que proporciona una API moderna y basada en corrutinas para el almacenamiento de datos clave-valor. Para datos más complejos como el carrito de compras se utiliza un enfoque en memoria con respaldo en DataStore para persistencia entre sesiones.

### 3.5 Uso de RecyclerView en la App

La implementación del RecyclerView se ha realizado de manera sofisticada, soportando múltiples tipos de vista (grid y lista) y utilizando el patrón ViewHolder para una reutilización eficiente de las vistas. Se ha implementado DiffUtil para optimizar las actualizaciones de la lista, calculando las diferencias entre la lista actual y la nueva, y realizando solo las actualizaciones necesarias.

El adaptador del RecyclerView (ProductsAdapter) soporta dos tipos de vista diferentes: grid y lista, que pueden cambiarse dinámicamente según las preferencias del usuario. Cada tipo de vista tiene su propio ViewHolder con una implementación específica del método bind que configura la vista con los datos del producto.

Para la carga de imágenes se utiliza la biblioteca Picasso, que proporciona una API sencilla para la carga y caché de imágenes. La implementación incluye manejo de errores y placeholders para mejorar la experiencia de usuario en caso de problemas con la carga de imágenes.

### 3.6 Creación de Pantalla con Reproductor de Video Promocional

Se ha implementado una sección de video promocional sofisticada que utiliza la API oficial de YouTube para Android (com.pierfrancescosoffritti.androidyoutubeplayer) para reproducir videos de manera nativa dentro de la aplicación. Esta implementación proporciona una experiencia de usuario superior a la alternativa de WebView, con mejor rendimiento y más opciones de personalización.

El componente de video está encapsulado en un Composable (VideoSection) que maneja todo el ciclo de vida del reproductor, desde la inicialización hasta la liberación de recursos. El componente incluye una interfaz de usuario personalizada con controles de reproducción, indicadores de carga y manejo de errores.

La implementación también incluye una vista previa del video cuando no está reproduciéndose, con un diseño atractivo que invita al usuario a iniciar la reproducción. Esta vista previa se implementa utilizando un gradiente personalizado y animaciones sutiles para mejorar la experiencia visual.

## Tecnologías Utilizadas

### Jetpack Compose

Jetpack Compose ha sido utilizado extensivamente para la creación de interfaces de usuario modernas y declarativas. Esta tecnología ha permitido desarrollar interfaces complejas con menos código y mayor flexibilidad que el enfoque tradicional basado en XML. Las pantallas principales de la aplicación, como la pantalla de inicio (HomeScreen), la pantalla de autenticación (AuthScreen) y la pantalla de checkout (CheckoutScreen) están implementadas completamente con Compose.

La implementación con Compose incluye animaciones sofisticadas, como transiciones entre pantallas, animaciones de carga y efectos visuales que mejoran la experiencia de usuario. Se han utilizado componentes personalizados como GlassmorphicCard y BentoCard para crear una interfaz distintiva y atractiva.

El estado de la UI se gestiona utilizando el patrón de estado unidireccional (Unidirectional Data Flow) con StateFlow y corrutinas, lo que proporciona una gestión de estado predecible y fácil de razonar. Los ViewModels exponen el estado a través de StateFlow, y las pantallas Compose observan estos flujos y actualizan la UI en consecuencia.

### Kotlin y Corrutinas

Kotlin ha sido el lenguaje principal de desarrollo, aprovechando sus características modernas como funciones de extensión, delegados de propiedades, clases de datos y null safety para crear un código más conciso, expresivo y seguro. Las corrutinas de Kotlin se han utilizado extensivamente para manejar operaciones asíncronas de manera eficiente, reemplazando enfoques más tradicionales como callbacks o RxJava.

Las corrutinas se utilizan en todas las capas de la aplicación: en la capa de datos para realizar llamadas a la API, en la capa de dominio para ejecutar casos de uso, y en la capa de presentación para realizar operaciones asíncronas sin bloquear el hilo principal. El uso de corrutinas ha simplificado significativamente el código asíncrono, haciéndolo más legible y mantenible.

## Koin para Inyección de Dependencias

Koin ha sido utilizado como framework de inyección de dependencias, proporcionando una solución ligera y pragmática para la gestión de dependencias en la aplicación. La configuración de Koin se organiza en módulos temáticos (networkModule, repositoryModule, useCaseModule, viewModelModule, sessionManagerModule) que facilitan la organización y el mantenimiento del código.

La inyección de dependencias con Koin ha permitido una mayor modularidad y testabilidad del código, facilitando la sustitución de implementaciones reales por mocks durante las pruebas. Además, ha simplificado la gestión del ciclo de vida de los objetos, especialmente en el caso de los ViewModels, que se crean y destruyen automáticamente siguiendo el ciclo de vida de los componentes de UI.

## Retrofit, OkHttp y Gson

Para la comunicación con la API REST se ha utilizado Retrofit en combinación con OkHttp y Gson. Retrofit proporciona una interfaz tipo-segura para definir los endpoints de la API, OkHttp maneja las conexiones HTTP de bajo nivel, y Gson se encarga de la serialización y deserialización de los datos JSON.

La configuración de Retrofit incluye interceptores personalizados para añadir headers de autenticación, logging para facilitar la depuración, y timeouts configurables para manejar condiciones de red desfavorables. También se ha implementado un manejo sofisticado de errores que convierte los errores HTTP en excepciones tipadas que pueden ser manejadas adecuadamente por la aplicación.

## Almacenamiento Local

El almacenamiento local de la aplicación combina tecnologías modernas y robustas para gestionar datos estructurados y preferencias del usuario:

Room Database: Se ha implementado Room como ORM oficial de Android sobre SQLite para almacenar datos estructurados, como órdenes de compra, productos y sucursales. Room permite definir relaciones uno-a-muchos, migraciones entre versiones y consultas SQL type-safe.

DataStore: Como reemplazo moderno de SharedPreferences, DataStore se ha utilizado para almacenar datos clave-valor como tokens de autenticación, preferencias de usuario y configuraciones generales. DataStore proporciona una API basada en corrutinas que permite un acceso eficiente, seguro y reactivo.

Security Crypto: Los datos sensibles, como credenciales y tokens, se almacenan en DataStore utilizando encriptación a nivel de archivo, protegiendo la información contra accesos no autorizados.

Gestión de Sesiones: La clase SessionManager centraliza todas las operaciones relacionadas con la sesión del usuario. Administra el almacenamiento y recuperación de tokens JWT, implementa flujos reactivos con StateFlow para notificar cambios en tiempo real y soporta el refresco automático de tokens expirados.

Esta arquitectura permite un almacenamiento seguro, modular y escalable, manteniendo la separación entre datos persistentes complejos (Room) y preferencias o sesiones rápidas (DataStore).

## Carga de Imágenes: Migración a Coil y Jetpack Compose

Inicialmente, el proyecto contemplaba el uso de Picasso para la carga y gestión de imágenes desde URLs externas. Sin embargo, considerando que el desarrollo de la aplicación está basado en Jetpack Compose, el uso de Picasso resulta inadecuado por las siguientes razones:

Picasso está diseñado principalmente para trabajar con ImageView en vistas basadas en XML, no con composables de Compose.

El flujo de trabajo con Picasso implica más código boilerplate y adaptaciones innecesarias en un entorno Compose.

Por este motivo, se ha optado por migrar a Coil, una biblioteca moderna recomendada por Google y la comunidad Android para aplicaciones Kotlin y Compose.  
Coil ofrece las siguientes ventajas clave:

Carga de imágenes nativa para Jetpack Compose mediante el componente AsyncImage.

Soporte integrado para caché, transformaciones y placeholders sin código adicional.

Mayor rendimiento y menor uso de recursos comparado con bibliotecas tradicionales.

## YouTube Player API

Para la reproducción de videos de YouTube se ha utilizado la biblioteca com.pierfrancescosoffritti.androidyoutubeplayer, que proporciona una implementación nativa del reproductor de YouTube para Android. Esta biblioteca ofrece un control completo sobre la reproducción del video, incluyendo opciones de personalización de la UI, control de eventos y manejo de errores.

La implementación incluye un componente Composable (NativeYouTubePlayer) que encapsula el reproductor de YouTube y proporciona una API sencilla para su uso en pantallas Compose. Este componente maneja automáticamente el ciclo de vida del reproductor, liberando recursos cuando no se necesita.

## Seguridad y Manejo de Sesiones

La seguridad de la aplicación se ha implementado de manera robusta, con un enfoque especial en la protección de datos sensibles y la gestión segura de sesiones de usuario. El sistema de autenticación utiliza tokens JWT que se almacenan de forma segura utilizando DataStore con encriptación.

La clase SessionManager proporciona métodos para almacenar y recuperar información de sesión, verificar la validez de los tokens y manejar casos de tokens expirados. La implementación incluye un mecanismo de refresco automático de tokens cuando están a punto de expirar, evitando interrupciones en la experiencia de usuario.

Las contraseñas nunca se almacenan localmente, y todos los datos sensibles se transmiten utilizando conexiones HTTPS. Además, se han implementado mecanismos para detectar y prevenir ataques comunes como inyección SQL y cross-site scripting.

## Manejo de Estado y Flujo de Datos

El manejo de estado en la aplicación sigue el patrón de estado unidireccional (Unidirectional Data Flow), donde el estado fluye desde los ViewModels hacia la UI, y las acciones del usuario se propagan desde la UI hacia los ViewModels. Este enfoque proporciona un flujo de datos predecible y fácil de razonar, facilitando la depuración y el mantenimiento del código.

Cada ViewModel expone su estado a través de un StateFlow inmutable, que puede ser observado por la UI para reflejar cambios en el estado. Las acciones del usuario se manejan mediante métodos en el ViewModel, que pueden actualizar el estado o desencadenar efectos secundarios como llamadas a la API.

Este enfoque se complementa con el uso de clases de estado inmutables (como AuthUiState, HomeUiState, CartUiState) que encapsulan todo el estado necesario para renderizar una pantalla específica. Estas clases utilizan el patrón de copia de datos de Kotlin (copy) para crear nuevas instancias con cambios específicos, manteniendo la inmutabilidad del estado.

## Gestión Geográfica y Mapas

La aplicación incorpora funcionalidades de geolocalización y visualización de mapas con el objetivo de brindar a los usuarios información sobre las sucursales disponibles del marketplace y su ubicación en tiempo real.

Tecnologías implementadas:

OSMDroid: Biblioteca open-source que permite visualizar mapas basados en OpenStreetMap, proporcionando una alternativa ligera a Google Maps.

Google Play Services Location: Para obtener la ubicación actual del usuario con precisión y eficiencia.

Algoritmo de Haversine: Implementado para calcular distancias geográficas entre el usuario y las sucursales.

Funcionalidades clave:

Visualización de sucursales como marcadores en el mapa.

Solicitud y gestión de permisos de localización (ACCESS\_FINE\_LOCATION, ACCESS\_COARSE\_LOCATION) siguiendo las buenas prácticas de Android.

Cálculo de distancia para mostrar las sucursales más cercanas.

1. Procesamiento de Pagos

El sistema de pagos de la aplicación permite a los usuarios completar transacciones utilizando diversos métodos, buscando flexibilidad y seguridad en el proceso.

Métodos de pago soportados:

Yape: Aplicación de pagos móviles ampliamente utilizada en Perú.

Plin: Plataforma de pago digital.

Tarjetas de crédito/débito: Procesamiento mediante plataformas compatibles.

Funcionalidades clave:

Validación del stock antes de proceder con el pago.

Subida y validación de comprobantes digitales como evidencia de pago.

Verificación de transacciones para garantizar la integridad y éxito de las operaciones.

Seguridad:

La información de pago no se almacena localmente.

Las transacciones se realizan sobre conexiones seguras (HTTPS).

Se implementa un control de integridad para verificar que los pagos registrados correspondan efectivamente a las órdenes creadas.

PARA INICIAR EL PROYECTO  
  
  
 DEBEMOS INICIAR PHP   
  
A screenshot of a computer

AI-generated content may be incorrect.  
  
IMPORTAMOS LA BASE DE DATOS CON EL NOMBRE “supermercadomass.SQL”  
  
A screenshot of a computer

AI-generated content may be incorrect.  
  
  
LUEGO UTILIZAREMOS VISUAL STUDIO PARA ABRIR EL ARCHIVO SUPERMERCADO.RAR YA EXTRAIDO   
A screenshot of a computer

AI-generated content may be incorrect.  
abriremos una terminal para validar si tenemos instalado el composer en caso de no haberlo instalado   
  
https://getcomposer.org/  
  
instalarlo manualmente o utilizar power Shell para instalarlo la guía estará incluida en la pagina de arriba.  
  
  
  
validamos en visual studio la versión de Composer  
  
con el comando  
composer –version  
  
  
A screen shot of a computer

AI-generated content may be incorrect.  
  
  
Instalas globalmente el instalador de Laravel  
  
con el comando   
composer global require laravel/installer  
  
A screen shot of a computer program

AI-generated content may be incorrect.  
  
  
  
Iniciamos el servidor Laravel  
  
con el comando   
php artisan serve  
  
A black background with white dots

AI-generated content may be incorrect.  
  
ahora el servidor para aceptar conexiones externas  
con el comando  
  
php artisan serve --host=0.0.0.0  
  
A screen shot of a computer

AI-generated content may be incorrect.  
  
cómo se observa en la imagen la API generada con Swagger en la línea de documentos. Swagger UI es una interfaz web interactiva que permite visualizar y probar esos endpoints desde el navegador.  
  
podemos observar   
http:://localhost:8000/api/documentation  
  
A screenshot of a computer

AI-generated content may be incorrect.  
  
finalmente abrimos el proyecto Android para visualizar la aplicación  
  
A screenshot of a computer

AI-generated content may be incorrect.  
  
  
A screenshot of a computer

AI-generated content may be incorrect.  
  
  
**Referencias**

* Arquitectura Clean con MVVM en aplicaciones Android modernas
* Principios y patrones de diseño en el desarrollo de software
* Organización de proyectos Android con Clean Architecture
* Implementación de servicios REST en aplicaciones Android con Retrofit y OkHttp
* Diseño e implementación de APIs REST para aplicaciones móviles
* Patrones de acceso a datos en aplicaciones Android
* Optimización de RecyclerView para aplicaciones Android de alto rendimiento
* Integración de reproductores multimedia en aplicaciones Android
* Desarrollo de interfaces de usuario con Jetpack Compose
* Programación asíncrona con Corrutinas de Kotlin
* Inyección de dependencias en Android con Koin
* Comunicación con APIs REST en Android
* Almacenamiento de datos local en Android con DataStore
* Carga y caché de imágenes en Android con Picasso
* Integración de YouTube Player API en aplicaciones Android
* Seguridad y autenticación en aplicaciones Android
* Arquitectura de estado unidireccional en aplicaciones Android
* Mejores prácticas para el desarrollo de aplicaciones Android escalables