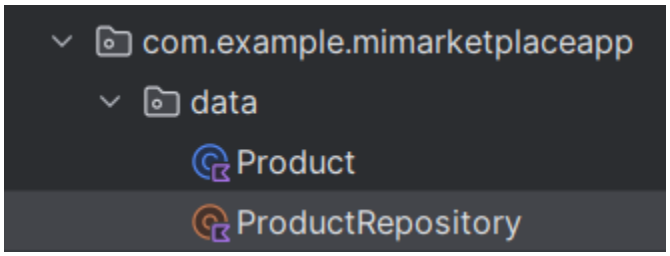
Desarrollo de Mi Marketplace (Android con Jetpack Compose)

Documentación del Paquete data (Modelo y Repositorio)

El paquete data es el Model de la arquitectura, encargado de definir la estructura de la información (qué es un producto) y la fuente de datos (de dónde se obtienen los productos).



1. Archivo Product.kt (El Modelo)

Este archivo define la estructura fundamental de los datos que la aplicación gestiona. Utiliza una Data Class de Kotlin, optimizada para almacenar datos y que automáticamente proporciona métodos como equals(), hashCode(), y toString().

```kotlin
package com.example.mimarketplaceapp.data

data class Product(
    val id: Int,
    val name: String,
    val shortDescription: String,
    val longDescription: String,
    val price: Double,
    val imageUrl: String
)
```

id	Int	Identificador único del producto (clave para la navegación).
name	String	Nombre del producto (ej. "Smartphone Pro X").

shortDescription	**String**	Descripción breve usada en la lista de productos.
longDescription	**String**	Descripción detallada usada en la pantalla de detalle.
price	**Double**	Precio del producto antes de aplicar formato de moneda.
imageUrl	**String**	URL o nombre de recurso de la imagen del producto.

2. Archivo Product Repository.kt (El Repositorio)

Esta clase implementa el patrón Repository. Su función es aislar la lógica de acceso a los datos del resto de la aplicación. En un proyecto real, esta clase gestionaría llamadas a una API o a una base de datos.

Se declara como un object para ser un Singleton, asegurando que solo exista una instancia del repositorio en toda la aplicación.

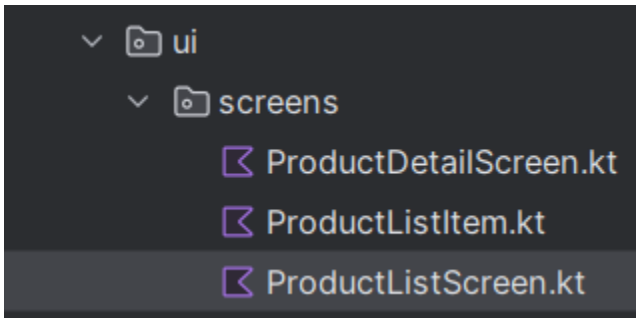
```
4 Usages
object ProductRepository {

    // Lista de productos simulados
    2 Usages
    private val products = listOf(
        Product( id = 1, name = "Smartphone Pro X", shortDescription = "Potente y con gran cámara.", long
        Product( id = 2, name = "Auriculares Wireless Go", shortDescription = "Sonido Hi-Fi sin cables.",
        Product( id = 3, name = "Smartwatch Lite SE", shortDescription = "Controla tu salud y actividad."
        Product( id = 4, name = "Laptop UltraBook 3000", shortDescription = "Delgada, ligera y rápida.",
        // NUEVOS PRODUCTOS
        Product( id = 5, name = "Teclado Mecánico RGB", shortDescription = "Máxima precisión para gamers.
        Product( id = 6, name = "Mouse Ergonómico Vertical", shortDescription = "Comodidad para largas jo
        Product( id = 7, name = "Monitor Curvo 27\"", shortDescription = "Inmersión total en 4K.", longDe
    )
```

getAllProducts()	List<Product>	**Devuelve la lista completa de productos para la pantalla principal.**

getProductById(id: Int)	Product	**Busca un producto por su ID. Es esencial para cargar los datos en la pantalla de detalle después de la navegación.**

Documentación del Paquete ui.screens (Vistas Composable)

Esta es la pantalla principal, diseñada para mostrar de manera eficiente el catálogo de productos y permitir la interacción básica de búsqueda y navegación.



1. Archivo ProductListScreen.kt (Lista de Productos y Búsqueda)

Esta es la pantalla principal, diseñada para mostrar de manera eficiente el catálogo de productos y permitir la interacción básica de búsqueda y navegación.

```
2 Usages
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun ProductListScreen(
    navController: NavHostController
) {
    // 1. Estado para almacenar el texto que escribe el usuario
    var searchText by remember { mutableStateOf( value = "") }

    // 2. Obtiene la lista completa de productos
    val allProducts = ProductRepository.getAllProducts()

    // 3. Aplica el filtro CADA VEZ que cambia el searchText
    val filteredProducts = remember( key1 = allProducts, key2 = searchText) {
        if (searchText.isBlank()) {
            allProducts // Si la búsqueda está vacía, devuelve todos
        } else {
            allProducts.filter { product ->
                // Filtra por nombre O por descripción breve (sin importar mayúsculas)
                product.name.contains( other = searchText, ignoreCase = true) ||
                        product.shortDescription.contains( other = searchText, ignoreCase = true)
            }
        }
    }
```

Lista Eficiente	**LazyColumn**	**Solo renderiza los elementos visibles, optimizando el rendimiento.**
Búsqueda en Tiempo Real	**remember { mutableStateOf("") }**	**Almacena el texto. remember(allProducts, searchText) recalcula la lista solo cuando cambia el texto, filtrando por nombre o descripción. (Extra Opcional: Filtros/Búsqueda)**
Navegación	**navController.navigate("product_detail/${product.id}")**	**Inicia la transición a la pantalla de detalle, pasando el id del producto.**
UI	**OutlinedTextField**	**Proporciona un campo de búsqueda claro con el icono de lupa.**

2. Archivo ProductListItem.kt (Elemento de la Lista)

Define el aspecto de cada tarjeta individual dentro del LazyColumn. Es el responsable de la interacción visual del usuario.

```
Composable
un ProductListItem(product: Product, onItemClick: () -> Unit) {

    val interactionSource = remember { MutableInteractionSource() }
    val isPressed by interactionSource.collectIsPressedAsState()


    val animatedElevation by animateDpAsState(
        targetValue = if (isPressed) 2.dp else 6.dp, // 6.dp normal, 2.dp al presionar
        label = "elevationAnimation"
    )

    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp, horizontal = 16.dp)
            .clickable(
                interactionSource = interactionSource,
                indication = null,
                onClick = onItemClick
            ),
        // 3. elevación animada
        elevation = CardDefaults.cardElevation(defaultElevation = animatedElevation),
        shape = RoundedCornerShape( size = 10.dp) // Bordes redondeados Tabulador to complete
    ) {
        Row(
            modifier = Modifier.padding( all = 12.dp),
            verticalAlignment = Alignment.CenterVertically
```

Animación Táctil	animateDpAsState + collectIsPressedAsState	**Al presionar la tarjeta, la elevación se reduce de 6.dp a 2.dp, dando una sensación de click y mejorando la UX (Extra Opcional: Animaciones).**

Estructura	Card, Row, Column	**Organiza la imagen y el texto de manera horizontal y vertical.**
Formato de Moneda	Locale("es", "ES")	**Asegura que el precio se muestre en Euros (€) con la coma como separador decimal.**

3. Archivo ProductDetailScreen.kt (Vista de Detalle)

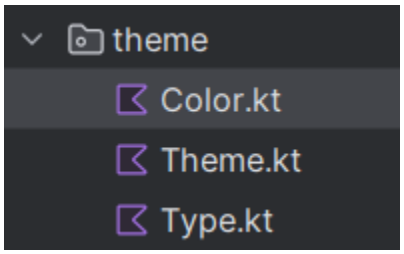
Muestra la información completa de un producto específico, cargada a través del productId recibido por navegación.

```
2 Usages
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun ProductDetailScreen(
    productId: Int,
    navController: NavHostController
) {
    val product = ProductRepository.getProductById(productId)

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text( text = product?.name ?: "Detalle") },
                colors = TopAppBarDefaults.topAppBarColors(
                    containerColor = MaterialTheme.colorScheme.primary,
                    titleContentColor = MaterialTheme.colorScheme.onPrimary,
                    navigationIconContentColor = MaterialTheme.colorScheme.onPrimary
                ),
                navigationIcon = {
                    IconButton(onClick = { navController.popBackStack() }) {
                        Icon( imageVector = Icons.AutoMirrored.Filled.ArrowBack, contentDescription = "Volver")
                    }
```

Obtención de Datos	ProductRepository.getProductById(productId)	**Utiliza el ID recibido para cargar la información específica.**
Diseño Desplazable	rememberScrollState() y verticalScroll()	**Asegura que todo el contenido, incluyendo la descripción larga, sea accesible.**
Navegación de Retorno	navController.popBackStack()	**Vinculada al icono de flecha en TopAppBar para volver a la lista.**
Interacción	Button(onClick = { /* ... */ })	**Implementación del requisito de un botón de interacción ("Añadir al Carrito").**

Documentación del Paquete ui.theme (Diseño y Estilo)

El paquete ui.theme gestiona la paleta de colores (Color.kt), los esquemas de tema (Theme.kt) y las fuentes y estilos de texto (Typography.kt). Esto asegura que el diseño sea consistente (responsive y con tema oscuro).



1. Archivo Color.kt (Paleta Personalizada)

Este archivo define la paleta de colores utilizada por la aplicación, dando nombres descriptivos a los valores hexadecimales.

```
val EmeraldPrimary = Color( color = 0xFF004D40) // #004D40
// Naranja/Oro para precios y acentos
1 Usage
val OrangeAccent = Color( color = 0xFFFF9800)  // #FF9800

2 Usages
val TealLight = Color( color = 0xFF4DB6AC)     // #4DB6AC
```

EmeraldPrimary	#004D40	primary (Barra superior, botones)	N/A
OrangeAccent	#FF9800	secondary (Precios, acentos)	N/A
TealLight	#4DB6AC	N/A	primary (Barra superior, botones)

DarkSurface	#1C1C1E	N/A	background y surface (Fondo)
LightAccent	#FFCC80	N/A	secondary (Precios, acentos)

2. Archivo Theme.kt (Implementación del Tema)

Este archivo utiliza los colores definidos para construir los esquemas de color y decide qué tema aplicar en función de la configuración del sistema.

```
1 Usage
private val DarkColorScheme = darkColorScheme(
    // 1. Primario (Barra superior)
    primary = TealLight,
    onPrimary = Black,

    // 2. Secundario (Precios)
    secondary = LightAccent,
    onSecondary = Black,

    // 3. Fondo y Superficie (Tarjetas).
    background = DarkSurface,
    surface = DarkSurface,
    onBackground = White,
    onSurface = White
)
```

Adaptación de Tema	isSystemInDarkTheme()	Cumple con el Extra Opcional de Tema Oscuro. La aplicación se adapta dinámicamente al modo claro u oscuro establecido por el usuario en su dispositivo.

Esquemas de Color	darkColorScheme y lightColorScheme	Mapea la paleta de colores personalizada a los roles de Material Design 3 (primary, secondary, background, etc.) para asegurar la consistencia.
Barra de Estado	SideEffect	Asegura que el color de la barra de estado del sistema (donde están la batería y la hora) coincida con el color primary de la TopAppBar.

3. Archivo Typography.kt (Estilos de Texto)

Define los estilos de fuente, peso y tamaño utilizados en toda la aplicación. Se utiliza MaterialTheme.typography para establecer un sistema de tipos consistente.

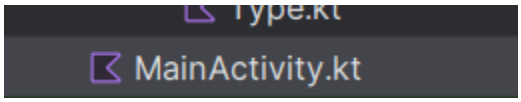
```
val Typography = Typography(
    bodyLarge = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 16.sp,
        lineHeight = 24.sp,
        letterSpacing = 0.5.sp
    )
```

Consistencia: Aunque solo se sobrescribe bodyLarge en el ejemplo, esta estructura permite definir todos los estilos de texto (títulos, subtítulos, etiquetas) de forma centralizada.

Integración: La variable Typography se pasa a MaterialTheme en Theme.kt, lo que permite que todos los Text Composable de la aplicación utilicen estos estilos por defecto.

Documentación de MainActivity.kt (Componente Raíz y Navegación)

La clase MainActivity es el punto de entrada de la aplicación Android. En Jetpack Compose, su función principal es configurar el entorno Compose, aplicar el tema y definir el sistema de navegación global.



1. Clase MainActivity (El Punto de Entrada)

Esta clase extiende ComponentActivity, la base para las actividades modernas en Android que usan Compose.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MiMarketplaceAppTheme { // Llama al tema
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    val navController = rememberNavController()
                    AppNavigation(navController = navController)
```

setContent { ... }	**Método de la ComponentActivity**	**Define la raíz de la interfaz de usuario de Compose.**
MiMarketplaceAppTheme	**Composable Personalizado**	**Aplica los estilos, la tipografía y el esquema de color (incluyendo el Tema Oscuro condicional).**
Surface	**Composable de Material Design 3**	**Actúa como contenedor de fondo, usando el color background del tema.**

rememberNavController()	**Navigation Compose**	**Crea y recuerda el controlador de navegación (NavHostController), que gestiona el estado y la pila de las pantallas.**

2. Función AppNavigation (Sistema de Rutas)

Esta función Composable define el grafo de navegación (NavHost), que mapea las rutas (URLs internas) a las pantallas Composable correspondientes.

```
@Composable
fun AppNavigation(navController: NavHostController) {
    NavHost(navController = navController, startDestination = "product_list") {

        // 1. Ruta de la Lista de Productos
        composable( route = "product_list") {
            ProductListScreen(navController = navController)
        }

        // 2. Ruta de Detalle
        composable(
            route = "product_detail/{productId}",
            arguments = listOf(navArgument( name = "productId") { type = NavType.IntType })
        ) { backStackEntry ->
            val productId = backStackEntry.arguments?.getInt( key = "productId")

            if (productId != null) {
                ProductDetailScreen(productId = productId, navController = navController)
```

NavHost	**Navigation Compose**	**Contenedor que gestiona las transiciones entre destinos.**
startDestination	**Propiedad de NavHost**	**Establece la primera pantalla**

		visible, que es la lista de productos.
Ruta product_list	composable	Carga el ProductListScreen, pasando el navController para permitir la navegación.
Ruta product_detail	composable con navArgument	Cumple el Requisito 2: Navegación a detalle con argumento. Define que la ruta espera un parámetro ({productId}).
Argumentos Seguros	navArgument { type = NavType.IntType }	Declara explícitamente que el argumento debe ser un número entero, previniendo errores de tipo en tiempo de ejecución.
Paso de Datos	backStackEntry.arguments?.getInt("productId")	Extrae el ID del producto de la ruta y se lo pasa directamente al ProductDetailScreen.

PRODUCTO FINAL: