Project Title: Mobile Shopping App Development in Kotlin

Date: 20.12.2024

Student: Issayev Abzal

Institution/Organization Name: KBTU

GitHub:
https://github.com/Karisbala/MobDev2024/tree/master/ShoppingApp

Executive Summary

This report presents the development of a mobile shopping application built with Kotlin and modern Android development principles. The core objectives were to create a functional prototype that allows user registration, product browsing, category-based filtering, cart management, order placement, and order cancellation. Emphasis was placed on Jetpack Compose for UI, Room for local data persistence, Retrofit for network operations, and a clean, layered architecture for maintainability.

Key outcomes include a fully functional front-end with Compose-based screens, state management using ViewModels and reactive flows, stable navigation patterns, and efficient local database handling. The application demonstrates how Kotlin's concise syntax and null-safety enhance productivity, while architectural separations ensure that business logic remains testable and scalable. It is recommended to further refine the UI/UX, integrate secure user authentication, and introduce additional backend features in future iterations.

Table of Contents

Introduction.....4
System Architecture.....5
Table Descriptions.....6
Overview of Android Development: Intro to Kotlin.....7
Functions and Lambdas in Kotlin.....8
OOP in Kotlin.....9
Working with Collections in Kotlin.....10
Android Layout.....11
Activity: Handling User Input and Events.....12
Activity Lifecycle.....13
Fragments and Fragment Lifecycle.....15
RecyclerView and Adapters.....16
ViewModel and LiveData.....17
Working with Databases.....18
Retrofit.....19
WebSockets.....20
Challenges and Solutions.....21
Conclusion.....22
References.....23
Appendices.....24

Introduction

The Android ecosystem encourages building applications using modern frameworks and languages. Kotlin's expressive syntax and safety features, combined with Jetpack libraries, have become the standard for Android development [1]. Developing a shopping app scenario showcases handling user interactions, managing local and remote data, and rendering complex UIs efficiently.

The project aimed to create a basic mobile shopping experience, focusing on:

- User registration and login workflows.
- Product listings, including filtering products by categories.
- Cart management, including adding, removing, and adjusting item quantities.
- Order placement with confirmation dialogs to prevent accidental actions.
- Order cancellation with a confirmation step.

These goals highlight handling data consistently across different states, ensuring a responsive and user-friendly interface, and demonstrating Kotlin's advantages in Android development.

Within this project, essential e-commerce operations were implemented. Sophisticated authentication methods, online payment gateways, and advanced backend integrations were not included. Instead, focus remained on local data operations, static product data retrieval, UI state management, and ensuring stable navigation and interaction flows.

System Architecture

A layered architecture was adopted, dividing the application into domain, data, and UI layers [2]:

- Domain Layer: Contains pure Kotlin use cases and domain models. This layer holds the business logic independent of frameworks. For example, a PlaceOrderUseCase orchestrates verifying cart contents and initiating order placement logic.
- Data Layer: Manages data retrieval and storage through Room database DAOs and Retrofit service interfaces. Repositories implement domain-defined contracts by delegating calls to Room DAOs or API endpoints. Entities (for Room) and DTOs (for Retrofit) are mapped to domain models.
- UI Layer: Employs Jetpack Compose for UI and ViewModels for state management. The MVI pattern ensures a single source of truth for UI states and reduces complexity. State is updated reactively through flows. The UI listens to state changes and renders screens accordingly.

Components interact as follows: the UI dispatches user intents (e.g., add product to cart), the ViewModel uses use cases to process data, and repositories fetch or store data. The result is returned to the ViewModel, updating state flows, which the UI observes and renders.

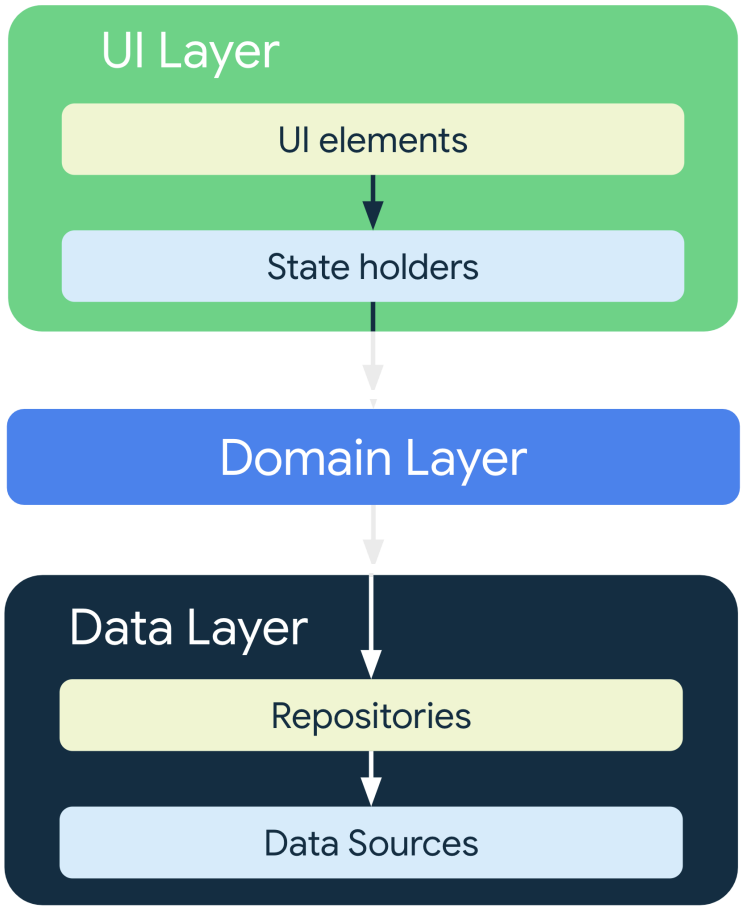

Figure 1. Diagram of an app architecture.

Table Descriptions

Users: (userId, username, email, passwordHash)
Manages user credentials and profiles. userId is a primary key. Relationships: Orders and ShoppingCart reference a userId.

Products: (productId, name, description, price, imageUrl, category)
Holds product information. productId is a primary key. category field references a known category name. Used to populate product listings and cart items.

Categories: (categoryId, categoryName)
Defines product categories. A one-to-many relationship exists from Categories to Products based on categoryName. This table supports category filtering.

Orders: (orderId, userId, orderDate, totalAmount, status)
Represents placed orders, linking them to a userId. status may be "in delivery" or "canceled." A one-to-many relationship with OrderItems exists.

OrderItems: (orderItemId, orderId, productId, quantity, price)
Details each product in an order. orderId references Orders, productId references Products. Ensures itemized order details.

ShoppingCart: (cartId, userId, createdAt)
Tracks a user’s active cart. One-to-many relationship with CartItems. Acts as a container before an order is placed.

CartItems: (cartItemId, cartId, productId, quantity)
Contains products and their quantities for a user’s ongoing cart session. cartId references ShoppingCart.

Overview of Android Development: Intro to Kotlin

Android Studio was set up with the latest Android SDK. The Kotlin plugin was integrated, ensuring first-class support. A minimal "Hello World" activity validated the environment.

Kotlin’s key advantages, such as null-safety (? operator), data classes for immutable value objects, and extension functions for cleaner code, were leveraged. Kotlin coroutines were introduced to handle asynchronous calls smoothly. The shift from Java’s verbose syntax to Kotlin’s more concise style improved development speed and reduced boilerplate [3].

Functions and Lambdas in Kotlin

Functions in Kotlin were defined using the fun keyword, and return types were inferred where possible [4]. Lambdas allow passing behavior as parameters, commonly used in filtering products by category. For instance, a lambda in filter { it.category in selectedCategories } clarified logic without verbose anonymous classes. Such lambdas improved code readability and maintainability.

Code snippet showcasing the lambda usage:

```
val filteredProducts = remember(state.products,
state.selectedCategories) {
   if (state.selectedCategories.isEmpty()) {
       state.products
   } else {
       state.products.filter { it.category in
state.selectedCategories }
   }
}
```

OOP in Kotlin

Object-oriented programming principles guided the creation of domain models. Classes and interfaces supported abstraction, while inheritance provided reuse. Data classes (e.g., data class Product(...)) minimized boilerplate, automatically generating equals, hashCode, and toString. Interfaces defined repository contracts, enabling multiple implementations (e.g., UserRepository, ProductRepository). By applying OOP principles, domain logic remained organized, promoting scalability and future enhancements.

Product class implementation:

```
data class Product(
   val productId: String,
   val name: String,
   val description: String,
   val price: Double,
   val imageUrl: String,
   val category: String
)
```

Repository interfaces (UserRepository, ProductRepository):

```
interface UserRepository {
   suspend fun register(username: String, email: String, password:
String): User
   suspend fun login(email: String, password: String): User
   suspend fun getUserById(userId: String): User?
}
```

```
interface ProductRepository {
   suspend fun getProducts(): List<Product>
   suspend fun getProductById(productId: String): Product?
   suspend fun getCategories(): List<Category>
}
```

Working with Collections in Kotlin

Kotlin's collection functions like filter, map simplified data transformations. Category filtering, price sorting, and cart item aggregation were achieved by chaining collection operations. This reduced boilerplate and made data manipulation more intuitive and maintainable.

Code snippets showcasing collection operations:

```
val totalAmount = items.sumOf { it.quantity *
getProductPrice(it.productId) }
```

```
val filteredProducts = remember(state.products,
state.selectedCategories) {
   if (state.selectedCategories.isEmpty()) {
       state.products
   } else {
       state.products.filter { it.category in
state.selectedCategories }
   }
}
```

Android Layout

Jetpack Compose replaced traditional XML layouts. Each screen (e.g., ProductsScreen, CartScreen, OrdersScreen) was built from composables that defined UI elements declaratively. Material Design 3 components were used for consistent theming and responsiveness. For example, category filters were implemented as FilterChips, and product lists were displayed using LazyColumn. Compose previews in Android Studio allowed rapid iteration on UI design.

Code snippet of composable function rendering a product card:

```
@Composable
fun ProductItemCard(product: Product, onAddToCart: () -> Unit) {
   Card(modifier = Modifier.fillMaxWidth().padding(8.dp)) {
       Row(verticalAlignment = Alignment.CenterVertically) {
           AsyncImage(
               model = product.imageUrl,
               contentDescription = product.name,
               modifier = Modifier.size(64.dp).padding(8.dp)
           )
           Column(modifier = Modifier.weight(1f)) {
               Text(text = product.name, style =
MaterialTheme.typography.titleMedium)
               Text(text = "$${product.price}", style =
MaterialTheme.typography.bodyMedium)
           }
           Button(onClick = onAddToCart, modifier =
Modifier.padding(8.dp)) {
               Text("Add to Cart")
           }
       }
   }
}
```

Activity: Handling User Input and Events

While Compose reduced the reliance on Activities for UI handling, the MainActivity still set up content and integrated top-level navigation. User input events such as button clicks or text input were forwarded to ViewModels, ensuring that Activities remained lightweight. The Activity's role was limited to theming, navigation host setup, and initiating the UI's initial state.

Code snippet of main activity and app navigation:

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContent {
           AppTheme {
               AppNavigation()
           }
       }
   }
}
```

```
@Composable
fun AppNavigation(startDestination: String = "login") {
   val navController = rememberNavController()
   NavHost(navController, startDestination = startDestination) {
       composable("login") {
           LoginScreen(
               onLoginSuccess = { navController.navigate("main") {
popUpTo("login") { inclusive = true } } },
               onRegisterClick = { navController.navigate("register")
}
           )
       }
       composable("register") {
           RegisterScreen(onRegisterSuccess = {
navController.popBackStack() })
       }
       composable("main") {
           MainScreen(onLogout = {
               navController.navigate("login") {
                   popUpTo("main") { inclusive = true }
               }
           })
       }
   }
}
```

Activity Lifecycle

Managing the lifecycle was simplified by ViewModels, which preserved state across configuration changes. This minimized explicit lifecycle handling in Activities. By observing lifecycle-aware components, background work was automatically managed. Composables used LaunchedEffect or rememberCoroutineScope() as needed, ensuring no leaks or redundant operations occurred during lifecycle transitions.

Code snippet of main screen composable:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MainScreen(onLogout: () -> Unit) {
   val navController = rememberNavController()
   val drawerState = rememberDrawerState(initialValue =
DrawerValue.Closed)
   val scope = rememberCoroutineScope()
   var selectedItem by remember { mutableStateOf("Products") }

   ModalNavigationDrawer(
       drawerContent = {
           Surface(color = MaterialTheme.colorScheme.surface) {
               Box(modifier = Modifier.fillMaxWidth(0.3f)) {
                   Column {
                       NavigationDrawerItem(
                           label = { Text("Products") },
                           selected = (selectedItem == "Products"),
                           onClick = {
                               selectedItem = "Products"
                                scope.launch { drawerState.close() }
                               navController.navigate("products") {
                                   popUpTo("products") { inclusive =
false }
                               }
                           }
                       )
                        Spacer(modifier = Modifier.height(16.dp))
                        NavigationDrawerItem(
                           label = { Text("Orders") },
                           selected = (selectedItem == "Orders"),
                           onClick = {
                               selectedItem = "Orders"
                                scope.launch { drawerState.close() }
                               navController.navigate("orders") {
                                   popUpTo("products") { inclusive =
false }
                               }
                           }
                       )
                        Spacer(modifier = Modifier.height(16.dp))
                        NavigationDrawerItem(
```

```
                            label = { Text("Log Out") },
                            selected = (selectedItem == "Log Out"),
                            onClick = {
                                onLogout()
                                 scope.launch { drawerState.close() }
                            }
                        )
                    }
                }
            }
        },
        drawerState = drawerState,
        content = {
            Scaffold(
                topBar = {
                    TopAppBar(
                        title = { Text(selectedItem) },
                        navigationIcon = {
                            IconButton(onClick = { scope.launch {
drawerState.open() } }) {
                                Icon(Icons.Default.Menu,
contentDescription = "Menu")
                            }
                        }
                    )
                }
            ) { padding ->
                NavHost(navController, startDestination = "products",
modifier = Modifier.padding(padding)) {
                    composable("products") { ProductsScreen(onCartClick
= { navController.navigate("cart") }) }
                    composable("orders") { OrdersScreen() }
                    composable("cart") { CartScreen(onOrderPlaced = {
navController.navigate("orders") }) }
                }
            }
        }
    )
}
```

Fragments and Fragment Lifecycle

Fragments were not implemented in the final version of the project. The entire UI was developed using Jetpack Compose within activities, and no code or files related to Fragments or Fragment lifecycle management were included in the project's final codebase.

RecyclerView and Adapters

With Compose, LazyColumn replaced RecyclerView. If a RecyclerView was used, a custom adapter bound product entities to item layouts, ensuring stable performance and smooth scrolling. In Compose, LazyColumn and items blocks replaced this pattern, removing the need for a traditional adapter and simplifying UI updates.

Code snippet of LazyColumn usage for displaying product cards:

```
LazyColumn {
   items(filteredProducts) { product ->
       ProductItemCard(product = product) {

viewModel.handleIntent(ProductsIntent.AddToCart(product.productId))
       }
   }
}
```

ViewModel and LiveData

ViewModels stored UI data and exposed it through LiveData or StateFlow. For example, the ProductsViewModel maintained a StateFlow<ProductsState> that included a list of products and selected categories. The UI collected this state, recomposing whenever the state changed. This enabled a reactive UI that automatically updated in response to data changes without manual refresh calls. Using StateFlows also ensured easy integration with coroutines, improving async data handling.

Code snippet of StateFlow:

```
private val _state = MutableStateFlow(ProductsState())
val state: StateFlow<ProductsState> get() = _state
```

Code snippet of UI collecting this state:

```
val state by viewModel.state.collectAsState()
```

Working with Databases

Room was integrated to store products, users, orders, and cart items locally. Entities annotated with @Entity corresponded to database tables, while DAOs provided methods such as getCartItems(userId), insertOrder(), or updateCartItemQuantity(). By returning Flow objects, DAOs allowed the UI to reactively observe database changes, ensuring that updates to product stock or order status were reflected on screen instantly. This offline-first approach improved reliability and performance [5].

Code snippet of Product entity:

```
@Entity(tableName = "Products")
data class ProductEntity(
   @PrimaryKey val productId: String,
   val name: String,
   val description: String,
   val price: Double,
   val imageUrl: String,
   val category: String
)
```

Code snippet of Product Dao:

```
@Dao
interface ProductDao {
   @Insert(onConflict = OnConflictStrategy.REPLACE)
   suspend fun insertProducts(products: List<ProductEntity>)

   @Query("SELECT * FROM Products")
   suspend fun getAllProducts(): List<ProductEntity>

   @Query("SELECT * FROM Products WHERE productId = :productId LIMIT
1")
   suspend fun getProductById(productId: String): ProductEntity?
}
```

Retrofit

Retrofit was set up to handle potential remote data calls, such as fetching product lists from a FakeStore API. Even though the final implementation relied mostly on local data, the code structure for Retrofit endpoints existed. ApiService interfaces defined endpoints and data models. If integrated fully, calling apiService.getProducts() inside a UseCase fetched remote data, which was then mapped into domain models and stored locally. This arrangement made it easy to add real endpoints later without restructuring the code [6,7].

Code snippet of a Retrofit interface definition:

```
interface ApiService {

   @POST("register")
   suspend fun register(@Body request: RegisterRequest):
LoginResponse

   @POST("login")
   suspend fun login(@Body request: LoginRequest): LoginResponse

   @GET("products")
   suspend fun getProducts(): List<ProductDto>

   @GET("products/{id}")
   suspend fun getProductById(@Path("id") productId: String):
ProductDto?

   @GET("products/categories")
   suspend fun getCategories(): List<String>
}
```

Code snippet of viewModel making a network call through the usecase (getProductsUseCase):

```
private fun loadData() {
   viewModelScope.launch {
       _state.value = _state.value.copy(isLoading = true)
       try {
           val products = getProductsUseCase()
           val categories = productRepository.getCategories()
           _state.value = _state.value.copy(
               isLoading = false,
               products = products,
               categories = categories
           )
       } catch (e: Exception) {
           _state.value = ProductsState(isLoading = false, error =
e.message)
       }
   }
}
```

WebSockets

No WebSocket client setup or code related to establishing, maintaining, or parsing WebSocket connections was added to the repository's code. Consequently, there are no files or functions in the project that include WebSocket client code, listener implementations, or message handling logic. Thus, no dependencies related to WebSockets (such as OkHttp's WebSocket feature or third-party WebSocket libraries) were integrated into the Gradle configuration or utilized in the application modules.

Challenges and Solutions

A few significant challenges and their solutions:

1. Complex Category Filtering Not Triggering Recomposition:

Initially, toggling categories did not update the UI. Converting states to StateFlow and computing filteredProducts inside a composable fixed this, ensuring immediate recomposition.

2. Ensuring Order Placement and Cancellation Confirmation:

Without confirmation dialogs, users could accidentally place or cancel orders. Adding AlertDialog composables before performing these actions provided a safety check. This improved user experience and prevented unintended actions.

3. Navigating with a Navigation Drawer and Ensuring Unified UI Style:

Implementing a navigation drawer that looked cohesive and only took up 30% of screen width required careful Compose UI styling. Using Surface, Box with fillMaxWidth(0.3f), and theming ensured a solid background and unified look. State management with DrawerState and rememberCoroutineScope() ensured the drawer opened and closed predictably.

Conclusion

This project demonstrated the use of Kotlin, Jetpack Compose, Room, and Retrofit in building a mobile shopping application. Achieved functionalities included user registration logic (mocked), product browsing, category filtering, cart operations, order placement, and order cancellation—all managed through a clean architecture that separated domain logic from data handling and UI. The result was a stable, responsive, and maintainable application prototype. Future enhancements might incorporate secure remote authentication, payment integration, and richer product data, further showcasing the benefits of Kotlin and modern Android development techniques.

References

1. Android Developers. (n.d.). Kotlin on Android. Retrieved from https://developer.android.com/kotlin
2. Android Developers. (n.d.). Guide to app architecture. Retrieved from https://developer.android.com/topic/architecture
3. JetBrains. (n.d.). Kotlin programming language. Retrieved from https://kotlinlang.org/
4. Android Developers. (n.d.). Basic Android Kotlin Compose: Function types and lambda (Codelab). Retrieved from https://developer.android.com/codelabs/basic-android-kotlin-compose-function-types-and-lambda#0
5. Android Developers. (n.d.). Save data in a local database using Room. Retrieved from https://developer.android.com/training/data-storage/room
6. Fake Store API. (n.d.). Fake Store API. Retrieved from https://fakestoreapi.com/
7. Square. (n.d.). Retrofit: A type-safe HTTP client for Android and Java. Retrieved from https://square.github.io/retrofit/

Appendices

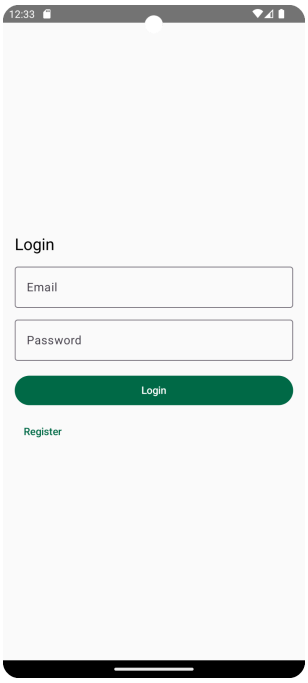

Figure 2. Login screen.

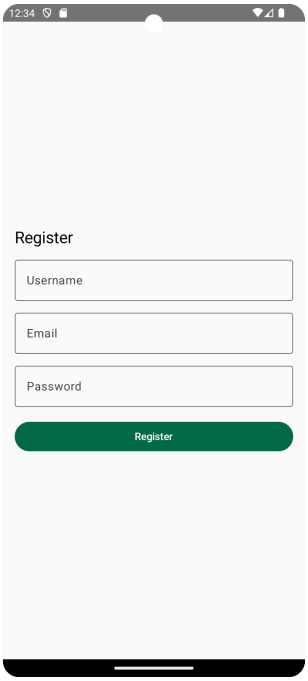

Figure 3. Register screen.

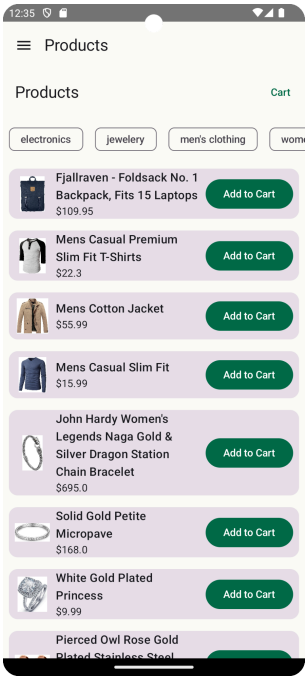

Figure 4. Main screen displaying Products screen.

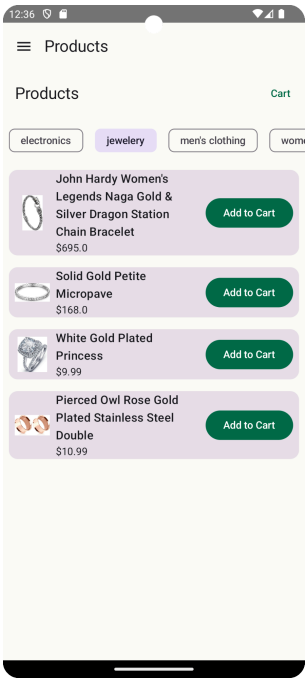

Figure 5. Selecting “jewelry” filter to display only jewelry.

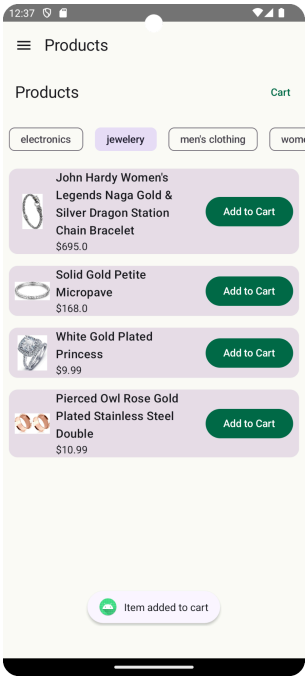

Figure 6. Toast snackbar after adding an item to the cart.

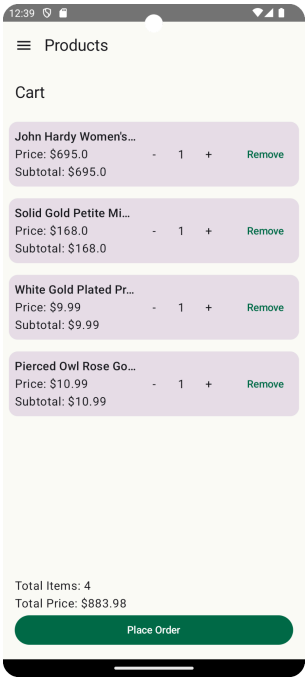

Figure 7. Cart screen.

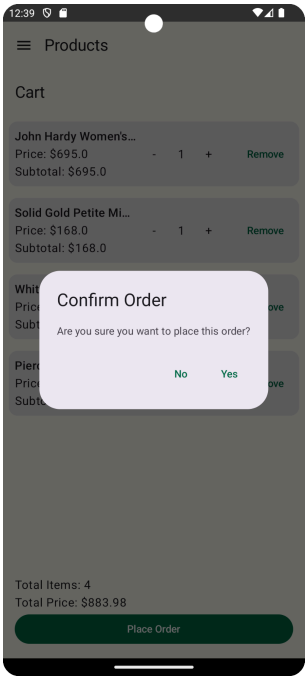

Figure 8. Confirmation screen after clicking “Place Order”.

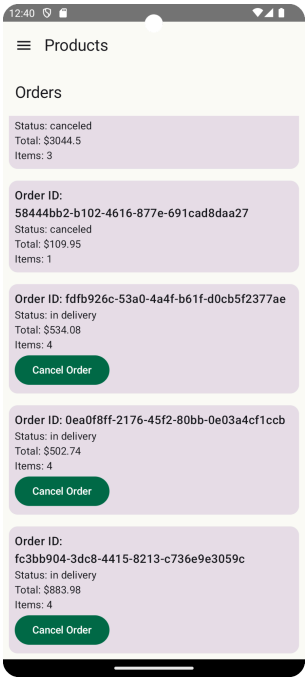

Figure 9. Order screen.

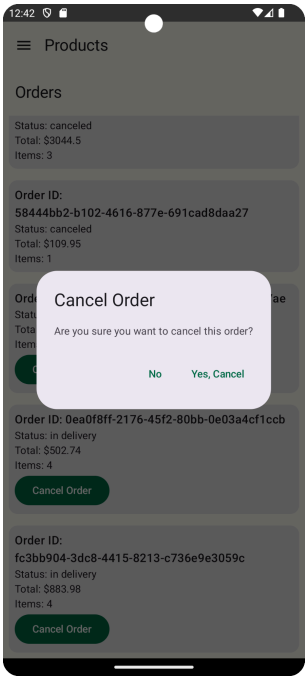

Figure 10. Confirmation dialog after clicking “Cancel order” button.

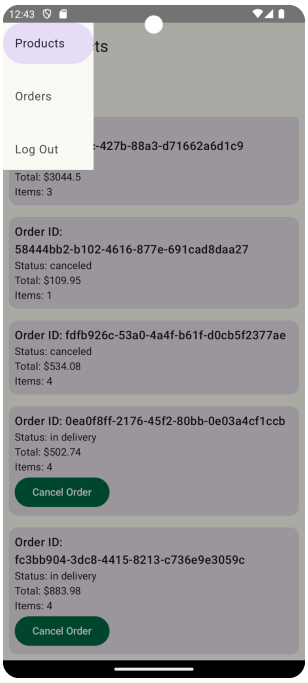

Figure 11. Navigation drawer after clicking the navigation drawer button.

Code snippets for model implementations (other than Product):

```
data class User(
   val userId: String,
   val username: String,
   val email: String
```

```
)
```

```
data class OrderItem(
   val productId: String,
   val quantity: Int,
   val price: Double
)
```

```
data class Order(
   val orderId: String,
   val userId: String,
   val orderDate: Long,
   val totalAmount: Double,
   val status: String,
   val items: List<OrderItem>
)
```

```
data class Category(
   val categoryId: String,
   val categoryName: String
)
```

```
data class CartItem(
   val productId: String,
   val quantity: Int
)
```

Repository implementations (other than UserRepository, ProductRepository):

```
interface CartRepository {
   suspend fun getCartItems(userId: String): List<CartItem>
   suspend fun addCartItem(userId: String, productId: String,
quantity: Int)
   suspend fun removeCartItem(userId: String, productId: String)
   suspend fun clearCart(userId: String)
   suspend fun updateCartItemQuantity(userId: String, productId:
String, newQuantity: Int)
}
```

```
interface OrderRepository {
   suspend fun placeOrder(userId: String, items: List<CartItem>):
Order
   suspend fun getOrders(userId: String): List<Order>
   suspend fun getOrderById(orderId: String): Order?
   suspend fun cancelOrder(orderId: String): Int
}
```

Code snippet of ProductViewModel:

```
@HiltViewModel
class ProductsViewModel @Inject constructor(
```

```
    private val getProductsUseCase: GetProductsUseCase,
    private val addToCartUseCase: AddToCartUseCase,
    private val productRepository: ProductRepository,
) : ViewModel() {

    private val _state = MutableStateFlow(ProductsState())
    val state: StateFlow<ProductsState> get() = _state

    fun handleIntent(intent: ProductsIntent) {
        when (intent) {
            ProductsIntent.LoadProducts -> loadData()
            is ProductsIntent.AddToCart -> addToCart(intent.productId)
            is ProductsIntent.ToggleCategory ->
toggleCategory(intent.categoryId)
        }
    }

    private fun loadData() {
        viewModelScope.launch {
            _state.value = _state.value.copy(isLoading = true)
            try {
                val products = getProductsUseCase()
                val categories = productRepository.getCategories()
                _state.value = _state.value.copy(
                    isLoading = false,
                    products = products,
                    categories = categories
                )
            } catch (e: Exception) {
                _state.value = ProductsState(isLoading = false, error
= e.message)
            }
        }
    }

    private fun addToCart(productId: String) {
        viewModelScope.launch {
            try {
                addToCartUseCase("currentUserId", productId, 1)
                _state.value = _state.value.copy(cartAddMessage =
"Item added to cart")
            } catch (_: Exception) {

            }
        }
    }

    fun resetCartMessage() {
        _state.value = _state.value.copy(cartAddMessage = null)
    }

    private fun toggleCategory(categoryId: String) {
```

```
        val currentSelected =
_state.value.selectedCategories.toMutableSet()
        if (currentSelected.contains(categoryId)) {
            currentSelected.remove(categoryId)
        } else {
            currentSelected.add(categoryId)
        }
        _state.value = _state.value.copy(selectedCategories =
currentSelected)
    }
}
```

Code Snippet of Product Screen composable:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ProductsScreen(onCartClick: () -> Unit, viewModel:
ProductsViewModel = hiltViewModel()) {
    val state by viewModel.state.collectAsState()
    val context = LocalContext.current

    LaunchedEffect(state.cartAddMessage) {
        val msg = state.cartAddMessage
        if (msg != null) {
            Toast.makeText(context, msg, Toast.LENGTH_SHORT).show()
            viewModel.resetCartMessage()
        }
    }
    LaunchedEffect(Unit) {
        viewModel.handleIntent(ProductsIntent.LoadProducts)
    }

    val filteredProducts = remember(state.products,
state.selectedCategories) {
        if (state.selectedCategories.isEmpty()) {
            state.products
        } else {
            state.products.filter { it.category in
state.selectedCategories }
        }
    }

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text("Products") },
                actions = {
                    TextButton(onClick = onCartClick) {
                        Text("Cart")
                    }
                }
```

```
            )
        }
    ) { padding ->
        if (state.isLoading) {
            LoadingIndicator()
        } else if (state.error != null) {
            ErrorMessage(state.error!!)
        } else {
            Column(modifier = Modifier.padding(padding)) {
                Row(modifier =
Modifier.horizontalScroll(rememberScrollState())) {
                    state.categories.forEach { category ->
                        val selected = category.categoryId in
state.selectedCategories
                        FilterChip(
                            selected = selected,
                            onClick = {
viewModel.handleIntent(ProductsIntent.ToggleCategory(category.categor
yId)) },
                            label = { Text(category.categoryName) },
                            modifier = Modifier.padding(8.dp)
                        )
                    }
                }

                LazyColumn {
                    items(filteredProducts) { product ->
                        ProductItemCard(product = product) {

viewModel.handleIntent(ProductsIntent.AddToCart(product.productId))
                        }
                    }
                }
            }
        }
    }
}
```

Code snippet of MyApp:

```
@HiltAndroidApp
class MyApp : Application() {
}
```

Code snippet of Orders Screen:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun OrdersScreen(viewModel: OrdersViewModel = hiltViewModel()) {
    val state by viewModel.state.collectAsState()
```

```
    var orderToCancel by remember { mutableStateOf<String?>(null) }
    var showCancelDialog by remember { mutableStateOf(false) }

    LaunchedEffect(Unit) {
        viewModel.handleIntent(OrdersIntent.LoadOrders)
    }

    Scaffold(topBar = { TopAppBar(title = { Text("Orders") }) }) {
padding ->
        if (state.isLoading) {
            LoadingIndicator()
        } else if (state.error != null) {
            ErrorMessage(state.error!!)
        } else {
            LazyColumn(contentPadding = padding) {
                items(state.orders) { order ->
                    OrderItemCard(order = order, onCancelClick = {
                        orderToCancel = order.orderId
                         showCancelDialog = true
                     })
                }
            }
        }
    }

    if (showCancelDialog && orderToCancel != null) {
        AlertDialog(
            onDismissRequest = { showCancelDialog = false },
            title = { Text("Cancel Order") },
            text = { Text("Are you sure you want to cancel this
order?") },
            confirmButton = {
                TextButton(onClick = {
                    showCancelDialog = false
                     orderToCancel?.let { viewModel.cancelOrder(it) }
                    orderToCancel = null
                }) {
                    Text("Yes, Cancel")
                }
            },
            dismissButton = {
                TextButton(onClick = {
                    showCancelDialog = false
                     orderToCancel = null
                }) {
                    Text("No")
                }
            }
        )
    }
}
```

Code snippet of Orders ViewModel:

```
@HiltViewModel
class OrdersViewModel @Inject constructor(
   private val getOrdersUseCase: GetOrdersUseCase,
   private val cancelOrderUseCase: CancelOrderUseCase
) : ViewModel() {

   private val _state = MutableStateFlow(OrdersState())
   val state: StateFlow<OrdersState> get() = _state

   fun handleIntent(intent: OrdersIntent) {
       when (intent) {
           is OrdersIntent.LoadOrders -> loadOrders()
       }
   }

   private fun loadOrders() {
       viewModelScope.launch {
           _state.value = OrdersState(isLoading = true)
           try {
               val orders = getOrdersUseCase("currentUserId")
               _state.value = OrdersState(isLoading = false, orders =
orders)
           } catch (e: Exception) {
               _state.value = OrdersState(isLoading = false, error =
e.message)
           }
       }
   }

   fun cancelOrder(orderId: String) {
       viewModelScope.launch {
           try {
               cancelOrderUseCase(orderId)
               loadOrders()
           } catch (e: Exception) {
               _state.value = _state.value.copy(error = e.message)
           }
       }
   }
}
```

Code Snippet of Cart Screen:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun CartScreen(
   onOrderPlaced: () -> Unit,
   viewModel: CartViewModel = hiltViewModel()
```

```
) {
    val state by viewModel.state.collectAsState()
    val context = LocalContext.current

    var showPlaceOrderDialog by remember { mutableStateOf(false) }

    LaunchedEffect(Unit) {
        viewModel.handleIntent(CartIntent.LoadCart)
    }

    LaunchedEffect(state.orderPlaced) {
        if (state.orderPlaced) {
            onOrderPlaced()
        }
    }

    Scaffold(
        topBar = { TopAppBar(title = { Text("Cart") }) },
        bottomBar = {
            Column(modifier = Modifier.padding(16.dp)) {
                Text("Total Items: ${state.totalItems}")
                Text("Total Price: \$${state.totalPrice}")
                Button(
                    onClick = {
                        if (state.items.isEmpty()) {
                            Toast.makeText(context, "Cart is empty.
Add items before ordering.", Toast.LENGTH_SHORT).show()
                        } else {
                            showPlaceOrderDialog = true
                        }
                    },
                    modifier = Modifier.fillMaxWidth()
                ) {
                    Text("Place Order")
                }
            }
        }
    ) { padding ->
        Box(modifier = Modifier.padding(padding).fillMaxSize()) {
            when {
                state.isLoading -> LoadingIndicator()
                state.error != null -> ErrorMessage(state.error!!)
                else -> {
                    if (state.items.isEmpty()) {
                        Text("Cart is empty", modifier =
Modifier.align(Alignment.Center))
                    } else {
                        LazyColumn {
                            items(state.items) { item ->
                                Card(modifier =
Modifier.fillMaxWidth().padding(8.dp)) {
                                    Row(
```

```
                                        modifier = Modifier
                                            .fillMaxWidth()
                                            .padding(8.dp),
                                        verticalAlignment =
Alignment.CenterVertically
                                    ) {
                                        Column(modifier =
Modifier.weight(1f)) {
                                            Text(
                                                text =
item.productName,
                                                style =
MaterialTheme.typography.titleMedium,
                                                maxLines = 1,
                                                overflow =
TextOverflow.Ellipsis
                                            )
                                            Text("Price:
\$${item.price}")
                                            Text("Subtotal:
\$${item.price * item.quantity}")
                                        }

                                        // Quantity controls
                                        Row(verticalAlignment =
Alignment.CenterVertically) {
                                            IconButton(onClick = {
viewModel.handleIntent(CartIntent.DecreaseQuantity(item.productId))
}) {
                                                Text("-")
                                            }
                                            Text("${item.quantity}",
modifier = Modifier.padding(horizontal = 8.dp))
                                            IconButton(onClick = {
viewModel.handleIntent(CartIntent.IncreaseQuantity(item.productId))
}) {
                                                Text("+")
                                            }
                                            Spacer(modifier =
Modifier.width(16.dp))
                                            TextButton(onClick = {

viewModel.handleIntent(CartIntent.RemoveItem(item.productId))
                                            }) {
                                                Text("Remove")
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
```

```
                }
            }
        }
    }
    if (showPlaceOrderDialog) {
        AlertDialog(
            onDismissRequest = { showPlaceOrderDialog = false },
            title = { Text("Confirm Order") },
            text = { Text("Are you sure you want to place this
order?") },
            confirmButton = {
                TextButton(onClick = {
                    showPlaceOrderDialog = false
                    viewModel.handleIntent(CartIntent.PlaceOrder)
                }) {
                    Text("Yes")
                }
            },
            dismissButton = {
                TextButton(onClick = { showPlaceOrderDialog = false })
{
                    Text("No")
                }
            }
        )
    }
}
```

Code snippet in Cart ViewModel:

```
@HiltViewModel
class CartViewModel @Inject constructor(
    private val getCartItemsUseCase: GetCartItemsUseCase,
    private val removeFromCartUseCase: RemoveFromCartUseCase,
    private val placeOrderUseCase: PlaceOrderUseCase,
    private val productRepository: ProductRepository,
    private val updateCartItemQuantityUseCase:
UpdateCartItemQuantityUseCase
) : ViewModel() {

    private val _state = MutableStateFlow(CartState())
    val state: StateFlow<CartState> get() = _state

    fun handleIntent(intent: CartIntent) {
        when (intent) {
            CartIntent.LoadCart -> loadCart()
            is CartIntent.RemoveItem -> removeItem(intent.productId)
            CartIntent.PlaceOrder -> placeOrder()
            is CartIntent.IncreaseQuantity ->
changeQuantity(intent.productId, +1)
```

```
            is CartIntent.DecreaseQuantity ->
changeQuantity(intent.productId, -1)
        }
    }

    private fun loadCart() {
        viewModelScope.launch {
            _state.value = _state.value.copy(isLoading = true,
orderPlaced = false)
            try {
                val cartItems = getCartItemsUseCase("currentUserId")
                val detailedItems = cartItems.map { cartItem ->
                    val product =
productRepository.getProductById(cartItem.productId)
                    DetailedCartItem(
                        productId = cartItem.productId,
                        productName = product?.name ?: "Unknown",
                        price = product?.price ?: 0.0,
                        quantity = cartItem.quantity
                    )
                }
                _state.value = CartState(isLoading = false, items =
detailedItems)
            } catch (e: Exception) {
                _state.value = CartState(isLoading = false, error =
e.message)
            }
        }
    }

    private fun removeItem(productId: String) {
        viewModelScope.launch {
            removeFromCartUseCase("currentUserId", productId)
            loadCart()
        }
    }

    private fun placeOrder() {
        viewModelScope.launch {
            _state.value = _state.value.copy(isLoading = true)
            try {
                placeOrderUseCase("currentUserId")
                _state.value = _state.value.copy(isLoading = false,
orderPlaced = true)
            } catch (e: Exception) {
                _state.value = _state.value.copy(isLoading = false,
error = e.message)
            }
        }
    }

    private fun changeQuantity(productId: String, delta: Int) {
```

```
        val item = _state.value.items.find { it.productId == productId
} ?: return
        val newQty = (item.quantity + delta).coerceAtLeast(1)
        viewModelScope.launch {
            updateCartItemQuantityUseCase("currentUserId", productId,
newQty)
            loadCart()
        }
    }
}
```

Code snippet of Register Screen:

```
@Composable
fun RegisterScreen(
    onRegisterSuccess: () -> Unit,
    viewModel: RegisterViewModel = hiltViewModel()
) {
    val state by viewModel.state.collectAsState()

    LaunchedEffect(state.success) {
        if (state.success) {
            onRegisterSuccess()
        }
    }

    var email by remember { mutableStateOf("") }
    var username by remember { mutableStateOf("") }
    var password by remember { mutableStateOf("") }

    if (state.isLoading) {
        LoadingIndicator()
    } else {
        Column(
            modifier = Modifier.fillMaxSize().padding(16.dp),
            verticalArrangement = Arrangement.Center
        ) {
            Text("Register", style =
MaterialTheme.typography.titleLarge)
            Spacer(modifier = Modifier.height(8.dp))

            OutlinedTextField(value = username, onValueChange = {
username = it }, label = { Text("Username") }, modifier =
Modifier.fillMaxWidth())
            Spacer(modifier = Modifier.height(8.dp))
            OutlinedTextField(value = email, onValueChange = { email =
it }, label = { Text("Email") }, modifier = Modifier.fillMaxWidth())
            Spacer(modifier = Modifier.height(8.dp))
            OutlinedTextField(value = password, onValueChange = {
password = it }, label = { Text("Password") }, modifier =
```

```
Modifier.fillMaxWidth(), visualTransformation =
androidx.compose.ui.text.input.PasswordVisualTransformation())

            Spacer(modifier = Modifier.height(16.dp))
            Button(onClick = {
viewModel.handleIntent(RegisterIntent.Register(username, email,
password)) },
                modifier = Modifier.fillMaxWidth()) {
                Text("Register")
            }

            state.error?.let {
                Spacer(modifier = Modifier.height(8.dp))
                ErrorMessage(message = it)
                LaunchedEffect(Unit) {
                    viewModel.handleIntent(RegisterIntent.ResetError)
                }
            }
        }
    }
}
```

Code snippet of Register ViewModel:

```
@HiltViewModel
class RegisterViewModel @Inject constructor(
    private val registerUserUseCase: RegisterUserUseCase
) : ViewModel() {

    private val _state = MutableStateFlow(RegisterState())
    val state: StateFlow<RegisterState> get() = _state

    fun handleIntent(intent: RegisterIntent) {
        when (intent) {
            is RegisterIntent.Register -> register(intent.username,
intent.email, intent.password)
            is RegisterIntent.ResetError -> resetError()
        }
    }

    private fun register(username: String, email: String, password:
String) {
        viewModelScope.launch {
            _state.value = _state.value.copy(isLoading = true, error =
null, success = false)
            try {
                registerUserUseCase(username, email, password)
                _state.value = _state.value.copy(isLoading = false,
success = true)
            } catch (e: Exception) {
```

```
                _state.value = _state.value.copy(isLoading = false,
error = e.message)
            }
        }
    }

    private fun resetError() {
        _state.value = _state.value.copy(error = null)
    }
}
```

Code snippet of Login Screen:

```
@Composable
fun LoginScreen(
    onLoginSuccess: () -> Unit,
    onRegisterClick: () -> Unit,
    viewModel: LoginViewModel = hiltViewModel()
) {
    val state by viewModel.state.collectAsStateWithLifecycle()

    LaunchedEffect(state) {
        if (!state.isLoading && state.success && state.error == null)
{
            onLoginSuccess()
        }
    }

    var email by remember { mutableStateOf("") }
    var password by remember { mutableStateOf("") }

    if (state.isLoading) {
        LoadingIndicator()
    } else {
        Column(
            modifier = Modifier.fillMaxSize().padding(16.dp),
            verticalArrangement = Arrangement.Center
        ) {
            Text("Login", style = MaterialTheme.typography.titleLarge)
            Spacer(modifier = Modifier.height(8.dp))

            OutlinedTextField(
                value = email,
                onValueChange = { email = it },
                label = { Text("Email") },
                modifier = Modifier.fillMaxWidth()
            )
            Spacer(modifier = Modifier.height(8.dp))

            OutlinedTextField(
                value = password,
```

```
                onValueChange = { password = it },
                label = { Text("Password") },
                modifier = Modifier.fillMaxWidth(),
                visualTransformation =
androidx.compose.ui.text.input.PasswordVisualTransformation()
            )
            Spacer(modifier = Modifier.height(16.dp))

            Button(onClick = {
viewModel.handleIntent(LoginIntent.Login(email, password)) },
                modifier = Modifier.fillMaxWidth()) {
                Text("Login")
            }
            Spacer(modifier = Modifier.height(8.dp))

            TextButton(onClick = onRegisterClick) {
                Text("Register")
            }

            state.error?.let {
                Spacer(modifier = Modifier.height(8.dp))
                ErrorMessage(message = it)
                LaunchedEffect(Unit) {
                    viewModel.handleIntent(LoginIntent.ResetError)
                }
            }
        }
    }
}
```

Code snippet of Login ViewModel:

```
@HiltViewModel
class LoginViewModel @Inject constructor(
    private val loginUseCase: LoginUseCase
) : ViewModel() {

    private val _state = MutableStateFlow(LoginState())
    val state: StateFlow<LoginState> get() = _state

    fun handleIntent(intent: LoginIntent) {
        when (intent) {
            is LoginIntent.Login -> login(intent.email,
intent.password)
            is LoginIntent.ResetError -> resetError()
        }
    }

    private fun login(email: String, password: String) {
        viewModelScope.launch {
```

```
            _state.value = _state.value.copy(isLoading = true, error =
null, success = false)
            try {
                loginUseCase(email, password)
                _state.value = _state.value.copy(isLoading = false,
success = true)
            } catch (e: Exception) {
                _state.value = _state.value.copy(isLoading = false,
error = e.message)
            }
        }
    }

    private fun resetError() {
        _state.value = _state.value.copy(error = null)
    }
}
```

Code snippet of Order card:

```
@Composable
fun OrderItemCard(order: Order, onCancelClick: () -> Unit) {
    Card(modifier = Modifier.fillMaxWidth().padding(8.dp)) {
        Column(modifier = Modifier.padding(8.dp)) {
            Text(text = "Order ID: ${order.orderId}", style =
MaterialTheme.typography.titleMedium)
            Text(text = "Status: ${order.status}", style =
MaterialTheme.typography.bodyMedium)
            Text(text = "Total: $${order.totalAmount}", style =
MaterialTheme.typography.bodyMedium)
            Text(text = "Items: ${order.items.size}", style =
MaterialTheme.typography.bodyMedium)
            if (order.status != "canceled") {
                Button(onClick = onCancelClick) {
                    Text("Cancel Order")
                }
            }
        }
    }
}
```

Code snippets of all usecase:

```
class AddToCartUseCase @Inject constructor(
    private val cartRepository: CartRepository
) {
    suspend operator fun invoke(userId: String, productId: String,
quantity: Int) {
        cartRepository.addCartItem(userId, productId, quantity)
    }
}
```

```
class CancelOrderUseCase @Inject constructor(
   private val orderRepository: OrderRepository
) {
   suspend operator fun invoke(orderId: String) {
       orderRepository.cancelOrder(orderId)
   }
}
```

```
class GetCartItemsUseCase @Inject constructor(
   private val cartRepository: CartRepository
) {
   suspend operator fun invoke(userId: String): List<CartItem> {
       return cartRepository.getCartItems(userId)
   }
}
```

```
class GetOrdersUseCase @Inject constructor(
   private val orderRepository: OrderRepository
) {
   suspend operator fun invoke(userId: String): List<Order> {
       return orderRepository.getOrders(userId)
   }
}
```

```
class GetProductsUseCase @Inject constructor(
   private val productRepository: ProductRepository
) {
   suspend operator fun invoke(): List<Product> {
       return productRepository.getProducts()
   }
}
```

```
class LoginUseCase @Inject constructor(
   private val userRepository: UserRepository
) {
   suspend operator fun invoke(email: String, password: String): User
{
       return userRepository.login(email, password)
   }
}
```

```
class PlaceOrderUseCase @Inject constructor(
   private val cartRepository: CartRepository,
   private val orderRepository: OrderRepository
) {
   suspend operator fun invoke(userId: String): Order {
       val cartItems = cartRepository.getCartItems(userId)
       val order = orderRepository.placeOrder(userId, cartItems)
       cartRepository.clearCart(userId)
       return order
```

```
    }
}
```

```
class RegisterUserUseCase @Inject constructor(
    private val userRepository: UserRepository
) {
    suspend operator fun invoke(username: String, email: String,
password: String): User {
        return userRepository.register(username, email, password)
    }
}
```

```
class RemoveFromCartUseCase @Inject constructor(
    private val cartRepository: CartRepository
) {
    suspend operator fun invoke(userId: String, productId: String) {
        cartRepository.removeCartItem(userId, productId)
    }
}
```

```
class UpdateCartItemQuantityUseCase @Inject constructor(
    private val cartRepository: CartRepository
) {
    suspend operator fun invoke(userId: String, productId: String,
newQuantity: Int) {
        cartRepository.updateCartItemQuantity(userId, productId,
newQuantity)
    }
}
```

Code snippets of DI modules:

```
@Module
@InstallIn(SingletonComponent::class)
object NetworkModule {

    @Provides
    @Singleton
    fun provideGson(): Gson = GsonBuilder().create()

    @Provides
    @Singleton
    fun provideRetrofit(gson: Gson): Retrofit {
        return Retrofit.Builder()
            .baseUrl("https://fakestoreapi.com/")
            .addConverterFactory(GsonConverterFactory.create(gson))
            .build()
    }

    @Provides
    @Singleton
```

```
    fun provideApiService(retrofit: Retrofit): ApiService =
retrofit.create(ApiService::class.java)
}
```

```
@Module
@InstallIn(SingletonComponent::class)
object DatabaseModule {

    @Provides
    @Singleton
    fun provideDatabase(@ApplicationContext context: Context):
AppDatabase {
        return Room.databaseBuilder(context, AppDatabase::class.java,
"app_db").build()
    }

    @Provides
    fun provideUserDao(db: AppDatabase): UserDao = db.userDao()

    @Provides
    fun provideProductDao(db: AppDatabase): ProductDao =
db.productDao()

    @Provides
    fun provideCartDao(db: AppDatabase): CartDao = db.cartDao()

    @Provides
    fun provideOrderDao(db: AppDatabase): OrderDao = db.orderDao()
}
```

```
@Module
@InstallIn(SingletonComponent::class)
abstract class RepositoryModule {
    @Binds
    @Singleton
    abstract fun bindUserRepository(impl: UserRepositoryImpl):
UserRepository

    @Binds
    @Singleton
    abstract fun bindProductRepository(impl: ProductRepositoryImpl):
ProductRepository

    @Binds
    @Singleton
    abstract fun bindCartRepository(impl: CartRepositoryImpl):
CartRepository

    @Binds
    @Singleton
```

```
    abstract fun bindOrderRepository(impl: OrderRepositoryImpl): OrderRepository
}
```

Code snippets of Repository implementations:

```
@Singleton
class CartRepositoryImpl @Inject constructor(
    private val cartDao: CartDao
) : CartRepository {
    override suspend fun getCartItems(userId: String): List<CartItem> = withContext(Dispatchers.IO) {
        val entities = cartDao.getCartItemsForUser(userId)
        entities.map { it.toDomain() }
    }

    override suspend fun addCartItem(userId: String, productId: String, quantity: Int) {
        withContext(Dispatchers.IO) {
            val cartId = ensureCartForUser(userId)
            val cartItemId = UUID.randomUUID().toString()
            val entity = CartItemEntity(
                cartItemId = cartItemId,
                cartId = cartId,
                productId = productId,
                quantity = quantity
            )
            cartDao.insertCartItem(entity)
        }
    }

    override suspend fun removeCartItem(userId: String, productId: String) {
        withContext(Dispatchers.IO) {
            cartDao.removeCartItem(userId, productId)
        }
    }

    override suspend fun clearCart(userId: String) {
        withContext(Dispatchers.IO) {
            cartDao.clearCart(userId)
        }
    }

    private suspend fun ensureCartForUser(userId: String): String {
        val cartId = UUID.randomUUID().toString()
        val cartEntity = CartEntity(
            cartId = cartId,
            userId = userId,
            createdAt = System.currentTimeMillis()
        )
```

```
        cartDao.insertCart(cartEntity)
        return cartId
    }

    override suspend fun updateCartItemQuantity(userId: String,
productId: String, newQuantity: Int) {
        cartDao.updateCartItemQuantity(userId, productId, newQuantity)
    }
}
```

```
@Singleton
class OrderRepositoryImpl @Inject constructor(
    private val orderDao: OrderDao,
    private val productDao: ProductDao
) : OrderRepository {
    override suspend fun placeOrder(userId: String, items:
List<CartItem>): Order = withContext(Dispatchers.IO) {
        val orderId = UUID.randomUUID().toString()
        val totalAmount = items.sumOf { it.quantity *
getProductPrice(it.productId) }
        val orderEntity = OrderEntity(
            orderId = orderId,
            userId = userId,
            orderDate = System.currentTimeMillis(),
            totalAmount = totalAmount,
            status = "in delivery"
        )
        orderDao.insertOrder(orderEntity)

        val orderItems = items.map {
            OrderItemEntity(
                orderItemId = UUID.randomUUID().toString(),
                orderId = orderId,
                productId = it.productId,
                quantity = it.quantity,
                price = getProductPrice(it.productId)
            )
        }
        orderDao.insertOrderItems(orderItems)

        orderEntity.toDomain(orderItems)
    }

    override suspend fun getOrders(userId: String): List<Order> =
withContext(Dispatchers.IO) {
        val orderEntities = orderDao.getOrdersForUser(userId)
        orderEntities.map { order ->
            val orderItems = orderDao.getOrderItems(order.orderId)
            order.toDomain(orderItems)
        }
```

```
    }

    override suspend fun getOrderById(orderId: String): Order? =
withContext(Dispatchers.IO) {
        val orderEntity = orderDao.getOrderById(orderId) ?:
return@withContext null
        val orderItems = orderDao.getOrderItems(orderId)
        orderEntity.toDomain(orderItems)
    }

    override suspend fun cancelOrder(orderId: String) =
withContext(Dispatchers.IO) {
        orderDao.cancelOrderById(orderId)
    }

    private suspend fun getProductPrice(productId: String): Double {
        val product = productDao.getProductById(productId)
        return product?.price ?: 0.0
    }
}
```

```
@Singleton
class ProductRepositoryImpl @Inject constructor(
    private val apiService: ApiService,
    private val productDao: ProductDao
) : ProductRepository {
    override suspend fun getProducts(): List<Product> =
withContext(Dispatchers.IO) {
        val productDtos = apiService.getProducts()
        val products = productDtos.map { it.toDomain() }
        productDao.insertProducts(products.map { it.toEntity() })
        products
    }

    override suspend fun getProductById(productId: String): Product? =
withContext(Dispatchers.IO) {
        val localProduct = productDao.getProductById(productId)
        if (localProduct != null) {
            return@withContext localProduct.toDomain()
        }

        val dto = apiService.getProductById(productId) ?:
return@withContext null
        val product = dto.toDomain()
        productDao.insertProducts(listOf(product.toEntity()))
        product
    }

    override suspend fun getCategories(): List<Category> =
withContext(Dispatchers.IO) {
```

```
        val categories = apiService.getCategories()
        categories.map { it.toCategory() }
    }
}
```

```
@Singleton
class UserRepositoryImpl @Inject constructor(
    private val userDao: UserDao
) : UserRepository {
    override suspend fun register(username: String, email: String,
password: String): User {
        return withContext(Dispatchers.IO) {
            val userId = UUID.randomUUID().toString()
            val passwordHash = password.reversed()
            val userEntity = User(
                userId = userId,
                username = username,
                email = email
            ).toEntity(passwordHash)
            userDao.insertUser(userEntity)
            userEntity.toDomain()
        }
    }

    override suspend fun login(email: String, password: String): User
{
        return withContext(Dispatchers.IO) {
            val userEntity = userDao.getUserByEmail(email)
                ?: throw Exception("User not found")

            val passwordHash = password.reversed()
            if (userEntity.passwordHash == passwordHash) {
                userEntity.toDomain()
            } else {
                throw Exception("Invalid credentials")
            }
        }
    }

    override suspend fun getUserById(userId: String): User? {
        return withContext(Dispatchers.IO) {
            userDao.getUserById(userId)?.toDomain()
        }
    }
}
```

Code snippet of AppDatabse:

```
@Database(
    entities = [
```

```
        UserEntity::class,
        ProductEntity::class,
        CartEntity::class,
        CartItemEntity::class,
        OrderEntity::class,
        OrderItemEntity::class
    ],
    version = 1,
    exportSchema = false
)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    abstract fun productDao(): ProductDao
    abstract fun cartDao(): CartDao
    abstract fun orderDao(): OrderDao
}
```

Code snippets of DAO's (other than ProductDao):

```
@Dao
interface CartDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCart(cart: CartEntity)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCartItem(cartItem: CartItemEntity)

    @RewriteQueriesToDropUnusedColumns
    @Query("SELECT CartItems.* FROM CartItems INNER JOIN ShoppingCart
ON CartItems.cartId = ShoppingCart.cartId WHERE ShoppingCart.userId =
:userId")
    suspend fun getCartItemsForUser(userId: String):
List<CartItemEntity>

    @Query("DELETE FROM CartItems WHERE productId = :productId AND
cartId IN (SELECT cartId FROM ShoppingCart WHERE userId = :userId)")
    suspend fun removeCartItem(userId: String, productId: String)

    @Query("DELETE FROM CartItems WHERE cartId IN (SELECT cartId FROM
ShoppingCart WHERE userId = :userId)")
    suspend fun clearCart(userId: String)

    @Query("UPDATE CartItems SET quantity = :newQuantity WHERE
productId = :productId AND cartId IN (SELECT cartId FROM ShoppingCart
WHERE userId = :userId)")
    suspend fun updateCartItemQuantity(userId: String, productId:
String, newQuantity: Int)
}
```

```
@Dao
interface OrderDao {
```

```
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertOrder(order: OrderEntity)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertOrderItems(items: List<OrderItemEntity>)

    @Query("SELECT * FROM Orders WHERE userId = :userId")
    suspend fun getOrdersForUser(userId: String): List<OrderEntity>

    @Query("SELECT * FROM OrderItems WHERE orderId = :orderId")
    suspend fun getOrderItems(orderId: String): List<OrderItemEntity>

    @Query("SELECT * FROM Orders WHERE orderId = :orderId LIMIT 1")
    suspend fun getOrderById(orderId: String): OrderEntity?

    @Query("UPDATE Orders SET status = 'canceled' WHERE orderId =
:orderId")
    suspend fun cancelOrderById(orderId: String): Int
}
```

```
@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUser(user: UserEntity)

    @Query("SELECT * FROM Users WHERE email = :email LIMIT 1")
    suspend fun getUserByEmail(email: String): UserEntity?

    @Query("SELECT * FROM Users WHERE userId = :userId LIMIT 1")
    suspend fun getUserById(userId: String): UserEntity?
}
```