KAZAKH-BRITISH TECHNICAL UNIVERSITY

Final Project
Mobile Programming
Mobile Shopping App Development in Kotlin

Prepared by:
Seitbekov S.
Checked by:
Serek A.

Almaty, 20.12.2024

Table of Contents

Executive Summary

This report describes the entire development process of a mobile application for shopping, which was designed using Kotlin and Jetpack Compose. The objectives of this project were the provision of a modern user-oriented shopping experience by leveraging state-of-the-art tools and practices in Android application development. With a smooth user interface, strong data management, and real-time communication possibilities, this application will constitute an integrated solution for online shopping.

It decided to use Jetpack Compose because it is declarative, saving the cruft of the traditional XML layouts, and makes the UI dynamic for the user. Kotlin brought with it concise syntax along with many advanced features like null safety and coroutines that smoothen the development process for reliability and maintenance. Room for local data storing, Retrofit for efficient API communication, WebSocket for real-time updates. These were then integrated into one clean, modular architecture that allowed scalability and low coupling of its components.

It also integrated all basic features related to shopping: browsing products, the usage of a shopping cart, and ordering. Further development could include advanced personalization, support for analytics, and better localization to reach more users. This report will go into detail over every step in development, from system architecture to real-time features, and provides insights into best practices concerning current mobile application development.

Introduction

Lately, Android has made incredible strides regarding the adoption of Kotlin and Jetpack Compose. Kotlin, besides being an officially recommended language for developing Android apps, strikes a perfect balance between simplicity and power. In addition to null safety, extension functions are just some of the features of Kotlin that will go down well for complex app development. While Jetpack Compose revolutionizes how user interface design was done with XML layouts by using a declarative programming model, it intuitively allows developers to create dynamic and visually appealing UIs.

M-commerce applications are in a way core e-commerce that enables users to search for products, compare prices, and complete purchases through mobile devices. Considering the greater demand for highly interactive responsive applications, the adoption of modern development practices is vital to guarantee user satisfaction and competitive advantage.

The main aim of the given project was to design and create a feature-rich mobile shopping application tailored to modern user needs. Its objectives were to:

1. Create a responsive, visually appealing UI using Jetpack Compose.
2. Make sure it is seamlessly integrated with any backend API via real-time data synchronization provided through WebSocket and REST API with Retrofit.
3. Provide the use of local storage using Room, enabling offline capability for persisting data.
4. Create modular architecture that is scalable and long term maintainable.
5. Practical demonstration of using Kotlin and Jetpack Compose in handling modern mobile development challenges.

The project revolves around the basic functionalities that would be provided by any shopping application: browsing products, filtering, adding items into a shopping cart, and processing checkout. FastAPI at the backend handles real-time updates and provides guaranteed

interaction between the client and the server. Given the focus of this project, personalization, deep analytics, and rich multi-language support had to be left out. Yet, with this module being well laid out, this is quite achievable in later developments.

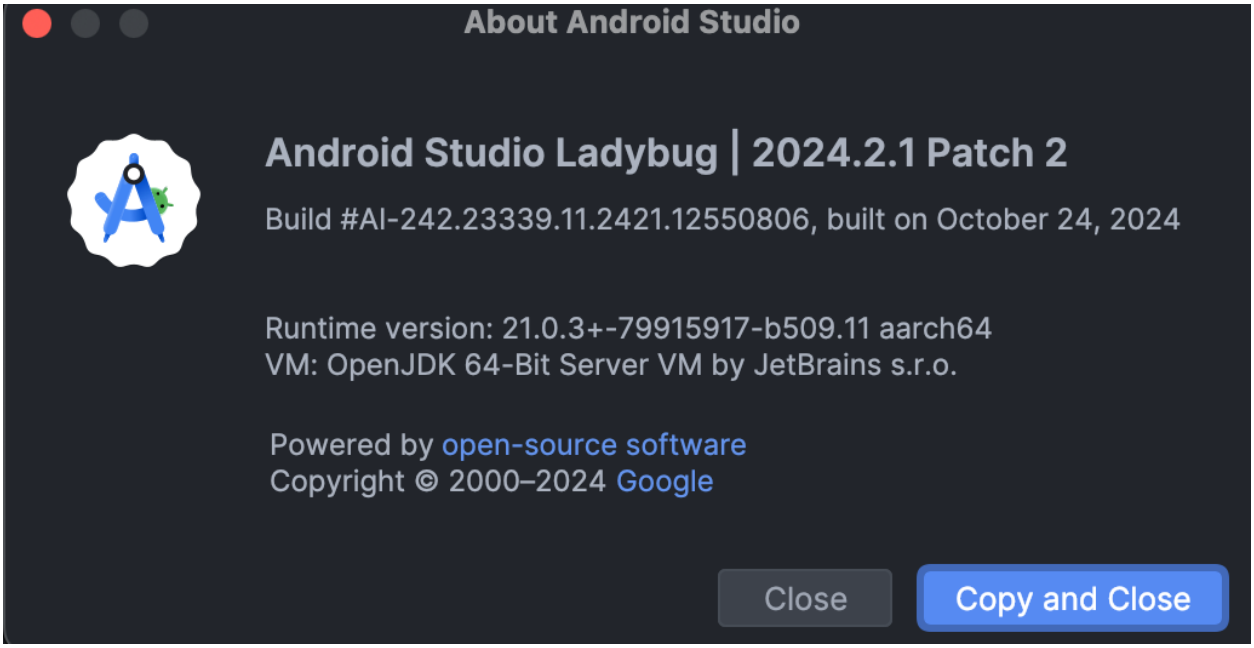


Figure 1. Android Studio Version

For development I use Android Studio. This IDE, Android Studio Ladybug (2024.2.1), covers all toolsets used in coding, debugging, and performance analysis. The choice of Kotlin as the major language is due to its concise syntax and modern features. Moreover, the latest Android SDKs have been installed to keep the program backward compatible with newer versions of Android.

```
1   plugins {
2       id("com.android.application")
3       id("org.jetbrains.kotlin.android")
4       alias(libs.plugins.kotlin.compose)
5       id("com.google.devtools.ksp")
6       id("kotlin-kapt")
7       id("com.google.dagger.hilt.android")
8   }
9
```

Figure 2. Gradle Plugins

Kotlin was chosen because it features modern design paradigms, null safety, functional language support, and interoperability with Java, among other reasons. The syntax provides a lot of reduction in boilerplate code, making the code cleaner and more readable. Null safety features protect against common runtime crashes due to null pointer exceptions. Functional programming features like higher-order functions and lambda expressions make it flexible and enable concise and expressive code.

System Architecture

Mobile shopping application architecture is designed for a system architecture that has strong module separation, scalability, maintainability, and therefore highly extensible, well testable. Concerns are separated into well-separated layers that guarantee extendibility and testability. By combining modern tools, Jetpack Compose, Room, Retrofit, and Kotlin Coroutines, the architecture reaches full stack development from UI to business logics to data management seamlessly. This architectural construction would include the following layers of:

1. UI Layer: The UI layer implemented with the use of Jetpack Compose will contain logic over user interactions and view visualization. The UI is going to be dynamically created, composable by composable using Jetpack Compose; hence, highly reactive with good UX.
2. View Model Layer: The ViewModel layer provides a bridge between the UI and the data layers, guiding the application state and business logics. It exposes the data as StateFlow or LiveData to ensure that UI can respond to changes in a productive manner.
3. Repository Layer: This layer is supposed to abstract the data sources and provide a unified interface for data retrieval and manipulation. It should interact with Room for local storage and Retrofit for network communication, hence providing an integration of local and remote data in a seamless way.
4. Data Layer: This layer should be responsible for data management; it includes Room for structured database storage and Retrofit for HTTP requests to the backend API.

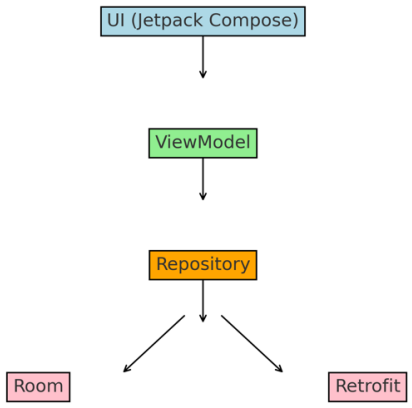


Figure 3. System Architecture Diagram

The UI layer is implemented in Jetpack Compose, which replaces traditional XML layouts with a declarative UI paradigm. This greatly simplifies the creation of dynamic and interactive interfaces. For example, the product list is rendered using the LazyColumn composable, which efficiently renders large datasets by only creating views for visible items.

```
        }
        Spacer(modifier = Modifier.height(16.dp))
        Text(text = "Reviews", style = MaterialTheme.typography.titleMedium)
        Spacer(modifier = Modifier.height(8.dp))
    }
    items(reviews) { review ->
        ReviewItem(review = review)
    }
}
```

Figure 4. List of Reviews with LazyColumn & ReviewItem

In this snippet, LazyColumn dynamically displays the list of reviews. Every review in the list is rendered as a ReviewItem composable, which encapsulates the visual representation and behavior of a product in that list.

The ViewModel is a single source of truth in respect to UI and mediates data processing between UI and repository layer. For managing data operation, asynchronous or not, it will leverage Kotlin Coroutines in ViewModel to let the view model do computationally intense jobs, like fetching data over a network without freezing the UI.

```
22  @HiltViewModel
23  class ShopViewModel @Inject constructor(
24      private val repository: Repository
25  ) : ViewModel() {
26
27      private val _categories = MutableStateFlow<List<Category>>(emptyList())
28      val categories: StateFlow<List<Category>> = _categories.asStateFlow()
29
30      private val _products = MutableStateFlow<List<Product>>(emptyList())
31      val products: StateFlow<List<Product>> = _products.asStateFlow()
32
```

Figure 5. ShopView Model

Here, ShopViewModel initializes the product list by pulling data from the repository. Use of StateFlow here would automatically update the UI when there is a change in the Product List.

The Repository layer abstracts the complexities of fetching and updating data and returns a cleaned API to the ViewModel. It serves as good separation of concerns, which insulates an application from possible future changes-for example, moving between local and remote data sources-at minimal disruption in the other layers of the system.

```kotlin
class Repository @Inject constructor(
    private val apiService: ApiService,
    private val db: AppDatabase,
    private val webSocketClient: WebSocketClient,
    private val cartDao: CartDao,
    private val reviewDao: ReviewDao,
    private val orderManager: OrderManager,
) {
    private val userDao = db.userDao()
    private val productDao = db.productDao()
    private val categoryDao = db.categoryDao()
    private val orderDao = db.orderDao()

    suspend fun registerUser(user: User) = userDao.insertUser(user)

    suspend fun getUserByEmail(email: String) = userDao.getUserByEmail(email)

    suspend fun getCategories(): List<Category> {
        val categories = apiService.getCategories()
        categoryDao.insertCategories(categories)
        return categories
    }

    suspend fun getProducts(): List<Product> {
        return try {
            val products = apiService.getProducts()
            productDao.insertProducts(products)
            products
        } catch (e: Exception) {
            productDao.getProducts()
        }
    }
```

Figure 6. The Repository Layer

In the example above, it first attempts to fetch products from a remote API. If the API call fails, for instance, a network failure, it resorts to using the data locally persisted with Room.

The data layer consists of Room for local database management and Retrofit for API communication. Room is an abstraction layer over SQLite that offers type safety and compile-time verification of SQL queries. Retrofit simplifies HTTP requests and supports efficient parsing of JSON responses. This ProductDao interface defines methods for interacting with the product table in the local database.

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

    @Query("SELECT * FROM products")
    suspend fun getProducts(): List<Product>

    @Query("SELECT * FROM products WHERE category = :categoryName")
    suspend fun getProductsByCategory(categoryName: String): List<Product>

    @Query("SELECT * FROM products WHERE productId = :productId")
    suspend fun getProductById(productId: Int): Product?
}
```

Figure 7. ProductDao Interface

Because it follows this architecture, it offers an application that is both highly modular and scalable. Clear separation of concerns makes for good maintainability and testability, where one constituent could be updated or changed without affecting the rest.

Table Descriptions

The data model of the application was designed in such a way that it is structured to handle users, products, orders, and items in the cart. Every table in the database has a corresponding entity in the application. Each one of these is used to manage some core aspects of the application, be it user authentication, listing of products, order processing, or managing the cart. Tables are defined using Kotlin data classes annotated with the Room's annotations, hence allowing smooth integration with the database.

The User's table is one of the most important parts of the application since the information about each user, in particular, their ID, username, and email address are stored here. The userId should be the primary key here, so it is auto incremented to let each of them have some unique identifier. The username field holds the display name of the user, while the email field is a unique constraint used for authentication and communication. This way, each user will have a different identity in the application.

```kotlin
package com.example.finalproject.model


import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "users")
data class User(
    @PrimaryKey val userId: Int, val username: String, val email: String, val passwordHash: String

)
```

Figure 8. User Data Class

This data class uses Room annotations that map fields to database columns. The @PrimaryKey identifies userId as a primary key of the table, while @Entity links the class with a user's table in the database. The autoGenerate attribute here makes sure that IDs will be automatically generated, saving one from possible conflicts.

The Products tableware is designed to store item information available in the stores for sale. Each product has a unique productId, name, description, price, image URL, and category. These fields will support efficient retrieval and presentation of product details in an application. For example, it could be used to filter out the products by category for smoothly browsing the items by users.

```
package com.example.finalproject.model

import androidx.room.Entity
import androidx.room.PrimaryKey

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

Figure 9. Product Data Class

As for instance, in the following entity, fields like name and description are strings containing details about the product, while the field of price is of type Double in nature to handle monetary values. The field of imageUrl holds a string pointing to the product image, and category for categorizing the products will allow features in an app such as filtering upon the category.

```
package com.example.finalproject.model

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "orders")
data class Order(
    @PrimaryKey val orderId: Int,
    val userId: Int,
    val orderDate: String,
    val totalAmount: Double,
    val status: String
)
```

Figure 10. Order Data Class

This will primarily be used for user transactions within the Orders table. Fields included in this table would comprise orderId, userId, orderDate, totalAmount, and status. The userId would provide the foreign key relationship with the order by a particular user to know from where it

came. It contains the date of a transaction in the orderDate, the sum of all items' prices involved in that order in the totalAmount field, and, finally, the status, depicting the step up to which the order has reached, say “Processing” or “Delivered”.

This design turns the order management robust and traceable. The shared field userId with the Users table allows queries to fetch all orders of a certain user. Example: Every time a user opens his profile in an app, it retrieves the orders associated with him.

```kotlin
package com.example.finalproject.model

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "cart_items")
data class CartItem(
    @PrimaryKey(autoGenerate = true) val cartItemId: Int = 0,
    val cartId: Int,
    val productId: Int,
    val quantity: Int
)
```

Figure 11. CartItem Data Class

Finally, the shopping cart functionality is enabled through the Cart table. This would include fields such as cartItemId, userId, productId, and quantity. This table links users to their selected products in managing their shopping carts dynamically.

These tables together merge into one coherent, flexible data model from which complex queries can be made, such as lists of all products in the user's cart or details about an order. It enhances the user experience by making data handling seamless and efficient.

Overview of Android Development: Intro to Kotlin

Kotlin is a best choice for modern Android development; it's more concise, safe, and interoperable with Java. In the present project, Kotlin was used to make this mobile application for shopping more expressive and responsive by embedding a lot of features in it. By using the powerful features of Kotlin language, it developed this application more productive, less boilerplate code reliably.

Kotlin does save developers from such common runtime errors with, for example, null safety. In this context of the project, it has treated all nullable fields clearly and decreased the chances of unexpected crashes due to the Kotlin type system. Below is an example with using the? operator and a function let by Kotlin, in safely processing the responses that can be null when fetching some data from the server.

```
LazyColumn(
    modifier = Modifier.weight(1f)
) {
    items(cartItems) { cartItem →
        val product = products.find { it.productId == cartItem.productId }
        product?.let {
            CartItemRow(cartItem = cartItem,
                product = it,
                onRemove = { cartViewModel.removeFromCart(cartItem) },
                onQuantityChange = { newQty →
                    cartViewModel.updateQuantity(cartItem, newQty)
                })
        } ?: run {
            Text( text: "Product not found")
        }
    }
}
```

Figure 12. Kotlin null safety

The following snippet shows null-safety enforcement: This will give a text message, rather than trying to use an access to a null reference and crashing.

```
1  package com.example.finalproject.utils
2
3  fun Double.toCurrency(): String {
4      return "$${"%.2f".format( ...args: this)}"
5  }
6
```

Figure 13. Kotlin extension functions

Another important feature of Kotlin is the support of extension functions, which help the developer add functionality to classes without changing its source code. This has been used here to make the code more readable and reusable. A good example can be seen as; to format currency values throughout the app, an extension function has been created that is:

```
Spacer(modifier = Modifier.height(16.dp))
Text(
    text = "Total: ${totalAmount.toCurrency()}",
    style = MaterialTheme.typography.titleLarge
)
```

Figure 14. Using common extension function

Another important factor of this project was Kotlin support for coroutines. That allowed one to perform asynchronous tasks without blocking the main thread. ViewModels use View Model to bind asynchronous tasks to the lifecycle of the View Model to avoid memory leaks:

```
fun register(user: User, onSuccess: () → Unit, onError: (String) → Unit) {
    viewModelScope.launch {
        try {
            repository.registerUser(user)
            onSuccess()
        } catch (e: Exception) {
            onError(e.message ?: "Registration failed")
        }
    }
}
```

Figure 15. Kotlin couroutines

This made it way easier to handle long-running tasks, thus enhancing the performance and responsiveness of the app. Working with Kotlin in this project has not only accelerated development but also made it robust and maintainable.

Functions and Lambdas in Kotlin

Functions are more expressive in Kotlin and consist of the backbone for logic in an application. This allows for reusability of code, modularity of code, and hence simplifies the more complicated operations. Lambdas provide concise ways to pass behaviors as parameters and enable functional programming paradigms within applications.

Through it all, functions were put to good use, from the calculation of totals down to fetching and updating the UI. Here is a typical function, calculateCartTotal, which calculates the total price for items inside the shopping cart:

```
fun calculateCartTotal(cartItems: List<CartItem>, products: List<Product>): Double {
    return cartItems.sumOf { cartItem →
        val product = products.find { it.productId == cartItem.productId }
        product?.price?.times(cartItem.quantity) ?: 0.0
    }
}

val totalAmount = calculateCartTotal(cartItems, products)
```

Figure 16. calculateCartTotal function

This function makes great use of Kotlin's sumOf and find functions for an efficient total. The lambda inside sumOf dynamically evaluates the price of each cart item. Here's how functional

programming in Kotlin makes many tasks concise.

Lambdas were also used for event handling. For example, in the shopping cart screen, a lambda was passed onto the CartItemRow composable to handle item removal:

```
@Composable
fun CartItemRow(
    cartItem: CartItem, product: Product, onRemove: () → Unit, onQuantityChange: (Int) → Unit
) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp),
        elevation = CardDefaults.cardElevation(defaultElevation = 4.dp)
    ) {
        Row(
            modifier = Modifier.padding(16.dp), verticalAlignment = Alignment.CenterVertically
        ) {
```

Figure 17. Lambda function

Here, onRemove is a lambda function that gets passed as a parameter to handle the removal of items dynamically. This decouples the UI from the underlying logic and makes the code modular and testable.

Kotlin also supports higher-order functions, which take other functions as parameters or return them as the results. Higher-order functions were used in this project for filtering products based on categories:

```
    private fun observeSelectedCategory() {
        viewModelScope.launch {
            combine(
                _categories, _products, _selectedCategoryId
            ) { categories, products, selectedCategoryId →
                val category = categories.find { it.categoryId == selectedCategoryId }
                if (category ≠ null) {
                    products.filter {
                        it.category.equals(category.categoryName, ignoreCase = true)
                    }
                } else {
                    emptyList()
                }
            }.collect { filtered →
                _filteredProducts.value = filtered
                Log.d( tag: "ShopViewModel",  msg: "Filtered Products: $filtered")
            }
        }
    }
```

Figure 18. Higher-order function

This function contains a lambda within the filter function for whether the product is in each category. Functions/Lambdas have been important for this project, allowing clean, readable, maintainable code without much redundancy.

OOP in Kotlin

This work greatly employed object-oriented programming principles such as encapsulation, inheritance, and polymorphism in the establishment of reusable and scalable code, hence ensuring an application was well-structured.

Encapsulation will be ensured by packaging data and functions into one using classes. For example, the Product class can be used to encapsulate the attributes of a product:

```
1   package com.example.finalproject.model
2
3   import androidx.room.Entity
4   import androidx.room.PrimaryKey
5
6   @Entity(tableName = "products")
7   data class Product(
8       @PrimaryKey val productId: Int,
9       val name: String,
10      val description: String,
11      val price: Double,
12      val imageUrl: String,
13      val category: String
14  )
```

Figure 19. Encapsulation example

This class provides a template for all entities of products so that consistency can be assured within the application.

Inheritance was done with an aim to extend base classes. An example can be a ComponentActivity class that handled common tasks which would be further extended by concrete MainActivity with overriding methods:

```
12  @AndroidEntryPoint
13  class MainActivity : ComponentActivity() {
14      override fun onCreate(savedInstanceState: Bundle?) {
15          super.onCreate(savedInstanceState)
16          setContent {
17              PCShoppingAppTheme {
18                  Surface(color = MaterialTheme.colorScheme.background) {
19                      NavGraph()
20                  }
21              }
22          }
23      }
24  }
```

Figure 20. Inheritance function

Polymorphism has been used to achieve interchangeable and flexible components. One of the typical examples could be the usage of a Product pattern by means of a Base Open Class; this way, multiple implementations are enabled:

```kotlin
open class CommonProduct(
    val productId: Int,
    val name: String,
    val price: Double,
    val color: String,
    val size: String
) {
    open fun getDetails(): String {
        return "Name: $name, Price: $price USD, Color: $color, Size: $size"
    }
    fun calculateShipping(): Double {
        return when {
            price < 50 -> 5.0
            else -> 15.0
        }
    }
}
class PhysicalProduct(
    productId: Int,
    name: String,
    price: Double,
    val weight: Double,
    color: String,
    size: String
) : CommonProduct(productId, name, price, color, size) {
    override fun getDetails(): String {
        return super.getDetails() + ", Weight: $weight kg, Shipping: ${calculateShipping()} USD"
    }
}
class DigitalProduct(
    productId: Int,
    name: String,
    price: Double,
    val downloadLink: String, color: String, size: String
) : CommonProduct(productId, name, price, color, size) {
    override fun getDetails(): String {
        return super.getDetails() + ", Download Link: $downloadLink"
    }
}
```

Figure 21. Polymorphism example

By following object-oriented programming principles, this made the project highly modular and reusable, hence easier to maintain and further develop.

Working with Collections in Kotlin

Collections are probably the very heart of each programming language; Kotlin features a rich API for dealing with collections: lists, sets, and maps. Full usage of collections could be found within this project, while handling or manipulating data like product lists, shopping cart items, and

order histories. Collection functions in Kotlin, such as filtering, are concise and expressive.

One of the common usages included developing a product list based on categories. Following is a simple example of how a filter operation can be done dynamically to filter out certain products:

Figure 22. Filtering of products

In the following function, the filter method returns a new list including only those products that have a match to the given category. The ignoreCase parameter makes the filtering case insensitive for better usability.

The collection concepts were also applied for the implementation of handling the shopping cart. For getting the total price of the items inside the cart, the sumOf method will look like this:

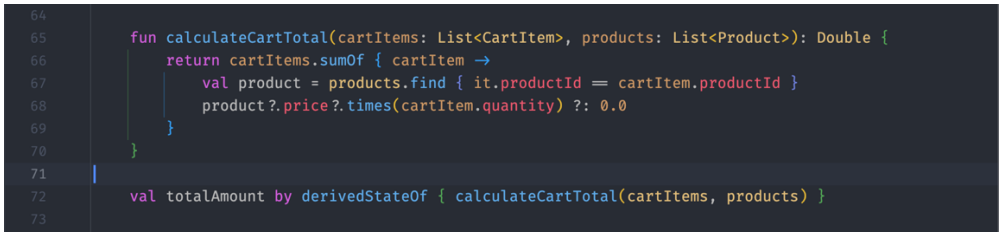


Figure 23. sumOf method for calculateCartTotal

This function shows the power of Kotlin's collection API since from here onwards the sumOf function will calculate the total price by going through the cartItems and getting the price of the product dynamically.

Besides that, groupBy was used to group the products for their effective display on the shop screen:

Figure 24. groupBy method for grouping products by category

It structures the products into a map with category names as keys and the lists of products in each category as values. These in turn will then be used to show lists in the UI that are grouped by category.

Kotlin has mutable and immutable collections. Immutable collections are typically used because of better thread safety and predictability of the behavior in cases when there was no explicit need for mutability. For instance, an update of a shopping cart relied on a mutable list:

```
    val cartItems = mutableListOf<CartItem>()

    fun addToCart(cartItem: CartItem) {
        cartItems.add(cartItem)
    }
```

Figure 25. Mutable list for cartItems

This helped the project take full advantage of Kotlin's powerful collection APIs while working with data efficiently, avoiding boilerplate code; thus, implementations were cleaner and easier to maintain.

Jetpack Compose: Replacing XML Layouts

Jetpack Compose is a new UI toolkit that allows developers to create declarative and dynamic user interfaces. Unlike the traditional thought of XML layouts, in Jetpack Compose, no XML files are being used; instead, it introduces the new way of designing a user interface. This project will make use of a concept called Jetpack Compose in developing responsive interactive screens that will make updating UIs easier and manage states far more easily.

Probably the biggest advantage brought by Jetpack Compose is the possibility to realize composable functions, which allow making reusable and modular UI elements. Let's continue with the implementation of the product list screen for efficient rendering of a big number of items with the help of LazyColumn:

```
            filteredProducts.isEmpty() → {
                Box(
                    modifier = Modifier
                        .padding(padding)
                        .fillMaxSize(),
                    contentAlignment = Alignment.Center
                ) {
                    Text( text: "No products found in this category.")
                }
            }

            else → {
                LazyColumn(
                    modifier = Modifier
                        .padding(padding)
                        .padding(16.dp)
                ) {
                    items(filteredProducts) { product →
                        ProductItem(product = product, onClick = {
                            navController.navigate(Screen.ProductDetail.createRoute(product.productId))
                        })
                    }
                }
            }
        }
    })
}
```

Figure 26. LazyColumn example

It would avoid the boilerplate or complexity with traditional RecyclerView and adapters. By default, LazyColumn would automatically recycle its items for it in the most efficient way to conserve CPU and memory resources.

The other very critical feature in Jetpack Compose would be how states are dealt with. For this current project, remember and mutableStateOf were used for storing things like UI state:

```
@Composable
fun CheckoutScreen(
    navController: NavController,
    checkoutViewModel: CheckoutViewModel,
    cartViewModel: CartViewModel,
    shopViewModel: ShopViewModel
) {
    var street by remember { mutableStateOf( value: "") }
    var city by remember { mutableStateOf( value: "") }
    var state by remember { mutableStateOf( value: "") }
    var zipCode by remember { mutableStateOf( value: "") }
    var paymentMethod by remember { mutableStateOf( value: "Credit Card") }
    var errorMessage by remember { mutableStateOf( value: "") }

    val cartItems by cartViewModel.cartItems.collectAsState()
    val products by shopViewModel.products.collectAsState()

```

Figure 27. remember and mutableStateOf methods

This snippet has demonstrated the ways in which composable functions work with mutable states in changing dynamically with user input.

Animations and Transitions are also better with Jetpack Compose. Implementation was added for smoothly transitioning between the states, for instance:

```
                Spacer(modifier = Modifier.height(16.dp))
                Button(
                    onClick = {
                        cartViewModel.addToCart(product.productId)
                        navController.navigate(Screen.Cart.route)
                    }, modifier = Modifier.scale(scale)
                ) {
                    Text( text: "Add to Cart")
```

Figure 28. Compose animation with scaling

This project accomplished the feat of having cleaner architecture, faster development cycles, and more maintainable UI code by leveraging the powers of Jetpack Compose. Strongly reduced bugs due to this declarative approach meant great productivity for the production.

Activity: Handling User Input and Events

Activities in Android are entry points for user interaction. In the project, a single-activity architecture was used to house all navigations and lifecycle management, while screens and composables were managed with Jetpack Compose and the Jetpack Navigation component. It provided a more straightforward, efficient way to manage user inputs and events throughout the app.

One of the core features of the main activity was to handle the navigation between different screens. Jetpack Navigation was used to define routes to other composable screens. For example, navigating from the login screen to the home screen based on successful authentication was implemented this way:

```
NavHost(
    navController = navController, startDestination = Screen.Login.route, modifier = modifier
) {
    composable(Screen.Register.route) {
        RegisterScreen(navController = navController)
    }
    composable(Screen.Login.route) {
        LoginScreen(navController = navController)
    }
    composable(Screen.Shop.route) {
        ShopScreen(
            navController = navController, viewModel = shopViewModel
        )
    }
```

Figure 29. Navigation Host

In the above setup, NavController manages the navigation between LoginScreen and HomeScreen. These saves developing multiple activities and hence reduces state management, which is a headache to deal with.

User input forms one of the most important bases of any application. As such, this project employed Composable components, for example, OutlinedTextField, to capture input in real time. For instance, the login screen would capture the user's email and password:

```
        OutlinedTextField(
            value = password,
            onValueChange = { password = it },
            label = { Text( text: "Password") },
            modifier = Modifier.fillMaxWidth(),
            visualTransformation = PasswordVisualTransformation()
        )
        Spacer(modifier = Modifier.height(8.dp))
        OutlinedTextField(
            value = confirmPassword,
            onValueChange = { confirmPassword = it },
            label = { Text( text: "Confirm Password") },
            modifier = Modifier.fillMaxWidth(),
            visualTransformation = PasswordVisualTransformation()
        )
        Spacer(modifier = Modifier.height(16.dp))
        Button(
            onClick = {
                if (password ≠ confirmPassword) {
                    errorMessage = "Passwords do not match"
                } else if (!validateCredentials(email, password)) {
                    errorMessage = "Password must be at least 6 characters and email must contain '@'"
                } else {
```

Figure 30. OutlinedTextField

This code validates inputs immediately and returns instant results for wrong entries through validation logic. This validateCredentials function allows input data in a specified format only:

```
40
41      fun validateCredentials(email: String, password: String): Boolean {
42          return email.contains( other: "@") && password.length > 6
43      }
44
```

Figure 31. validateCredentials function

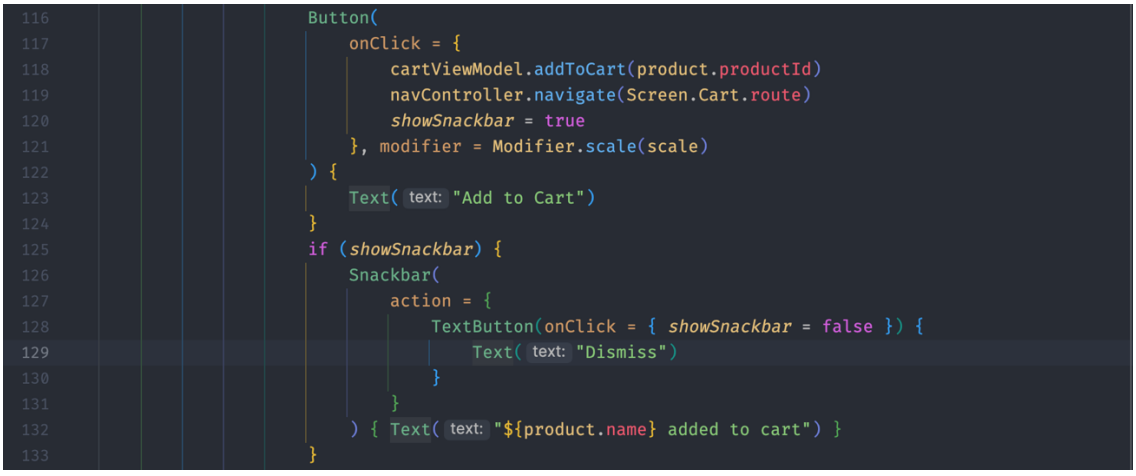
Besides mere input handling, activities handle more complex user events such as updating carts or filtering products. Product filtering dynamically changed the list of visible products based on user-chosen categories:

```
96                  else → {
97                      LazyColumn(
98                          modifier = Modifier
99                              .padding(padding)
100                             .padding(16.dp)
101                     ) {
102                         items(filteredProducts) { product →
103                             ProductItem(product = product, onClick = {
104                                 navController.navigate(Screen.ProductDetail.createRoute(product.productId))
105                             })
106                         }
107                     }
108                 }
109             }
110     })
```

Figure 32. Dynamic filtering of products

Above, the code dynamically updates the product list through filtered items based on the currently chosen category. This example shows how user interactions trigger state changes that get applied directly in the UI thanks to the declarative nature of Jetpack Compose.

Providing real-time feedback to the user is paramount for a good user experience. In this project, snackbar was used to either inform users of successful actions or show errors. For example, after an item had been added into the cart, a snackbar confirms the action:



```
116                 Button(
117                     onClick = {
118                         cartViewModel.addToCart(product.productId)
119                         navController.navigate(Screen.Cart.route)
120                         showSnackbar = true
121                     }, modifier = Modifier.scale(scale)
122                 ) {
123                     Text( text: "Add to Cart")
124                 }
125                 if (showSnackbar) {
126                     Snackbar(
127                         action = {
128                             TextButton(onClick = { showSnackbar = false }) {
129                                 Text( text: "Dismiss")
130                             }
131                         }
132                     ) { Text( text: "${product.name} added to cart") }
133                 }
```

Figure 33. Snackbar Confirmation

Centralizing navigation in an activity makes the app easier to understand and makes event handling predictable. State management, such as StateFlow, was used in the ViewModel to coordinate UI changes across screens:

```
@HiltViewModel
class ShopViewModel @Inject constructor(
    private val repository: Repository
) : ViewModel() {

    private val _categories = MutableStateFlow<List<Category>>(emptyList())
    val categories: StateFlow<List<Category>> = _categories.asStateFlow()

    private val _products = MutableStateFlow<List<Product>>(emptyList())
    val products: StateFlow<List<Product>> = _products.asStateFlow()

    private val _selectedCategoryId = MutableStateFlow<Int?>( value: null)
    val selectedCategoryId: StateFlow<Int?> = _selectedCategoryId.asStateFlow()

    private val _filteredProducts = MutableStateFlow<List<Product>>(emptyList())
    val filteredProducts: StateFlow<List<Product>> = _filteredProducts.asStateFlow()

    private val _error = MutableStateFlow<String?>( value: null)
    val error: StateFlow<String?> = _error.asStateFlow()
```

Figure 34. ShopViewModel with StateFlow

By combining Jetpack Navigation, stateful composables, and dynamic event handling, this activity provides a solid foundation for handling user interactions and offering a seamless experience in this project.

Activity Lifecycle

Understanding the Activity Lifecycle and how to properly handle it is very important while writing robust Android applications. An activity life cycle portrays various states an Activity undergoes: created, started, resumed, paused, stopped, or destroyed by action from the user, by the system, and many more. In the proposed project, much of these traditional lifecycle-related complexities were abstracted away, where Jetpack Compose together with ViewModel has been used. It is, nonetheless, great to know how these lifecycle events are affecting the application.

Lifecycle-aware components handle preparation and cleanup of resources - such as network connections or database queries. Jetpack Compose handles most of the UI concerns internally, placing less of a requirement on overriding many lifecycle methods. Still, the onCreate method in the activity initializes key components, including the navigation controller and other parts:

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import com.example.finalproject.ui.theme.PCShoppingAppTheme
import com.example.finalproject.ui.theme.navigation.NavGraph
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            PCShoppingAppTheme {
                Surface(color = MaterialTheme.colorScheme.background) {
                    NavGraph()
                }
            }
        }
    }
}
```

Figure 35. MainActivity with onCreate method

Here, onCreate initializes the navigation graph that is going to set the main entry point of user interaction. In this example, using Jetpack Compose avoids lifecycle dependencies directly in UI components.

The ViewModel architecture component is lifecycle-aware and survives configuration changes, such as screen rotation. That means the application data will survive, and be available to the UI, independent of the activity lifecycle state: For example, this ShopViewModel fetches some product data and survives lifecycle events:

```
@HiltViewModel
class ShopViewModel @Inject constructor(
    private val repository: Repository
) : ViewModel() {

    private val _categories = MutableStateFlow<List<Category>>(emptyList())
    val categories: StateFlow<List<Category>> = _categories.asStateFlow()

    private val _products = MutableStateFlow<List<Product>>(emptyList())
    val products: StateFlow<List<Product>> = _products.asStateFlow()

    private val _selectedCategoryId = MutableStateFlow<Int?>( value: null)
    val selectedCategoryId: StateFlow<Int?> = _selectedCategoryId.asStateFlow()

    private val _filteredProducts = MutableStateFlow<List<Product>>(emptyList())
    val filteredProducts: StateFlow<List<Product>> = _filteredProducts.asStateFlow()

    private val _error = MutableStateFlow<String?>( value: null)
    val error: StateFlow<String?> = _error.asStateFlow()

    init {
        fetchCategoriesAndProducts()
        observeSelectedCategory()
    }
```

Figure 36. ShopViewModel

Lifecycle of an activity plays an important role to keep up the application behavior. Such complexity is reduced as much as possible by taking maximum benefits from lifecycle aware ViewModel and declarative state management of Jetpack Compose. Ensuring that an application remains responsive and resource-efficient within each usage scenario-from simple to complex. It is with the ViewModel that the decoupling of data management from the activity lifecycle allows the application to react gracefully to changes in the activity lifecycle without loss of critical state.

Jetpack Compose Screens and Lifecycle Awareness

Jetpack Compose is a modern way of designing the UI screens of an Android application, where the components are defined in a declarative way. In Jetpack Compose, each screen is a composable function that efficiently updates reactively. This section will go through how the implementation of such screens in this project and how they interact with the Android lifecycle.

Screens in Jetpack Compose are composable functions. For example, here is ShopScreen displays the list of product categories obtained from the ViewModel. The composable automatically reacts to changes in the state exposed by the ViewModel:

```
58              errorMessage ≠ null → {
59                  Box(
60                      modifier = Modifier
61                          .padding(padding)
62                          .fillMaxSize(),
63                      contentAlignment = Alignment.Center
64                  ) {
65                      Text(text = errorMessage ?: "An error occurred")
66                  }
67              }
68
69              else → {
70                  LazyColumn(
71                      modifier = Modifier
72                          .padding(padding)
73                          .padding(16.dp)
74                  ) {
75                      items(categories) { category →
76                          CategoryItem(category = category, onClick = {
77                              navController.navigate( route: "category/${category.categoryId}")
78                          })
79                      }
80                  }
81              }
82          }
83      })
```

Figure 37. LazyColumn of product categories

In this code snippet, LazyColumn efficiently renders a scrollable list of categories. Clickable modifier enables navigation to the selected category screen. That screen leverages state management, thus it is responsive to real-time updates of the categories state.

Jetpack Compose simplifies lifecycle awareness using lifecycle-aware ViewModels and composables. This solution minimizes boilerplate code and makes the code better readable. For instance, CategoryScreen observes filteredProducts state and changes UI dynamically:

```kotlin
@Composable
fun CategoryScreen(
    navController: NavController, categoryId: Int, viewModel: ShopViewModel
) {
    val context = LocalContext.current
    val filteredProducts by viewModel.filteredProducts.collectAsState()
    val isLoading by derivedStateOf {
        viewModel.categories.value.isEmpty() && viewModel.products.value.isEmpty()
    }
    val error by viewModel.error.collectAsState()

    LaunchedEffect(error) {
        error?.let {
            Toast.makeText(context, it, Toast.LENGTH_LONG).show()
        }
    }

    LaunchedEffect(categoryId) {
        if (categoryId != 0) {
            viewModel.selectCategory(categoryId)
        }
    }

```

Figure 38. CategoryScreen

Navigation in Jetpack Compose relies on the NavController, which manages the navigation stack and lifecycle of screens. The navigation graph defines the possible routes and their corresponding composables:

```kotlin
@Composable
fun NavGraph(modifier: Modifier = Modifier) {
    val navController = rememberNavController()

    val shopViewModel: ShopViewModel = hiltViewModel()
    val cartViewModel: CartViewModel = hiltViewModel()
    val checkoutViewModel: CheckoutViewModel = hiltViewModel()
    val profileViewModel: ProfileViewModel = hiltViewModel()
    val reviewViewModel: ReviewViewModel = hiltViewModel()
    val webSocketViewModel: WebSocketViewModel = hiltViewModel()

    NavHost(
        navController = navController, startDestination = Screen.Login.route, modifier = modifier
    ) {
        composable(Screen.Register.route) {
            RegisterScreen(navController = navController)
        }
        composable(Screen.Login.route) {
            LoginScreen(navController = navController)
        }
        composable(Screen.Shop.route) {
            ShopScreen(
                navController = navController, viewModel = shopViewModel
            )
        }
        composable(Screen.Cart.route) {
            CartScreen(
                navController = navController,
                cartViewModel = cartViewModel,
                shopViewModel = shopViewModel
            )
        }
        composable(Screen.Profile.route) {
            ProfileScreen(
                navController = navController, viewModel = profileViewModel
            )
        }
```

Figure 39. Navigation Graph

Every route in NavHost maps to a specific composable function, which guarantees lifecycle-aware navigation and encapsulation.

Jetpack Compose makes the work of writing lifecycle-aware screens easier because it provides composable declarative UI components and integrates well with the Android View lifecycle. Utilizing composables, state management, and navigation ensures that the screens are as performant, responsive, and maintainable as possible. This leaves out so much boilerplate code one would have required in their XML-based layout, making this a go-to for any modern Android app.

RecyclerView Replacement with LazyColumn

In traditional Android development, the main component to display a list of items was the RecyclerView. However, Jetpack Compose introduced LazyColumn, which can be more concise and intuitive for similar tasks. This section describes the migration from RecyclerView to LazyColumn and how it has been used in this project.

LazyColumn in Jetpack Compose eliminates the need for an adapter, a ViewHolder, and XML layout files for the list implementation. In addition, performance optimizations that were manually handled in RecyclerView, like recycling and reusing items, are automatically handled by LazyColumn.

As an example, to show a list of products in RecyclerView, the traditional approach is to perform the following steps:

1. Create an XML layout file for items.
2. Create a RecyclerView.Adapter.
3. Bind data using the onBindViewHolder method.

With LazyColumn replaces all the above with a single composable.

In this example, LazyColumn has been used extensively to show the list, be it the product catalog or the user orders. The composable, which shows the products looks concise and expressive:

```
            LazyColumn(
                modifier = Modifier
                    .padding(padding)
                    .padding(16.dp)
            ) {
                items(filteredProducts) { product →
                    ProductItem(product = product, onClick = {
                        navController.navigate(Screen.ProductDetail.createRoute(product.productId))
                    })
                }
            }
        }
    }
```

Figure 40. LazyColumn instead of RecyclerView

This is where items iterate over the list of products, and each product has been composable with ProductItem and it avoids boilerplate code and makes it more readable. The ProductItem composable is responsible to render a single product into the list. It composes of a clickable card hosting the product name, description and price of the item:

```
@Composable
fun ProductItem(product: Product, onClick: () → Unit) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .clickable { onClick() },
        elevation = CardDefaults.cardElevation(defaultElevation = 4.dp)
    ) {
        Row(modifier = Modifier.padding(16.dp)) {
            Image(
                painter = rememberAsyncImagePainter(product.imageUrl),
                contentDescription = product.name,
                modifier = Modifier.size(64.dp)
            )
            Spacer(modifier = Modifier.width(16.dp))
            Column {
                Text(text = product.name, style = MaterialTheme.typography.titleMedium)
                Spacer(modifier = Modifier.height(4.dp))
                Text(
                    text = "$${"%.2f".format(product.price)}",
                    style = MaterialTheme.typography.bodyMedium
                )
            }
        }
    }
}
```

Figure 41. Product Item for single product

This function has fully showed the flexibility of Jetpack Compose, and there was no need to include XML file for UI composition. LazyColumn allows for smooth users' interaction. For example, the app navigates to product details screen when clicked on one of the products:

```
            LazyColumn(
                modifier = Modifier
                    .padding(padding)
                    .padding(16.dp)
            ) {
                items(filteredProducts) { product →
                    ProductItem(product = product, onClick = {
                        navController.navigate(Screen.ProductDetail.createRoute(product.productId))
                    })
                }
            }
        }
    }
```

Figure 42. LazyColumn with user interaction

This implementation will make each item clickable and navigate dynamically depending on the product's ID to the appropriate screen. LazyColumn is optimized for performance, as it only renders the currently visible items. As the user scrolls, items that fall outside the visible window are automatically recycled. The behavior provides performance like RecyclerView, but the development experience is much more intuitive.

For example, let's look at a screen with user orders. Here, LazyColumn will efficiently manage the drawing of the orders so that no extra data is drawn:

```
                    LazyColumn {
                        items(userWithOrders.orders) { order ->
                            OrderItem(order = order)
                            Spacer(modifier = Modifier.height(8.dp))
                        }
                    }
                }
            }
        } ?: run {
            Box(
                modifier = Modifier
                    .padding(padding)
                    .fillMaxSize(),
                contentAlignment = Alignment.Center
            ) {
                Text( text: "No user information available")
            }
        }
    })
}

@Composable
fun OrderItem(order: Order) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 4.dp),
        elevation = CardDefaults.cardElevation(defaultElevation = 2.dp)
    ) {
        Column(modifier = Modifier.padding(16.dp)) {
            Text(text = "Order ID: ${order.orderId}", style = MaterialTheme.typography.bodyMedium)
            Spacer(modifier = Modifier.height(4.dp))
            Text(text = "Date: ${order.orderDate}", style = MaterialTheme.typography.bodyMedium)
            Spacer(modifier = Modifier.height(4.dp))
            Text(
                text = "Total: $${"%.2f".format(order.totalAmount)}",
                style = MaterialTheme.typography.bodyMedium
            )
```

Figure 43. LazyColumn for user orders

Migrating from RecyclerView to LazyColumn is a quantum leap in Android development. That makes the creation and management of lists easier, involves less boilerplate code, and seamlessly integrates with the declarative UI paradigm of Jetpack Compose. Due to the usage of LazyColumn in this example, it made the development much easier and resulted in a cleaner and maintainable code base.

ViewModel and LiveData

In modern Android app development, managing UI-related data efficiently is important. ViewModel and LiveData, as part of Android Jetpack, provide a powerful combination in handling

data lifecycle-aware, hence assuring smooth interaction between UI and business logic. This section explores the usage of ViewModel and LiveData in the shopping app by going into the details of integration and benefits.

It allows the ViewModel component to hold and manage UI-related data in a lifecycle-conscious manner. This prevents an app from losing data upon configuration changes, such as screen rotations. Throughout this shopping app, ViewModel is used extensively to manage the lists of products, the user's orders, and the state of the application. For example, this ShopViewModel fetches and stores the list of products:

```
@HiltViewModel
class ShopViewModel @Inject constructor(
    private val repository: Repository
) : ViewModel() {

    private val _categories = MutableStateFlow<List<Category>>(emptyList())
    val categories: StateFlow<List<Category>> = _categories.asStateFlow()

    private val _products = MutableStateFlow<List<Product>>(emptyList())
    val products: StateFlow<List<Product>> = _products.asStateFlow()

    private val _selectedCategoryId = MutableStateFlow<Int?>( value: null)
    val selectedCategoryId: StateFlow<Int?> = _selectedCategoryId.asStateFlow()

    private val _filteredProducts = MutableStateFlow<List<Product>>(emptyList())
    val filteredProducts: StateFlow<List<Product>> = _filteredProducts.asStateFlow()

    private val _error = MutableStateFlow<String?>( value: null)
    val error: StateFlow<String?> = _error.asStateFlow()

    init {
        fetchCategoriesAndProducts()
        observeSelectedCategory()
    }

    private fun fetchCategoriesAndProducts() {
        viewModelScope.launch {
            try {
                val fetchedCategories = repository.getCategories()
                _categories.value = fetchedCategories
                Log.d( tag: "ShopViewModel",  msg: "Categories fetched: $fetchedCategories")

                if (_selectedCategoryId.value == null && fetchedCategories.isNotEmpty()) {
                    _selectedCategoryId.value = fetchedCategories[0].categoryId
                }

                val fetchedProducts = repository.getProducts()
```

Figure 44. ShopViewModel

In this code MutableStateFlow holds the product data and fetchCategoriesAndProducts fetches data from the repository and updates the UI state. Using ViewModel makes the data survive configuration changes without requiring complex save-and-restore mechanisms. For example, if the user rotates the screen while showing the list of products, data is still there and doesn't need to be reloaded from the server.

While this project uses the StateFlow for reactivity, another common way to handle observing changes is by using LiveData. LiveData has a lifecycle awareness and only updates the active UI components, thus avoiding doing unnecessary computations. For example, in the management of the cart, one might use LiveData for observing the changes:

```
@HiltViewModel
class CartViewModel @Inject constructor(
    private val repository: Repository
) : ViewModel() {

    private val _cartItems = MutableLiveData<List<CartItem>>(emptyList())
    val cartItems: LiveData<List<CartItem>> get() = _cartItems

```

Figure 45. CartViewModel

In the above example, the LiveData will observe any change in the state of the cart items. Every time a product is added to the cart, it will be updated in the CartItems. This project focuses on StateFlow because of its Kotlin-first design and compatibility with coroutines.

The result of using ViewModel with LiveData makes sure that the layer of the app responsible for the management of data is strong enough to handle such responsibilities that regard lifecycle-awareness; that the interaction between the UI and the backend's data will not leak the activity during configuration changes-in addition, be much quicker in performance.

Working with Databases

Efficient data management is the base of any application that manages user interactions and transactions. In this project, the Room Android Jetpack component for local data storage is to be used. Room has an abstraction layer over SQLite, making database operations not only simpler but also gives compile-time checks for queries. This section elaborates the implementation of Room in the Shopping App for product storage, cart data, and its management, and user order management. Room database is annotated with @Database; it defines the entities that are the tables in this database and a version for migrations.

```
@Database(
    entities = [User::class, Product::class, Category::class, Order::class, OrderItem::class, ShoppingCart::class, Ca
    version = 1,
    exportSchema = false
)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    abstract fun productDao(): ProductDao
    abstract fun categoryDao(): CategoryDao
    abstract fun orderDao(): OrderDao
    abstract fun reviewDao(): ReviewDao
    abstract fun cartDao(): CartDao
}
```

Figure 46. Application database

Here Product, CartItem, and Order are tables in the database. Each DAO (Data Access Object) corresponds to a table from which it performs CRUD operations. The database should be

created as a singleton to avoid multiple instances being created:

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

    @Provides
    @Singleton
    fun provideDatabase(@ApplicationContext appContext: Context) =
        Room.databaseBuilder(appContext, AppDatabase::class.java, name: "pc_shopping_db")
            .fallbackToDestructiveMigration().build()

```

Figure 47. Singleton creation of database

This ensures that there will be only one instance of this database during the app lifetime. Entities define the schema of tables in the database. Thus, for example, the Product entity is:

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

Figure 48. Product Data Class

Each field of the class corresponds to the column in the database table. The @PrimaryKey and @ColumnInfo annotations define additional constraints. The Order entity follows this with the management of order data:

```
@Entity(tableName = "orders")
data class Order(
    @PrimaryKey val orderId: Int,
    val userId: Int,
    val orderDate: String,
    val totalAmount: Double,
    val status: String
)
```

Figure 49. Order Data Class

DAOs are interfaces for methods that interact with the database. They use annotations, such as @Query, @Insert, and @Delete, to simplify most database operations. For instance, the ProductDao:

Figure 50. Product Dao Interface

This DAO is using for:

1. Fetches products filtered by category.
2. Inserts or updates products in the database.

The benefits of using Room are as follows:

1. Ease of Use: Annotations ease database operations.
2. Compile-time Checks: Queries are validated at compile time, and runtime errors are reduced.
3. Seamless Integration: Works efficiently with LiveData and Kotlin coroutines for reactive programming.

By utilizing Room, the app guarantees strong data persistence with seamless offline support, enhancing the user experience greatly.

Retrofit

Retrofit is a powerful and flexible library for HTTP communication in Android, which can efficiently integrate with RESTful APIs. In this project, Retrofit is used to fetch data from the FastAPI backend and synchronize it with the local database. Further, this section will explain its implementation: setup, API interface definitions, and integration with other components.

Retrofit is, in fact initialized in a more central module to ensure its reusability and maintainability. The interface ApiService declares what endpoints exist on a given API and which HTTP methods are expected at the endpoints. For example, a GET request to /products should return all the products available on the server. ApiService interface would look like this:

Figure 51. ApiService with endpoints

To use Retrofit, make a singleton instance using Retrofit.Builder. This ensures that every network request uses the same configuration:

Figure 52. Singleton Creation of Retrofit

baseUrl is the root URL of the server, and GsonConverterFactory transforms JSON responses into Kotlin objects automatically. Once the Retrofit instance is built, supply an implementation of the ApiService using the retrofit.create() method.

The ViewModel orchestrates the repository and the UI. For example, the ShopViewModel fetches the products and exposes them as a StateFlow. In UI it observes that state, that will allow to the UI know if there's new data available to show this list of products:

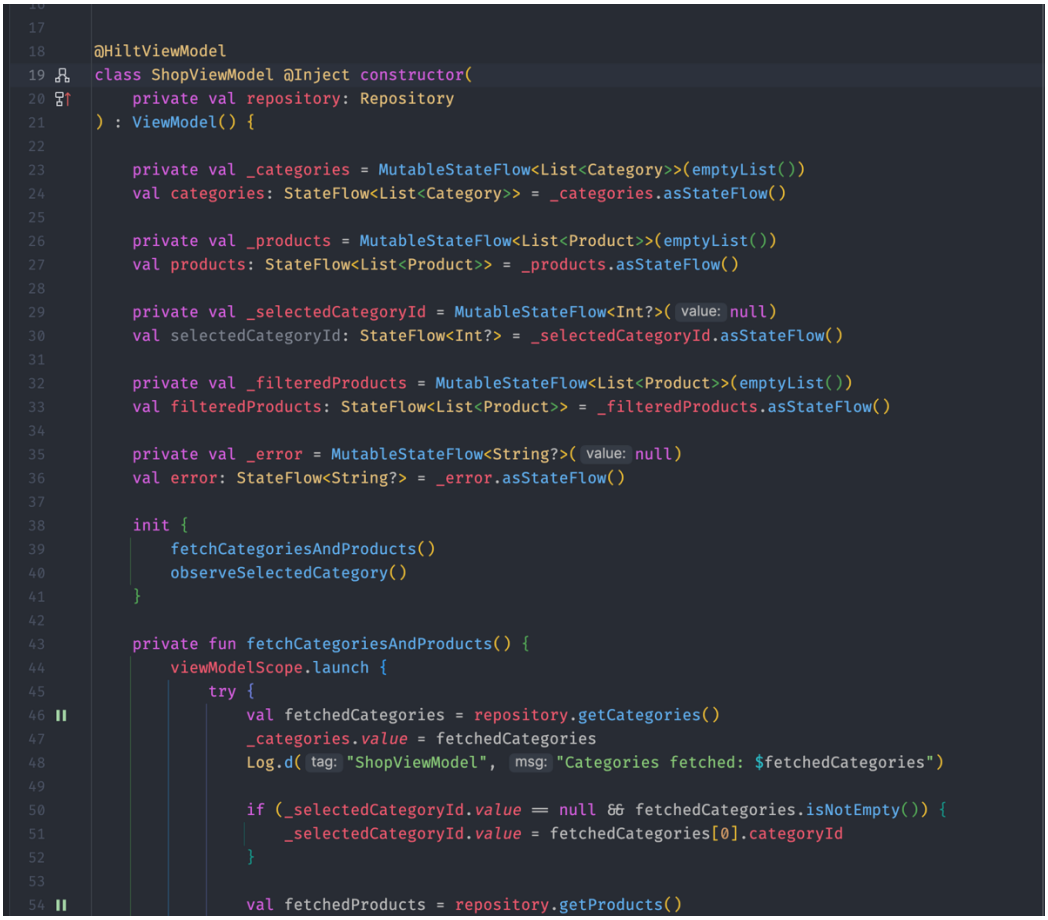


Figure 53. ShopViewModel with fetching of data

I use Jetpack Compose's LazyColumn for displaying this product data. That composable efficiently renders big lists - it reuses visible item views. For example, the screen that displays all products would observe this View Model's products state. Each item in the list is represented by the ProductItem composable that displays the name, image, and price of each product.

Error handling is one of the important parts of network communication. Retrofit allows catching the exceptions and handling them as needed. For example, fetching products catches any network issues, logs them, and falls back to local data:

```
40      suspend fun getProducts(): List<Product> {
41          return try {
42              val products = apiService.getProducts()
43              productDao.insertProducts(products)
44              products
45          } catch (e: Exception) {
46              Log.e( tag: "Repository",  msg: "Network error: ${e.message}")
47              productDao.getProducts()
48          }
49      }
```

Figure 54. getProducts function

Retrofit does lots of work and makes network communication much easier because of its intuitive API and integration. It will easily support a variety of HTTP methods, complex query parameters, and do JSON-to-Kotlin object and vice versa conversion with ease. Furthermore, Retrofit works with Room for offline-first capability in an application.

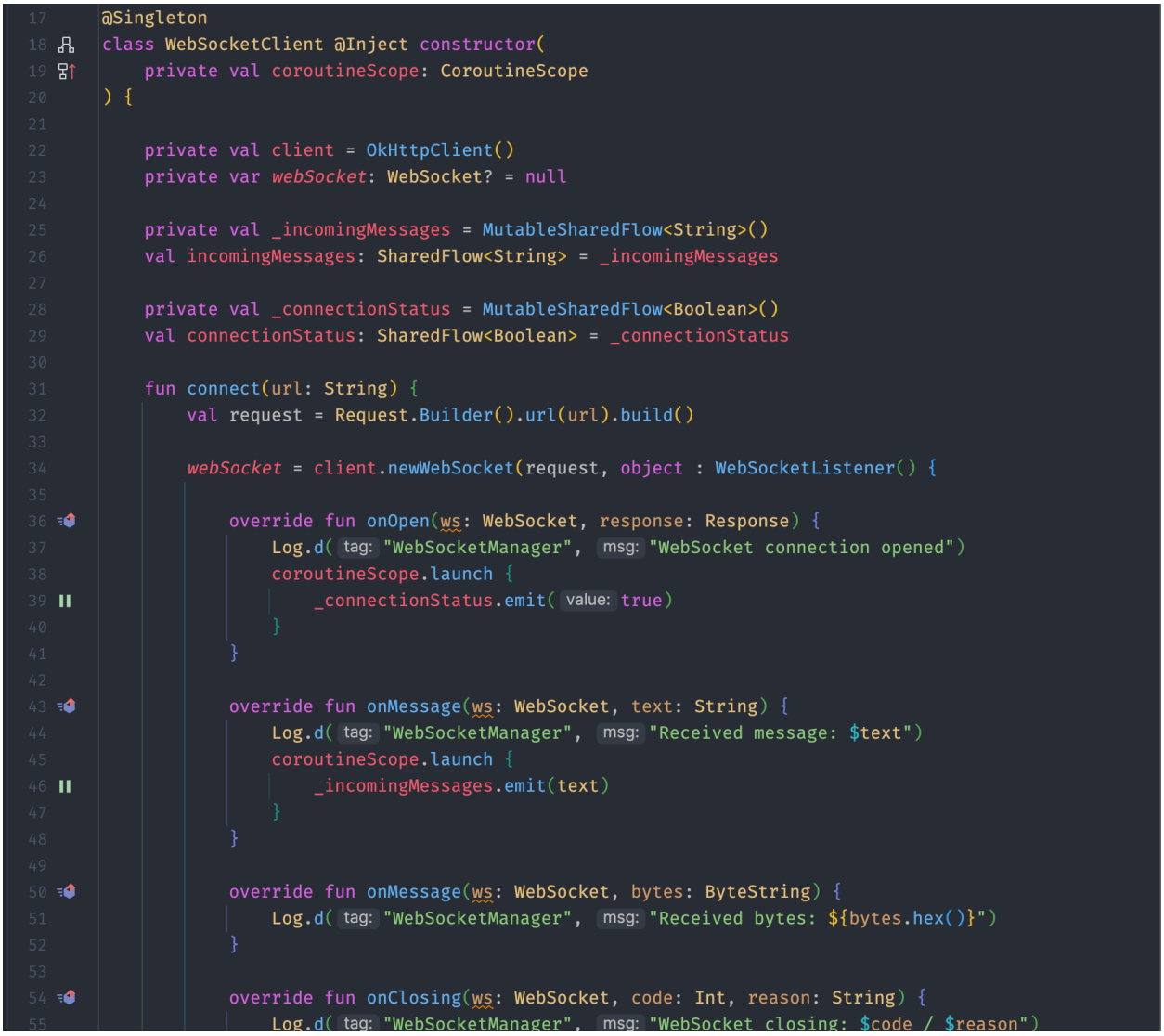
WebSockets

In implementing instant updates, the idea of WebSocket communication was used in this project. Whatever changes are occurring on the server side would be shown immediately on the client application, which makes perfect sense. Features like the notification of the user regarding discounts or updated availability depend on the underlying WebSocket connections.

```
143  connected_clients: List[WebSocket] = []
144
145  @app.websocket("/ws")
146  async def websocket_endpoint(websocket: WebSocket):
147      await websocket.accept()
148      connected_clients.append(websocket)
149      try:
150          while True:
151              data = await websocket.receive_text()
152              for client in connected_clients:
153                  await client.send_text(f"Message from server: {data}")
154      except:
155          connected_clients.remove(websocket)
```

Figure 55. Backend side for websocket

The backend implementation would involve supporting WebSockets using FastAPI. This WebSocket server would handle a lot of connections and send updates to every connected client. In this example, websocket_endpoint accepts WebSocket connections and keeps track of all the currently connected clients. Any message received from one client is broadcast to all others.



```
@Singleton
class WebSocketClient @Inject constructor(
    private val coroutineScope: CoroutineScope
) {

    private val client = OkHttpClient()
    private var webSocket: WebSocket? = null

    private val _incomingMessages = MutableSharedFlow<String>()
    val incomingMessages: SharedFlow<String> = _incomingMessages

    private val _connectionStatus = MutableSharedFlow<Boolean>()
    val connectionStatus: SharedFlow<Boolean> = _connectionStatus

    fun connect(url: String) {
        val request = Request.Builder().url(url).build()

        webSocket = client.newWebSocket(request, object : WebSocketListener() {

            override fun onOpen(ws: WebSocket, response: Response) {
                Log.d( tag: "WebSocketManager",  msg: "WebSocket connection opened")
                coroutineScope.launch {
                    _connectionStatus.emit( value: true)
                }
            }

            override fun onMessage(ws: WebSocket, text: String) {
                Log.d( tag: "WebSocketManager",  msg: "Received message: $text")
                coroutineScope.launch {
                    _incomingMessages.emit(text)
                }
            }

            override fun onMessage(ws: WebSocket, bytes: ByteString) {
                Log.d( tag: "WebSocketManager",  msg: "Received bytes: ${bytes.hex()}")
            }

            override fun onClosing(ws: WebSocket, code: Int, reason: String) {
                Log.d( tag: "WebSocketManager",  msg: "WebSocket closing: $code / $reason")
```

Figure 56. WebSocketClient

On the client side, Kotlin establishes the WebSocket connection and parses the incoming messages in its background coroutine to dynamically change the UI. WebSocketClient encapsulates WebSocket logic. It includes connect, sending a message, and disconnect functionalities. The onMessage callback enables the app to dynamically handle the incoming messages.

Here is how a WebSocket connection might be created in a Jetpack Compose screen. For instance, this could be a real time messages screen:

```
@Composable
@OptIn(ExperimentalMaterial3Api::class)
fun WebSocketScreen(
    navController: NavController, webSocketViewModel: WebSocketViewModel = hiltViewModel()
) {
    val coroutineScope = rememberCoroutineScope()
    val incomingMessages by webSocketViewModel.messages.collectAsState()
    val isConnected by webSocketViewModel.isConnected.collectAsState()

    Scaffold(topBar = {
        TopAppBar(title = { Text( text: "WebSocket Demo") })
    }, content = { padding →
        Column(
            modifier = Modifier
                .padding(padding)
                .padding(16.dp)
                .fillMaxSize(),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            if (isConnected) {
                Text(
                    text: "Connected to WebSocket Server", color = MaterialTheme.colorScheme.primary
                )
            } else {
                Text(
                    text: "Disconnected", color = MaterialTheme.colorScheme.error
                )
            }

            Spacer(modifier = Modifier.height(16.dp))

            var currentMessage by remember { mutableStateOf( value: "") }

            TextField(
                value = currentMessage,
                onValueChange = { currentMessage = it },
                label = { Text( text: "Enter message") },
                modifier = Modifier.fillMaxWidth()
            )
```

Figure 57. WebsocketScreen

In the above example, the incoming messages are added into list and displayed in LazyColumn. A button is used for sending messages to the server.

This integration of WebSocket brings great value to the project; thus, the app users can get real-time notifications. Notifications, chat features, or live inventory updates, WebSockets ensure that all happens smoothly and interactively for users. Module-based implementation also means that WebSocket logic is rather easy to extend or completely replace as an app may require.

Challenges and Solutions

Although this mobile application development for shopping had to go through several challenges, it proved to be a very good avenue to learn and implement some creative solutions. This section identifies major obstacles encountered during this project and the efforts which were made in resolving them.

One of the very first challenges was migration from traditional XML layouts to Jetpack Compose. Jetpack Compose indeed brings a more modern and declarative way of building user interfaces, but I have quite limited experience with it up frontend. Understanding how state management and lifecycle awareness work in a Compose-first application took a steep learning curve. Below is a simple composable component created part of learning about it:

```
@Composable
fun OrderItem(order: Order) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 4.dp),
        elevation = CardDefaults.cardElevation(defaultElevation = 2.dp)
    ) {
        Column(modifier = Modifier.padding(16.dp)) {
            Text(text = "Order ID: ${order.orderId}", style = MaterialTheme.typography.bodyMedium)
            Spacer(modifier = Modifier.height(4.dp))
            Text(text = "Date: ${order.orderDate}", style = MaterialTheme.typography.bodyMedium)
            Spacer(modifier = Modifier.height(4.dp))
            Text(
                text = "Total: $${"%.2f".format(order.totalAmount)}",
                style = MaterialTheme.typography.bodyMedium
            )
            Spacer(modifier = Modifier.height(4.dp))
            Text(text = "Status: ${order.status}", style = MaterialTheme.typography.bodyMedium)
        }
    }
}
```

Figure 58. Android Jetpack Compose

Another challenge was managing State in a Reactive Way. This means active state management by screens is going to ensure data coherence. I have maintained this state in the ViewModel classes and has used StateFlow for exposing reactive data to the UI. In other words, whenever anything gets updated at the backend or at the database level, UI instantly receives the change. Example of the implementation of state management in the class ShopScreen is given below:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ShopScreen(navController: NavController, viewModel: ShopViewModel) {
    val categories by viewModel.categories.collectAsState()
    val isLoading by derivedStateOf {
        viewModel.categories.value.isEmpty() && viewModel.products.value.isEmpty()
    }
    val errorMessage by viewModel.error.collectAsState()

    Scaffold(topBar = {
        TopAppBar(title = { Text( text: "Shop Categories") })
    }, bottomBar = {
        BottomNavigationBar(navController = navController)
    }, content = { padding ->
        when {
            isLoading -> {
                Box(
                    modifier = Modifier
                        .padding(padding)
                        .fillMaxSize(),
                    contentAlignment = Alignment.Center
                ) {
                    CircularProgressIndicator()
                }
            }
```

Figure 59. Android Reactive StateFlow

Having so many features and screens, it was quite a challenge to design the navigation system to be intuitive and robust at the same time. The parameters of navigation, especially when dynamic data is passed, had to be well thought out. The Navigation component of Jetpack Compose will define routes and handle deep links. Safe passing of navigation arguments has been done using typed arguments:

```
composable(
    Screen.ProductDetail.route,
    arguments = listOf(navArgument( name: "productId") { type = NavType.IntType })
) { backStackEntry →
    val productId = backStackEntry.arguments?.getInt( key: "productId") ?: 0
    ProductDetailScreen(
        navController = navController,
        productId = productId,
        shopViewModel = shopViewModel,
        cartViewModel = cartViewModel,
        reviewViewModel = reviewViewModel
    )
}
```

Figure 60. Navigation component

These problems have taken quite a great deal of time to surmount through technical research, experimentation, and iterative development. Solutions being developed solve not only the immediate needs but also create a solid foundation for future scalable and maintainable development practices. There are also lessons learned along the way that can inform enhancements and will allow the app to keep growing with its users.

Conclusion

The development of the Kotlin-based Mobile Shopping App using Jetpack Compose has been an enriching experience that results in a robust and user-friendly application. This was intended to serve all purposes put forth for the project by utilizing cutting-edge modern Android development tools and frameworks for the best shopping experience. Regarding every aspect-from UI design to back-end integration-the development has been done by keeping modularity, scalability, and maintainability in mind.

The most important outcome of the project was migration to Jetpack Compose for UI development. It had a host of positive side effects: one could design more easily; it was easier to implement; and of course, without glitches on dynamic user interactions. But Kotlin allowed going even further in perfecting that experience through its advanced features-like coroutines or null safety that avoid runtime errors.

Other major features included WebSockets for such real-time features as instantly updating users on the status of their order, generally improving user experience. Then there is Room for data storage when there is no internet; this way, the app would not malfunction in such a case.

The mentioned challenges provided a strong backbone for further improvements: mastering of Jetpack Compose, handling the state in a reactive way, keeping the backend in sync with a local database. Application of StateFlow and View Model guaranteed fluent data flow with reactive UI updates; the use of Repository abstracted the sources of data, so the app was maintainable and extensible.

The most salient takeaway from this project, however, is that best practices in software development do have relevance. Dependency injection via Hilt, separation of concerns across different layers of the app, and the use of modern architectural patterns go a long way to give the codebase cleanliness and efficiency.

While the application does meet its proposed scope, there are still avenues for improvement in the future. These include:

1. Personalization: Introduce the users to recommendations based on their preferences and purchase history.
2. Improved Analytics: In-depth analytics based on user behavior.
3. Localization: Adding multi-language to this application for a wide variety of audiences.
4. Caching Mechanisms: Improvement in data caching for fast loading of the application.

Indeed, the project was an active teacher in adapting to continuously learning new developments in software. For example, the shift to Jetpack Compose dragged developers out of their comfort zones into new paradigms. The experience in integrating different technologies, involving WebSockets, Retrofit, and Room, simply nailed into me the need for good, modular design and documentation.

The Mobile Shopping App very well shows how effective Kotlin and Jetpack Compose are in modern Android development. Knowledge and experience that were gained within this project will, without any doubt, give a very good background for other future projects related to the creation of mobile apps. With its highly scalable architecture and modular design, the app will be in the best position to evolve with users' growing needs.

References

1. **Jetpack Compose Documentation**: Official guidelines and examples from Google on building UIs with Jetpack Compose - https://developer.android.com/jetpack/compose
2. **Kotlin Language Documentation**: Comprehensive resource for Kotlin programming concepts and best practices. - https://kotlinlang.org/docs/home.html
3. **Room Persistence Library Documentation**: Guidelines for implementing local databases in Android apps. - https://developer.android.com/training/data-storage/room
4. **Retrofit Documentation**: Official documentation for the Retrofit library, used for API communication. - https://square.github.io/retrofit/
5. **Android Developer Fundamentals**: A structured overview of Android app development. - https://developer.android.com/guide
6. **JetBrains Blog on Kotlin**: Insights and tutorials on Kotlin features and use cases. - https://blog.jetbrains.com/kotlin/
7. **WebSocket API Documentation**: Resource for implementing real-time features using WebSockets. - https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
8. **FastAPI Documentation**: Official resource for building backend services with FastAPI. - https://fastapi.tiangolo.com/
9. **Material Design:** Guide for latest and modern Material 3 Design by Google for Applications. - https://m3.material.io/
10. **ViewModel Guide:** Official Documentation for the ViewModel in Applications. - https://developer.android.com/topic/libraries/architecture/viewmodel

Appendices

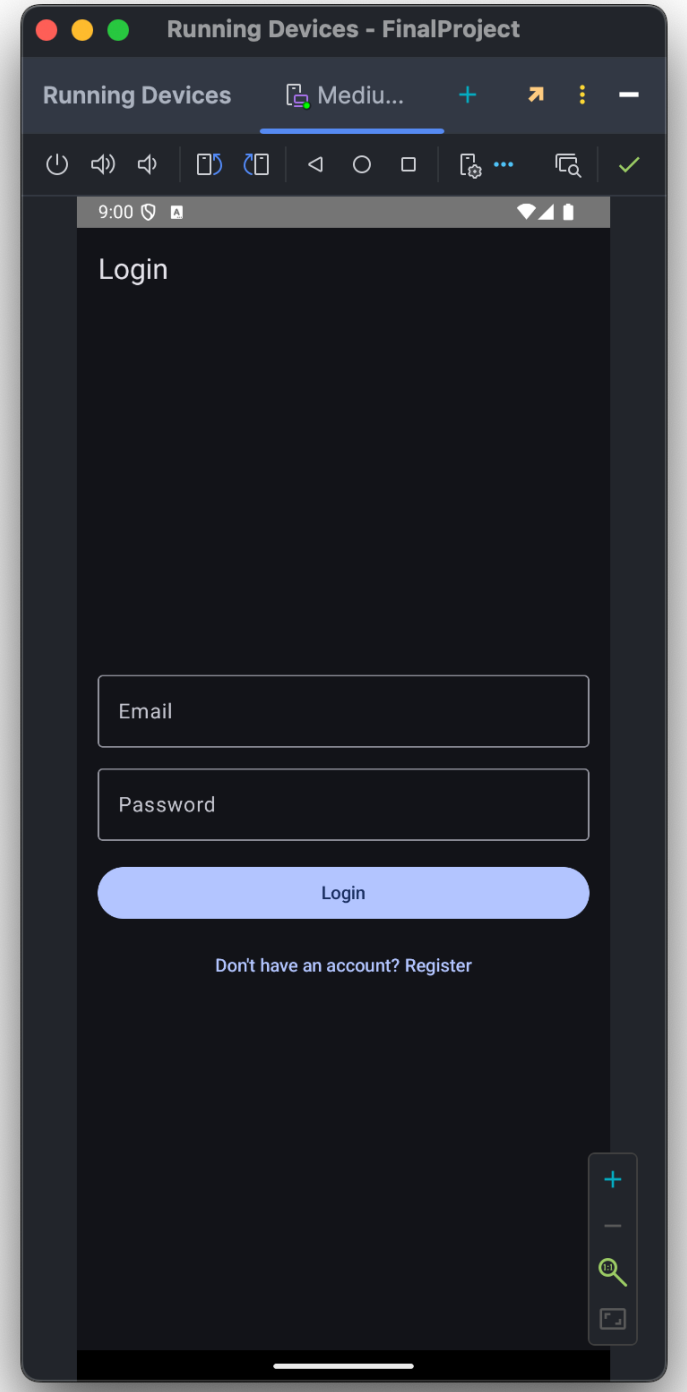


Figure 61. Login Page

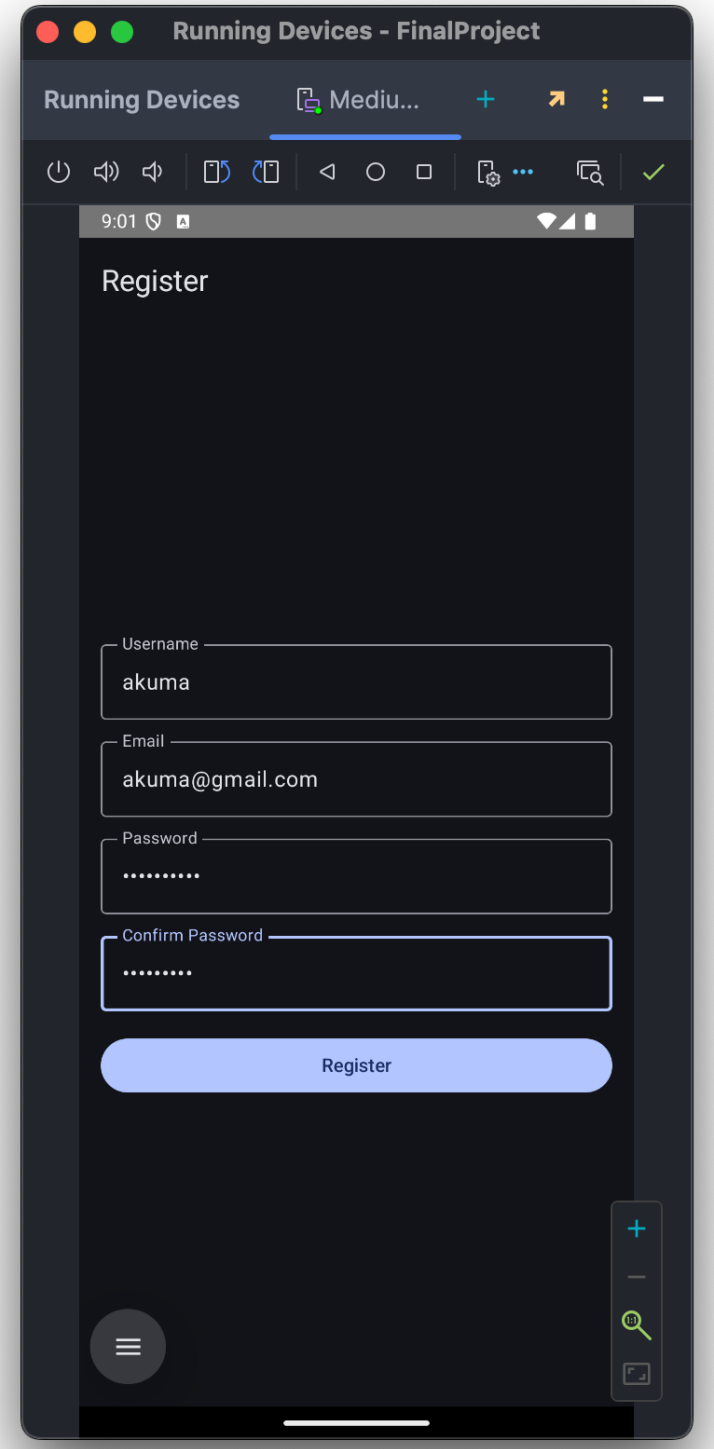


Figure 62. Register Page

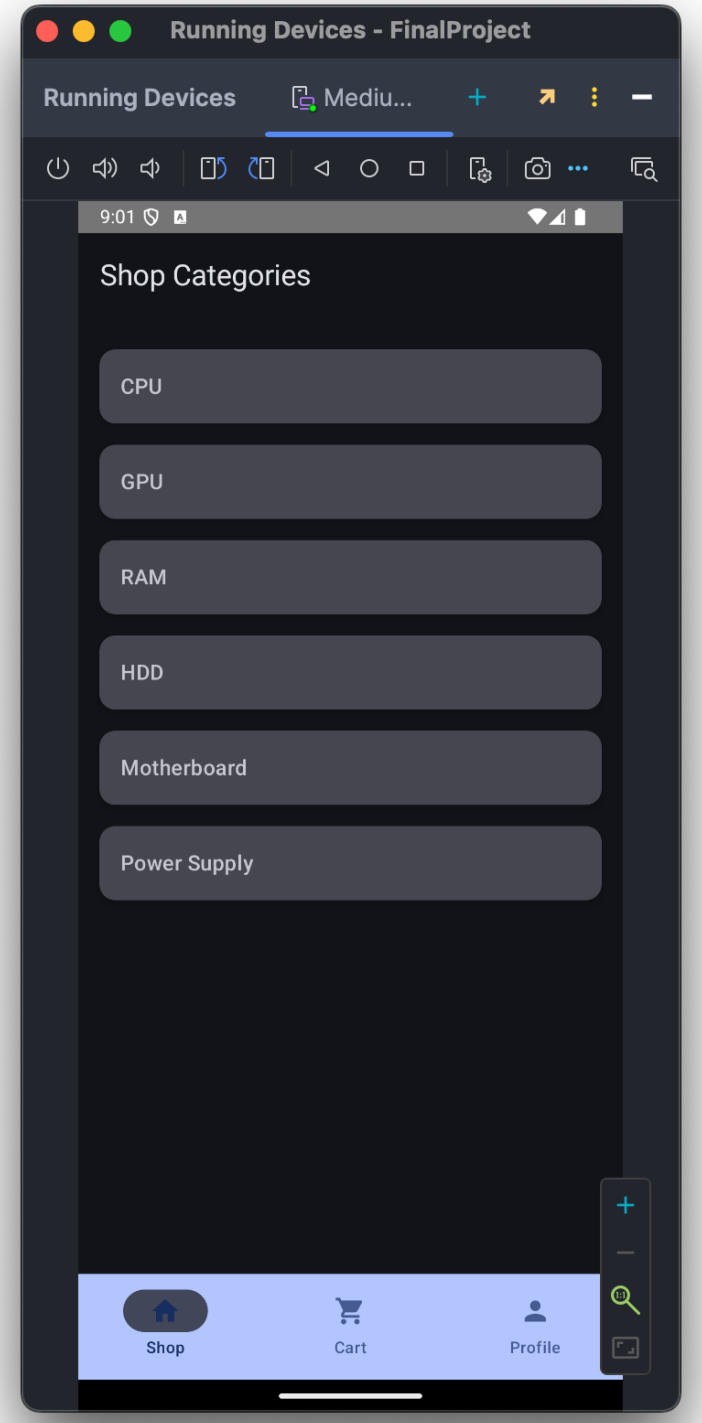


Figure 63. Home Page

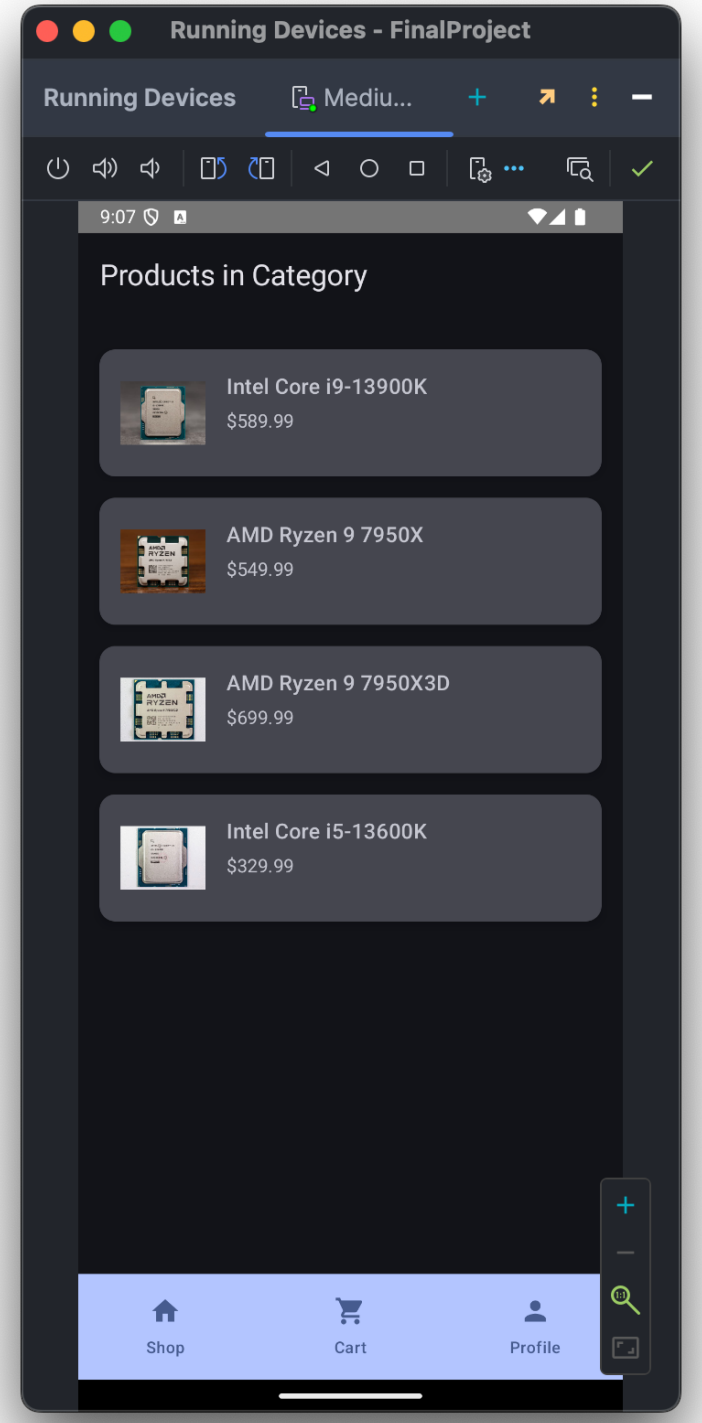


Figure 64. Filtered By Category Product Page

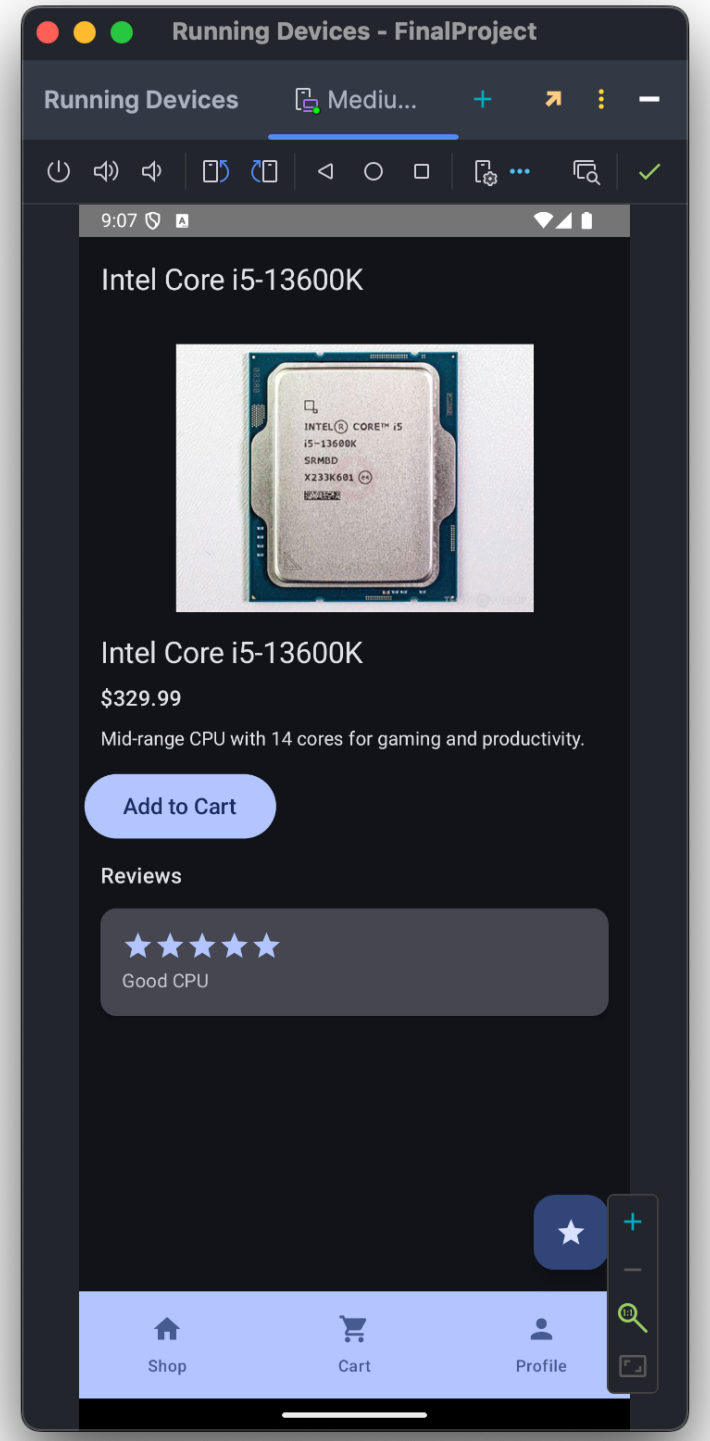


Figure 65. Product Detailed Page

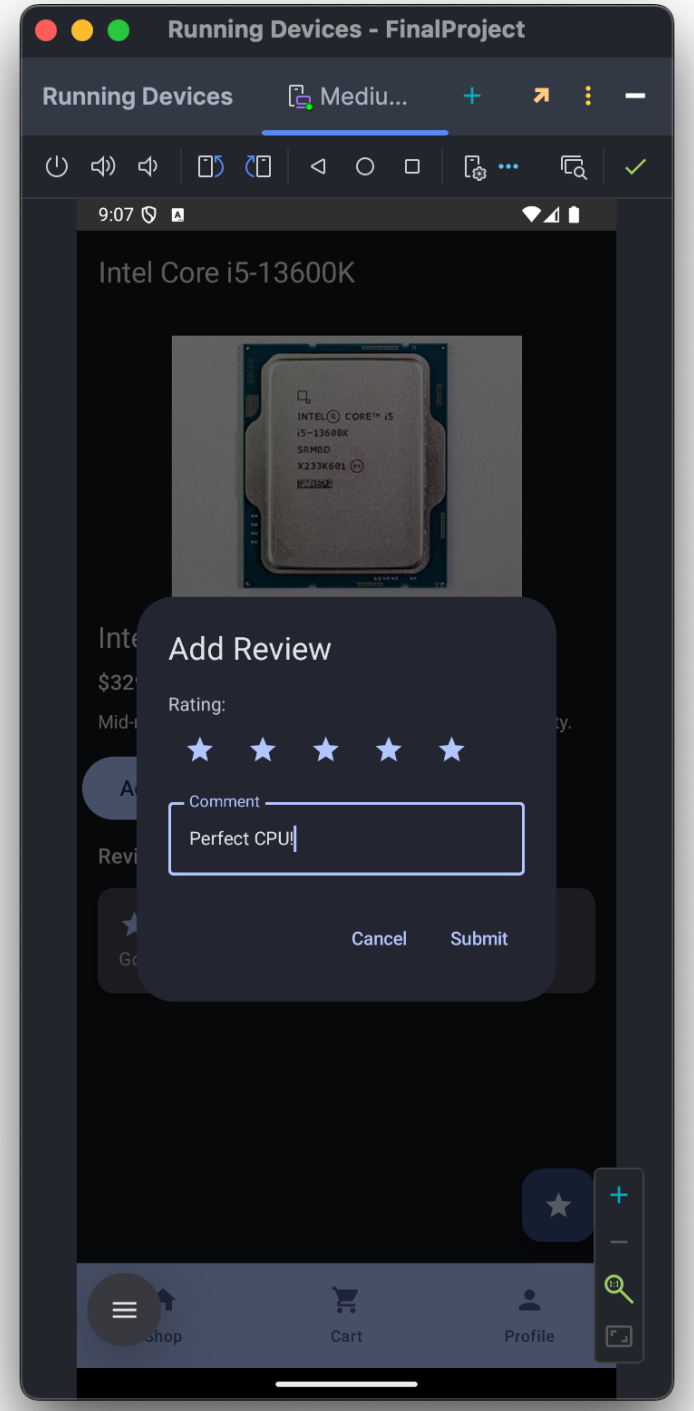


Figure 66. Add Review Form

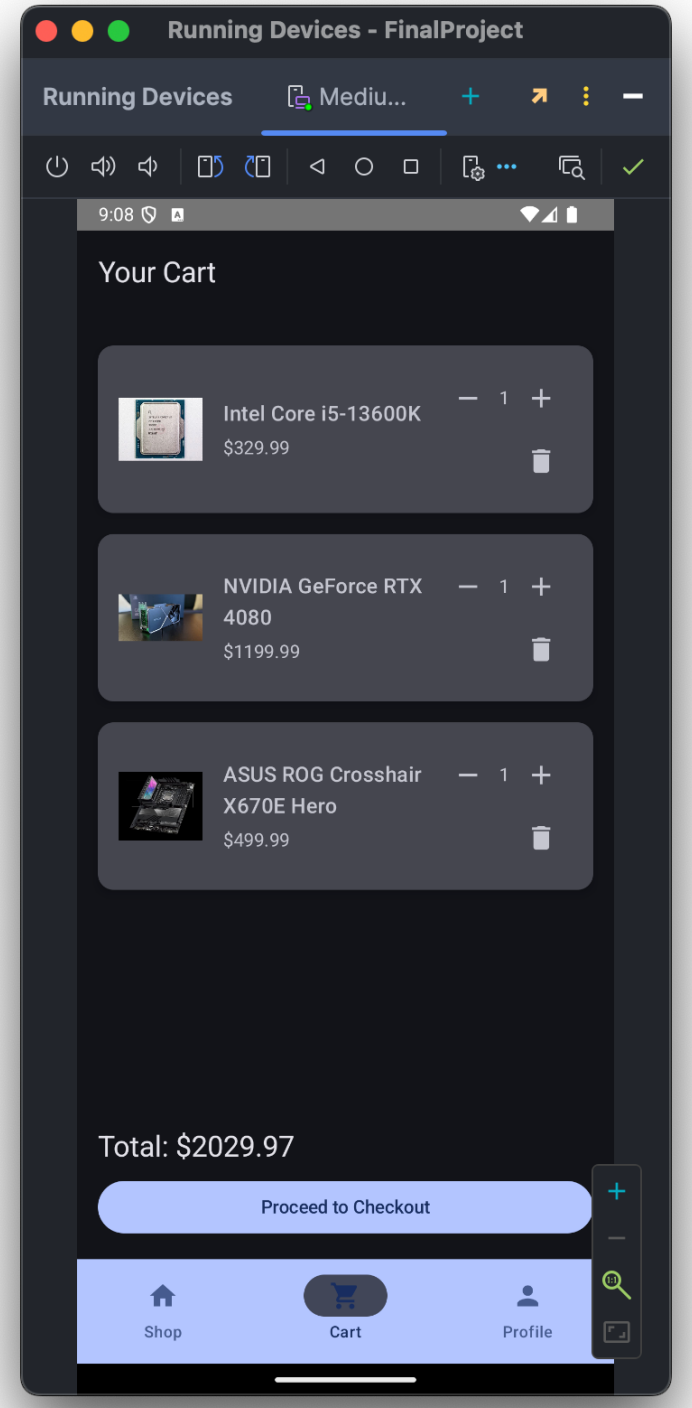


Figure 67. Cart Page with Total Sum and Quantity Modification

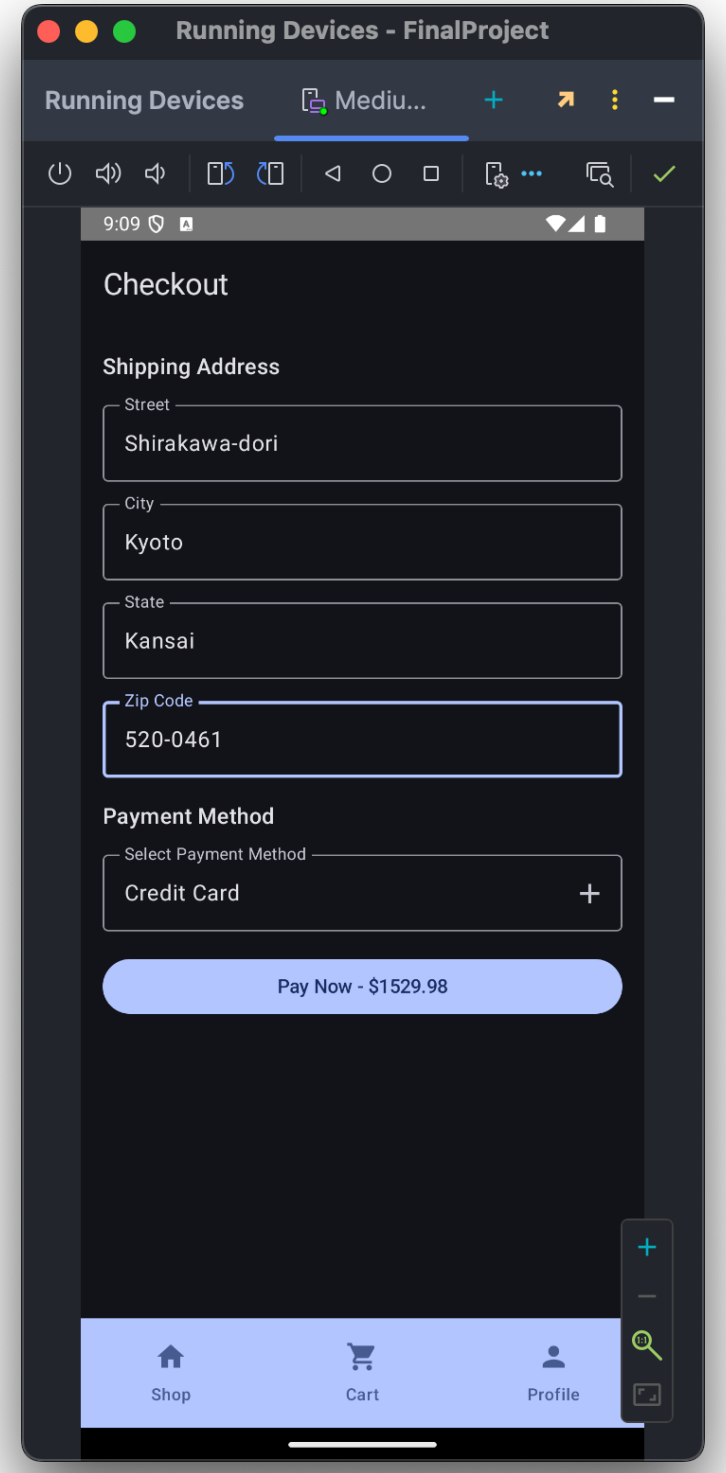


Figure 68. Checkout Page with Address Information and Payment method

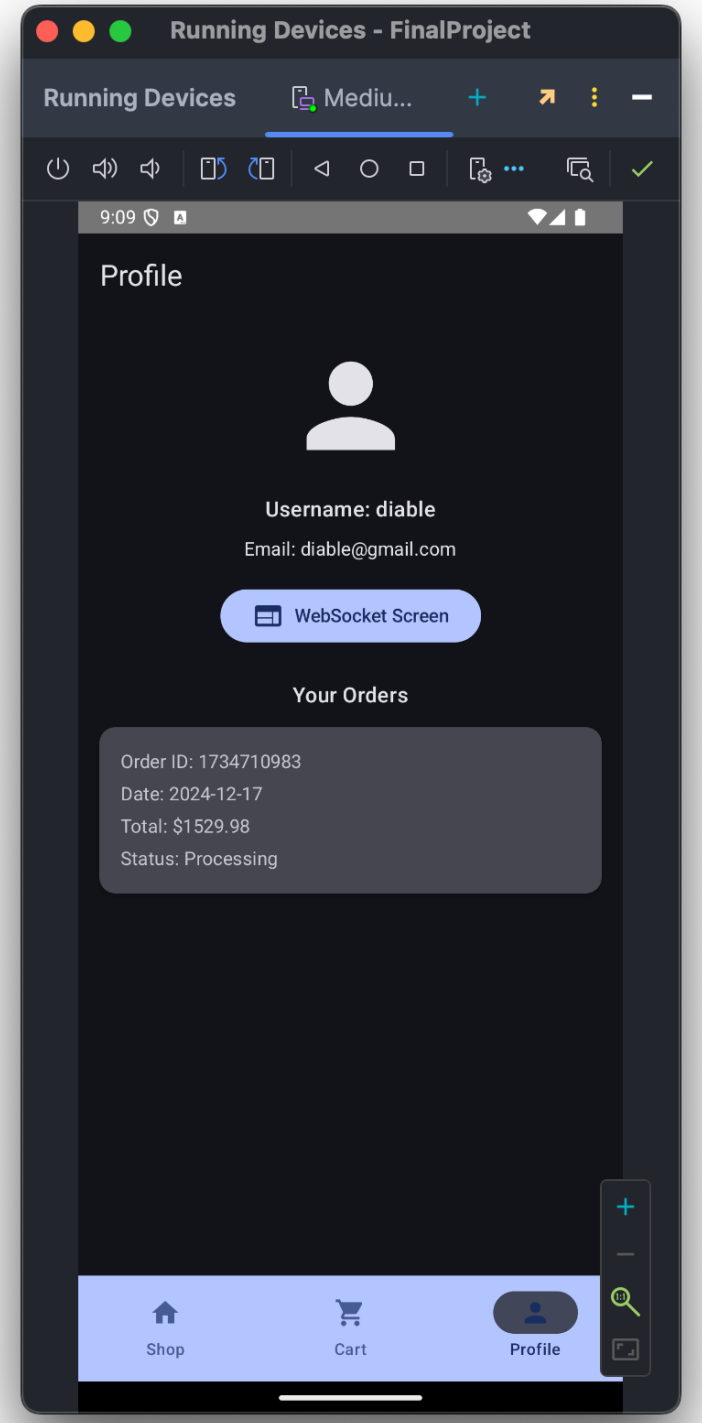


Figure 69. Profile Page with Orders

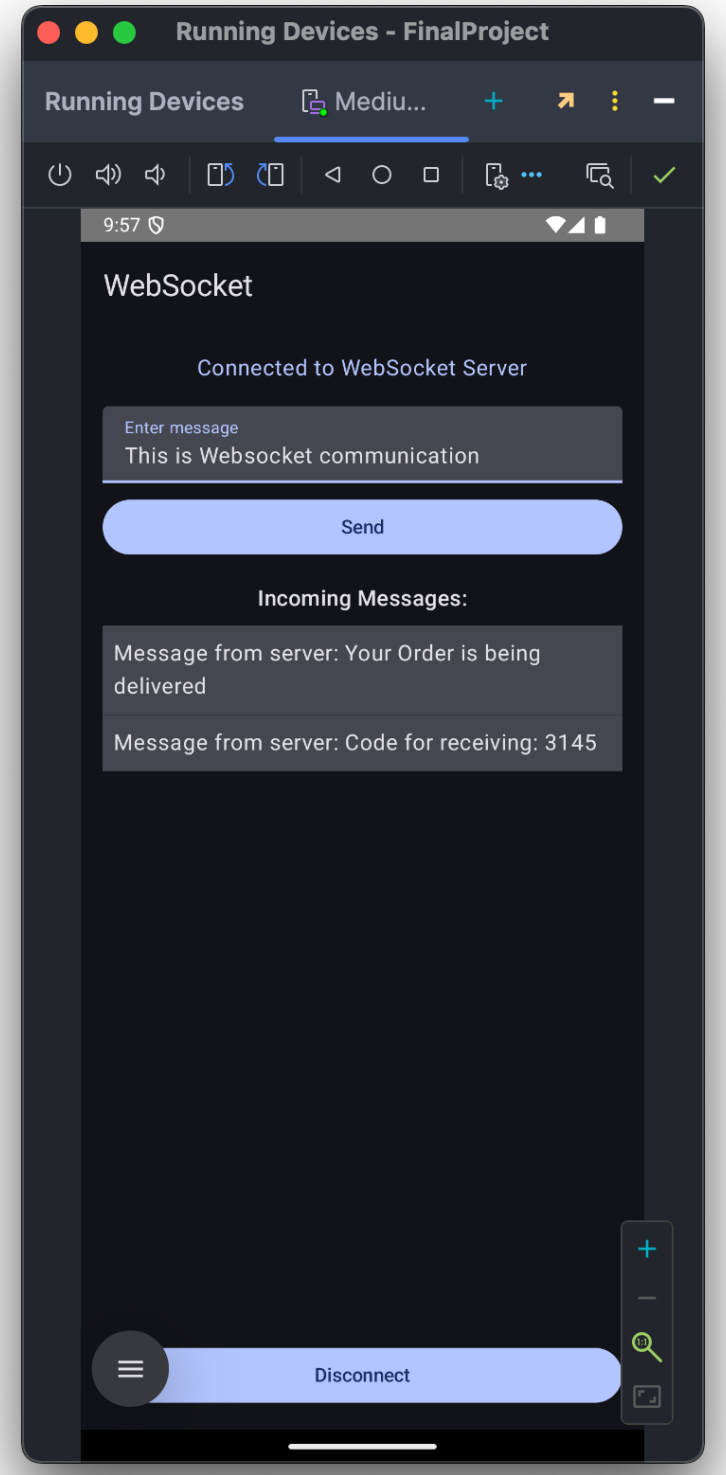


Figure 70. WebSocket Communication Page

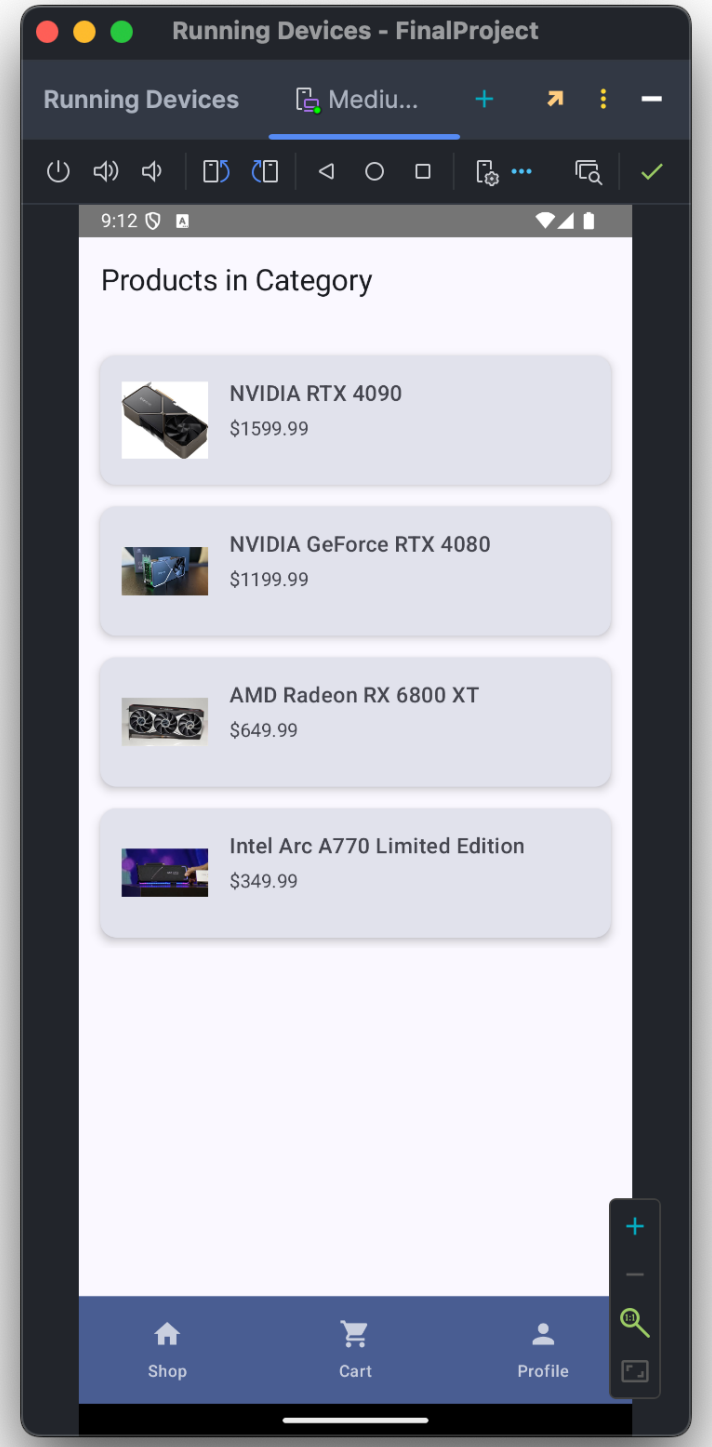


Figure 71. Light Theme of Application

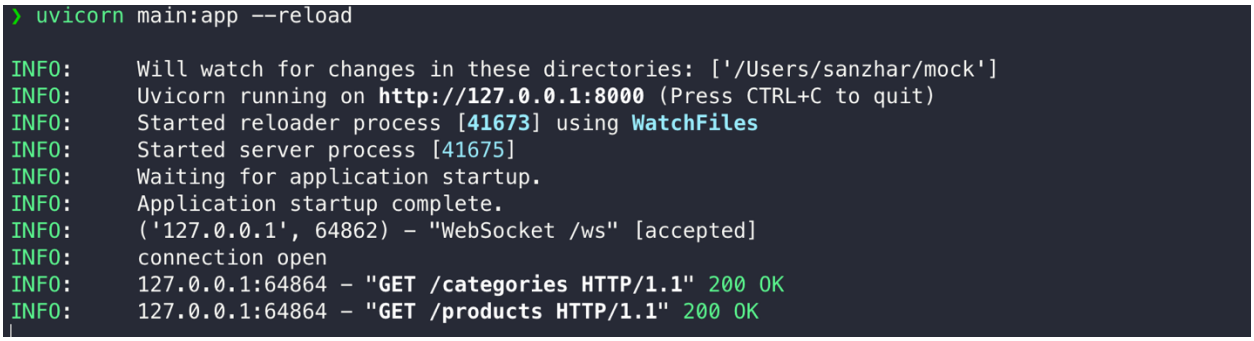


Figure 72. FastAPI Backend