KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Final Project Report

Mobile Shopping App

Serzhan Yerasyl

23MD0434

18.12.2024

Almaty,
2024

Executive Summary

The goal of this project is to create a good functional shopping app that includes product listings, product details, shopping cart functionality, and user reviews. Additionally, it includes product detail screens with user reviews and an option to add new reviews. To manage data locally, I used API service, callint it with http client. Key outcomes include a smooth user experience and clean code organization using MVVM architecture.

2.Table of Contents:

Introduction

The goal of this project is to create a good functional shopping app that includes product listings, product details, shopping cart functionality, and user reviews.The app covers product viewing, filtering, and review submission with integrating a client, that takes data from API service.

System Architecture

The architecture of this mobile shopping app is based on the Model-View-ViewModel design pattern. Model represents the app's data, including Product, Review, Categories and CartItem. Data is stored Room database. ViewModel manages UI-related logic and state using mutableStateOf. View the UI is developed using Jetpack Compose for modern, declarative interface building. The app consists of three main screens: product list, product details, and the shopping cart. Navigation between these screens is managed through state-based conditions.

Table Descriptions

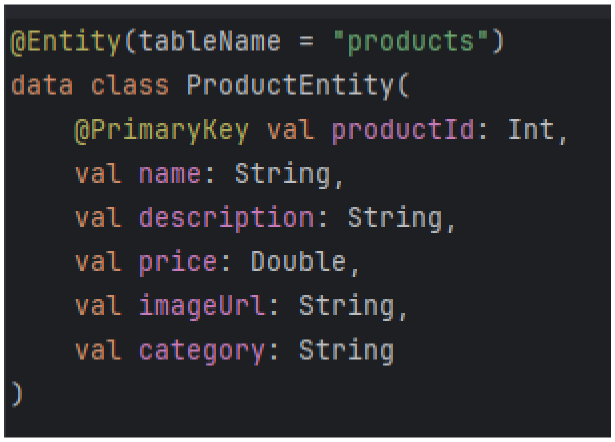
Products. Contains product details such as productId, name, description, price, category, and imageUrl.
Categories. Lists all product categories, allowing filtering on the product list screen.
CartItems: Manages items added to the shopping cart, including product ID and quantity.
Reviews. Allows users to leave reviews on specific products, including a rating and a comment.Orders: Represents orders placed by users, including total amount and order status.

Overview of Android Development: Intro to Kotlin

Android development is done using Android Studio, which provides all the necessary tools. Kotlin is the preferred programming language due to its simplicity and null-safety features.
To set up the project, we created a new Android project and added dependencies for Jetpack Compose. Compose replaces XML layouts and allows us to create UI components declaratively.
The project structure includes MainActivity, data models, and UI screens organized for maintainability. Kotlin's concise syntax made the development process faster and more readable.

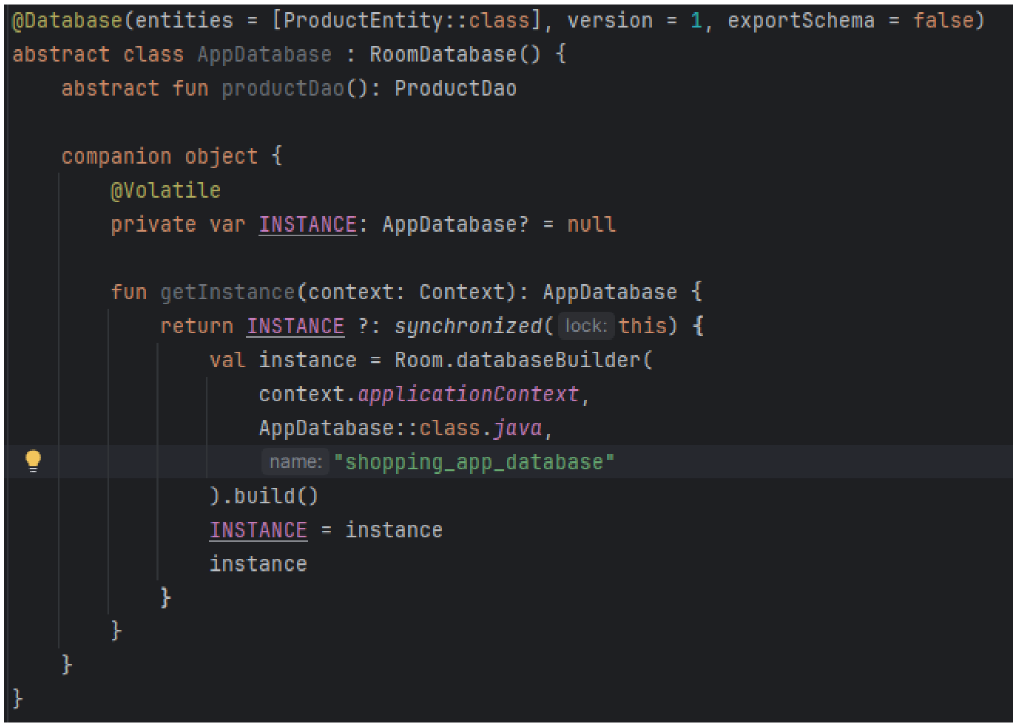
OOP in Kotlin

Object-oriented programming (OOP) principles like classes, data classes, and encapsulation are used in this project. The Product class is defined as a data class:



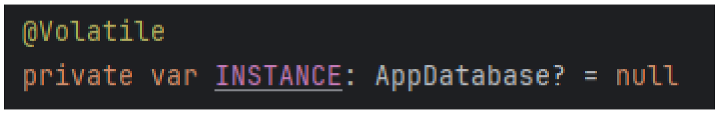
```kotlin
@Entity(tableName = "products")
data class ProductEntity(
    @PrimaryKey val productId: Int,
    val name: String,
    val description: String,
    val price: Double,
    val imageUrl: String,
    val category: String
)
```

This class like struct which will unmarshalling json's from external api.



```kotlin
@Database(entities = [ProductEntity::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun productDao(): ProductDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getInstance(context: Context): AppDatabase {
            return INSTANCE ?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                     name: "shopping_app_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

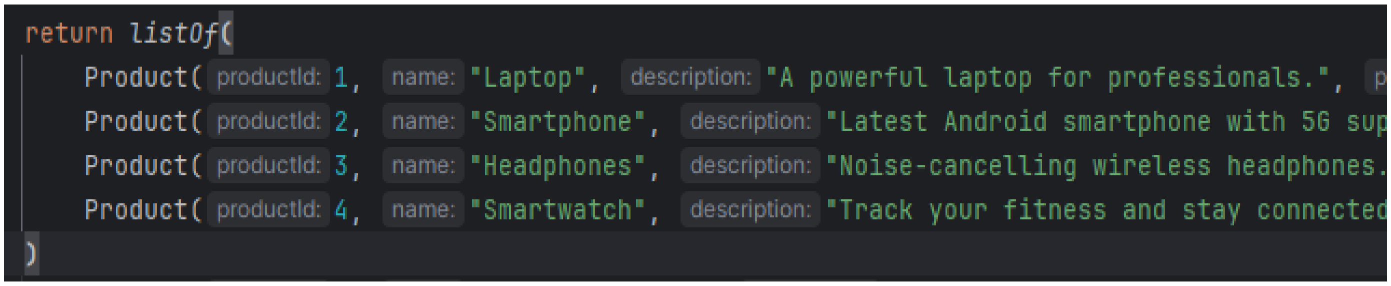
This AppDatabase class demonstrates several object-oriented principles (OOP) such as abstraction, encapsulation, and singleton. Let's explain in simple terms for the OOP section.
Encapsulation means hiding data and protecting it from outside interference.



```kotlin
@Volatile
private var INSTANCE: AppDatabase? = null
```

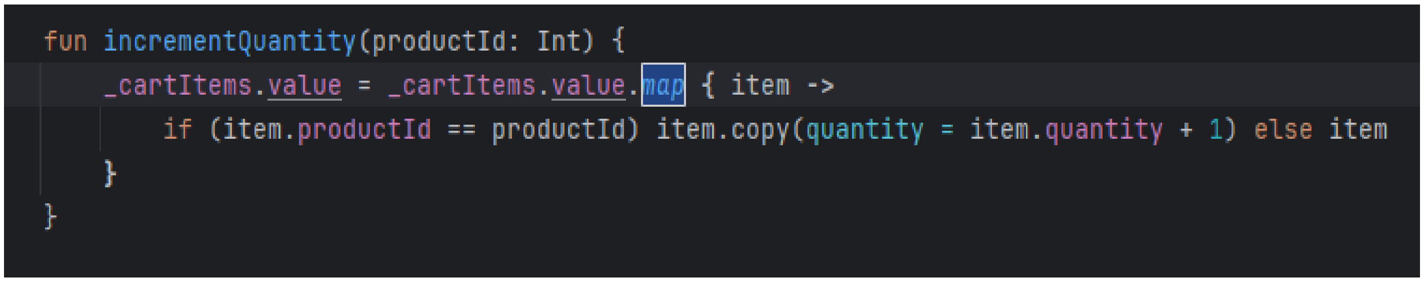
Working with Collections in Kotlin

Collections like List, Map, and Set are widely used in the app. For example, the product list is stored in a List:



```
return listOf(
    Product( productId: 1,  name: "Laptop",  description: "A powerful laptop for professionals.",  p
    Product( productId: 2,  name: "Smartphone",  description: "Latest Android smartphone with 5G sup
    Product( productId: 3,  name: "Headphones",  description: "Noise-cancelling wireless headphones.
    Product( productId: 4,  name: "Smartwatch",  description: "Track your fitness and stay connected
)
```

I used functions like filter for searching and map to transform data. This made managing data for products, reviews, and the cart simple and efficient.

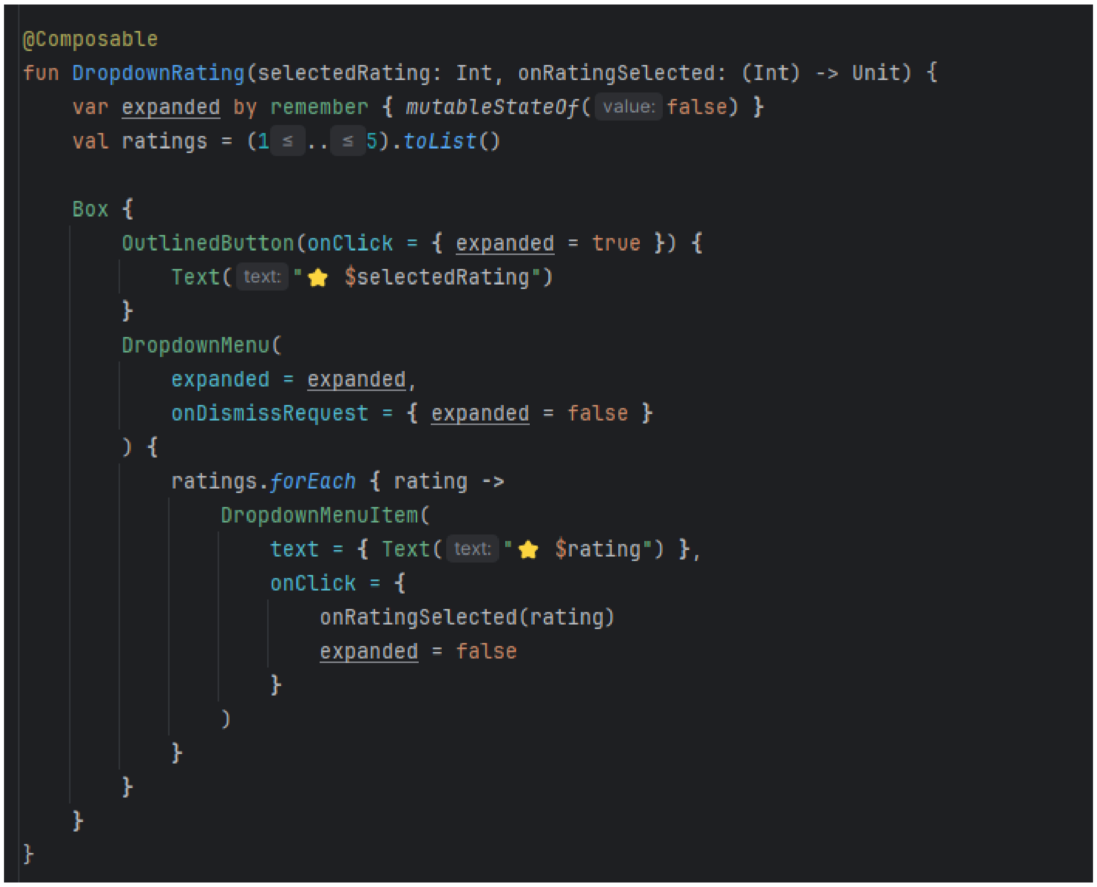


```
fun incrementQuantity(productId: Int) {
    _cartItems.value = _cartItems.value.map { item ->
        if (item.productId == productId) item.copy(quantity = item.quantity + 1) else item
    }
}
```

For example in adding to cartItem, for count amount of good I user map built-in function returns a list containing the results of applying the given transform function to each element in the original collection.

Android Layout

Jetpack Compose was used for the layout instead of traditional XML. Composable functions like ProductListScreen and ProductDetailScreen are defined for UI components.Compose simplifies UI creation with modifiers for styling, alignment, and spacing. The LazyColumn was used for displaying dynamic lists of products and reviews.

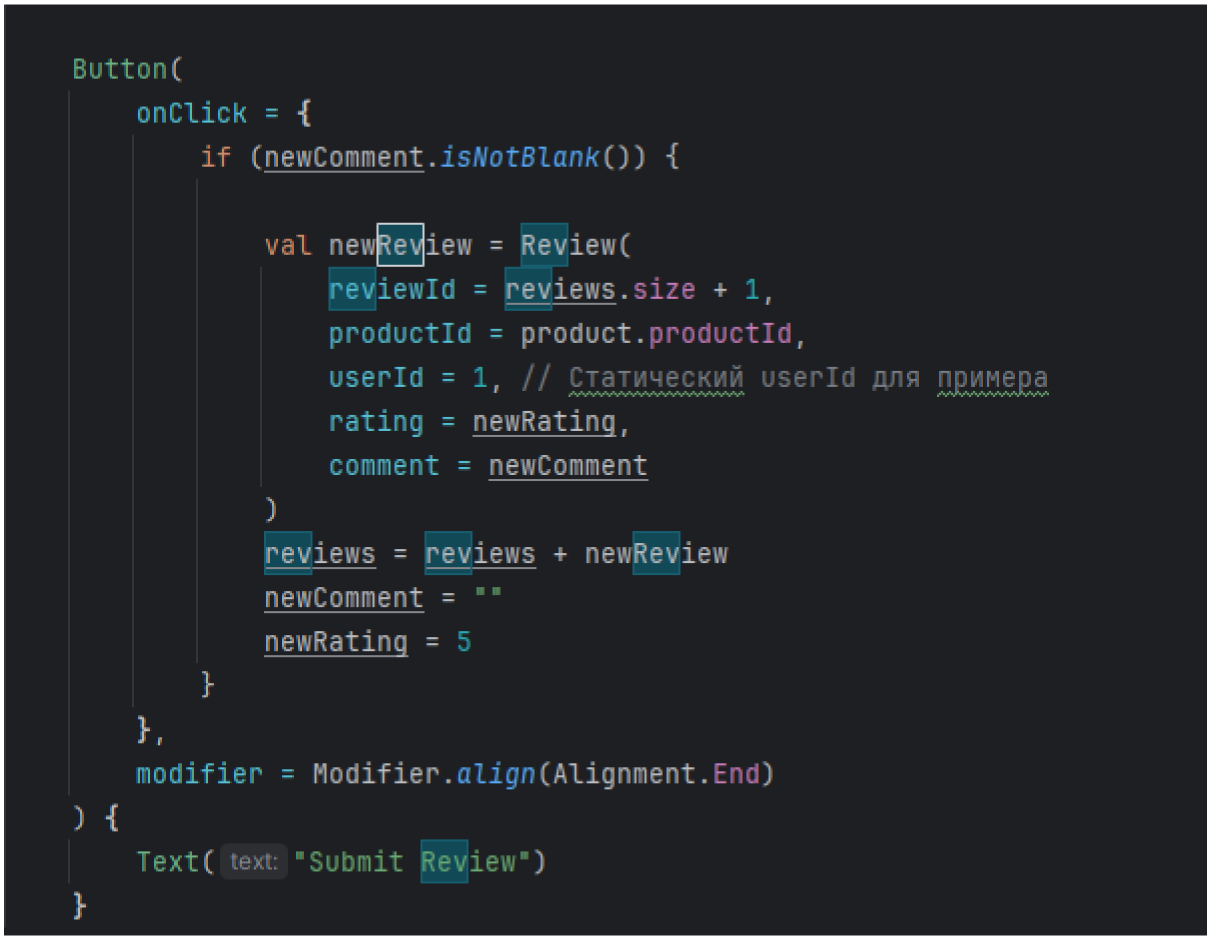


```kotlin
@Composable
fun DropdownRating(selectedRating: Int, onRatingSelected: (Int) -> Unit) {
    var expanded by remember { mutableStateOf( value: false) }
    val ratings = (1 ≤ .. ≤ 5).toList()

    Box {
        OutlinedButton(onClick = { expanded = true }) {
            Text( text: "⭐ $selectedRating")
        }
        DropdownMenu(
            expanded = expanded,
            onDismissRequest = { expanded = false }
        ) {
            ratings.forEach { rating ->
                DropdownMenuItem(
                    text = { Text( text: "⭐ $rating") },
                    onClick = {
                        onRatingSelected(rating)
                        expanded = false
                    }
                )
            }
        }
    }
}
```

This function DropdownRating is a composable function in Jetpack Compose that creates a dropdown list for selecting a rating. It uses modern Compose tools to create the interface. I will explain in simple terms for the Android Layout section.

Activity: Handling User Input and Events

Activities are responsible for handling user input, like button clicks and text input. For example, clicking "Add to Cart" updates the cart state.State management in Compose is done using remember and mutableStateOf to keep the UI updated. User input for reviews was captured through text fields.

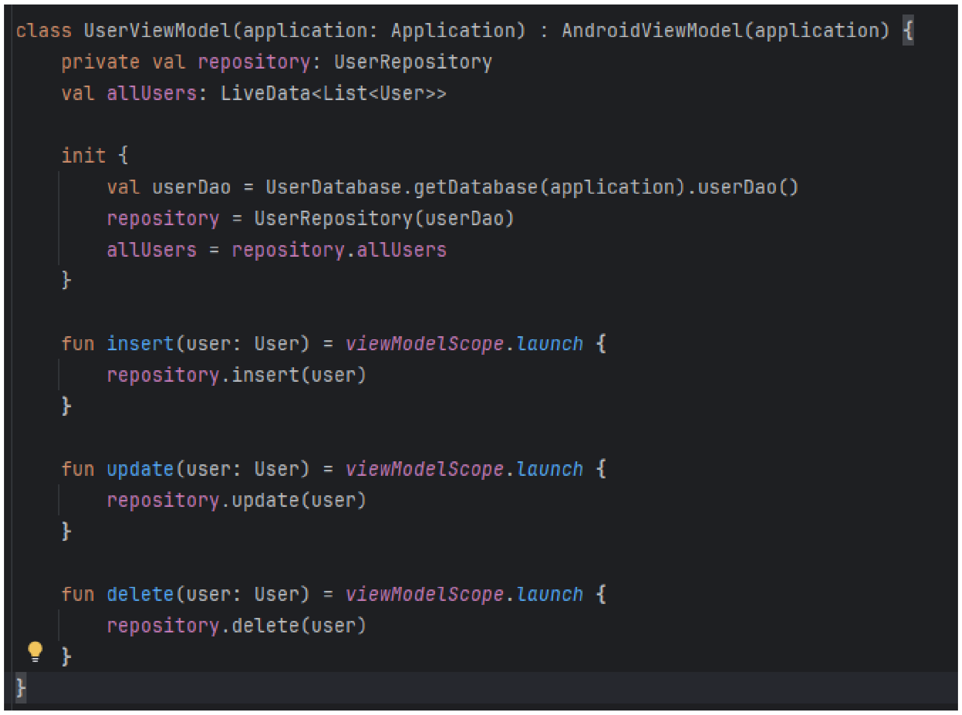


```
Button(
    onClick = {
        if (newComment.isNotBlank()) {

            val newReview = Review(
                reviewId = reviews.size + 1,
                productId = product.productId,
                userId = 1, // Статический userId для примера
                rating = newRating,
                comment = newComment
            )
            reviews = reviews + newReview
            newComment = ""
            newRating = 5
        }
    },
    modifier = Modifier.align(Alignment.End)
) {
    Text( text: "Submit Review")
}
```

This code snippet demonstrates handling user input and events in an Android app using Jetpack Compose. onClick this is the event that is executed when the button is clicked. The logic for adding a review is described here. A new Review object is created with a unique reviewId (based on the current size of the review list). A new Review object is created with a unique reviewId (based on the current size of the review list). The review stores the productId, a static userId, the entered comment (newComment), and the selected rating (newRating).

User Interface Integration

To display data in the UI, a UserViewModel is used, which efficiently handles large datasets. A LiveData object is tied to the database query, allowing the UI to automatically update when the data changes. The ViewModel holds the LiveData, ensuring that UI components are lifecycle-aware and do not access the database directly.

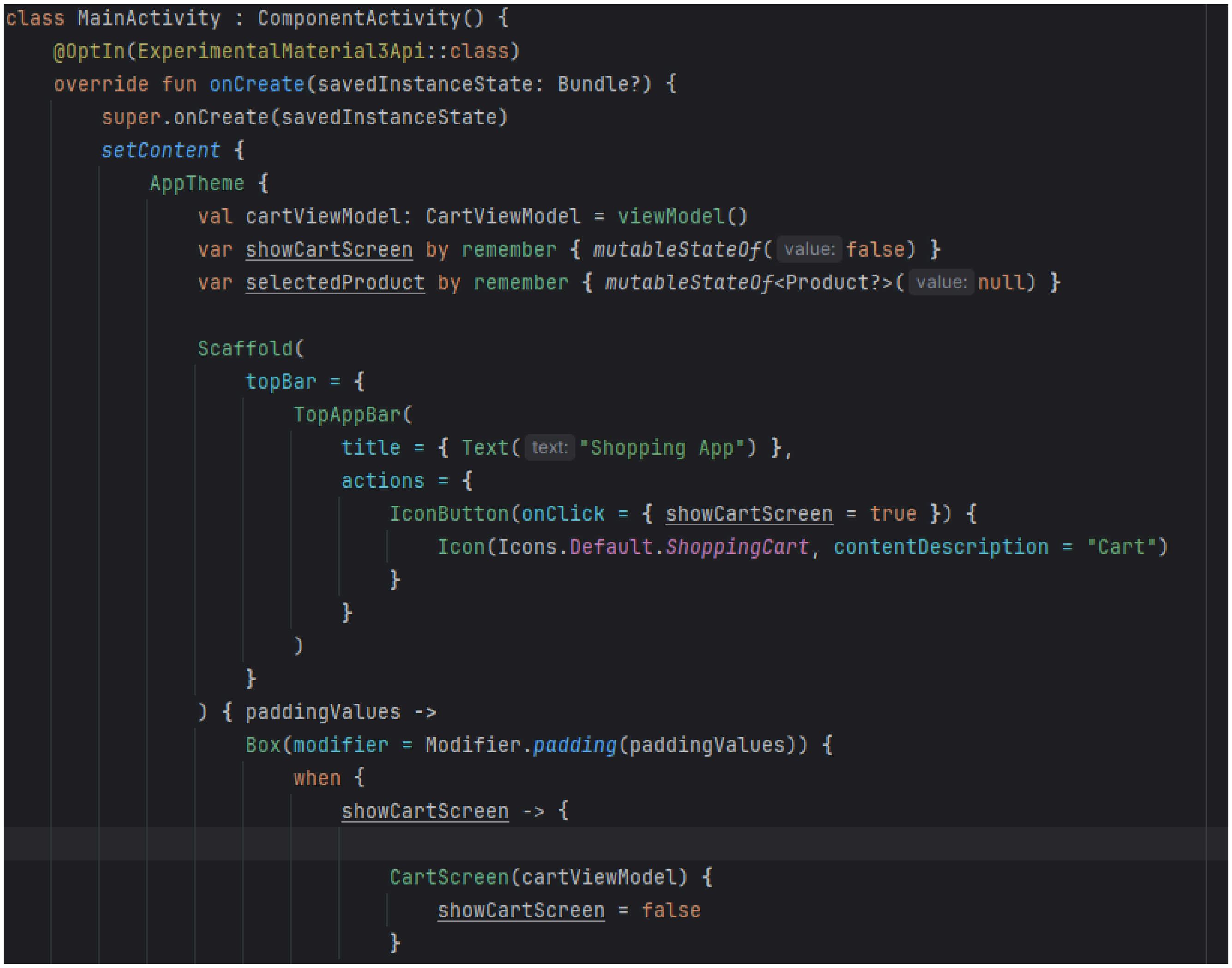


```
class UserViewModel(application: Application) : AndroidViewModel(application) {
    private val repository: UserRepository
    val allUsers: LiveData<List<User>>

    init {
        val userDao = UserDatabase.getDatabase(application).userDao()
        repository = UserRepository(userDao)
        allUsers = repository.allUsers
    }

    fun insert(user: User) = viewModelScope.launch {
        repository.insert(user)
    }

    fun update(user: User) = viewModelScope.launch {
        repository.update(user)
    }

    fun delete(user: User) = viewModelScope.launch {
        repository.delete(user)
    }
}
```

This UserViewModel class manages user data and acts as a link between the UI and the repository. It initializes the repository, which accesses data through the UserDao obtained from the database. The allUsers variable stores LiveData, which tracks all changes to the user list through the repository. The insert, update, and delete methods perform data operations asynchronously using coroutines, which allows updating the database without blocking the main thread. This approach helps keep data current and improves the performance of the application.

Activity Lifecycle

In Jetpack Compose, the activity lifecycle is still managed in the traditional way, but Compose simplifies UI state management and automatically restores data across repaints.

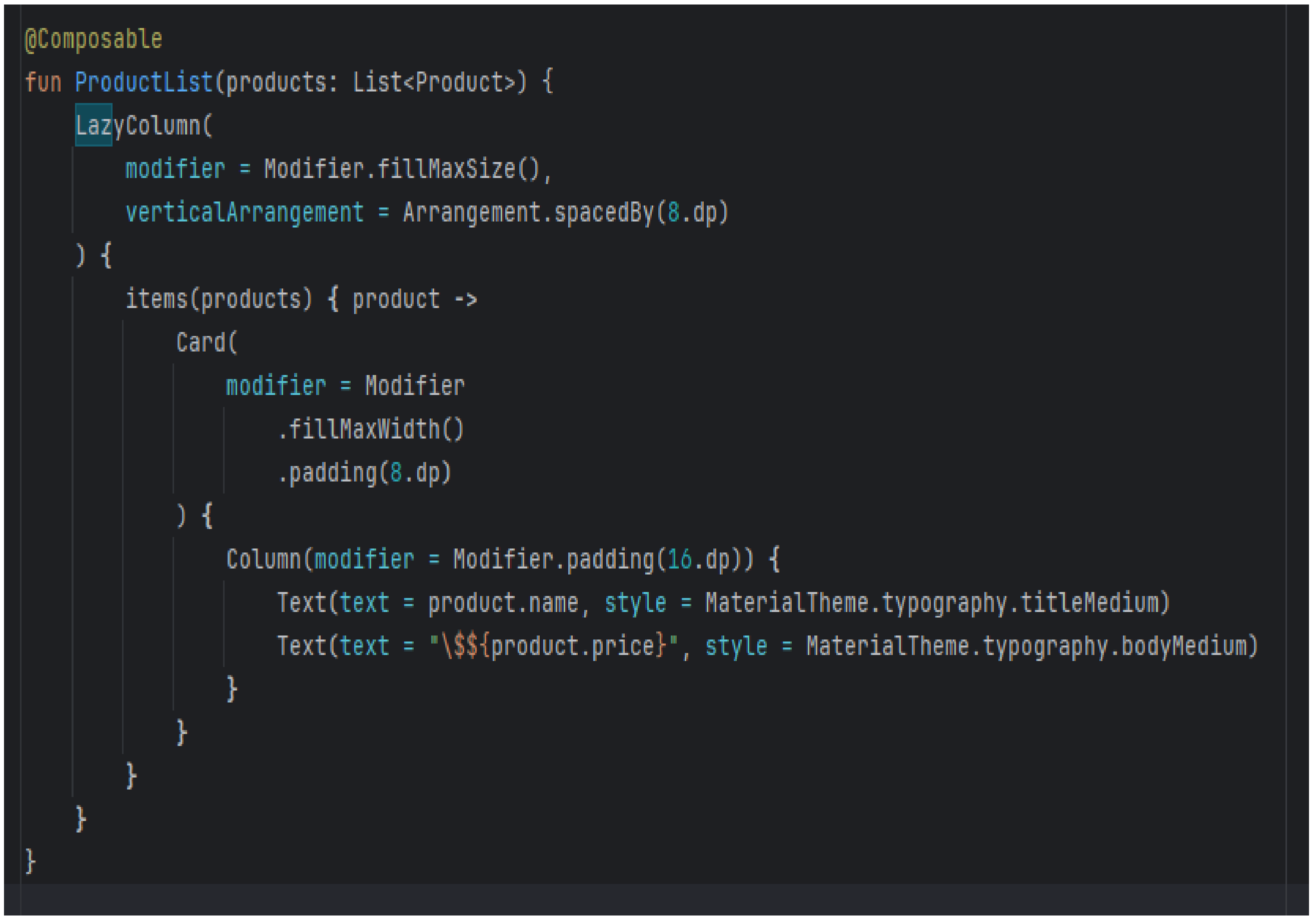


```kotlin
class MainActivity : ComponentActivity() {
    @OptIn(ExperimentalMaterial3Api::class)
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                val cartViewModel: CartViewModel = viewModel()
                var showCartScreen by remember { mutableStateOf( value: false) }
                var selectedProduct by remember { mutableStateOf<Product?>( value: null) }

                Scaffold(
                    topBar = {
                        TopAppBar(
                            title = { Text( text: "Shopping App") },
                            actions = {
                                IconButton(onClick = { showCartScreen = true }) {
                                    Icon(Icons.Default.ShoppingCart, contentDescription = "Cart")
                                }
                            }
                        )
                    }
                ) { paddingValues ->
                    Box(modifier = Modifier.padding(paddingValues)) {
                        when {
                            showCartScreen -> {

                                CartScreen(cartViewModel) {
                                    showCartScreen = false
                                }
```

In the onCreate method, setContent is called, which loads the Compose UI.
The state of counter is saved via remember, so even if the screen is redrawn, the value is not reset.
This replaces the need to use lifecycle methods to manually save the state.

RecyclerView and Adapters

Jetpack Compose uses LazyColumn instead of the traditional RecyclerView, which makes it much easier to display lists.

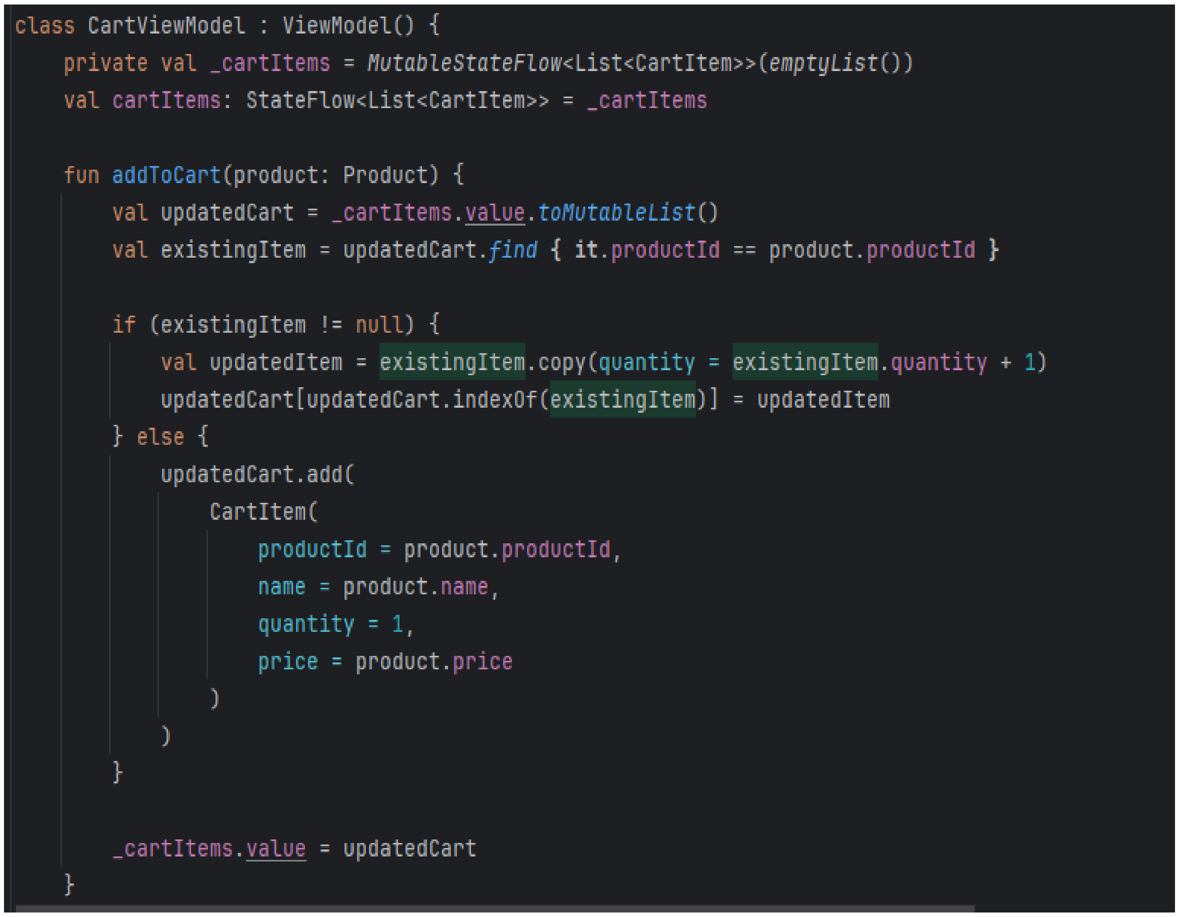


```kotlin
@Composable
fun ProductList(products: List<Product>) {
    LazyColumn(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(products) { product ->
            Card(
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(8.dp)
            ) {
                Column(modifier = Modifier.padding(16.dp)) {
                    Text(text = product.name, style = MaterialTheme.typography.titleMedium)
                    Text(text = "\$${product.price}", style = MaterialTheme.typography.bodyMedium)
                }
            }
        }
    }
}
```

LazyColumn displays list items as they appear on the screen.
Instead of adapters, each item is defined using a Composable function inside items.
Items are automatically optimized for performance.

ViewModel and LiveData

In Compose, the ViewModel manages UI data and persists it across Activity re-creations. Compose uses State and MutableState instead of LiveData to store and observe data.

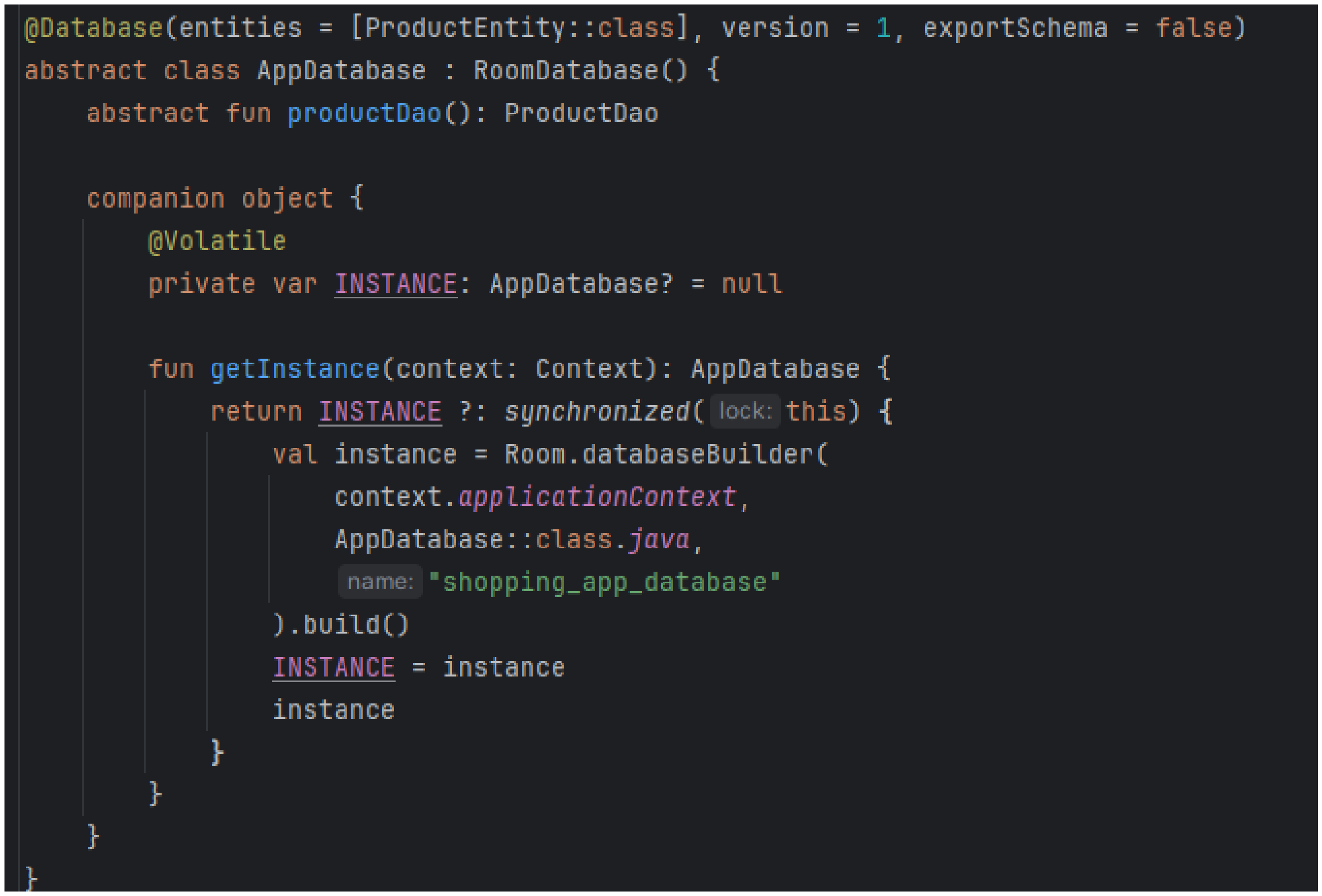


```
class CartViewModel : ViewModel() {
    private val _cartItems = MutableStateFlow<List<CartItem>>(emptyList())
    val cartItems: StateFlow<List<CartItem>> = _cartItems

    fun addToCart(product: Product) {
        val updatedCart = _cartItems.value.toMutableList()
        val existingItem = updatedCart.find { it.productId == product.productId }

        if (existingItem != null) {
            val updatedItem = existingItem.copy(quantity = existingItem.quantity + 1)
            updatedCart[updatedCart.indexOf(existingItem)] = updatedItem
        } else {
            updatedCart.add(
                CartItem(
                    productId = product.productId,
                    name = product.name,
                    quantity = 1,
                    price = product.price
                )
            )
        }

        _cartItems.value = updatedCart
    }
```

ViewModel stores data (list of products) and manages their state.
Instead of LiveData, MutableState is used, which automatically updates the UI in Compose.
viewModel() creates an instance of ViewModel, which saves data between Activity re-creations.

Working with Databases

For local data storage, this project uses Room Database, which provides a convenient abstraction layer on top of SQLite. Room makes it easy to create, use, and update a database in an Android app.

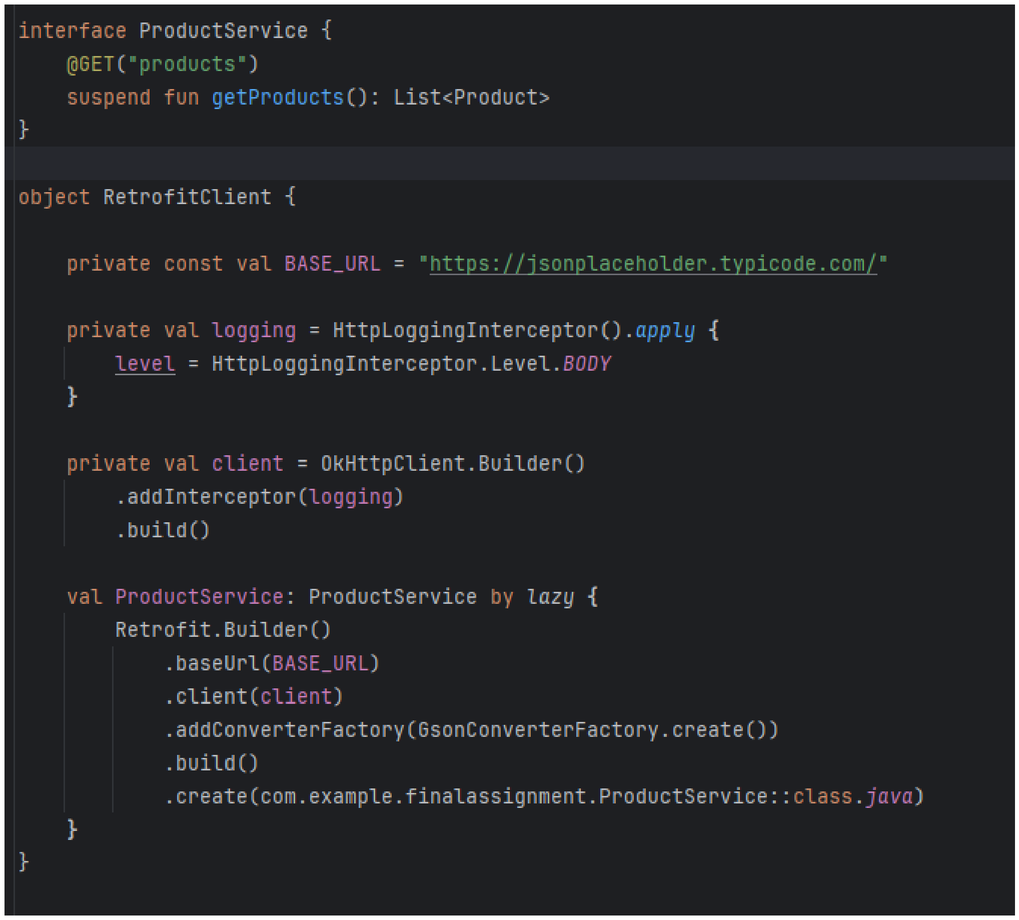


```
@Database(entities = [ProductEntity::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun productDao(): ProductDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getInstance(context: Context): AppDatabase {
            return INSTANCE ?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    name: "shopping_app_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

Room Database is a class that connects application data to tables in a database. In this project, I created a database to store a list of products. To use the database in code, we get an instance of the database and call the DAO methods. Using methods from ProductDao, we can insert, read and filter data in the database.

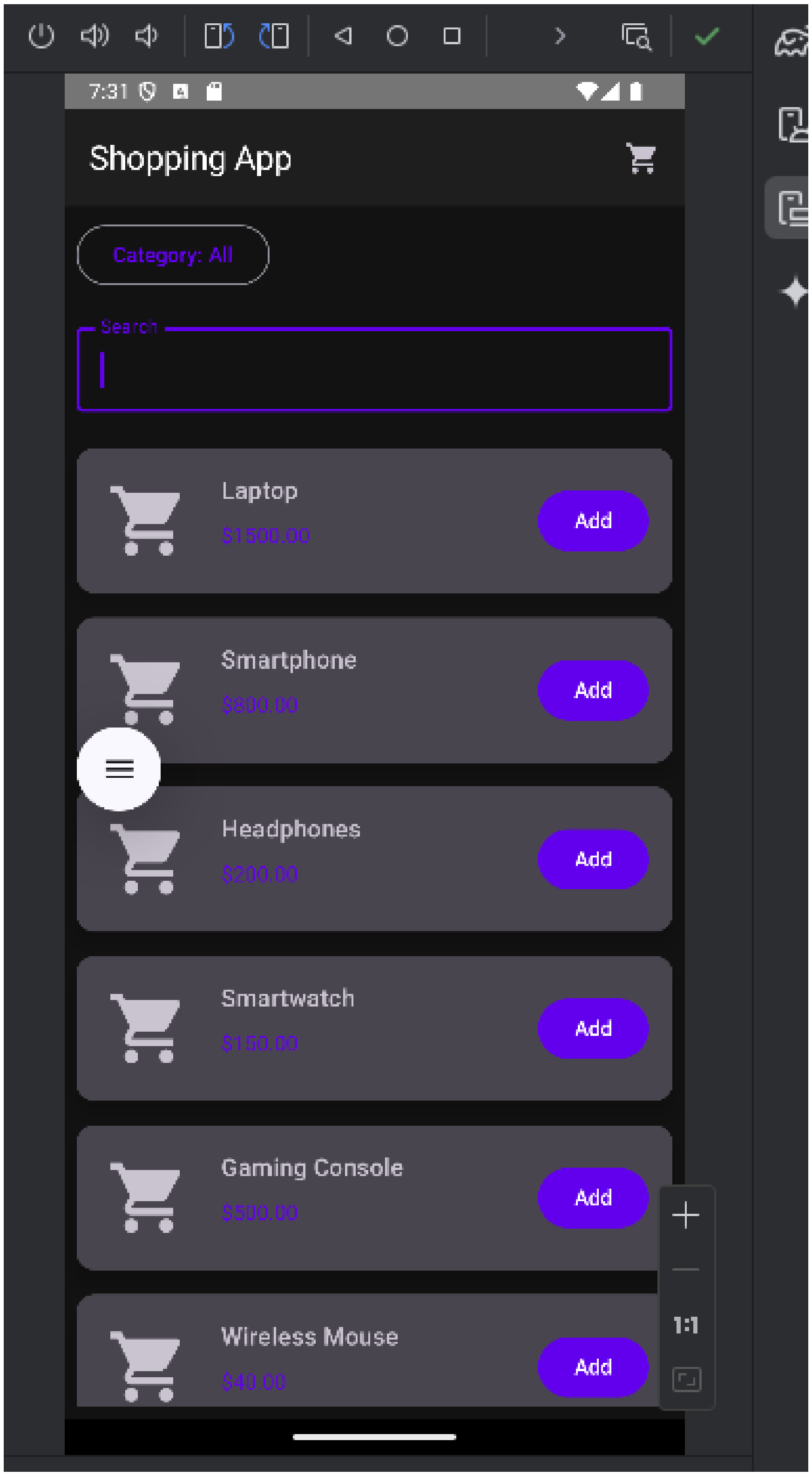
Retrofit

For local data storage, this project uses Room Database, which provides a convenient abstraction layer on top of SQLite. Room makes it easy to create, use, and update a database in an Android app.



```kotlin
interface ProductService {
    @GET("products")
    suspend fun getProducts(): List<Product>
}

object RetrofitClient {

    private const val BASE_URL = "https://jsonplaceholder.typicode.com/"

    private val logging = HttpLoggingInterceptor().apply {
        level = HttpLoggingInterceptor.Level.BODY
    }

    private val client = OkHttpClient.Builder()
        .addInterceptor(logging)
        .build()

    val ProductService: ProductService by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .client(client)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(com.example.finalassignment.ProductService::class.java)
    }
}
```

Retrofit decorator that says to send a GET request to the "products" endpoint. suspend fun getProducts(): This is the function that will load the list of products from the server. List<Product>: Returns a list of Product objects that represent product data. OkHttpClient This is the client for sending HTTP requests. Retrofit.Builder() Creates a Retrofit instance. addConverterFactory adds Gson, which unmarshals JSON server responses into Kotlin objects


7:31

Shopping App

Category: All

Search

Laptop
$1500.00
Add

Smartphone
$800.00
Add

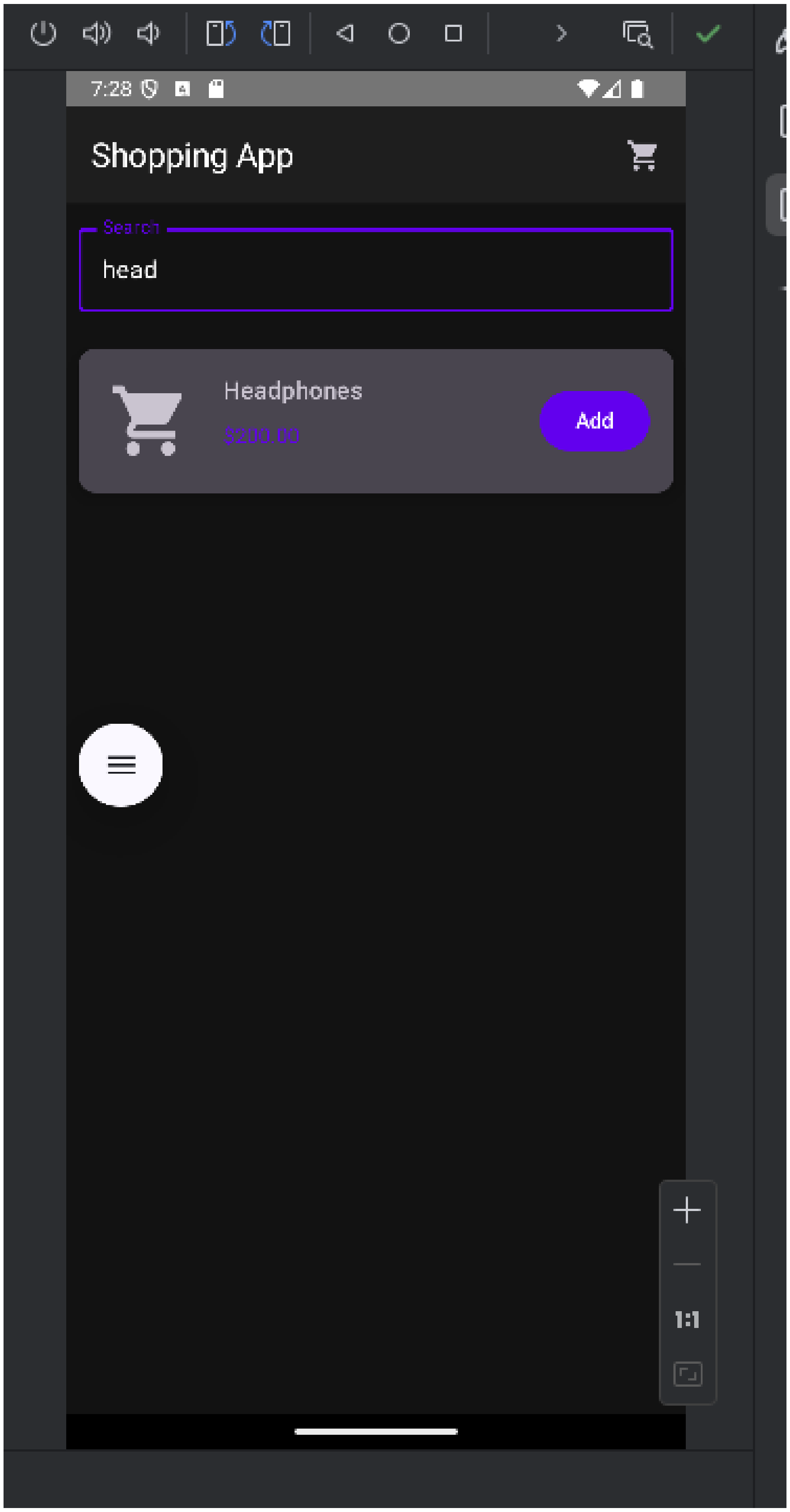
Headphones
$200.00
Add

Smartwatch
$150.00
Add

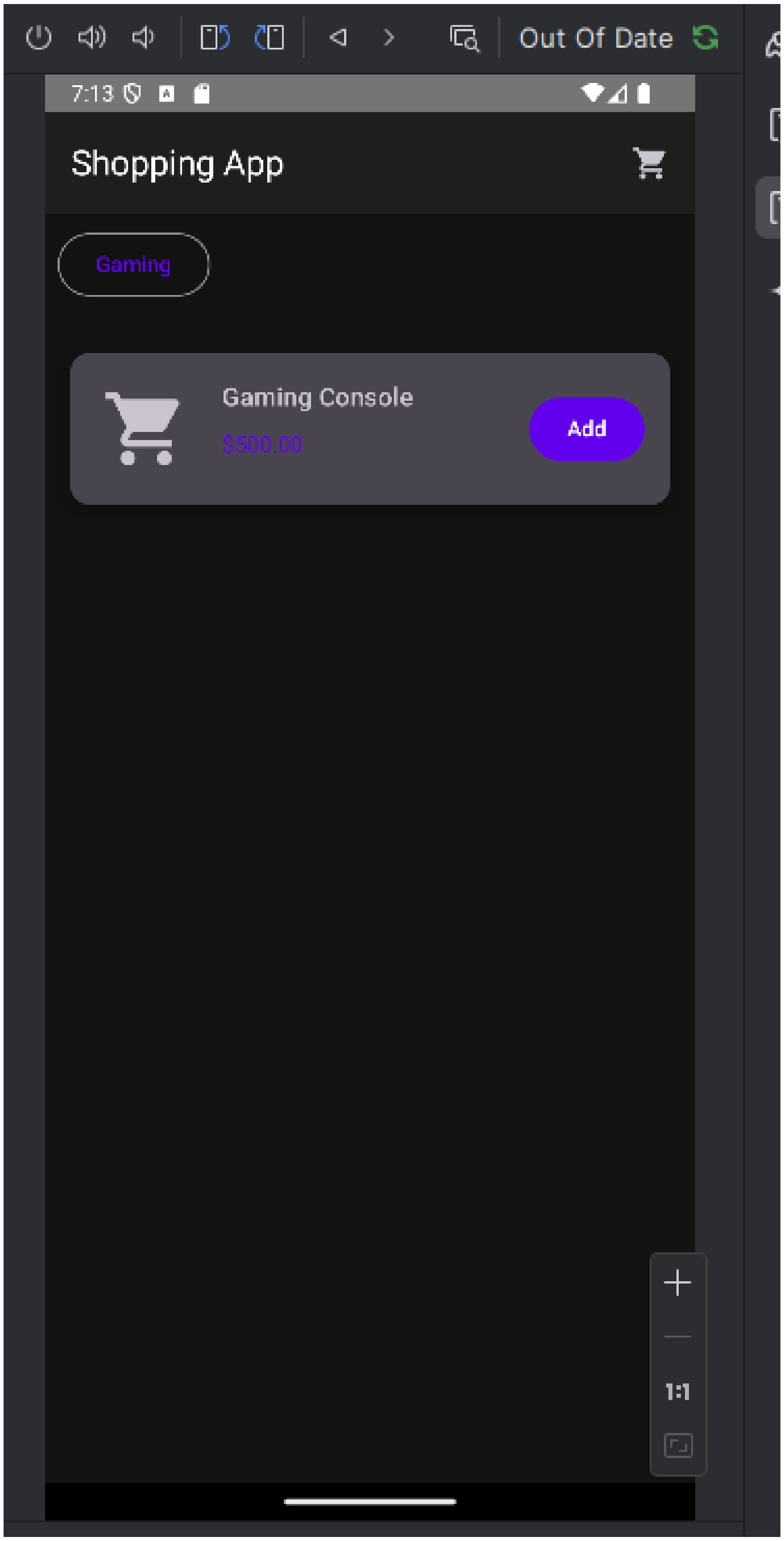
Gaming Console
$500.00
Add

Wireless Mouse
$40.00
Add

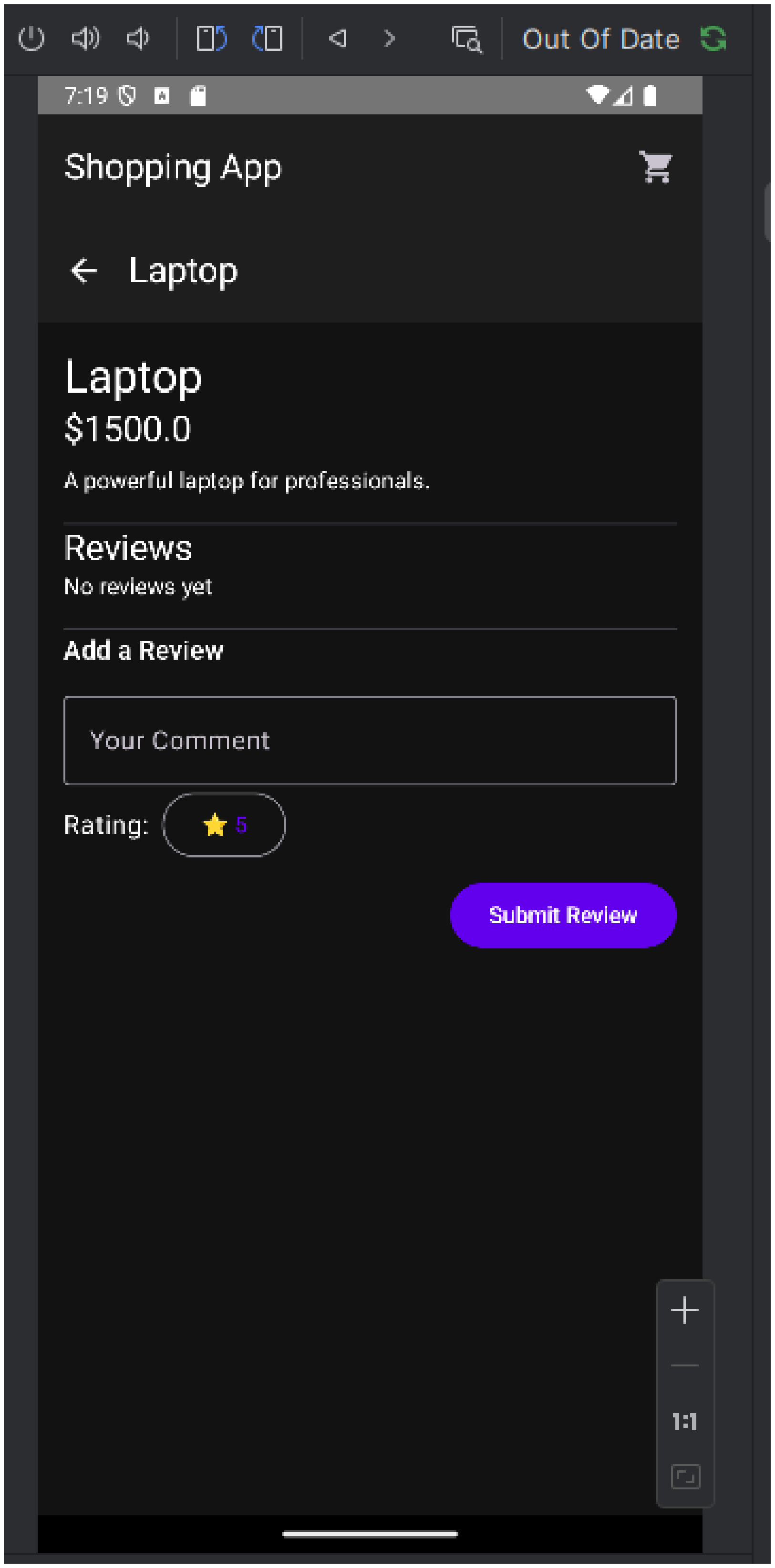
Main menu



Searching items



Filtered by category



Out Of Date

7:19

Shopping App

Laptop

Laptop

$1500.0

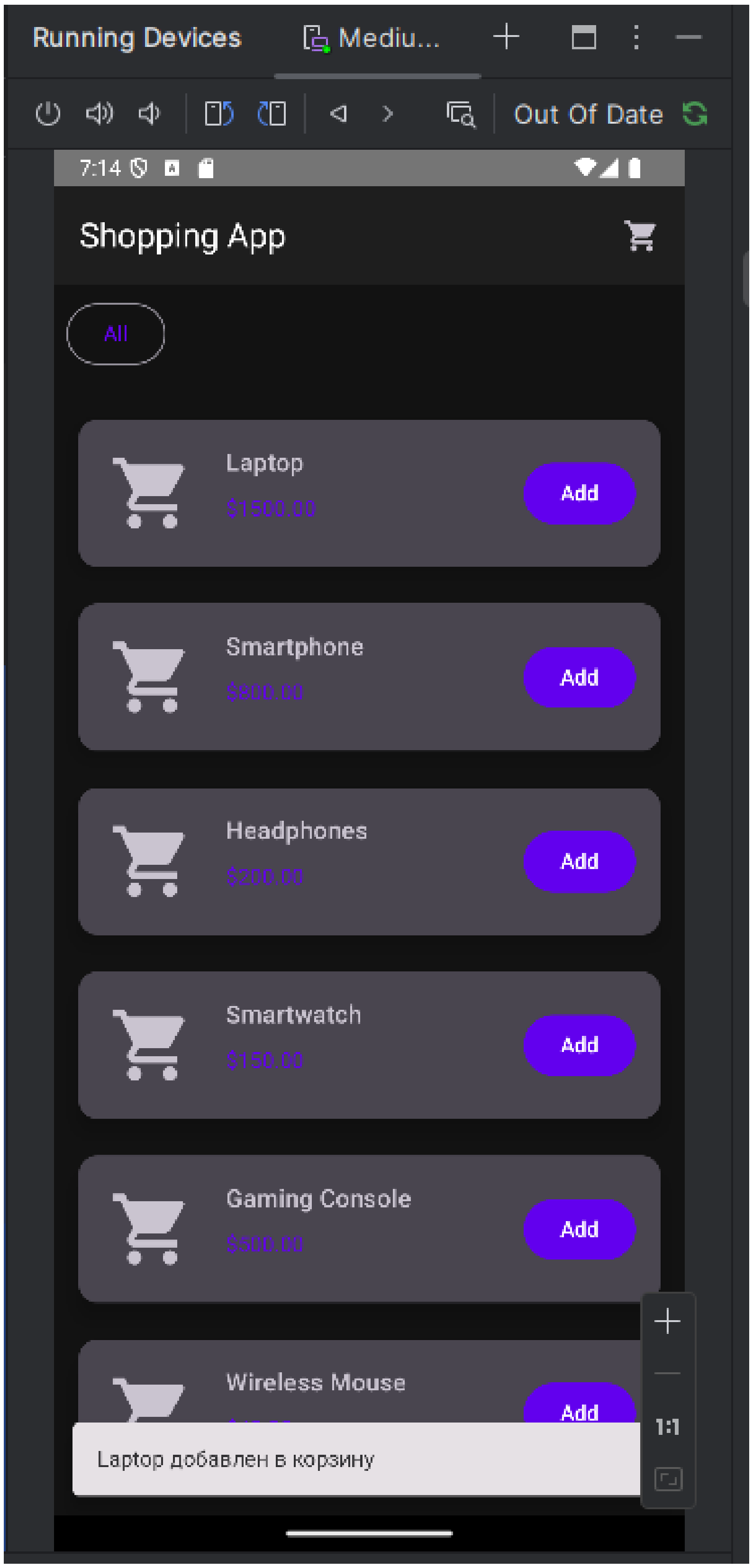
A powerful laptop for professionals.

Reviews

No reviews yet

Add a Review

Item description



Laptop
$1500.00
Add

Smartphone
$800.00
Add

Headphones
$200.00
Add

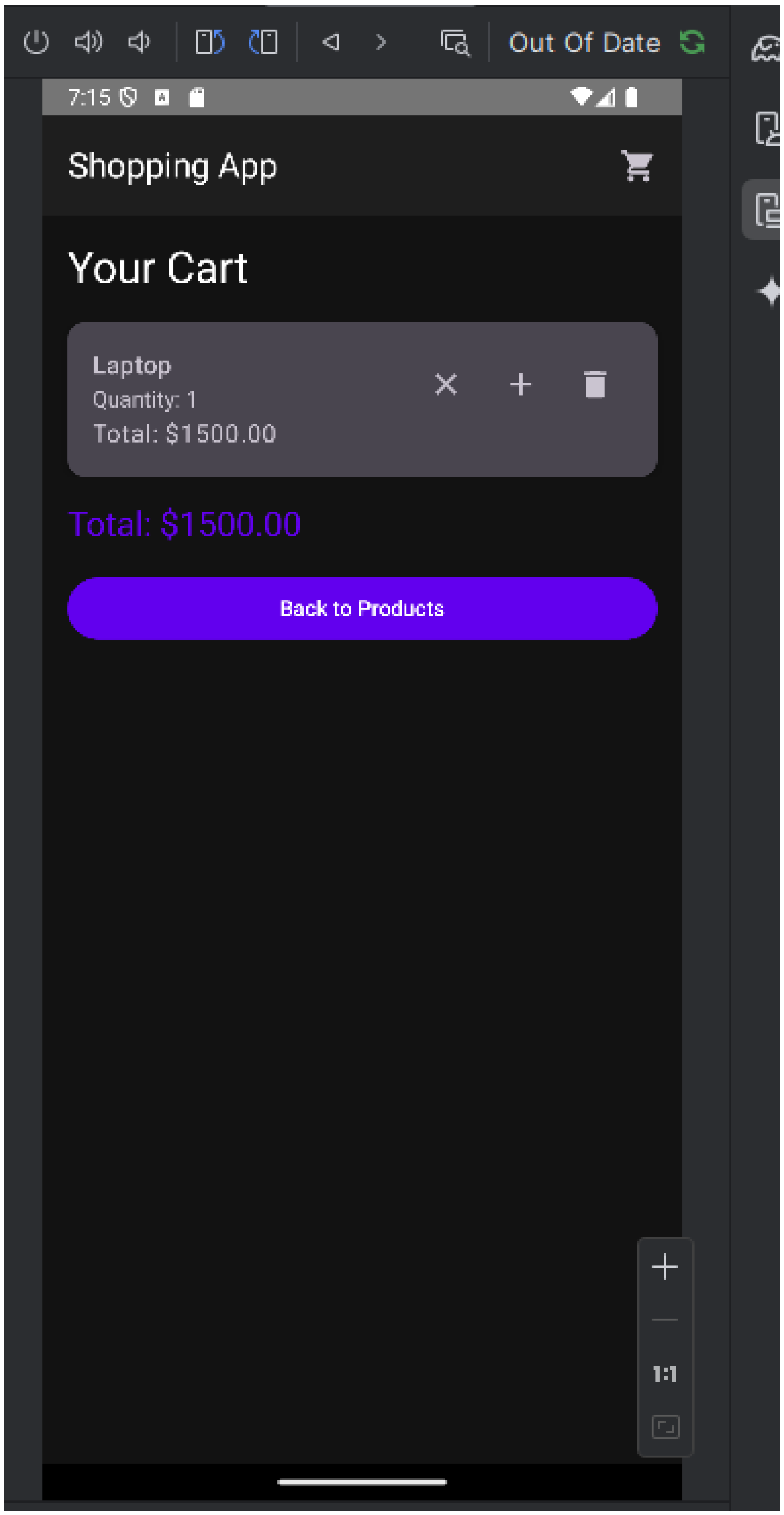
Smartwatch
$150.00
Add

Gaming Console
$500.00
Add

Wireless Mouse
Add

Laptop добавлен в корзину

Adding item to cart

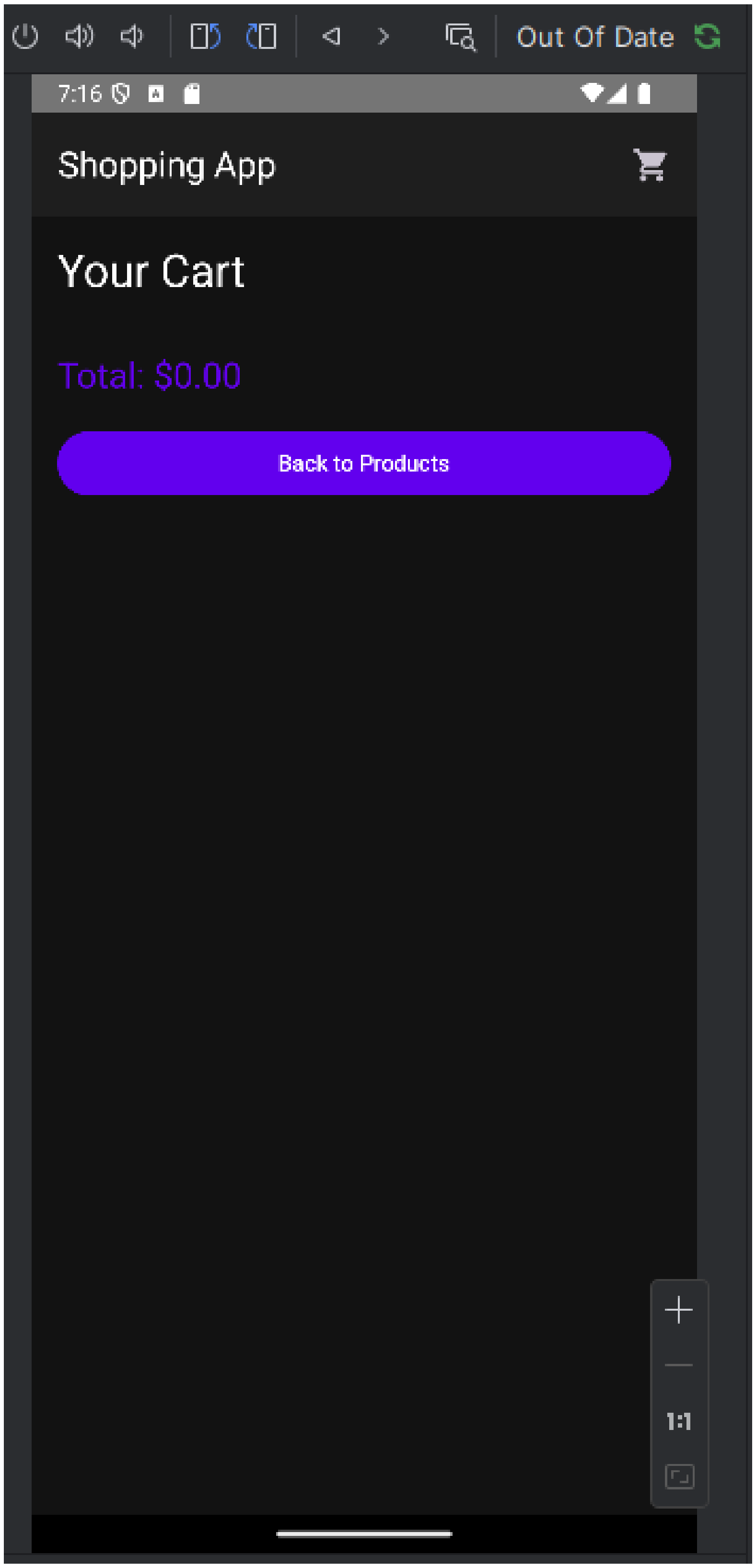


Your Cart

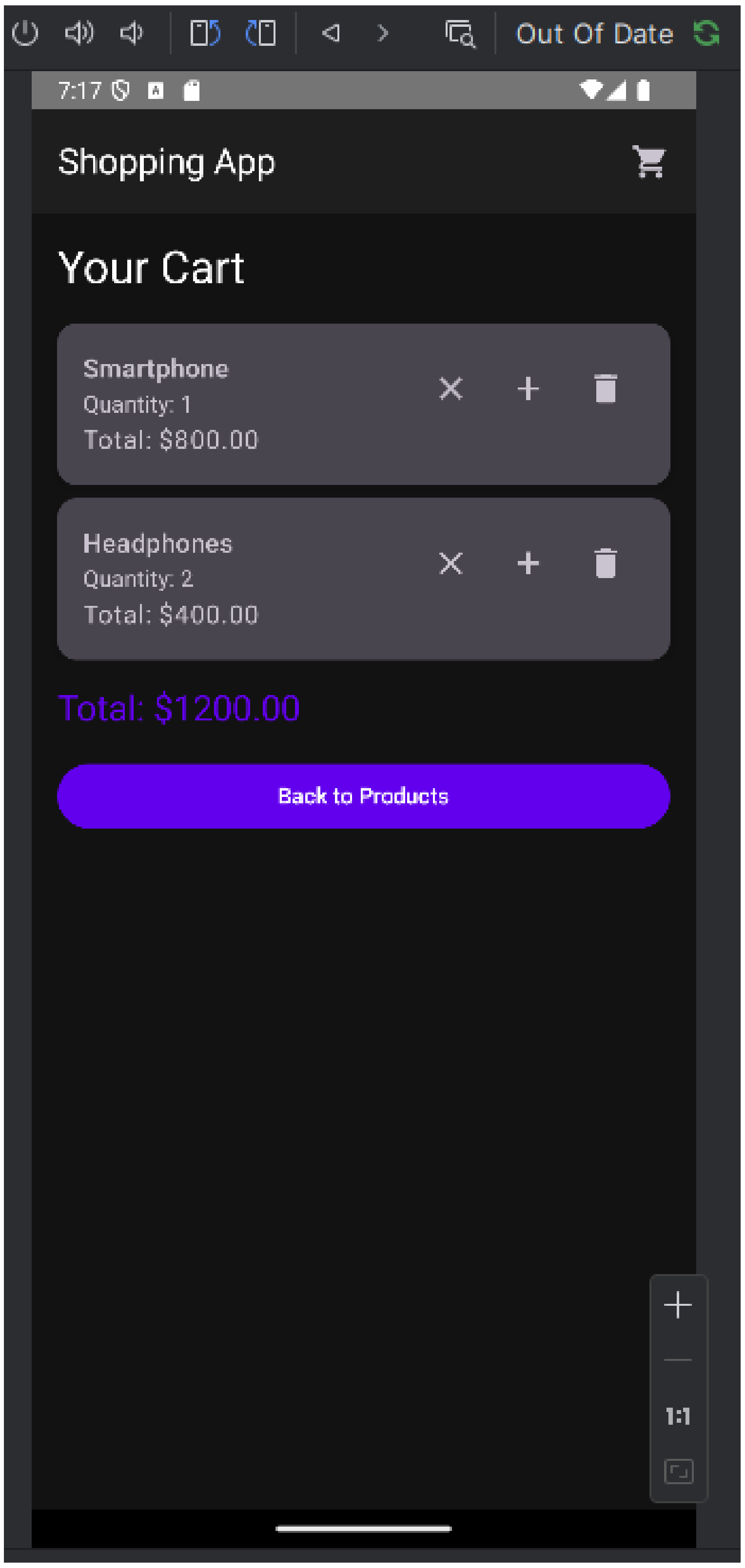
Laptop
Quantity: 1
Total: $1500.00

Total: $1500.00

Cart items


Out Of Date
7:16
Shopping App
Your Cart
Total: $0.00
Back to Products
1:1

Delete item



Shopping App

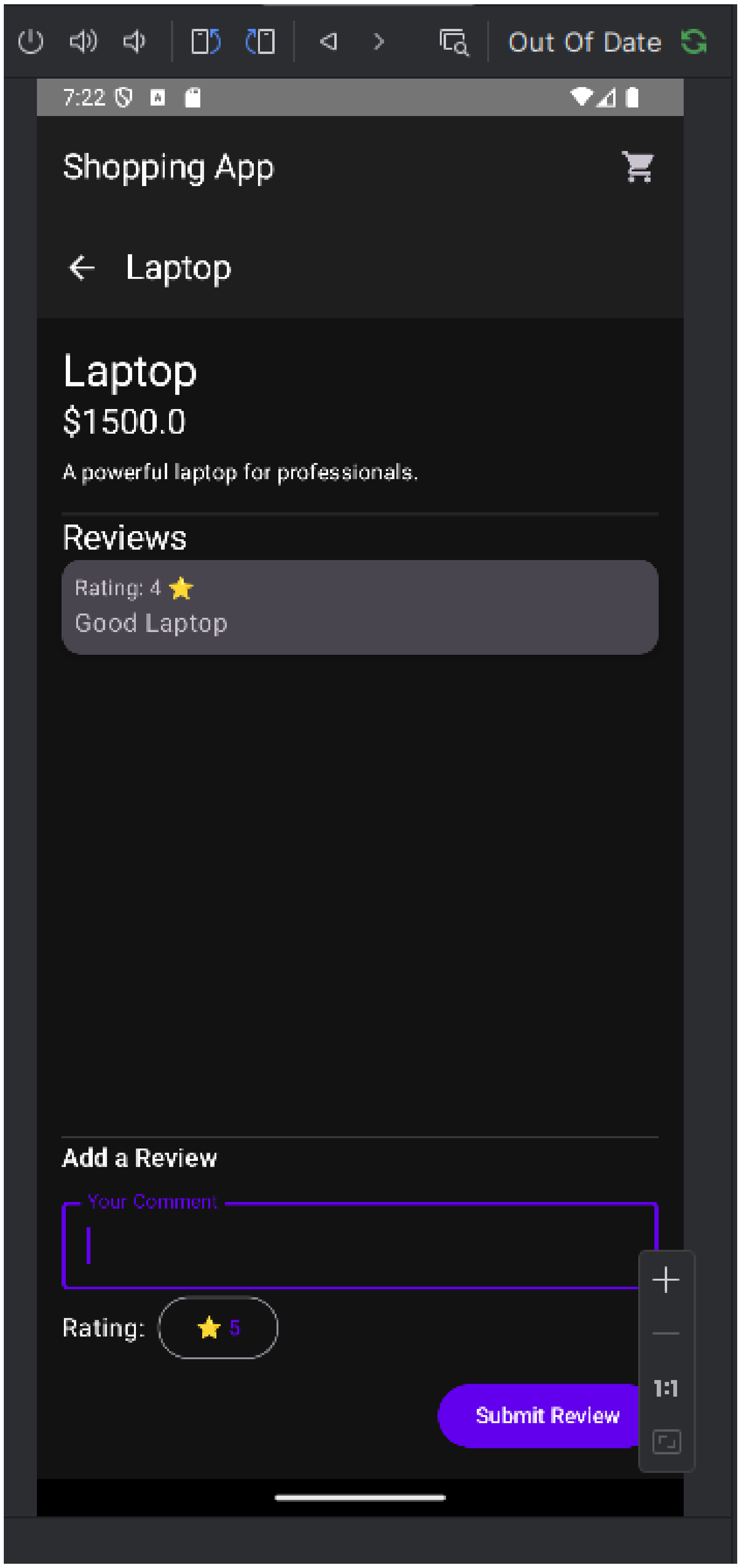
Your Cart

Smartphone
Quantity: 1
Total: $800.00

Headphones
Quantity: 2
Total: $400.00

Total: $1200.00

Increase/Decrease quantity



Out Of Date

7:22

Shopping App

Laptop

Laptop

$1500.0

A powerful laptop for professionals.

Reviews

Rating: 4 ⭐

Good Laptop

Add a Review

Your Comment

Rating: ⭐ 5

Submit Review

Reviewing item

Conclusion

The Mobile Shopping App Development project shows how to create Android app using actual technologies, Jetpack Compose, Room Database, and Retrofit. The app provides core features such as viewing a list of products, filtering them by categories, search, adding items to a shopping cart, and submitting user reviews. These functionalities showcase clean architecture practices, efficient state management, and user-friendly UI design.