**Expense Tracker App Notes**

### 1. Jatpack Compose

### 1. Overview of Jetpack Compose

* **Definition**: Jetpack Compose is a modern, fully declarative UI toolkit for building native Android applications.
* **Release**: Officially launched in **July 2021** by Google.
* **Purpose**: Simplifies UI development by using a declarative programming paradigm, making it easier to build, test, and maintain UI components.

**2. Advantages of Jetpack Compose**

* **Declarative Approach**: Instead of modifying views, developers describe what the UI should look like, and Compose takes care of updating it.
* **Less Boilerplate Code**: Requires fewer lines of code compared to the XML-based approach.
* **Interoperability**: Compose can coexist with XML-based UI, making it easier to migrate existing projects incrementally.
* **Reactive Programming**: Compose automatically updates the UI when data changes, eliminating the need for manual updates like notifyDataSetChanged().
* **Custom Components**: Easily create reusable and custom UI components.

**3. Core Concepts**

* **@Composable Annotation**: Marks a function as composable, enabling it to describe part of the UI.
* **State Management**:
  + Uses State and MutableState to track and update UI changes.
  + State hoisting is encouraged for better reusability and testability.
* **Recomposition**: Compose intelligently re-executes functions when their inputs change, updating only the affected parts of the UI.
* **Modifiers**:
  + Enhance composables with layout, appearance, and behavior.
  + Example: Modifier.padding(), Modifier.fillMaxSize().

**4. Key Components**

* **Scaffold**: A layout structure providing slots for top bars, bottom bars, floating action buttons, etc.
* **Lazy Components**:
  + LazyColumn and LazyRow for efficient vertical and horizontal scrolling lists.
  + Automatically manages memory by only rendering visible items.
* **Material Design**:
  + Compose integrates Material Design 3 (formerly Material You) for building aesthetically pleasing UIs.
  + Pre-built components like Button, TextField, Card, etc.

**5. Navigation in Jetpack Compose**

* **Navigation Component**: Uses NavHost and composable to define navigation graphs and routes.
* **Arguments Passing**: Easily pass and retrieve arguments between destinations.
* **Deep Links**: Supports deep linking for navigation.

**6. Jetpack Compose and MVVM**

* Works seamlessly with the MVVM architecture.
* **ViewModel Integration**:
  + hiltViewModel() is used for injecting ViewModel in composables.
  + StateFlow or LiveData can be observed in composables via collectAsState() or observeAsState().

**7. Performance**

* **Optimized Rendering**: Compose uses a single UI tree for rendering, improving efficiency.
* **No Overdraw**: Avoids traditional view hierarchy inefficiencies, like overdraws.
* **Lazy Components**: Optimized for large datasets.

**8. Testing**

* **Compose Testing**:
  + Provides testing tools like ComposeTestRule.
  + Supports writing UI tests using Jetpack’s testing framework.
  + Debug tools like Layout Inspector and UI Automator.

**9. Interoperability**

* **Compose in XML**:
  + Use ComposeView to embed a composable in a traditional XML layout.
* **XML in Compose**:
  + Use AndroidView to render XML layouts inside Compose.

**10. Migrating to Jetpack Compose**

* Compose can coexist with XML layouts, allowing for an incremental migration approach.
* Google provides tools and resources to ease the migration process.

**11. Frequently Asked Interview Questions**

1. **What is Jetpack Compose, and how is it different from XML?**
   * Answer: Compose is declarative and reactive, reducing boilerplate and improving UI updates compared to XML's imperative approach.
2. **Explain recomposition in Jetpack Compose.**
   * Answer: Recomposition is the process where a composable function re-runs to update the UI when its inputs (state or parameters) change.
3. **What is state hoisting, and why is it important?**
   * Answer: State hoisting is the practice of moving state to a higher composable to improve reusability, testability, and separation of concerns.
4. **How do you handle navigation in Jetpack Compose?**
   * Answer: Using NavHost, NavController, and composable functions, Compose manages navigation with argument passing and deep links.
5. **What are LazyColumn and LazyRow?**
   * Answer: These are optimized layouts for rendering large, scrollable lists, loading only visible items into memory.

### 2. Constraint Layout in Jatpack Compose

**ConstraintLayout** in Jetpack Compose is a powerful layout that allows developers to create complex, responsive UIs by defining constraints between components. It is inspired by the traditional ConstraintLayout in XML but adapted for the declarative nature of Compose.

**Key Features of ConstraintLayout in Compose**

1. **Flexible Constraints**: Define relationships between components (e.g., aligning one composable to another, setting margins, or specifying percentages of the parent container).
2. **Chain Behavior**: Create horizontal or vertical chains of components with specific alignment and distribution rules.
3. **Barrier Support**: Dynamically position elements relative to a group of views (e.g., aligning text next to the widest view in a group).
4. **Guidelines**: Use horizontal or vertical guidelines to align components to a percentage of the parent container.

**Dependency**

To use ConstraintLayout in Jetpack Compose, you need the following dependency:

kotlin

implementation("androidx.constraintlayout:constraintlayout-compose:1.0.1")

**How to Use ConstraintLayout**

Here’s a basic example of using ConstraintLayout in Compose:

kotlin

Copy code

import androidx.compose.foundation.layout.fillMaxSize

import androidx.compose.material3.Text

import androidx.compose.runtime.Composable

import androidx.compose.ui.Modifier

import androidx.compose.ui.unit.dp

import androidx.constraintlayout.compose.ConstraintLayout

@Composable

fun ConstraintLayoutExample() {

ConstraintLayout(

modifier = Modifier.fillMaxSize()

) {

// Create references for composables

val (title, subtitle, button) = createRefs()

// Title

Text(

text = "Welcome to Jetpack Compose",

modifier = Modifier.constrainAs(title) {

top.linkTo(parent.top, margin = 16.dp)

start.linkTo(parent.start)

end.linkTo(parent.end)

}

)

// Subtitle

Text(

text = "ConstraintLayout Example",

modifier = Modifier.constrainAs(subtitle) {

top.linkTo(title.bottom, margin = 8.dp)

start.linkTo(parent.start)

end.linkTo(parent.end)

}

)

// Button

androidx.compose.material3.Button(

onClick = { /\* Do something \*/ },

modifier = Modifier.constrainAs(button) {

top.linkTo(subtitle.bottom, margin = 16.dp)

start.linkTo(parent.start)

end.linkTo(parent.end)

}

) {

Text(text = "Get Started")

}

}

}

**Key Concepts**

1. **createRefs()**:
   * Creates references for composables used within the ConstraintLayout.
   * Each composable must have a reference to define its constraints.
2. **Constraints**:
   * Define relationships such as top.linkTo(), start.linkTo(), end.linkTo(), and bottom.linkTo().
   * Example: Align a composable to the parent or another composable.
3. **Chain**:
   * Create horizontal or vertical chains for distributed alignment of multiple composables.

kotlin

Copy code

val (box1, box2) = createRefs()

createHorizontalChain(box1, box2, chainStyle = ChainStyle.Spread)

1. **Guidelines**:
   * Define fixed or percentage-based guidelines for precise alignment.

kotlin

Copy code

val guideline = createGuidelineFromTop(0.25f) // 25% from the top

Text(

text = "Guideline Example",

modifier = Modifier.constrainAs(ref) {

top.linkTo(guideline)

}

)

1. **Barrier**:
   * Position elements relative to a group of views.

kotlin

Copy code

val barrier = createEndBarrier(box1, box2)

**Advantages of ConstraintLayout**

1. **Powerful Positioning**: Enables precise and complex layouts.
2. **Performance**: Optimized for nested layouts by reducing layout hierarchy.
3. **Responsive Design**: Supports percentage-based alignment and chains.

**Use Cases**

* Complex UIs with many interdependent components.
* Dynamic layouts where constraints depend on the size or position of other composables.
* Responsive designs with adaptive spacing and alignment.

Jetpack Compose’s ConstraintLayout is especially useful when you need the flexibility of constraint-based designs without introducing extra nesting. Let me know if you’d like to see a more complex example!

# ****Room Database Documentation****

This document provides an overview of the implementation and usage of the Room database in the expense tracker application. The database stores and manages expense-related data.

## ****1. Overview of Room****

Room is a persistence library provided by Android that acts as an abstraction layer over SQLite. It provides robust database management with compile-time validation, type safety, and seamless integration with Kotlin Coroutines.

## ****2. Components of the Implementation****

### ****a. Entity****

An entity represents a table in the database. Each instance corresponds to a row in the table.

#### ExpenseEntity

Defined in the com.example.expensetracker.data.model package:

kotlin

Copy code

@Entity(tableName = "expense\_table")

data class ExpenseEntity(

@PrimaryKey(autoGenerate = true)

val id: Int?, // Primary key, auto-generated

val title: String, // Title of the expense

val amount: Double, // Expense amount

val date: String, // Date of the expense

val category: String, // Category of the expense

val type: String // Type of the expense (e.g., Expense, Income)

)

### ****b. Data Transfer Object (DTO)****

Used to retrieve and process aggregated or summarized data from the database.

#### ExpenseSummary

Defined in the com.example.expensetracker.data.model package:

kotlin

Copy code

data class ExpenseSummary(

val type: String, // Type of expense (e.g., Expense, Income)

val date: String, // Date of the expense

val total\_amount: Double // Total amount for the type on the date

)

### ****c. Data Access Object (DAO)****

A DAO defines the methods for interacting with the database. It contains SQL queries and their mapping to methods.

#### ExpenseDao

Defined in the com.example.expensetracker.data.dao package:

kotlin

Copy code

@Dao

interface ExpenseDao {

@Query("SELECT \* FROM expense\_table")

fun getAllExpenses(): Flow<List<ExpenseEntity>>

@Query("SELECT \* FROM expense\_table WHERE type ='Expense' ORDER BY amount DESC LIMIT 5")

fun getTopExpenses(): Flow<List<ExpenseEntity>>

@Query("SELECT type, date, SUM(amount) AS total\_amount FROM expense\_table where type = :type GROUP BY type, date ORDER BY date")

fun getAllExpenseByDate(type: String = "Expense"): Flow<List<ExpenseSummary>>

@Insert

suspend fun insertExpense(expenseEntity: ExpenseEntity)

@Delete

suspend fun deleteExpense(expenseEntity: ExpenseEntity)

@Update

suspend fun updateExpense(expenseEntity: ExpenseEntity)

}

#### ****DAO Methods:****

1. **Retrieve All Expenses**: Fetches all rows in the table.
2. **Top Expenses**: Fetches the top 5 expenses by amount.
3. **Summary by Date**: Aggregates expense data grouped by type and date.
4. **Insert**: Inserts a new expense record.
5. **Delete**: Deletes a specific expense.
6. **Update**: Updates an existing expense.

### ****d. Database****

The database class serves as the main access point to the Room database.

#### ExpenseDataBase

Defined in the com.example.expensetracker.data package:

kotlin

Copy code

@Database(entities = [ExpenseEntity::class], version = 2)

abstract class ExpenseDataBase : RoomDatabase() {

abstract fun expenseDao(): ExpenseDao

companion object {

const val DATABASE\_NAME = "expense\_database"

fun getDatabase(context: Context): ExpenseDataBase {

return Room.databaseBuilder(

context.applicationContext,

ExpenseDataBase::class.java,

DATABASE\_NAME

).fallbackToDestructiveMigration().build()

}

}

}

#### ****Database Configuration:****

* **Entities**: ExpenseEntity table.
* **Version**: 2 (supports migrations with destructive fallback).
* **Instance Retrieval**: getDatabase method for singleton instance creation.

## ****3. How to Use the Room Database****

### ****Setup****

Add the following dependency to the build.gradle file:

gradle

Copy code

implementation "androidx.room:room-runtime:2.5.1"

kapt "androidx.room:room-compiler:2.5.1"

### ****Creating the Database Instance****

kotlin

Copy code

val database = ExpenseDataBase.getDatabase(context)

val expenseDao = database.expenseDao()

### ****Example Usage****

#### Insert an Expense:

kotlin

Copy code

val expense = ExpenseEntity(null, "Groceries", 50.0, "2024-12-24", "Food", "Expense")

CoroutineScope(Dispatchers.IO).launch {

expenseDao.insertExpense(expense)

}

#### Fetch All Expenses:

kotlin

Copy code

expenseDao.getAllExpenses().collect { expenses ->

// Handle the list of expenses

}

#### Update an Expense:

kotlin

Copy code

val updatedExpense = expense.copy(amount = 60.0)

CoroutineScope(Dispatchers.IO).launch {

expenseDao.updateExpense(updatedExpense)

}

## ****4. Advantages of Room****

1. **Simplified Database Access**: Removes boilerplate code for SQLite database management.
2. **Compile-Time Verification**: SQL queries are validated at compile time.
3. **Kotlin Integration**: Works seamlessly with Kotlin Coroutines and Flow.
4. **Easy Migration**: Supports database schema changes with migration strategies.

## ****5. Notes****

* Use Flow to observe changes in the database in real-time.
* For large datasets, consider using Paging to handle data efficiently.
* Avoid running database operations on the main thread.

## ****View Models****

## ****1. HomeViewModel****

HomeViewModel is a class that extends ViewModel and is responsible for managing UI-related data in a lifecycle-conscious way. It interacts with the ExpenseDao to fetch and process data from the Room database.

### ****Key Components****

#### ****a. Constructor****

kotlin

Copy code

class HomeViewModel(dao: ExpenseDao) : ViewModel() {

val expenses = dao.getAllExpenses()

}

* **Parameter:**
  + dao: An instance of ExpenseDao to perform database operations.
* **Property:**
  + expenses: A Flow of List<ExpenseEntity> fetched using getAllExpenses() from ExpenseDao. This provides real-time updates for any changes in the database.

#### ****b. Functions****

1. **getBalance**

kotlin

Copy code

fun getBalance(list: List<ExpenseEntity>): String {

var total = 0.0

list.forEach {

if (it.type == "Income") {

total += it.amount

} else {

total -= it.amount

}

}

return "Rs. ${Utils.formatToDecimalValue(total)}"

}

* + **Purpose:** Calculates the overall balance by subtracting total expenses from total income.
  + **Logic:**
    - If the type is Income, the amount is added to the balance.
    - If the type is Expense, the amount is subtracted from the balance.
  + **Returns:** A formatted string showing the balance.

1. **getTotalExpense**

kotlin

Copy code

fun getTotalExpense(list: List<ExpenseEntity>): String {

var total = 0.0

list.forEach {

if (it.type == "Expense") {

total += it.amount

}

}

return "Rs. ${Utils.formatToDecimalValue(total)}"

}

* + **Purpose:** Calculates the total expenses.
  + **Logic:** Adds up the amounts for all items where the type is Expense.
  + **Returns:** A formatted string showing the total expenses.

1. **getTotalIncome**

kotlin

Copy code

fun getTotalIncome(list: List<ExpenseEntity>): String {

var total = 0.0

list.forEach {

if (it.type == "Income") {

total += it.amount

}

}

return "RS. ${Utils.formatToDecimalValue(total)}"

}

* + **Purpose:** Calculates the total income.
  + **Logic:** Adds up the amounts for all items where the type is Income.
  + **Returns:** A formatted string showing the total income.

## ****2. HomeViewModelFactory****

The HomeViewModelFactory is a factory class used to create instances of HomeViewModel. It ensures the ViewModel is provided with the necessary dependencies.

### ****Key Components****

#### ****a. Constructor****

kotlin

Copy code

class HomeViewModelFactory(private val context: Context): ViewModelProvider.Factory

* **Parameter:**
  + context: The application context used to initialize the database.

#### ****b.**** create ****Method****

kotlin

Copy code

override fun <T : ViewModel> create(modelClass: Class<T>): T {

if (modelClass.isAssignableFrom(HomeViewModel::class.java)) {

val dao = ExpenseDataBase.getDatabase(context).expenseDao()

@Suppress("UNCHECKED\_CAST")

return HomeViewModel(dao) as T

}

throw IllegalArgumentException("Unknown ViewModel class")

}

* **Logic:**
  + Verifies if the requested modelClass is HomeViewModel.
  + Retrieves the ExpenseDao instance from ExpenseDataBase using the provided context.
  + Returns a new instance of HomeViewModel with the dao.
  + Throws an exception if the modelClass is not recognized.
* **Usage:** This method provides dependency injection for the HomeViewModel.

## ****3. Utilities Used****

### Utils.formatToDecimalValue

This utility function formats the total value to a specific decimal format, ensuring consistent display of monetary values. The exact implementation is not provided but is essential for formatting.

## ****4. Example Usage****

### ****ViewModel Initialization in Activity/Fragment****

kotlin

Copy code

val factory = HomeViewModelFactory(context)

val viewModel = ViewModelProvider(this, factory).get(HomeViewModel::class.java)

### ****Observing Expenses****

kotlin

Copy code

viewModel.expenses.collect { expenses ->

val balance = viewModel.getBalance(expenses)

val totalExpense = viewModel.getTotalExpense(expenses)

val totalIncome = viewModel.getTotalIncome(expenses)

// Update the UI with the computed values

}

## ****5. Advantages of This Implementation****

1. **Separation of Concerns:** The ViewModel handles business logic, while the UI remains responsible for rendering.
2. **Lifecycle Awareness:** The ViewModel survives configuration changes like screen rotations.
3. **Real-Time Updates:** Leveraging Flow from Room, the data is updated automatically in the UI when changes occur in the database.
4. **Dependency Injection:** Using ViewModelFactory ensures the ViewModel gets the required dependencies efficiently.

## ****2. AddExpenseViewModel****

AddExpenseViewModel is responsible for managing the logic and operations related to adding expenses to the database.

### ****Key Components****

#### ****a. Constructor****

kotlin

Copy code

class AddExpenseViewModel(val dao: ExpenseDao): ViewModel()

* **Parameter:**
  + dao: An instance of ExpenseDao that facilitates database operations.

#### ****b. Function:**** addExpense

kotlin

Copy code

suspend fun addExpense(expenseEntity: ExpenseEntity): Boolean {

return try {

dao.insertExpense(expenseEntity)

true

} catch (ex: Throwable) {

false

}

}

* **Purpose:**
  + Adds an expense record to the expense\_table.
* **Parameters:**
  + expenseEntity: An instance of ExpenseEntity containing details such as title, amount, date, category, and type.
* **Logic:**
  + Tries to insert the expense into the database using insertExpense from ExpenseDao.
  + Returns true if the operation succeeds, or false if an exception is thrown.
* **Return Type:**
  + A Boolean indicating whether the insertion was successful.

## ****2. AddExpenseViewModelFactory****

The AddExpenseViewModelFactory is a factory class used to create instances of AddExpenseViewModel. It ensures proper dependency injection for the ViewModel.

### ****Key Components****

#### ****a. Constructor****

kotlin

Copy code

class AddExpenseViewModelFactory(private val context: Context): ViewModelProvider.Factory

* **Parameter:**
  + context: The application context used to initialize the Room database.

#### ****b.**** create ****Method****

kotlin

Copy code

override fun <T : ViewModel> create(modelClass: Class<T>): T {

if (modelClass.isAssignableFrom(AddExpenseViewModel::class.java)) {

val dao = ExpenseDataBase.getDatabase(context).expenseDao()

@Suppress("UNCHECKED\_CAST")

return AddExpenseViewModel(dao) as T

}

throw IllegalArgumentException("Unknown ViewModel class")

}

* **Logic:**
  + Checks if the requested modelClass is AddExpenseViewModel.
  + If yes, retrieves the ExpenseDao instance from ExpenseDataBase using the provided context.
  + Returns a new instance of AddExpenseViewModel with the dao.
  + Throws an exception for unknown ViewModel classes.
* **Usage:**
  + This ensures the AddExpenseViewModel has access to the database for its operations.

## ****3. Example Usage****

### ****ViewModel Initialization in Activity/Fragment****

kotlin

Copy code

val factory = AddExpenseViewModelFactory(context)

val viewModel = ViewModelProvider(this, factory).get(AddExpenseViewModel::class.java)

### ****Adding an Expense****

kotlin

Copy code

val newExpense = ExpenseEntity(

id = null, // auto-generated

title = "Groceries",

amount = 1200.50,

date = "2024-12-25",

category = "Food",

type = "Expense"

)

lifecycleScope.launch {

val success = viewModel.addExpense(newExpense)

if (success) {

// Expense added successfully

} else {

// Handle error

}

}

## ****4. Advantages of This Implementation****

1. **Separation of Concerns:**
   * The ViewModel handles all data-related operations, while the UI handles user interactions and rendering.
2. **Error Handling:**
   * Encapsulates exceptions and provides a clear success/failure response (Boolean).
3. **Dependency Injection:**
   * The ViewModelFactory ensures that the ViewModel has access to the required ExpenseDao instance.
4. **Lifecycle Awareness:**
   * The ViewModel survives configuration changes like screen rotations.

## ****3. StatsViewModel****

StatsViewModel handles the logic for retrieving and processing statistical data related to expenses.

### ****Key Components****

#### ****a. Constructor****

kotlin

Copy code

class StatsViewModel(dao: ExpenseDao) : ViewModel()

* **Parameter:**
  + dao: An instance of ExpenseDao for performing database operations.

#### ****b. Properties****

1. **entities**

kotlin

Copy code

val entities = dao.getAllExpenseByDate()

* + Fetches a list of ExpenseSummary objects grouped by type and date from the database.
  + Returns a Flow<List<ExpenseSummary>>.

1. **topEntries**

kotlin

Copy code

val topEntries = dao.getTopExpenses()

* + Retrieves the top 5 expenses based on amount.
  + Returns a Flow<List<ExpenseEntity>>.

#### ****c. Function:**** getEntriesForChart

kotlin

Copy code

fun getEntriesForChart(entries: List<ExpenseSummary>): List<Entry> {

val list = mutableListOf<Entry>()

for (entry in entries) {

val formattedDate = Utils.getMilliFromDate(entry.date)

list.add(Entry(formattedDate.toFloat(), entry.total\_amount.toFloat()))

}

return list

}

* **Purpose:**
  + Converts a list of ExpenseSummary into chart entries compatible with the MPAndroidChart library.
* **Parameters:**
  + entries: A list of ExpenseSummary containing the date and total amount for each group.
* **Logic:**
  + Converts the date string into milliseconds using Utils.getMilliFromDate.
  + Creates an Entry object for each date and total amount, which can be used in MPAndroidChart.
* **Return Type:**
  + A List<Entry> containing formatted data for charting.

## ****2. StatsViewModelFactory****

The StatsViewModelFactory is a factory class that ensures proper dependency injection for creating StatsViewModel instances.

### ****Key Components****

#### ****a. Constructor****

kotlin

Copy code

class StatsViewModelFactory(private val context: Context): ViewModelProvider.Factory

* **Parameter:**
  + context: The application context used for initializing the Room database.

#### ****b.**** create ****Method****

kotlin

Copy code

override fun <T : ViewModel> create(modelClass: Class<T>): T {

if (modelClass.isAssignableFrom(StatsViewModel::class.java)) {

val dao = ExpenseDataBase.getDatabase(context).expenseDao()

@Suppress("UNCHECKED\_CAST")

return StatsViewModel(dao) as T

}

throw IllegalArgumentException("Unknown ViewModel class")

}

* **Logic:**
  + Checks if the modelClass is StatsViewModel.
  + Retrieves the ExpenseDao instance from ExpenseDataBase using the provided context.
  + Returns an instance of StatsViewModel with the dao.
  + Throws an exception if the modelClass is unknown.

## ****3. Example Usage****

### ****ViewModel Initialization in Activity/Fragment****

kotlin

Copy code

val factory = StatsViewModelFactory(context)

val viewModel = ViewModelProvider(this, factory).get(StatsViewModel::class.java)

### ****Observing Data****

kotlin

Copy code

viewModel.entities.collect { summaries ->

val chartEntries = viewModel.getEntriesForChart(summaries)

// Use chartEntries to populate the chart

}

### ****Top Expenses****

kotlin

Copy code

viewModel.topEntries.collect { expenses ->

// Display top expenses in the UI

}

## ****4. Advantages of This Implementation****

1. **Separation of Concerns:**
   * The ViewModel handles data processing, while the UI focuses on rendering.
2. **Chart Integration:**
   * Prepares data specifically for MPAndroidChart, simplifying chart implementation in the UI.
3. **Dependency Injection:**
   * Ensures the StatsViewModel is correctly initialized with its dependencies via StatsViewModelFactory.
4. **Room Integration:**
   * Retrieves data efficiently using DAO queries (getAllExpenseByDate and getTopExpenses).
5. **Lifecycle Awareness:**
   * The ViewModel survives configuration changes like screen rotations.

## ****5. Key Concepts Highlighted****

1. **ViewModel:** Manages UI-related data in a lifecycle-aware manner.
2. **ViewModelProvider.Factory:** Ensures proper dependency injection for the ViewModel.
3. **Room DAO:** Demonstrates advanced DAO queries for grouping and limiting data.
4. **MPAndroidChart Integration:** Shows how to prepare data for chart visualization.
5. **Coroutines and Flow:** Enables asynchronous and reactive data operations.