Mobile Application Development

Week6 MVVM Architecture
JSON, Retrofit, HTTP Requests and Restful APIs

Yi SUN
Kobe Institute of Computing

MVVM Architecture

- The **Model-View-ViewModel** (MVVM) architecture is a design pattern that helps separate your app's logic from its UI. This separation makes your code easier to understand, maintain, and test.

https://developer.android.com/topic/architecture

Model

- The Model represents your data layer. This includes data classes, database interactions, network calls, or any repository pattern that fetches and stores data.
- Role:

  It doesn't know anything about the UI. Its job is to provide data to the rest of your app.

ViewModel

- The ViewModel acts as a bridge between the Model and the View. It holds and manages UI-related data in a lifecycle-conscious way.
- Role
  - It fetches data from the Model.
  - It processes or transforms data if needed.
  - It exposes the state (using tools like LiveData, StateFlow, or Compose's mutableStateOf) that the UI can observe.
  - It handles user interactions forwarded by the View and updates the state accordingly.

View

- In Jetpack Compose, the View is made up of @Composable functions that define your UI.
- Role
  - It displays the UI based on the state it observes from the ViewModel.
  - It reacts to state changes automatically. When the state in the ViewModel changes, Compose will recompose the UI to reflect the new state.

Demo
Counter App

- We are building a simple counter app.
- The ViewModel holds the counter value and updates it, while the Composable functions render the UI.

1. Open AndroidStudio
2. Select `File > New > NewProject`
3. Choose `Empty Activity`
4. Name: CounterApp

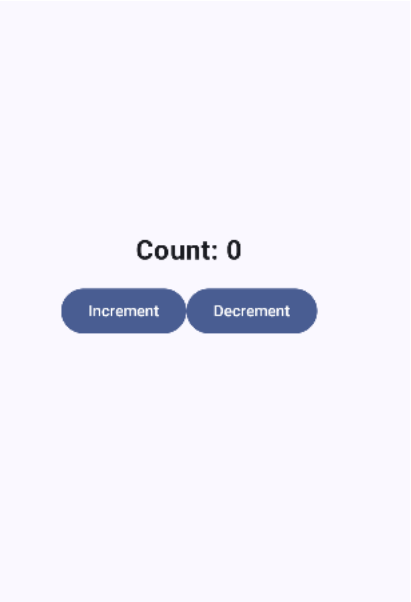
The Counter App without Model

```
@Composable
fun TheCounterApp(modifier: Modifier=Modifier){
    val count = remember { mutableStateOf(0) }
    fun increment(){
        count.value++
    }
    fun decrement(){
        count.value--
    }
    Column(modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment =  Alignment.CenterHorizontally
    ) {
        Text(text="Count: ${count.value}", fontSize = 24.sp, fontWeight =
FontWeight.Bold)
        Spacer(modifier= Modifier.height(16.dp))
        Row{
            Button(onClick = {increment()}) {
                Text("Increment")
            }
            Button(onClick = {decrement()}) {
                Text("Decrement")
```

Write a Compose to Count

```
Scaffold(modifier = Modifier.fillMaxSize()) {
innerPadding ->
    TheCounterApp(
        modifier = Modifier.padding(innerPadding)
    )
}
```

Run the TheCounterApp() at Main Screen

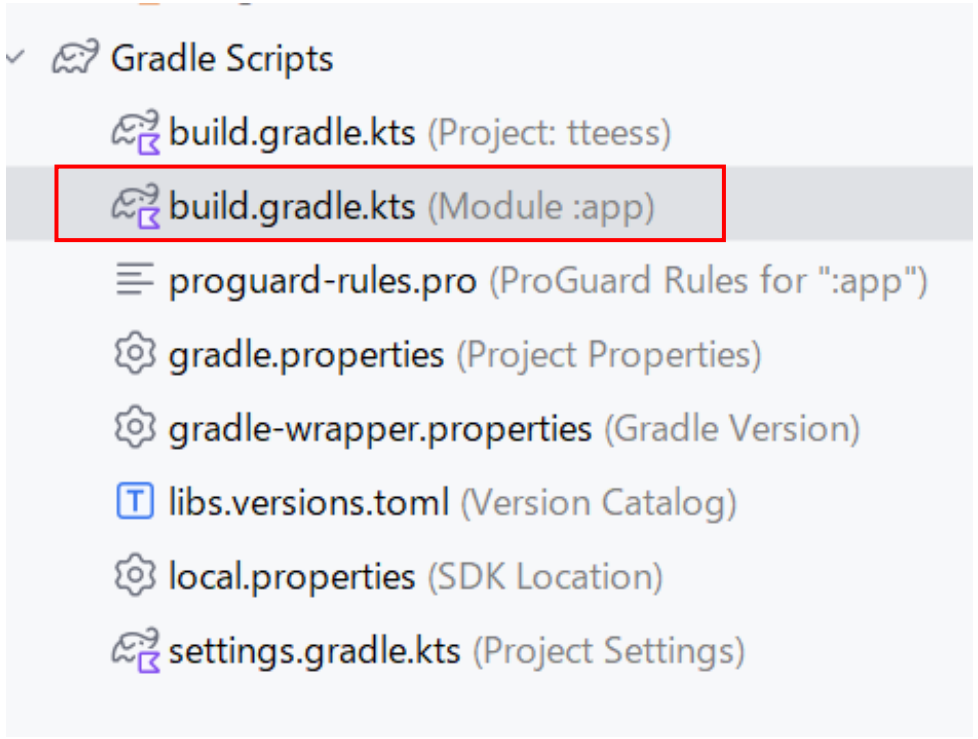


Convert the App by MVVM

- Add the Dependency
- Create a View Model
- Use the View Model at the View

Add the Dependency

- Make sure you've added the correct dependency in your build.gradle (app-level) file:




```
dependencies {

    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.8.7")
    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
```

- Sync the Project:After making these changes, sync your project in Android Studio to download the dependency.
- Cache/Index Issues:If everything seems correct but the dependency still isn't found, try cleaning the project and rebuilding it. You can also invalidate caches via File > Invalidate Caches / Restart... in Android Studio.

Create a View Model

- New Kotlin Class file CounterViewModel.kt

a variable name with an underscore (_) is a naming convention typically used to indicate that the variable is "private"

```
class CounterViewModel : ViewModel() {
    private val _count = mutableStateOf(0)
    val count: MutableState<Int> = _count
    fun increment(){
        _count.value++
    }
    fun decrement(){
        _count.value--
    }
}
```

_count is a private mutable state, while count is a public read-only state. External code can only access the value of _count through count.

Use the ViewModel at the View

```
fun TheCounterApp(
    viewModel: CounterViewModel,
    modifier: Modifier=Modifier){

    Column(modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment =  Alignment.CenterHorizontally
    ) {
        Text(text="Count: ${viewModel.count.value}",
            fontSize = 24.sp,
            fontWeight = FontWeight.Bold)
        Spacer(modifier= Modifier.height(16.dp))
        Row{
            Button(onClick = {viewModel.increment()}) {
                Text("Increment")
            }
            Button(onClick = {viewModel.decrement()}) {
                Text("Decrement")
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            val viewModel: CounterViewModel = viewModel()
            TteessTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) {
innerPadding ->
                    TheCounterApp(
                        viewModel,
                        modifier = Modifier.padding(innerPadding)
                    )
                }
```

Add the Model for CounterApp

- New Kotlin Class CounterModel.kt

```
data class CounterModel(var count : Int)

class CounterRepository{
    private var _counter = CounterModel(0)
    fun getCounter() = _counter
    fun incrementCounter(){
        _counter.count++
    }

    fun decrementCounter(){
        _counter.count--
    }
```

Define a data class to hold data, the only one Property is a mutable integer (var) that represents a counter's value.

The CounterRepository class manages an instance of CounterModel. It abstracts the logic for modifying the counter, which is useful for maintaining separation of concerns (a concept in MVVM architecture where data handling is separated from UI logic).

Modify the ViewModel

Add an instance of CounterRepository, name to repository

```
class CounterViewModel() : ViewModel() {
    private  val _repository: CounterRepository = CounterRepository()
    private val _count = mutableStateOf(_repository.getCounter().count)
    val count: MutableState<Int> = _count
    fun increment(){
        _repository.incrementCounter()
        _count.value = _repository.getCounter().count
    }
    fun decrement(){
        _repository.decrementCounter()
        _count.value = _repository.getCounter().count
    }
}
```

Use the function in the instance

Take the data from the instance

set the value from the instance

same as the increment()

Shopping App

- We will learn how to fetch data from the Internet for Android use by building a shopping app, as well as how to parse and utilize JSON files, and how to display images on the screen.

1. Open AndroidStudio
2. Select `File > New > NewProject`
3. Choose `Empty Activity`
4. Name: ShoppingApp

What is JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

https://www.json.org/json-en.html

https://jsonformatter.org

Understand the Structure of JSON

```
{
  "name": "Alice",
  "age": 30,
  "isStudent": false,
  "courses": ["Math", "Science", "History"],
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "zip": "10001"
  }
}
```

Objects: The entire JSON is an **object**, indicated by the curly braces { }. Inside, there are key/value pairs.

Keys and Values:

- "name": "Alice": The key is "name" and the value is "Alice", a string.
- "age": 30: The key is "age" with the numeric value 30.“
- isStudent": false: The key is "isStudent" with the boolean value false.

Arrays:

"courses": ["Math", "Science", "History"]: The key "courses" has an array as its value. The array is indicated by square brackets [ ] and contains a list of strings.

Nested Objects:

"address": { ... }: The key "address" holds another object as its value. This nested object contains its own set of key/value pairs, such as "street", "city", and "zip".

Use the Fake data API to get the JSON data

- https://fakestoreapi.com/
- fakeStoreApi can be used with any type of shopping project that needs products, carts, and users in JSON format. you can use examples below to check how fakeStoreApi works and feel free to enjoy it in your awesome projects!

Add the Dependencies for JSON & Network Calls

- Retrofit
  - Retrofit is a popular HTTP client library for Android (and also Java and Kotlin) that simplifies the process of making network requests to web services. It's developed by Square and is widely used in the Android development community.

```
dependencies {

    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.8.7")
    implementation("com.squareup.retrofit2:retrofit:2.11.0")
    implementation("com.squareup.retrofit2:converter-gson:2.11.0")
    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
```

https://square.github.io/retrofit/

Add the Dependencies for image loading

- Coil is an image loading library for Android and Compose Multiplatform.

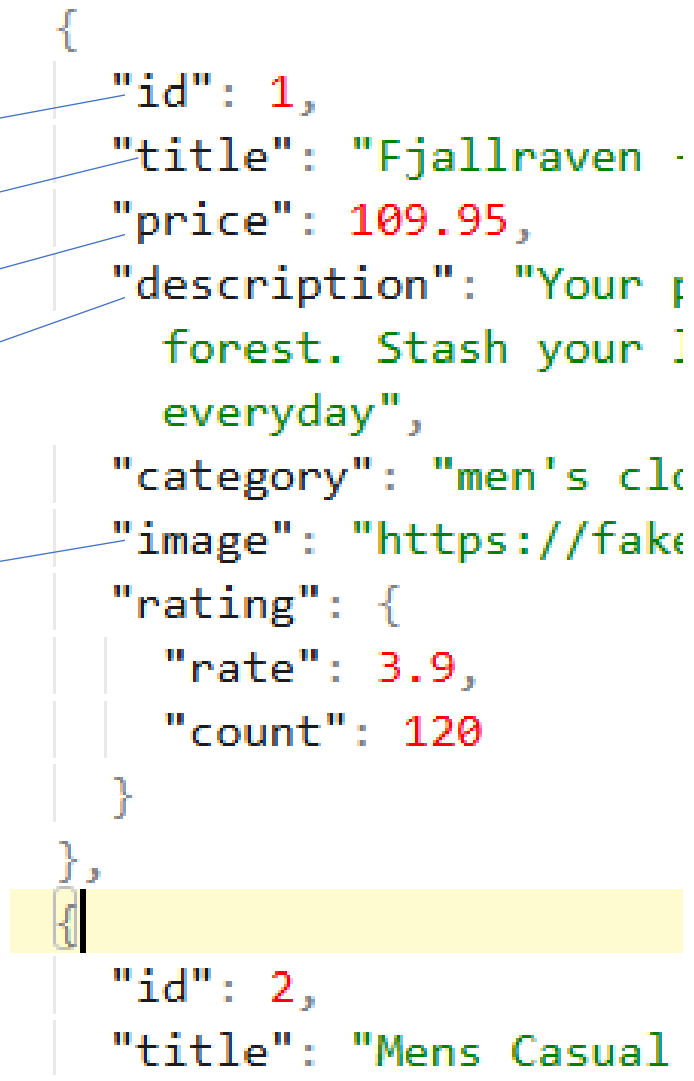
```
dependencies {

    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.8.7")
    implementation("com.squareup.retrofit2:retrofit:2.11.0")
    implementation("com.squareup.retrofit2:converter-gson:2.11.0")
    implementation("io.coil-kt.coil3:coil-compose:3.1.0")
    implementation("io.coil-kt.coil3:coil-network-okhttp:3.1.0")
    implementation(libs.androidx.core.ktx)
```

https://coil-kt.github.io/coil/

Create the Model for Product

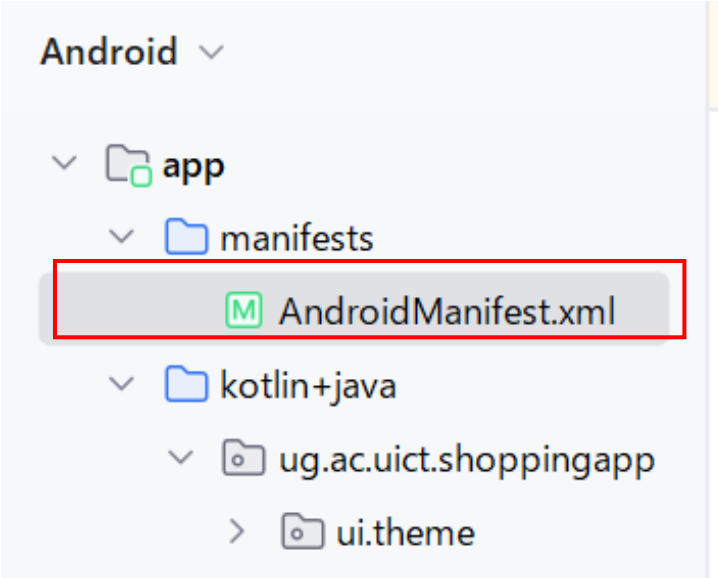
```
data class Product(
    val id: Int,
    val title: String,
    val price: Double,
    val description: String,
    val image: String
)
```



https://fakestoreapi.com/products

Add the Internet Permission

- Open the AndroidManifest.xml file from manifests folder, and add the <user-permission> for network access




```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
```

Allows your app to open network sockets and connect to the internet.

Lets your app check the network's state and connectivity status.

Write an Interface for access the Internet resource

- Create a new Kotlin file, named ApiService.kt

Creates a builder instance to configure Retrofit.

```
private val retrofit = Retrofit.Builder().baseUrl("https://fakestoreapi.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()

val shoppingService = retrofit.create(ApiService::class.java)

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

Adds a converter factory for serializing and deserializing JSON data. Gson converts JSON responses into Kotlin/Java objects.

Defining the API Service interface

This tells Retrofit to generate an implementation of the ApiService interface. we can use to call the API endpoints.

Declaring the method as suspend means it can be called within a Kotlin coroutine, allowing for asynchronous execution without blocking the main thread.

Create the MainViewModel

- Create a new Kotlin class, named MainViewModel.kt

Step1. prepare a data class to store the data from API

```
data class ProductState(
    val loading:Boolean = true,
    val list: List<Product> = emptyList(),
    val error: String? = null
)
```

Step2. prepare _productsState to hold the mutable state of ProductState(), and the public read-only productsState expose the state to UI.

```
private val _productsState =
mutableStateOf(ProductState())
val productState: State<ProductState>
= _productsState
```

Step3. write a fetchProducts method to access to API resource and store the response to _productsState

```
private fun fetchProducts(){
    viewModelScope.launch {
        try{
            val response = shoppingService.getProducts()
            _productsState.value = _productsState.value.copy(
                list = response,
                loading = false,
                error = null
            )
        }catch (e:Exception){
            _productsState.value = _productsState.value.copy(
                loading = false,
                error = "Error fetching Products ${e.message}"
```

Step4. execute the fetchProducts() to fetch the data

```
init {
    fetchProducts()
}
```

Create a view to show the products

- Create a new Kotlin file, named ShoppingtScreen.kt

```
@Composable
fun ShoppingScreen(modifier: Modifier = Modifier){
    val productViewModel: MainViewModel = viewModel()
    val viewstate by productViewModel.productState
    Box(modifier = modifier.fillMaxSize()){
        when{
            viewstate.loading -> {
                CircularProgressIndicator(modifier.align(Alignment.Center))
            }
            viewstate.error != null ->{
                Text("Error occurred")
            }
            else ->{
             // show the products at here
            }
```

Create a view for each product item in shopping view

```
@Composable
fun ProductItem(product: Product){
    Column(modifier = Modifier.padding(8.dp).fillMaxSize(),
        horizontalAlignment =  Alignment.CenterHorizontally)
    {
        Image(
            painter = rememberAsyncImagePainter(product.image),
            contentDescription = "the picture of our products",
            modifier = Modifier.fillMaxSize().aspectRatio(1f)
        )

        Text(
            text = product.title,
            color = Color.Black,
            style = TextStyle(fontWeight = FontWeight.Bold),
            modifier = Modifier.padding(top=4.dp)
        )
    }
}
```

The image of product

The title of product

Create a view of products contain all the product items

A layout that efficiently displays a grid of items, only composing and laying out visible items, which is ideal for long lists or grids.

Configures the grid to have a fixed number of columns

```
@Composable
fun ProductScreen(products: List<Product>){
    LazyVerticalGrid(GridCells.Fixed(2), modifier = Modifier.fillMaxSize()) {
        items(products){
            product ->
            ProductItem(product = product)
        }
    }
}
```

Iterates over the products list. For each Product in the list, the lambda function is executed.

Each element in the list is referred to as product.

For every product in the list, a ProductItem composable is called. This composable is responsible for rendering the individual product's UI

Add the ProductScreen view to ShoppingScreen view

```
fun ShoppingScreen(modifier: Modifier = Modifier){
    val productViewModel: MainViewModel = viewModel()
    val viewstate by productViewModel.productState
    Box(modifier = modifier.fillMaxSize()){
        when{
            viewstate.loading -> {
                CircularProgressIndicator(modifier.align(Alignment.Center))
            }
            viewstate.error != null ->{
                Text(viewstate.error.toString())
            }
            else ->{
                ProductScreen(products = viewstate.list)
            }
        }
```

Invoke the ProductScreen composable passing the list of producs from the state to display them.

Add the ShoppingScreen view to MainActivity

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            ShoppingAppTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    ShoppingScreen(
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
```