TÉCNICO LISBOA

Lab 2: Jetpack Compose + Lifecycle

CMU 2024/25

Learning objectives

In this lab you will:

- Understand Jetpack Compose fundamentals
- Build UI components with Compose
- Implement navigation in Compose applications
- Apply the MVVM architecture pattern
- Integrate ViewModel for state management
- Work with lifecycle-aware components

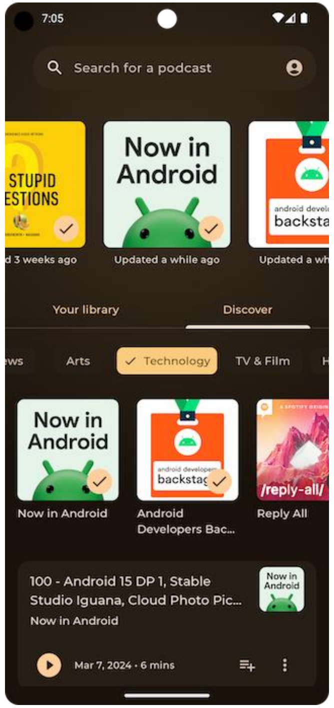
TÉCNICO LISBOA

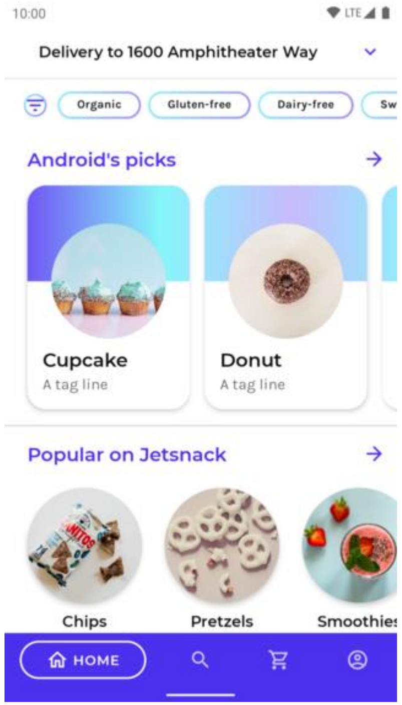
Part 1

Jetpack Compose Basics

What is Jetpack Compose?

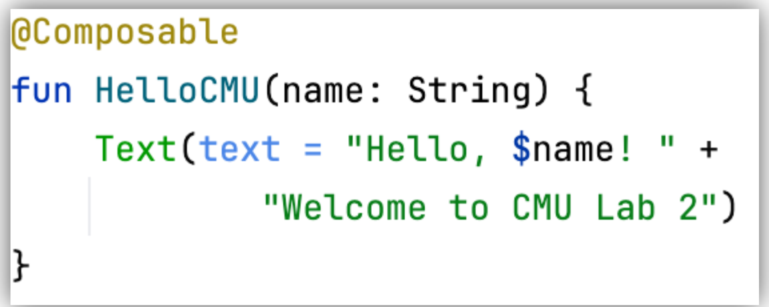
- Modern, declarative UI toolkit for Android
- Uses Kotlin to define UI components
- No more XML layouts
- Simpler, more intuitive UI development





Key concepts in Compose

- Composable functions
- Annotated with @Composable
- Define UI components
- Can contain other composables (nesting)
- Execute whenever data changes



```kotlin
@Composable
fun HelloCMU(name: String) {
    Text(text = "Hello, $name! " +
            "Welcome to CMU Lab 2")
}
```

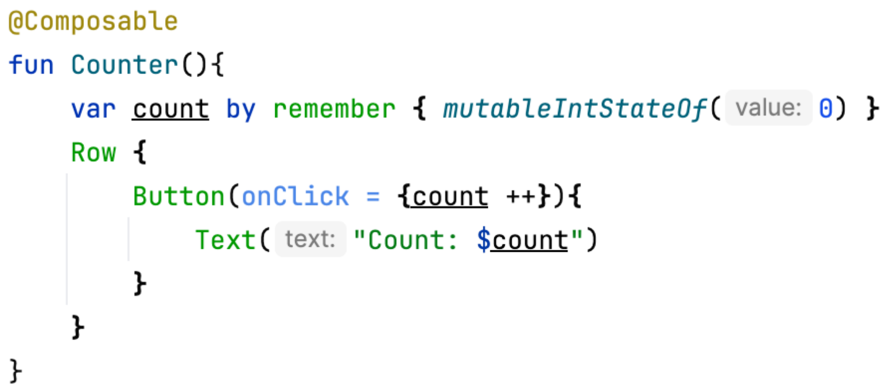
Composable properties

- Stateless vs. Stateful
  - Stateless: Display only, no internal data
  - Stateful: Maintain and update internal data

- Recomposition
  - UI updates when data changes
  - Intelligent redrawing of only affected components

State in compose

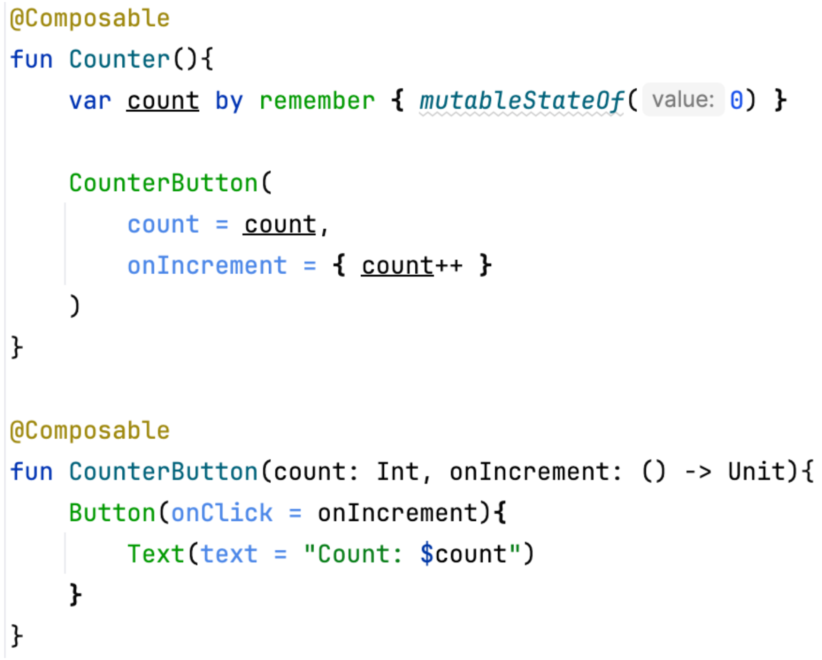
- remember
- Preserves state during recomposition
- Local to a composable



```
@Composable
fun Counter(){
    var count by remember { mutableIntStateOf( value: 0) }
    Row {
        Button(onClick = {count ++}){
            Text( text: "Count: $count")
        }
    }
}
```

State in Compose

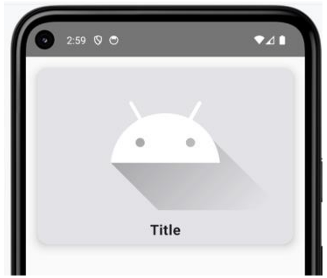
- Sate Hoisting
  - Lifting state to parent composables
  - Makes components more reusable
  - Enables shared state between components



```kotlin
@Composable
fun Counter(){
    var count by remember { mutableStateOf( value: 0) }

    CounterButton(
        count = count,
        onIncrement = { count++ }
    )
}

@Composable
fun CounterButton(count: Int, onIncrement: () -> Unit){
    Button(onClick = onIncrement){
        Text(text = "Count: $count")
    }
}
```

Common compose layouts

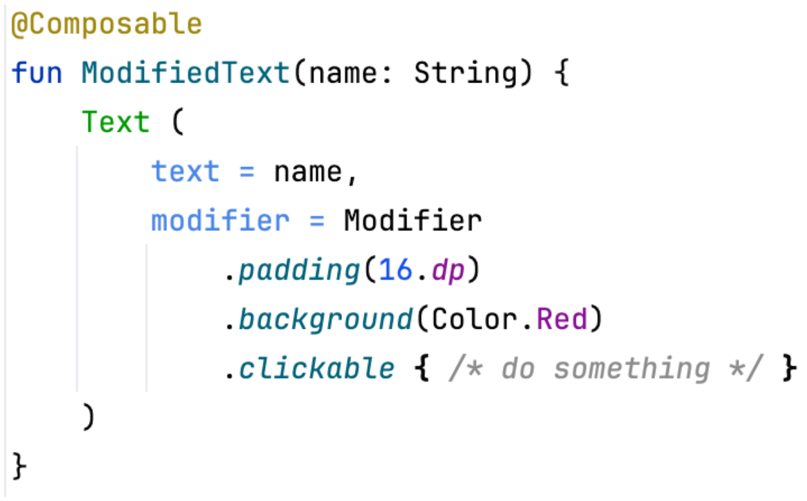
- Card
  - Display elements in a box
- Row
  - Horizontal arrangement
- Column
  - Vertical arrangement



```kotlin
@Composable
fun CardWithImageAndText(imageResId: Int, title: String) {
    Card(
        shape = RoundedCornerShape(16.dp),
        elevation = CardDefaults.cardElevation(defaultElevation = 8.dp),
        modifier = Modifier
            .padding(16.dp)
            .fillMaxWidth()
    ) {
        Column (
            modifier = Modifier.fillMaxWidth(),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Image(
                painter = painterResource(id = imageResId),
                contentDescription = null,
                contentScale = ContentScale.Crop,
                modifier = Modifier
                    .fillMaxWidth()
                    .height(200.dp)
            )
            Spacer(modifier = Modifier.height(8.dp))
            Text(
                text = title,
                fontSize = 20.sp,
                fontWeight = FontWeight.Bold,
                modifier = Modifier.padding(8.dp)
            )
        }
    }
}
```

Modifiers

- Control layout, appearance and behavior
- Can be chained for combined effects
- Apply to any composable



```
@Composable
fun ModifiedText(name: String) {
    Text (
        text = name,
        modifier = Modifier
            .padding(16.dp)
            .background(Color.Red)
            .clickable { /* do something */ }
    )
}
```

Part 2

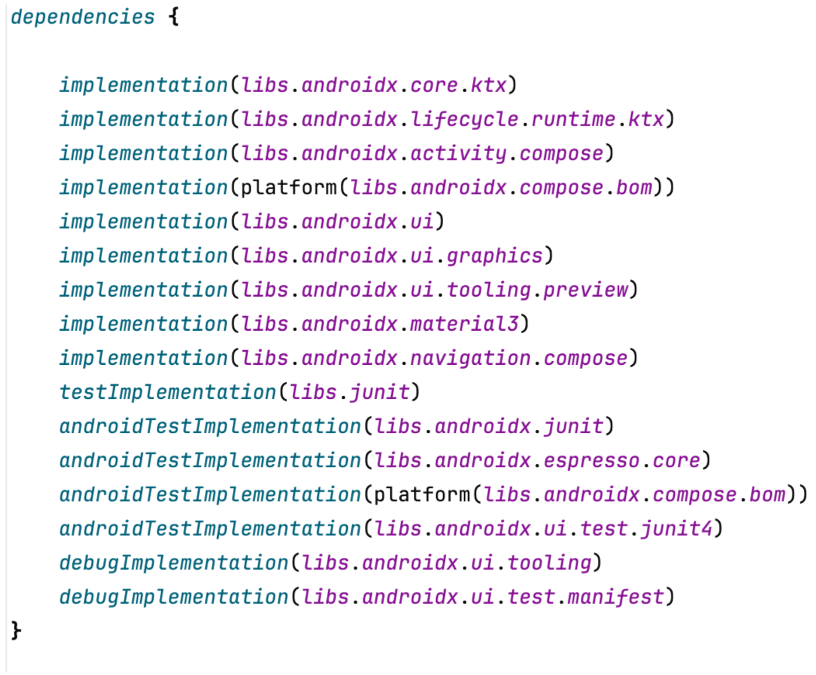
Navigation with compose

Navigation compose

- Purpose-built for Jetpack Compose
- Declarative Navigation System

Setting up navigation

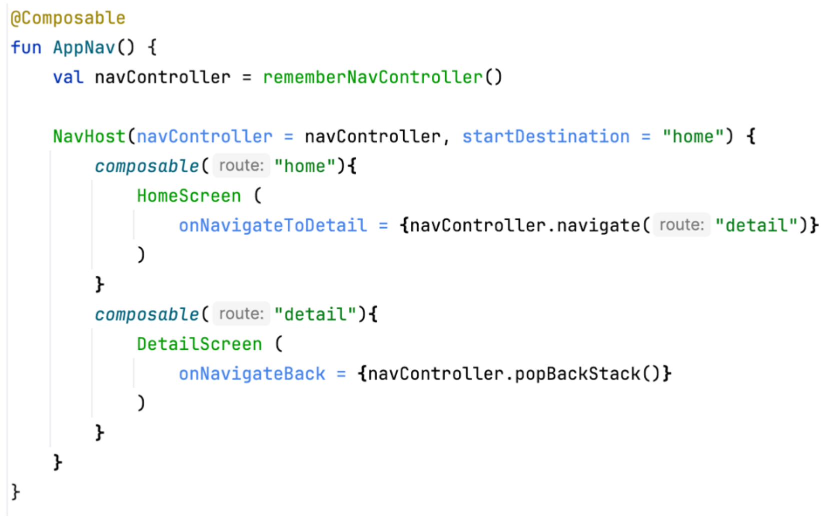
- Add dependencies
  - build.gradle file



```
dependencies {

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
    implementation(libs.androidx.activity.compose)
    implementation(platform(libs.androidx.compose.bom))
    implementation(libs.androidx.ui)
    implementation(libs.androidx.ui.graphics)
    implementation(libs.androidx.ui.tooling.preview)
    implementation(libs.androidx.material3)
    implementation(libs.androidx.navigation.compose)
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
    androidTestImplementation(platform(libs.androidx.compose.bom))
    androidTestImplementation(libs.androidx.ui.test.junit4)
    debugImplementation(libs.androidx.ui.tooling)
    debugImplementation(libs.androidx.ui.test.manifest)
}
```

Setting up navigation

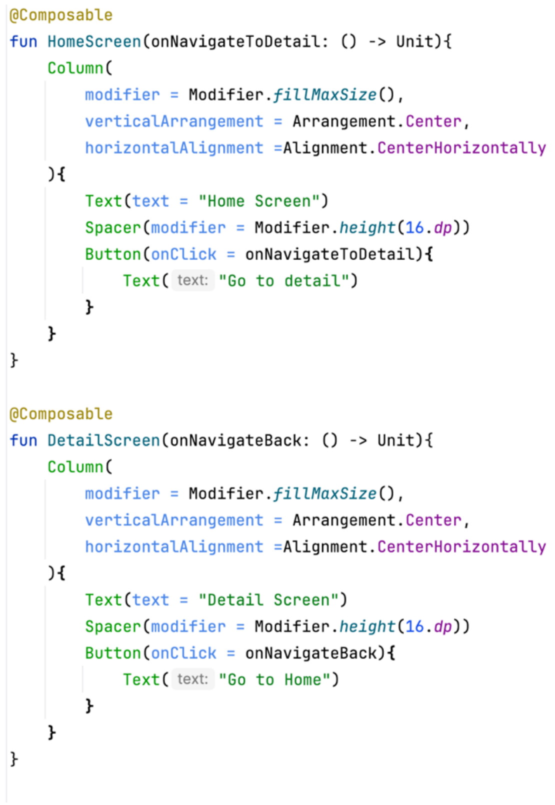
- Define navigation controller
  - navigate: goes to the designated screen
  - popBackStack: returns to the previous screen



```
@Composable
fun AppNav() {
    val navController = rememberNavController()

    NavHost(navController = navController, startDestination = "home") {
        composable( route: "home"){
            HomeScreen (
                onNavigateToDetail = {navController.navigate( route: "detail")}
            )
        }
        composable( route: "detail"){
            DetailScreen (
                onNavigateBack = {navController.popBackStack()}
            )
        }
    }
}
```

Setting up navigation

- Define the screens:
  - Home screen (main screen)
  - Detail screen (secondary screen triggered by the Home Screen)



```kotlin
@Composable
fun HomeScreen(onNavigateToDetail: () -> Unit){
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment =Alignment.CenterHorizontally
    ){
        Text(text = "Home Screen")
        Spacer(modifier = Modifier.height(16.dp))
        Button(onClick = onNavigateToDetail){
            Text( text: "Go to detail")
        }
    }
}

@Composable
fun DetailScreen(onNavigateBack: () -> Unit){
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment =Alignment.CenterHorizontally
    ){
        Text(text = "Detail Screen")
        Spacer(modifier = Modifier.height(16.dp))
        Button(onClick = onNavigateBack){
            Text( text: "Go to Home")
        }
    }
}
```

TÉCNICO LISBOA

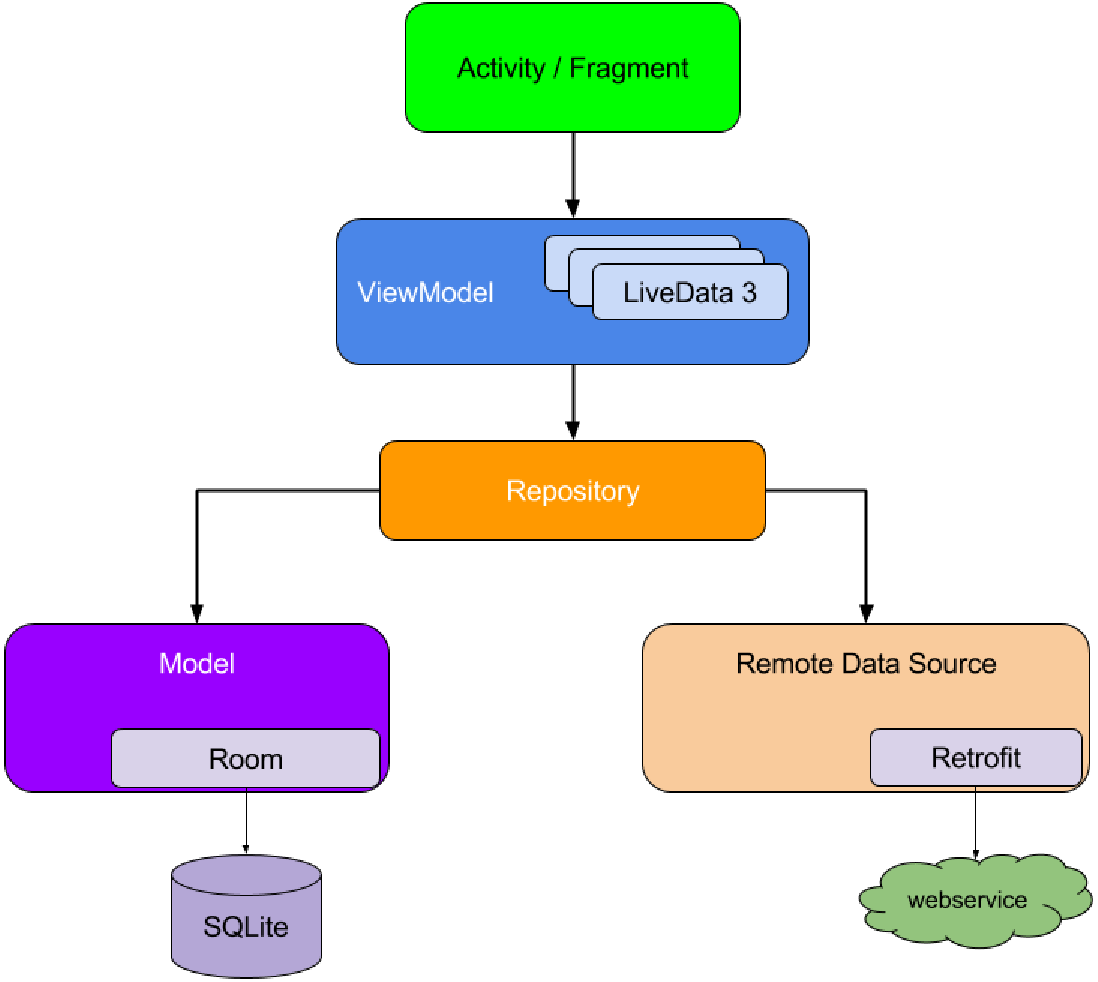
Part 3

The MVVM Architecture Pattern

What is MVVM

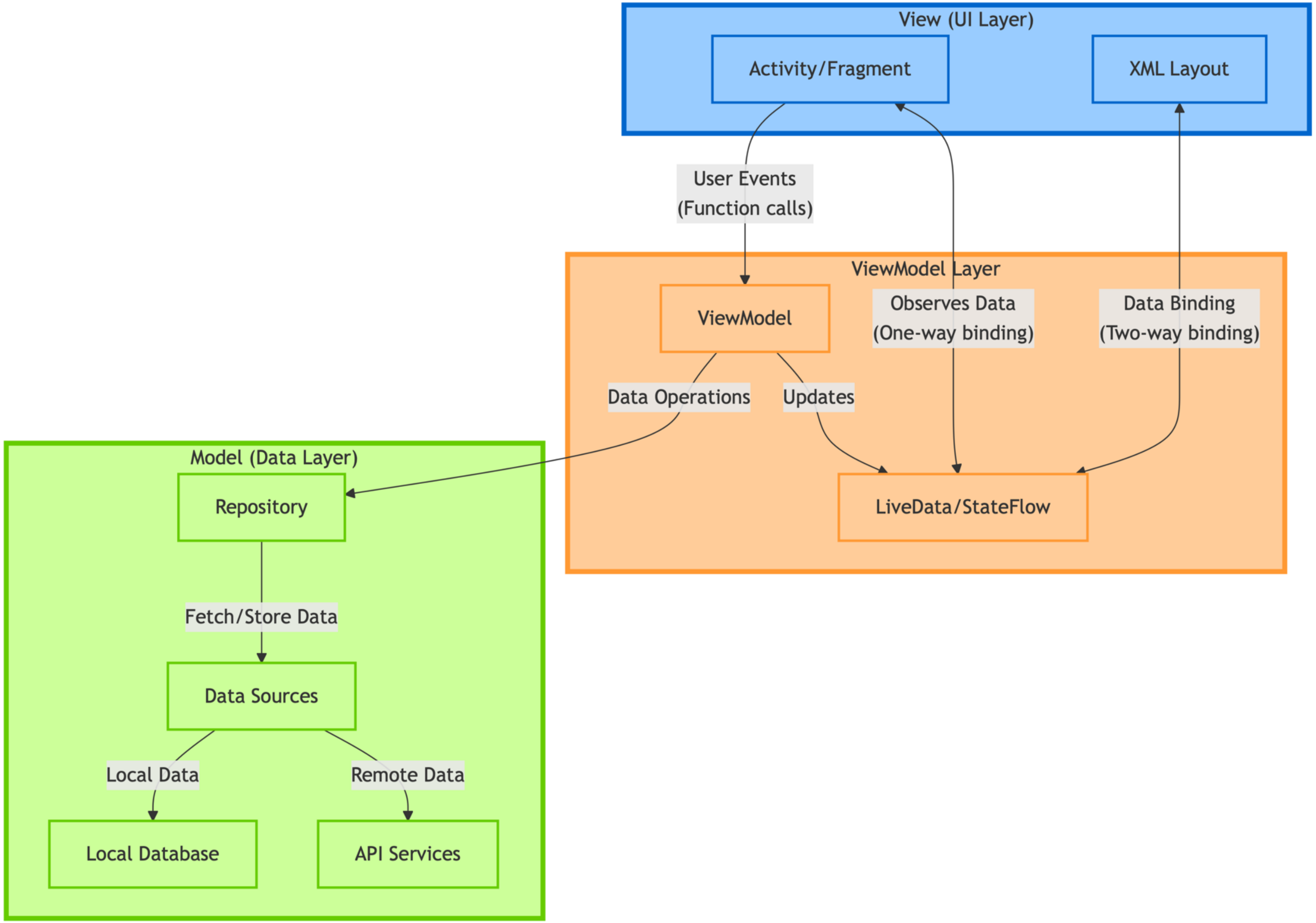
- Model-View-ViewModel architecture pattern
- Recommended by Google for Android app development
- Separates UI from business logic
- Key components:
  - **Model**: Data and business logic
  - **View**: UI elements (Activities/Fragments)
  - **ViewModel**: Bridge between Model and View

- Benefits
  - Better separation of concerns
  - Improved testability
  - UI components are less error-prone
  - Maintainable codebase



From: https://developer.android.com/topic/libraries/architecture/images/final-architecture.png

MVVI


View (UI Layer)
Activity/Fragment
XML Layout
User Events
(Function calls)
ViewModel Layer
ViewModel
Observes Data
(One-way binding)
Data Binding
(Two-way binding)
Data Operations
Updates
LiveData/StateFlow
Model (Data Layer)
Repository
Fetch/Store Data
Data Sources
Local Data
Remote Data
Local Database
API Services

The Model Layer

- **Responsibility**
    - Represents business logic and data
    - Manages data validation, storage, and retrieval
    - Communicates with data sources (local/remote)

Example data class

```kotlin
data class User(
    val id: String,
    val name: String,
    val email: String,
    val profileUrl: String? = null
)
```

Example repository

```kotlin
class UserRepository(
    private val remoteDataSource: UserRemoteDataSource,
    private val localDataSource: UserLocalDataSource
) {
    suspend fun getUser(userId: String): User {
        // Check if user exists in local storage
        localDataSource.getUser(userId)?.let {
            return it
        }

        // Fetch from remote source if not found locally
        val remoteUser = remoteDataSource.fetchUser(userId)

        // Cache the fetched user locally
        localDataSource.saveUser(remoteUser)

        return remoteUser
    }
}
```

The ViewModel Layer

- **Responsibility**
  - Acts as a bridge between the View and Model
  - Prepares and manages UI-related data
  - Survives configuration changes
  - Handles user interactions from the View

```kotlin
class UserProfileViewModel(
    private val userRepository: UserRepository
) : ViewModel() {

    // LiveData to observe user details
    private val _userData = MutableLiveData<User>()
    val userData: LiveData<User> = _userData

    // Loading state
    private val _isLoading = MutableLiveData<Boolean>()
    val isLoading: LiveData<Boolean> = _isLoading

    // Error handling
    private val _error = MutableLiveData<String?>()
    val error: LiveData<String?> = _error

    fun loadUserProfile(userId: String) {
        viewModelScope.launch {
            try {
                _isLoading.value = true
                _error.value = null

                val user = userRepository.getUser(userId)
                _userData.value = user

            } catch (e: Exception) {
                _error.value = "Failed to load user profile: ${e.message}"
            } finally {
                _isLoading.value = false
            }
        }
    }
}
```

The View Layer

- **Responsibility**
  - Renders UI components
  - Observes ViewModel data
  - Forwards user interactions to the ViewModel
  - No business logic

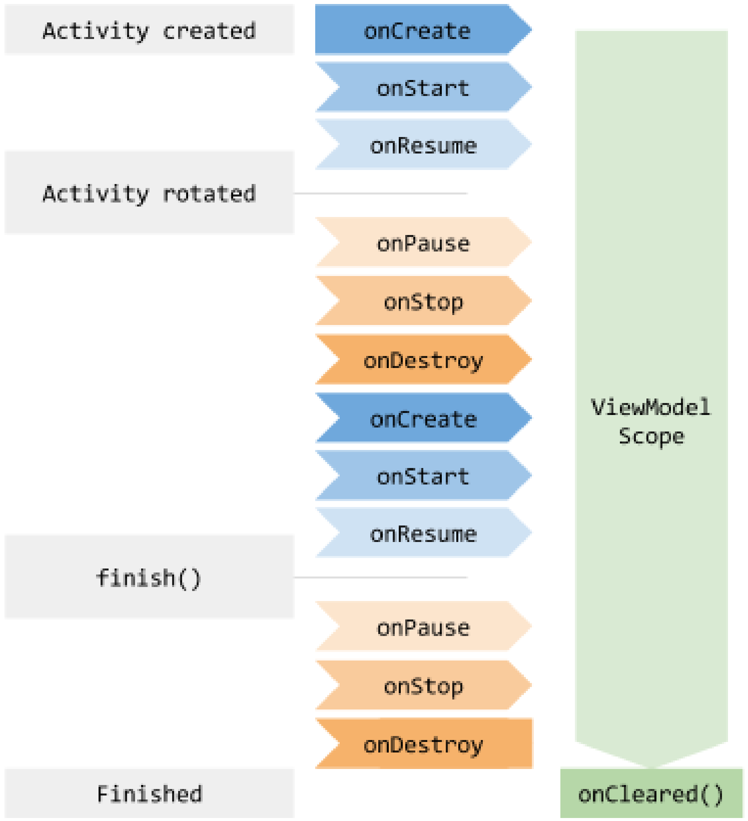
```kotlin
class UserProfileActivity : ComponentActivity() {

    private val viewModel: UserProfileViewModel by viewModels {
        UserProfileViewModelFactory(UserRepository(remoteDataSource, localDataSou
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Get user ID from intent
        val userId = intent.getStringExtra("USER_ID") ?: return

        // Load user profile
        viewModel.loadUserProfile(userId)

        // Set content with Compose
        setContent {
            MyAppTheme {
                UserProfileScreen(viewModel)
            }
        }
    }
```

```kotlin
@Composable
fun UserProfileScreen(viewModel: UserProfileViewModel) {
    // Observe LiveData values as State
    val userData by viewModel.userData.observeAsState()
    val isLoading by viewModel.isLoading.observeAsState(initial = false)
    val error by viewModel.error.observeAsState()

    // UI Layout with Compose
    Box(modifier = Modifier.fillMaxSize()) {
        // Error handling with SnackBar
        error?.let { errorMessage ->
            val snackbarHostState = remember { SnackbarHostState() }
            LaunchedEffect(errorMessage) {
                snackbarHostState.showSnackbar(message = errorMessage)
            }
            SnackbarHost(
                hostState = snackbarHostState,
                modifier = Modifier.align(Alignment.BottomCenter)
            )
        }

        // Loading indicator
        if (isLoading) {
            CircularProgressIndicator(
                modifier = Modifier.align(Alignment.Center)
            )
        }

        // User profile content
        userData?.let { user ->
            Column(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(16.dp),
                horizontalAlignment = Alignment.CenterHorizontally
            ) {
                // Profile image
                AsyncImage(
                    model = ImageRequest.Builder(LocalContext.current)
                        .data(user.profileUrl)
```

Lifecycle of a viewmodel

The lifecycle of a ViewModel is tied directly to its **scope**. A ViewModel remains in memory until:

- In the case of an activity, when it finishes.
- In the case of a fragment, when it detaches.
- In the case of a Navigation entry, when it's removed from the back stack.



TÉCNICO LISBOA