Kotlin Jetpack Compose

FPT Education
FPT POLYTECHNIC

Navigation và State

- Điều hướng giữa các màn hình với Navigation
- Quản lý trạng thái giao diện với State
- MVVM, ViewModel

Nội dung

- Điều hướng giữa các màn hình với Navigation
- Quản lý trạng thái giao diện với State
- MVVM, ViewModel

State

Trong Jetpack Compose, “state” là bất kỳ giá trị nào có thể thay đổi theo thời gian và ảnh hưởng đến UI.

Để quản lý state, Jetpack Compose cung cấp một số API và kỹ thuật như:

mutableStateOf: Tạo một đối tượng MutableState quan sát được, cho phép UI cập nhật tự động khi dữ liệu thay đổi.

remember: Lưu trữ một đối tượng trong Composition và giữ giá trị qua các quá trình recomposition.

State<T>: Một API giúp Compose tự động theo dõi state.

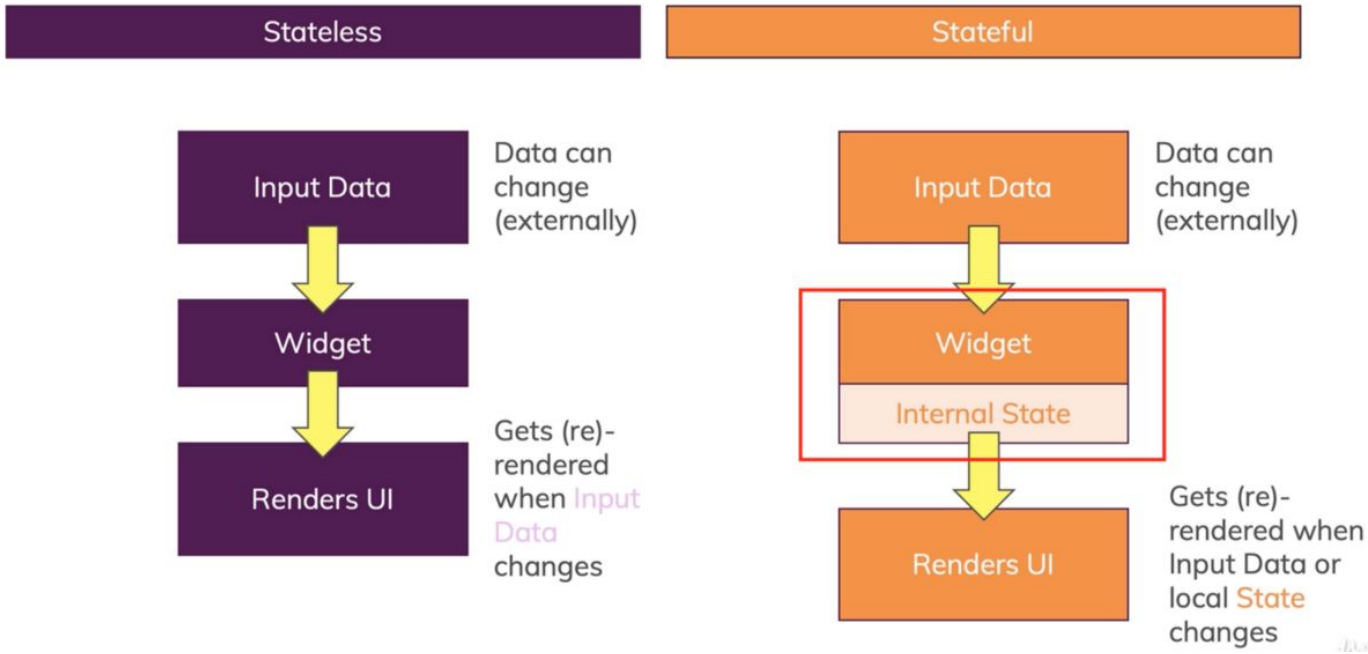
State Hoisting: Kỹ thuật nâng cao trạng thái lên một cấp độ cao hơn trong cây UI để làm cho composable trở nên stateless

Stateful vs Stateless

Các composable có thể là stateful (có trạng thái) hoặc stateless (không có trạng thái).

Stateful composable quản lý trạng thái bên trong, trong khi stateless composable nhận trạng thái từ bên ngoài

Stateful vs Stateless



State<T> và MutableState<T>

State<T> là một kiểu dữ liệu giữ giá trị chỉ đọc và thông báo cho composition khi giá trị thay đổi.

MutableState<T> là một phần mở rộng của State, cho phép cập nhật giá trị. Khi thuộc tính value được ghi vào và thay đổi, một recomposition của bất kỳ RecomposeScopes nào đăng ký sẽ được lên lịch (scheduled)

Ví dụ:

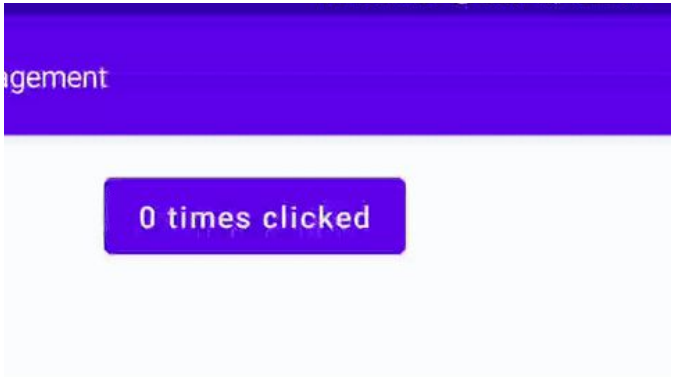
```
var selectedIndex by mutableStateOf(0)

//selectedIndex = 5
```

No State

Nếu chúng ta tạo một object trong có state, nó sẽ không cập nhật giao diện UI khi dữ liệu thay đổi

```
@Composable
fun NoState() {
    var clickCount = 0
    Column {
        Button(onClick = {
            clickCount++
            Log.d("TAG", "NoState: "+clickCount)
        }) {
            Text(text = ""+clickCount+" times clicked")
        }
    }
}
```

No State



Khi chúng ta nhấn nút gọi sự kiện onClick, chúng ta sẽ không thấy giao diện cập nhật số lần click. Nhưng nếu xem trong logcat, thì giá trị có thay đổi sau mỗi lần click

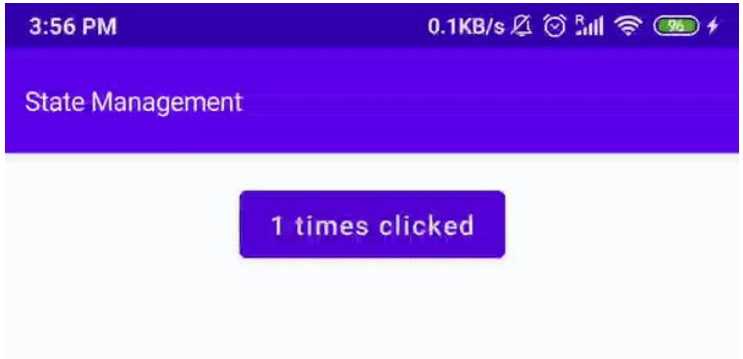
No State

Logcat

```
2024-04-10 09:35:26.480 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 1
2024-04-10 09:35:27.184 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 2
2024-04-10 09:35:27.384 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 3
2024-04-10 09:35:27.568 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 4
2024-04-10 09:35:27.736 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 5
2024-04-10 09:35:27.902 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 6
2024-04-10 09:35:28.210 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 7
2024-04-10 09:35:28.410 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 8
2024-04-10 09:35:28.478 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 9
2024-04-10 09:35:28.643 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 10
2024-04-10 09:35:28.817 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 11
2024-04-10 09:35:28.968 18236-18236 TAG                 com.dinhnt.lab01                D  NoState: 12
```

With State Object

```
@Composable
fun MutableStateClick() {
    var clickCount by mutableStateOf(0)//Not recommended
    Column {
        Button(onClick = { clickCount++ }) {
            Text(text = "" + clickCount + " times clicked")
        }
    }
}
```



With State Object

Nó hoạt động như mong đợi. Tuy nhiên, nó có một số vấn đề, nếu bạn sử dụng state object với child composable thì nó sẽ không hoạt động. Khi bạn sử dụng đoạn code này, Android studio sẽ đưa ra cảnh báo và khuyên bạn nên sử dụng đoạn code này cùng với remember.

Remember

Trong Jetpack Compose, **remember** là một API giúp lưu trữ các đối tượng trong bộ nhớ trong quá trình recomposition của một hàm composable. Nó được sử dụng để giữ giá trị của trạng thái có thể thay đổi mà không cần quản lý trạng thái một cách thủ công, remember có thể lưu trữ cả đối tượng bất biến và đối tượng có thể thay đổi

Remember

Trong Jetpack Compose, **remember** là một API giúp lưu trữ các đối tượng trong bộ nhớ trong quá trình recomposition của một hàm composable. Nó được sử dụng để giữ giá trị của trạng thái có thể thay đổi mà không cần quản lý trạng thái một cách thủ công, remember có thể lưu trữ cả đối tượng bất biến và đối tượng có thể thay đổi

Remember

Cú pháp:

```
val currentValue = remember { mutableStateOf(0) } //Int

val userName = remember { mutableStateOf("") } //String
```

Remember

```
@Composable
fun RememberSample() {
    var clickCount by remember { mutableStateOf(0) }
    Column {
        Button(onClick = { clickCount++ }) {
            Text(text = "" + clickCount + " times clicked")
        }
    }
}
```

Remember

Lưu ý:

Nếu bạn xoay thiết bị (từ dọc thành ngang, hoặc ngược lại), giá trị sẽ được đặt lại.

Nếu bạn muốn giữ lại dữ liệu ngay cả khi hoạt động được tạo lại/thay đổi hướng (xoay thiết bị) xảy ra, hãy sử dụng "**rememberSavaable**".

Remember Saveable

Trong Jetpack Compose, **rememberSaveable** là một phiên bản nâng cao của remember và được sử dụng để lưu trữ trạng thái qua các thay đổi cấu hình như xoay màn hình hoặc thay đổi ngôn ngữ hệ thống. Nó hoạt động bằng cách lưu trạng thái của một hàm composable vào một đối tượng Bundle, sau đó có thể được khôi phục lại khi hàm được gọi lại

Remember Saveable

```
@Composable
fun RememberSaveableSample() {
    var clickCount = rememberSaveable { mutableStateOf(0) }
    Column {
        Button(onClick = { clickCount.value++ }) {
            Text(text = "" + clickCount.value + " times clicked")
        }
    }
}
```

Remember Saveable

Navigaiton

Trong Jetpack Compose, hệ thống navigation cho phép bạn di chuyển giữa các composable trong ứng dụng của mình. Để sử dụng navigation, bạn cần thiết lập một **NavController**, tạo một **NavHost**, và định nghĩa các đường dẫn để di chuyển giữa các composable

NavController

NavController là API trung tâm cho thành phần Điều hướng. API này có trạng thái và theo dõi ngăn xếp lui của những thành phần kết hợp tạo nên các màn hình trong ứng dụng cũng như trạng thái của từng màn hình.

Bạn có thể tạo NavController bằng cách sử dụng phương thức **rememberNavController()** trong thành phần kết hợp:

```
val navController = rememberNavController()
```

NavController

Bạn nên tạo NavController ở vị trí trong hệ phân cấp thành phần kết hợp của mình. Tại đây, mọi thành phần kết hợp cần tham chiếu đến NavController đều có quyền truy cập vào NavController. Điều này tuân theo các nguyên tắc chuyển trạng thái lên trên (state hoisting) và cho phép bạn sử dụng NavController cũng như trạng thái mà nó cung cấp thông qua **currentBackStackEntryAsState**() để dùng làm nguồn xác thực cập nhật các thành phần kết hợp bên ngoài màn hình

NavHost

Mỗi NavController phải được liên kết với một thành phần kết hợp NavHost duy nhất. NavHost liên kết với NavController bằng một biểu đồ điều hướng, có tác dụng chỉ định các đích đến thành phần kết hợp mà bạn có thể điều hướng giữa chúng. Khi bạn điều hướng giữa các thành phần kết hợp, nội dung của NavHost sẽ tự động kết hợp lại. Mỗi thành phần kết hợp đóng vai trò điểm đến trong biểu đồ điều hướng của bạn được liên kết với một tuyến (route).

NavHost

Việc tạo NavHost cần phải có NavController được tạo trước đó thông qua rememberNavController() và tuyến của đích đến bắt đầu trong biểu đồ. Việc tạo NavHost sử dụng cú pháp lambda của DSL Kotlin điều hướng để tạo biểu đồ điều hướng. Bạn có thể thêm vào cấu trúc điều hướng bằng cách sử dụng phương thức composable(). Phương thức này yêu cầu bạn cung cấp tuyến và thành phần kết hợp phải được liên kết với đích đến

NavHost

```
NavHost(navController = navController, startDestination = "profile") {
    composable("profile") { Profile(/*...*/) }
    composable("friendslist") { FriendsList(/*...*/) }
    /*...*/
}
```

Điều hướng

Để điều hướng đến thành phần kết hợp đóng vai trò điểm đến trong biểu đồ điều hướng, bạn phải sử dụng phương thức navigate. navigate lấy một tham số String duy nhất đại diện cho tuyến của điểm đến. Để điều hướng từ một thành phần kết hợp trong biểu đồ điều hướng, hãy gọi navigate:

```
navController.navigate("friendslist")
```

Điều hướng

Theo mặc định, navigate sẽ thêm điểm đến mới vào ngăn xếp lui. Bạn có thể sửa đổi hành vi của navigate bằng cách đính kèm các tuỳ chọn điều hướng bổ sung vào lệnh gọi navigate():

```
// Pop everything up to the "home" destination off the back stack before
// navigating to the "friendslist" destination
navController.navigate("friendslist") {
    popUpTo("home")
}
```

Điều hướng

```
// Pop everything up to and including the "home" destination off
// the back stack before navigating to the "friendslist" destination
navController.navigate("friendslist") {
    popUpTo("home") { inclusive = true }
}

// Navigate to the "search” destination only if we’re not already on
// the "search" destination, avoiding multiple copies on the top of the
// back stack
navController.navigate("search") {
    launchSingleTop = true
}
```

Điều hướng bằng đối số

Thành phần Điều hướng trong Compose cũng hỗ trợ truyền đối số giữa các đích đến có khả năng kết hợp. Để thực hiện việc này, bạn cần thêm trình giữ chỗ đối số vào tuyến của mình, tương tự như cách bạn thêm đối số vào deep link khi sử dụng thư viện điều hướng cơ sở:

```
NavHost(startDestination = "profile/{userId}") {
    ...
    composable("profile/{userId}") {...}
}
```

Điều hướng bằng đối số

Theo mặc định, tất cả các đối số được phân tích cú pháp dưới dạng chuỗi. Tham số arguments của composable() chấp nhận danh sách NamedNavArgument. Bạn có thể nhanh chóng tạo NamedNavArgument bằng phương thức navArgument rồi chỉ định chính xác type:

```
NavHost(startDestination = "profile/{userId}") {
    ...
    composable(
        "profile/{userId}",
        arguments = listOf(navArgument("userId") { type = NavType.StringType
})
    ) {...}
}
```

Điều hướng bằng đối số

Theo mặc định, tất cả các đối số được phân tích cú pháp dưới dạng chuỗi. Tham số arguments của composable() chấp nhận danh sách NamedNavArgument. Bạn có thể nhanh chóng tạo NamedNavArgument bằng phương thức navArgument rồi chỉ định chính xác type:

```
NavHost(startDestination = "profile/{userId}") {
    ...
    composable(
        "profile/{userId}",
        arguments = listOf(navArgument("userId") { type = NavType.StringType
})
    ) {...}
}
```

Điều hướng bằng đối số

Bạn nên trích xuất đối số từ NavBackStackEntry có trong lambda của hàm composable().

```
composable("profile/{userId}") { backStackEntry →
    Profile(navController, backStackEntry.arguments?.getString("userId"))
}
```

Điều hướng bằng đối số

Để truyền đối số đến đích, bạn cần thêm đối số đó vào tuyến đường khi thực hiện lệnh gọi navigate:

```
navController.navigate("profile/user1234")
```

MVVM

MVVM (**Model-View-ViewModel**) là một mô hình kiến trúc phần mềm được sử dụng để tách biệt giao diện người dùng (UI) khỏi logic nghiệp vụ. Trong Jetpack Compose, ViewModel đóng vai trò quan trọng trong việc quản lý trạng thái UI và xử lý logic nghiệp vụ, giúp cho mã nguồn dễ dàng quản lý và bảo trì hơn

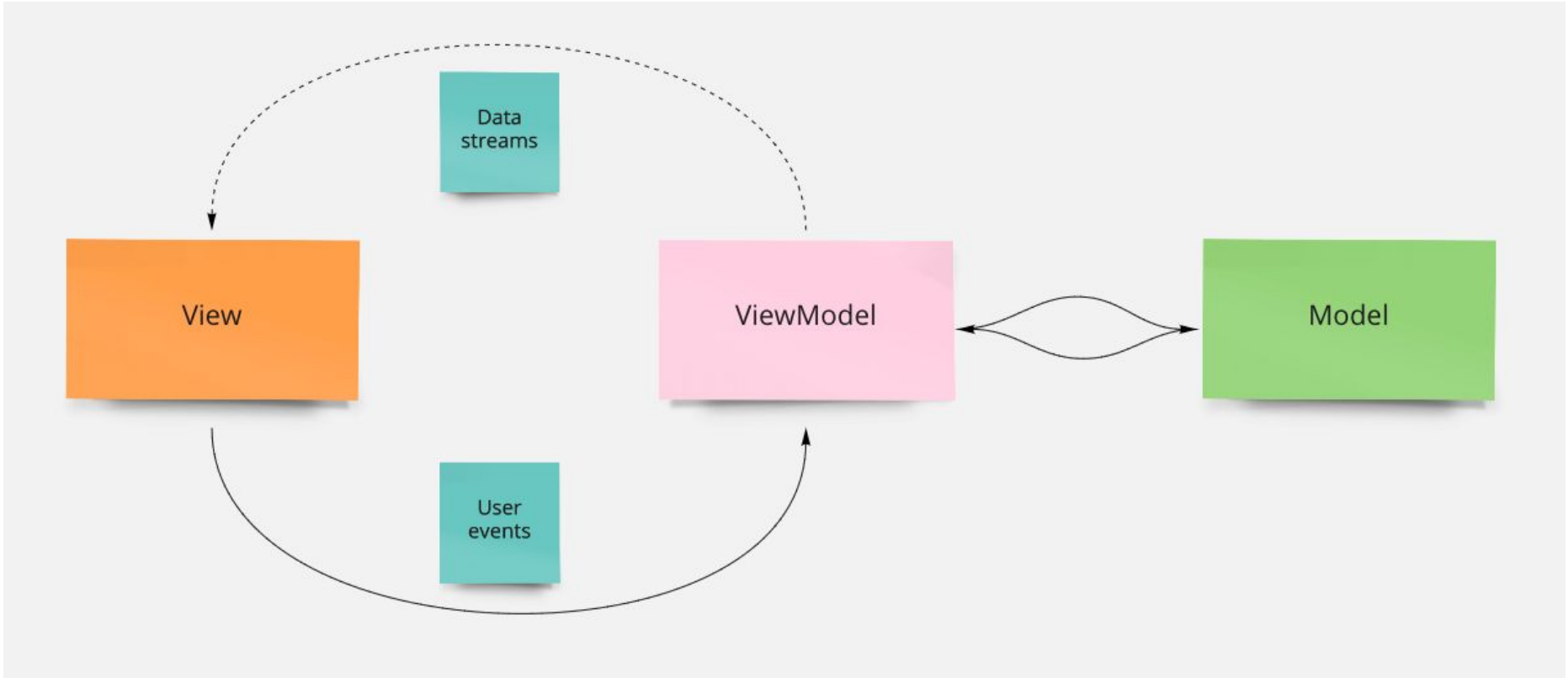
MVVM

Model: Đây là phần ‘não’ của ứng dụng, chịu trách nhiệm cho logic nghiệp vụ. Model xử lý các tác vụ như truy cập dữ liệu, xác thực, tính toán và cung cấp dữ liệu cho ViewModel.

View: Là phần hiển thị giao diện người dùng và thu thập input từ người dùng. Trong Jetpack Compose, View được xây dựng thông qua các hàm có khả năng kết hợp (@Composable functions) để định nghĩa UI.

ViewModel: Nằm giữa Model và View, ViewModel chứa logic trình bày và xử lý các sự kiện từ View để giao tiếp với Model. ViewModel cũng quản lý trạng thái UI và thực hiện các tác vụ như phản hồi lại các sự kiện người dùng hoặc xử lý các thay đổi trạng thái.

MVVM



MVVM kết hợp với Jetpack Compose

MVVM kết hợp cùng Jetpack Compose hoạt động như sau:

- **Model:** Model trong kiến trúc MVVM chứa dữ liệu và logic nghiệp vụ của ứng dụng. Bạn có thể đóng gói nguồn dữ liệu, kho lưu trữ, và các thực thể miền tại đây. Các thư viện Jetpack như Room và Retrofit có thể hỗ trợ triển khai lớp dữ liệu.
- **View:** Trong Jetpack Compose, lớp View là trái tim của giao diện người dùng. Bạn định nghĩa UI sử dụng các hàm có thể kết hợp, làm cho mã UI trở nên gọn gàng, dễ bảo trì và dễ đọc hơn. Lớp View trực tiếp quan sát dữ liệu từ ViewModel và hiển thị nó cho người dùng.

MVVM kết hợp với Jetpack Compose

ViewModel: ViewModel đóng vai trò trung gian giữa Model và View. Nó phơi bày dữ liệu cho View thông qua LiveData hoặc đối tượng State và xử lý tương tác người dùng. ViewModel cho phép View không biết gì về nguồn dữ liệu và giữ logic UI riêng biệt.

Data Binding: Jetpack Compose làm cho việc liên kết dữ liệu từ ViewModel trực tiếp đến các phần tử UI trở nên dễ dàng. Liên kết dữ liệu hai chiều này cho phép cập nhật tự động các thành phần UI khi dữ liệu trong ViewModel thay đổi.

Navigation: Navigation trong Jetpack Compose được xử lý thông qua thành phần Navigation, thân thiện với MVVM. ViewModel có thể kiểm soát navigation và chuyển dữ liệu giữa các điểm đến một cách liền mạch.

Lợi ích khi kết hợp MVVM với Jetpack Compose

- **Phân tách Mối quan tâm**: MVVM thực thi sự phân tách rõ ràng giữa dữ liệu, UI, và logic tương tác, làm cho mã nguồn trở nên có tổ chức và dễ bảo trì hơn.
- **Khả năng Kiểm thử**: Mỗi thành phần có thể được kiểm thử độc lập, cho phép kiểm thử đơn vị và UI toàn diện.
- **Tính Phản ứng**: Sự quan sát của ViewModel và sự tái cấu trúc của Jetpack Compose đảm bảo rằng giao diện người dùng của bạn luôn đồng bộ với dữ liệu.
- **Khả năng Mở rộng**: Khi ứng dụng của bạn phát triển, cấu trúc mô-đun của MVVM và các hàm có thể kết hợp của Jetpack Compose làm cho việc thêm và sửa đổi tính năng trở nên dễ dàng hơn.
- **Đọc Mã**: Mã UI khai báo trong Jetpack Compose rất dễ đọc, giúp cho việc hiểu và hợp tác trên các dự án trở nên dễ dàng hơn

Thanks!