Application Development II

420-5A6-AB

Instructor: Talib Hussain

Day 22:

Flows, ViewModels and DataStore: Persisting Data

StateFlow / MutableStateFlow

- StateFlow is a state-holder observable flow that emits the current and new state updates to its collectors. The current state value can also be read through its **value** property.
- To update state and send it to the flow, assign a new value to the value property of the MutableStateFlow class.

- https://farhan-tanvir.medium.com/stateflow-with-jetpack-compose-7d9c9711c286

- https://developer.android.com/jetpack/compose/state-saving
- https://proandroiddev.com/viewmodels-using-compose-mutablestateflows-or-mutablestates-64d34ba548c5

Don’t Forget

- Need this in your gradle in order to use ViewModels with Compose
    - implementation("androidx.lifecycle:lifecycle-viewmodel-compose:{latest_version}")

2. Multiple Flows

- We can define a second flow in our class, such as a counter.
- Add the following lines to your view model.

```
private val _counterFlow = MutableStateFlow<Int>(0)
val counterFlow: StateFlow<Int> get() = _counterFlow.asStateFlow()
fun incrementCounter() {
    _counterFlow.value = _counterFlow.value + 1
}
```

- Add the following lines to your composable.

```
val counter by myViewModel.counterFlow.collectAsState()
```

  - In an onClick:

```
myViewModel.incrementCounter()
```

- Run the program to make sure it works

3. Flow of objects

- In addition to storing individual value in a flow, we can store a more complex object, such as an instance of a data class, in a flow.
- Let's define a new data class called ProfileData

```
data class ProfileData(
    var name : String = "",
    var counter: Int = 0
)
```

- In our ViewModel, we can now store this as a flow instead of our previous two variables.

```
// private UI state (MutableStateFlow)
private val _uiState = MutableStateFlow(ProfileData())

// public getter for the state (StateFlow)
val uiState: StateFlow<ProfileData> = _uiState.asStateFlow()
```

Changing values

- A data class offers us a useful copy function that will return a copy with only the indicated fields changed.
    - E.g., oldObject.copy(name = newName) will return a new instance with the new name but an unchanged counter.
- We'd like to be able to write a simpler setter, such as:

```
fun setName(newName: String) {
    _uiState.value = _uiState.value.copy(name = newName)
  }
```

- However, there is an issue: Concurrency
- If another thread tries to update the StateFlow between the time copy function in the current thread completes and the StateFlow's new value is emitted, we could end up with results we were not expecting.
    - E.g., if the other thread updates counter (using copy) and the current thread is only updating name
- To avoid this, we can use the update() function
    - update() checks whether the previous value has changed, say by another thread, before setting a new value

```
_uiState.update { it.copy(name = newName) }
```

- https://proandroiddev.com/make-sure-to-update-your-stateflow-safely-in-kotlin-9ad023db12ba
    - Cached version: http://webcache.googleusercontent.com/search?q=cache:D6ZvHSHODUoJ:https://proandroiddev.com/make-sure-to-update-your-stateflow-safely-in-kotlin-9ad023db12ba&client=firefox-b-d&sca_esv=572463874&hl=en&gl=ca&strip=0&vwsrc=0

CoRoutineScope

- Note: Best to always use MutableStateFlow in a background thread to avoid blocking the main thread and causing UI freezes.
- You can use the viewModelScope or CoroutineScope to launch a flow in a background thread.

```
/* Increments the value of the counter stored in the state flow */
 fun increment() {
     viewModelScope.launch {
         _uiState.update { currentState ->
             currentState.copy(counter = currentState.counter + 1)
         }
     }
 }
```

- Or, even simpler:

```
uiState.update { it.copy(counter = it.counter + 1) }
```

- https://developermemos.com/posts/mutable-state-flow-android

Using state in Composable

- To access the state object, we simply use collectAsState like before. Except this time, the state value with be an instance of our data class.

```
@Composable
fun MySimpleScreen(myViewModel: MyViewModelSimple = viewModel()) {
    val myUiState by myViewModel.uiState.collectAsState()

    Column {
        Button(
            onClick = { myViewModel.setName(myUiState.name + "x") }
        ) {
            Text(
                text = "Append to Name",
                fontSize = 16.sp
            )
        }
        Button(
            onClick = { myViewModel.increment() }
        ) {
            Text(
                text = "Increment Counter",
                fontSize = 16.sp
            )
        }
        if (myUiState != null) {
            Text(text = "Name: ${myUiState.name}")
            Text(text = "Counter value ${myUiState.counter}")
        }
    }
}
```