- Retry Operator in Kotlin Flow
- Retrofit with Kotlin Flow
- Room Database with Kotlin Flow
- Kotlin Flow Zip Operator for Parallel Multiple Network Calls
- Instant Search Using Kotlin Flow Operators
- Exception Handling in Kotlin Flow
- callbackFlow - Callback to Flow API in Kotlin
- Unit Testing ViewModel with Kotlin Flow and StateFlow

Before jumping into the StateFlow and SharedFlow, we should have an understanding of the Cold Flow and Hot Flow. Refer to Cold Flow vs Hot Flow.

Let's start learning about the StateFlow and SharedFlow in Kotlin.

Both the StateFlow and SharedFlow are Hot Flows.

Let me tabulate the differences between StateFlow and SharedFlow for your better understanding so that you can decide which one to use based on your use case.

Also, we will learn all the below-mentioned points from the example code.

StateFlow vs SharedFlow

StateFlow	SharedFlow
Hot Flow	Hot Flow
Needs an initial value and emits it as soon as the collector starts collecting.	Does not need an initial value so does not emit any value by default.
`val stateFlow = MutableStateFlow(0)`	`val sharedFlow = MutableSharedFlow<Int>()`
Only emits the last known value.	Can be configured to emit many previous values using the replay operator.
It has the value property, we can check the current value. It keeps a history of one value that we can get directly without collecting.	It does not have the value property.
It does not emit consecutive repeated values. It emits the value only when it is distinct from the previous item.	It emits all the values and does not care about the distinct from the previous item. It emits consecutive repeated values also.
Similar to LiveData except for the Lifecycle awareness of the Android component. We should use repeatOnLifecycle scope with StateFlow to add the Lifecycle awareness to it, then it will become exactly like LiveData.	Not similar to LiveData.

Let's understand all of the above points from the example code.

StateFlow example

Suppose we have StateFlow as below:

```
val stateFlow = MutableStateFlow(0)
```

And, we start collecting on it:

```
stateFlow.collect {
    println(it)
}
```

As soon as we start collecting, we will get:

```
0
```

We get this "0" as it takes an initial value and emits it immediately.

And then, if we keep on setting the values as below:

```
stateFlow.value = 1
stateFlow.value = 2
stateFlow.value = 2
stateFlow.value = 1
stateFlow.value = 3
```

we will get the following output:

```
1
2
1
3
```

Notice here that we are getting "2" only once, not twice. As it does not emit consecutive repeated values.

Suppose we add a new collector now:

```
stateFlow.collect {
    println(it)
}
```

We will get:

```
3
```

As the StateFlow stores the last value and emits it as soon as a new collector starts collecting.

SharedFlow example

Suppose we have SharedFlow as below:

```
val sharedFlow = MutableSharedFlow<Int>()
```

And, we start collecting on it:

```
sharedFlow.collect {
    println(it)
}
```

As soon as we start collecting, we will not get anything as it does not take an initial value.

And then, if we keep on emitting the values as below:

```
sharedFlow.emit(1)
sharedFlow.emit(2)
sharedFlow.emit(2)
sharedFlow.emit(1)
sharedFlow.emit(3)
```

we will get the following output:

```
1
2
2
1
3
```

Notice here that we are getting "2" twice. As it emits consecutive repeated values also.

Suppose we add a new collector now:

```
sharedFlow.collect {
    println(it)
}
```

We will not get anything as the SharedFlow does not store the last value.

Now, that we have seen the examples of both of them. We can understand the below points.

StateFlow is a type of SharedFlow. StateFlow is a specialization of SharedFlow.

StateFlow is a SharedFlow with a fixed replay = 1 with some more additions. That means new collectors will immediately get the current state as soon as they start collecting.

In a simple way, we can say using the pseudo-code:

```
StateFlow = SharedFlow
            .withInitialValue(initialValue)
            .replay(count=1)
            .distinctUntilChanged()
```

In fact, we can get the StateFlow behavior using SharedFlow as below:

```
val sharedFlow = MutableSharedFlow<Int>(
    replay = 1,
    onBufferOverflow = BufferOverflow.DROP_OLDEST
)
```

```kotlin
sharedFlow.emit(0) // initial value
val stateFlow = sharedFlow.distinctUntilChanged()
```

This is how we can get the StateFlow behavior using SharedFlow.

We can customize the SharedFlow as per our requirement like we can do replay = 2 if we want based on our use case.

Now, it's time to learn where we can use StateFlow and ShareFlow in our Android Project.

Assume that we have a use case: Get the list of the users from the network and show them in the UI.

We have a StateFlow in our ViewModel.

```kotlin
val usersStateFlow = MutableStateFlow<UiState<List<User>>>(UiState.Loading)
```

And we have a collector in our Activity.

```kotlin
usersStateFlow.collect {

}
```

Now, as soon as we open the activity, the Activity will subscribe to collect. The following will be collected:

usersStateFlow: loading state as StateFlow takes the initial value and emits it immediately.

Now, when our viewModel fetches the data from the network. It will set the data to the usersStateFlow.

```kotlin
usersStateFlow.value = UiState.Success(usersFromNetwork)
```

Our Activity collector will get the data of the users and show them in the UI.

Now, if **orientation changes**, the ViewModel gets retained, and our collector present in the Activity will resubscribe to collect. The following will be collected:

usersStateFlow: List of users which was set from the network. (StateFlow keeps the last value).

Advantage: No need for a new network call.

Now, let's try to use the SharedFlow in place of the StateFlow.

We have a SharedFlow in our ViewModel.

```kotlin
val usersSharedFlow = MutableSharedFlow<UiState<List<User>>>()
```

And we have a collector in our Activity.

```kotlin
usersSharedFlow.collect {

}
```

Now, as soon as we open the activity, the Activity will subscribe to collect. Nothing will get collected here as SharedFlow is used.

Now, when our viewModel fetches the data from the network. It will set the data to the usersSharedFlow.

```
usersSharedFlow.emit(UiState.Success(usersFromNetwork))
```

Our Activity collector will get the data of the users and show them in the UI.

Now, if **orientation changes**, the ViewModel gets retained, and our collector present in the Activity will resubscribe to collect. Nothing will get collected here as SharedFlow is used which does not store any data. We will have to make a new network call.

Disadvantage: Unnecessary network call as we were already having the data.

So, in this case, we should use StateFlow instead of SharedFlow. However, we can modify SharedFlow to store data as we have seen above about getting the StateFlow behavior using the SharedFlow.

Now, we will take another example to learn where to use SharedFlow instead of StateFlow.

Suppose, suppose we are doing a task, if that task gets failed, we have to show Snackbar.

We have a SharedFlow in our ViewModel.

```
val showSnackbarSharedFlow = MutableSharedFlow<Boolean>()
```

And we have a collector in our Activity.

```
showSnackbarSharedFlow.collect {

}
```

Now, as soon as we open the activity, the Activity will subscribe to collect. Nothing will get collected here as SharedFlow is used.

Then, when our viewModel starts the task and gets failed. It will set the value true to showSnackbarSharedFlow.

```
showSnackbarSharedFlow.emit(true)
```

Our Activity collector will get the value as true and show the Snackbar.

Now, if **orientation changes**, the ViewModel gets retained, and our collector present in the Activity will resubscribe to collect. Nothing will get collected here as SharedFlow does not keep the last value. And that is fine. We should not show the Snackbar again on orientation changes.

Advantage: It does not show Snackbar again as intended.

Now, let's try to use the StateFlow in place of the SharedFlow.

We have a StateFlow in our ViewModel.

```
val showSnackbarStateFlow = MutableStateFlow(false)
```

And we have a collector in our Activity.

```
showSnackbarStateFlow.collect {

}
```

Now, as soon as we open the activity, the Activity will subscribe to collect. The following will get collected.

showSnackbarStateFlow: false as StateFlow takes the initial value and emits it immediately.

Now, when our viewModel starts the task and gets failed. It will set the value true to showSnackbarSharedFlow.

```
showSnackbarStateFlow.value = true
```

Our Activity collector will get the value as true and show the Snackbar.

Now, if **orientation changes**, the ViewModel gets retained, and our collector present in the Activity will resubscribe to collect. The following will be collected:

showSnackbarStateFlow: true will get collected here as StateFlow keeps the last value. It will show the Snackbar again. And that is not fine. We should not show the Snackbar again on orientation changes.

Disadvantage: It shows Snackbar again which is not required.

So, in this case, we should use SharedFlow instead of StateFlow.

Now, as we have a good understanding of StateFlow and SharedFlow, we can easily decide which one to use in which case.

This was all about StateFlow and SharedFlow in Kotlin.

Master Kotlin Coroutines from here: Mastering Kotlin Coroutines

That's it for now.

Thanks

Amit Shekhar

You can connect with me on:

- Twitter
- YouTube
- LinkedIn
- GitHub

Read all of my high-quality blogs here.

TAGS