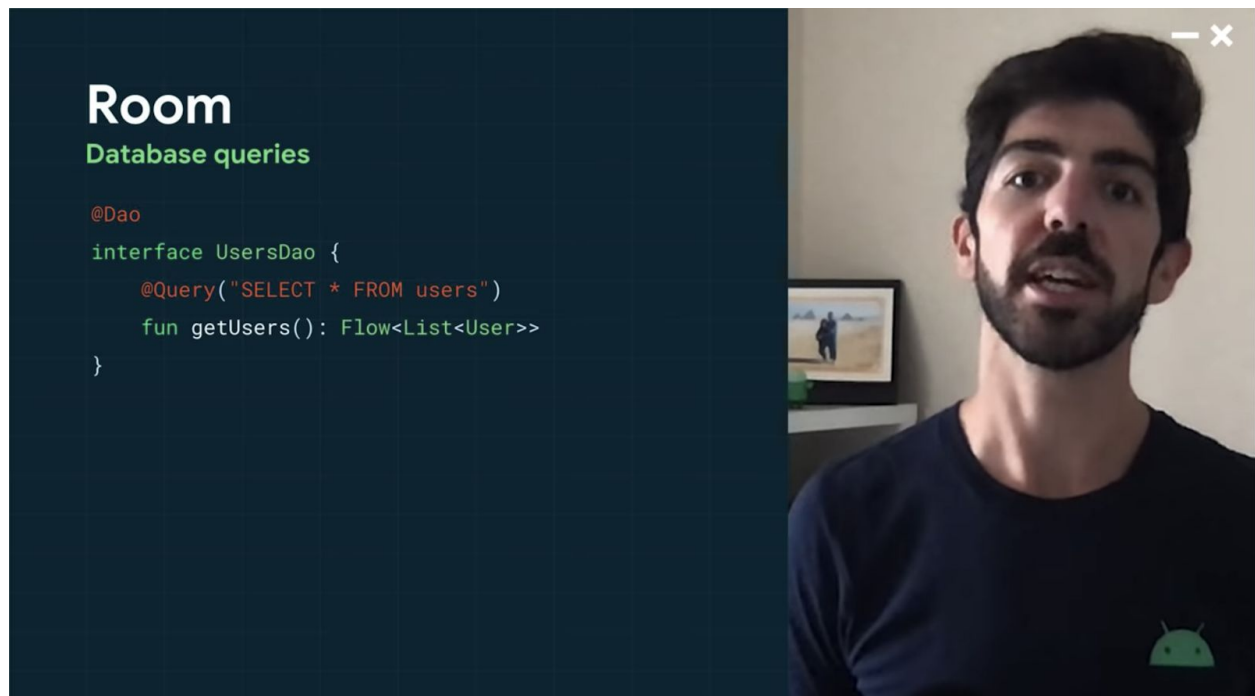


1. Kotlin Flows
 - a. Model streams of the data, not the single values.
 - b. Built upon the foundation of coroutines and suspend function
 - i. Cancellation
 - ii. Structure Concurrency
 - iii. Exception Transparency
 - iv. Natural back pressure handling



- v. Here, once the database is changed, it automatically updates the UI in a reactive way. (Observing the flow lets **the app react to changes** when the DB emits new data)
2. Comparison to the RxJava
 - a. Conceptually it's not different.
3. Comparison to the LiveData
 - a. Short Answer: little difference
 - b. Full Answer: more complicated

LiveData vs Flow

State holder vs cold data stream

```
MutableLiveData<T>:  
    var value: T  
    fun postValue(value:T)
```

```
LiveData<T>:  
    val value: T
```

```
flow { emit(...) }
```



4. Deep Dive into LiveData vs Flow

- Conceptually LiveData is a value holder, so the most important thing is to observe the current state.
- As UI only sees the latest result, hence it's fine to use LiveData
- Flow observes the 'current' of the data, so it's more appropriate for the lower layer of the app

LiveData vs StateFlow

State holder vs ...a Flow, for states

```
MutableLiveData<T>:  
    var value: T  
    fun postValue(value:T)
```

```
LiveData<T>:  
    val value: T
```

```
MutableStateFlow<T>:  
    var value: T
```

```
StateFlow<T>:  
    val value: T
```



5. StateFlow

- a. Found in the latest coroutine version
- b. It holds the current value like LiveData
- c. But LiveData has some drawbacks like being able to be observed only in the main thread.
- d. Trying converting one over the other, as converting is easy from regular Flow to LiveData or vice versa.

lifecycleScope

and the pausing dispatcher

```
// Observing a LiveData in an Activity
myFlow.asLiveData().observe ( ... )
// cancels when there are no active observers

// Collecting a Flow in an Activity
lifecycleScope.launchWhenStarted { myFlow.collect { ... } }
// pauses when not STARTED, cancels when Activity is destroyed
```



6. ViewModelScope, CoroutineScope and Android

- a. LifecycleScope for Activities and Fragments
 - i. When to cancel the livedata and flow?

callbackFlow

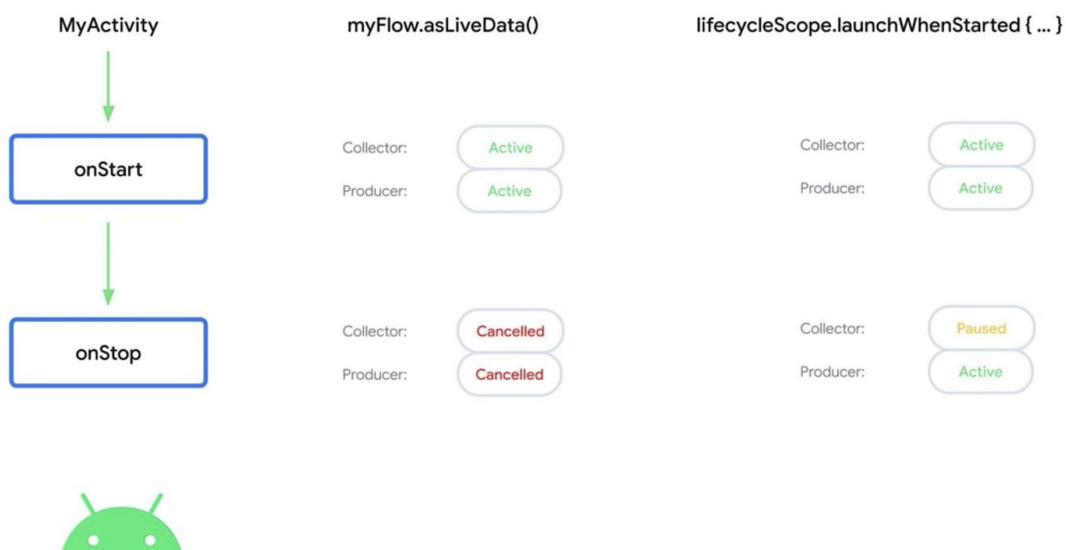
Hot source, cold flow

```
fun FusedLocationProviderClient.locationFlow() = callbackFlow<Location> {  
    val callback = object : LocationCallback() {  
        override fun onLocationResult(result: LocationResult?) {  
            result ?: return  
            try { offer(result.lastLocation) } catch(e: Exception) {}  
        }  
    }  
    requestLocationUpdates(createLocationRequest(), callback, Looper.getMainLooper())  
    .addOnFailureListener { e ->  
        close(e) // in case of exception, close the Flow  
    }  
    // clean up when Flow collection ends  
    awaitClose {  
        removeLocationUpdates(callback)  
    }  
}
```



7. callbackFlow

- A builder function that lets us convert callback or listener-based API to Flow.
- Hot source: Can it be understood as Hot Observable (?)
- Cold flow: the code block will not be executed until the flow is collected.




- d. So, even if the activity stays onStop, producers are still alive, which means it causes battery problems.
- e. <https://developer.android.com/topic/libraries/architecture/coroutines>

SharedFlow is coming

and `.shareIn()` for cold Flows

```
fun <T> Flow<T>.shareIn(
    scope: CoroutineScope,
    replay: Int,
    started: SharingStarted = SharingStarted.Eagerly,
    initialValue: T = NO_VALUE as T
): SharedFlow<T>

// Only keep upstream active when someone's listening
SharingStarted.WhileSubscribed
```



8. SharedFlow

- a. It can take the flow, and be shared among multiple subscribers.
- b. Useful when the flow is expensive to create.
- c. WhileSubscribed option: free up the upstream producer whenever no subscribers.