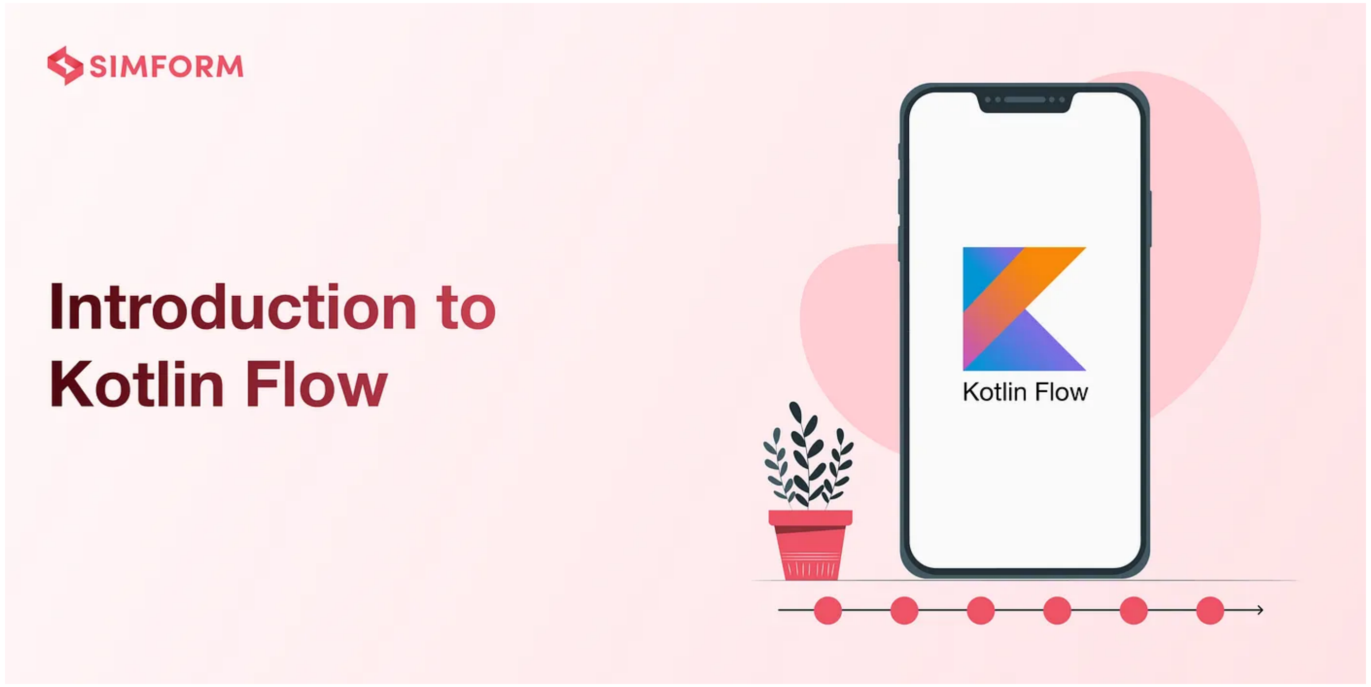
Introduction to Kotlin Flow

Explore the various facets of Kotlin Flow.

Sajid Juneja · Follow
Published in Simform Engineering · 8 min read · May 4, 2023

803 1



What comes to mind when you hear the word “flow”? It may occur to you as something continuous, such as a stream.

What is Flow?

- A stream of data that can be computed asynchronously is conceptually referred to as a Flow.
- It is constructed using Coroutines. An appropriate Kotlin type for modeling data streams is Flow.

- Flow, like LiveData and RxJava streams, allows you to implement the observer pattern: a software design pattern consisting of an object (source) that keeps a list of its dependents, called observers (collectors) and automatically notifies them of any state changes.
- A Flow uses suspended functions to consume and produce in an asynchronous manner.

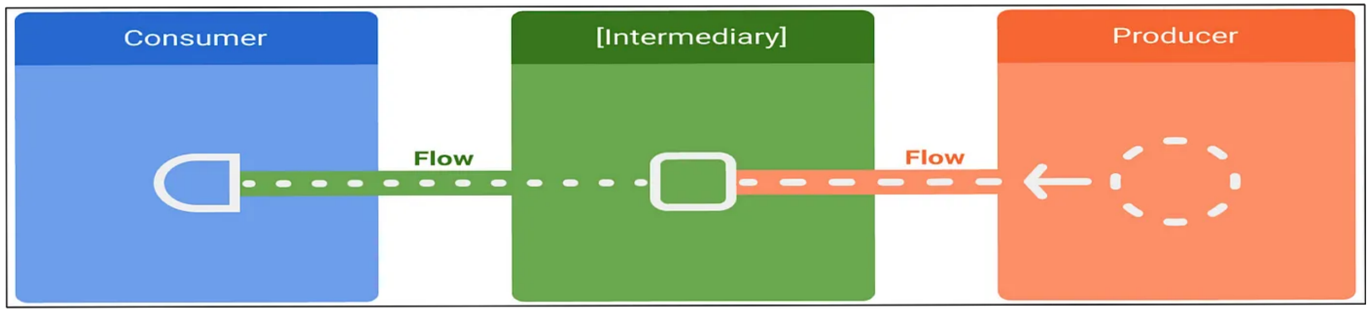
Why do we need Flow?

After Coroutines were introduced, people started enjoying them due to their simplicity and structured concurrency.

Coroutines, combined with the growing usage of Kotlin, led people to express their interest in having a pure Kotlin implementation of RxJava to leverage the power of Kotlin like Type Systems, Coroutines, etc. When these are combined, they form Flow.

We can say Flow takes advantage of LiveData and RxJava.

Working of Flow: Entities involved in Flow



There are three entities involved in flow:

1. A **producer** produces data that is added to the stream.

2. **Intermediaries**(Optional) can modify each value emitted into the stream or the stream itself without consuming the values.

3. A **consumer** consumes the values from the stream.

Creating a Flow and consuming values

To create a Flow, first you need to create a producer. The standard library provides you with several ways to create a flow, the easiest way is to use the *flow* operator:

```
val numbersFlow: Flow<Int> = flow {
    repeat(60) { it ->
        emit(it+1) //Emits the result of the request to the flow
        delay(1000) //Suspends the coroutine for some time
    }
}
```

To collect flow, first you will launch a Coroutine because flow operates on Coroutines under the hood. The collect ***operator*** is used to collect the values emitted by it.

```
lifecycleScope.launch {
    viewModel.numbersFlow.collect { it ->
        binding.textTimer.text = it.toString()
    }
}
```

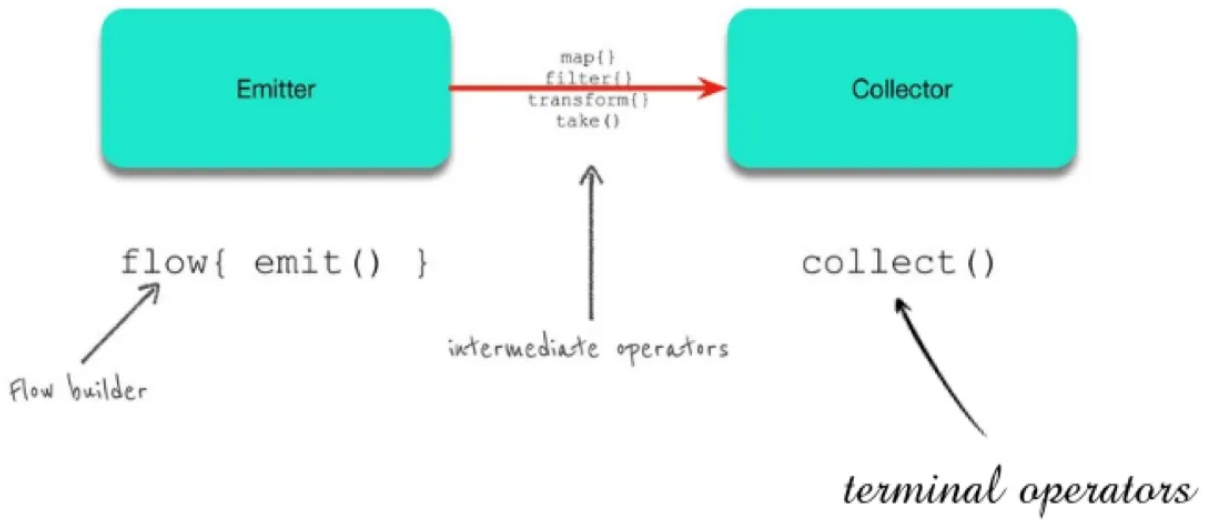
Types of Flow

- **Cold Flow** — It does not start producing values until one starts to collect them. It can have only one subscriber.
  e.g. flow

- **Hot Flow —** It will produce values even if no one is collecting them. e.g. StateFlow, SharedFlow

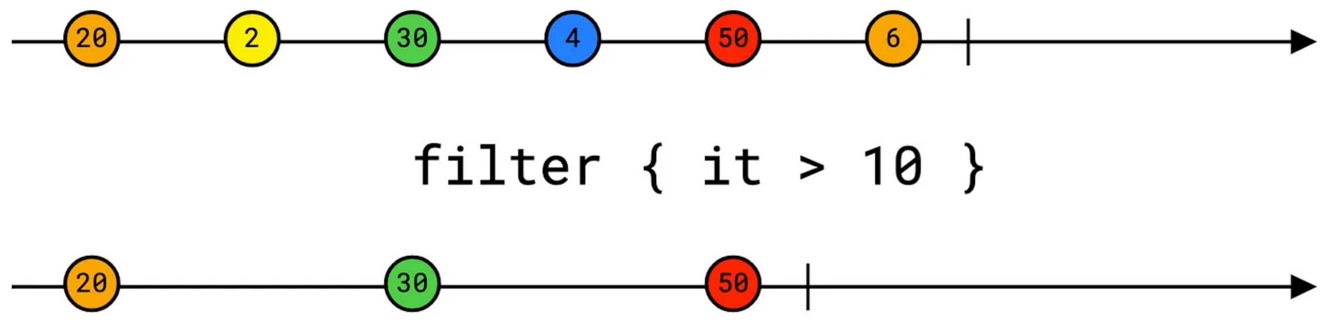
Commonly used operators and their types

- **Terminal Operators —** These complete normally or exceptionally depending on the successful or failed execution of all the flow operations upstream. The most basic terminal operator is *collect*.
- **Intermediate Operators —** These are map, filter, take, zip, etc. They only set up a chain of operations for future execution and quickly return.



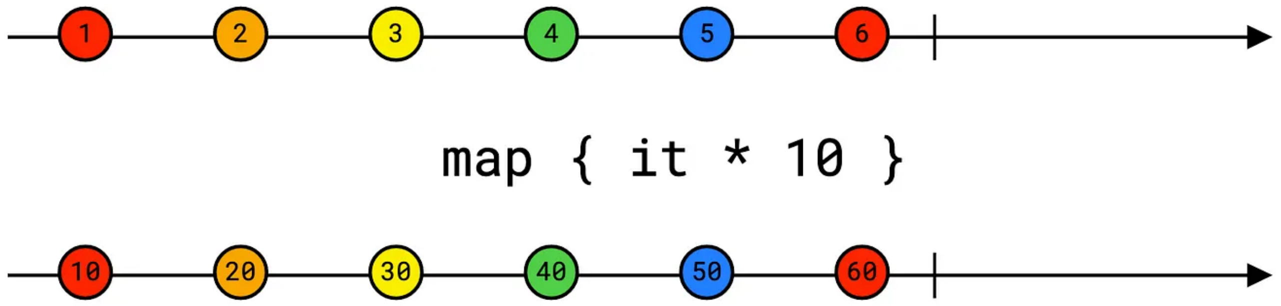
Filter Operator

Returns a flow containing only values of the original flow that match the given predicate.

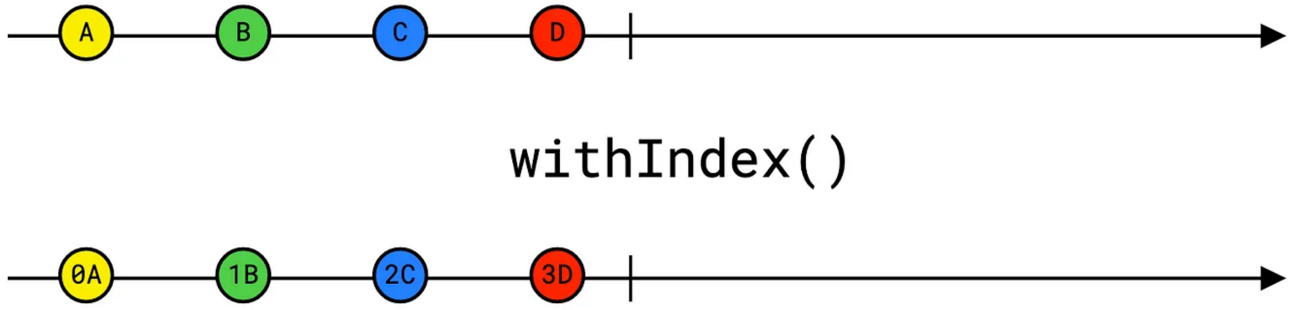


Map Operator

Returns a flow containing the results of applying the given transform function to each value of the original flow.



withIndex Operator
Returns a flow that wraps each element into IndexedValue, containing value and its index (starting from zero).



Exception Handling in Flow

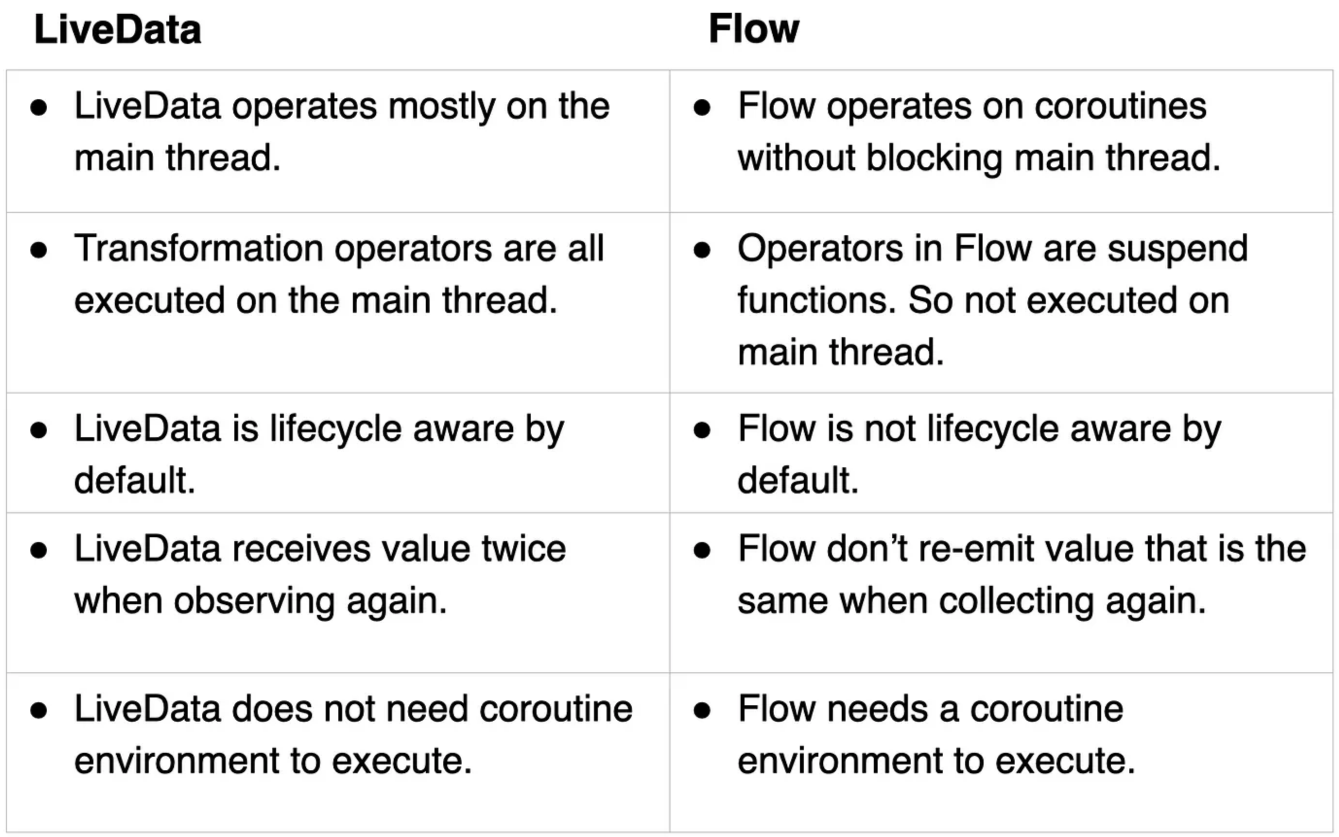
Flow collection can complete with an exception if it's thrown inside the building block or any of the operators.

The correct way to encapsulate exception handling in an emitter is to use the *catch* operator.

Flows must be transparent to exceptions, and it is a violation of the exception transparency principle to emit values in a flow builder from inside of a try/catch block. That is why try/catch is not recommended.

```
lifecycleScope.launch {
    viewModel.numbersFlow.map { it ->
        it * it
    }.filter { it ->
        it % 2 == 0
    }.catch { exception ->
        handleException(exception)
    }.collect { it ->
        binding.tvFlow.text = it.toString()
    }
}
```

Comparison of LiveData and Flow



LiveData	Flow
● LiveData operates mostly on the main thread.	● Flow operates on coroutines without blocking main thread.
● Transformation operators are all executed on the main thread.	● Operators in Flow are suspend functions. So not executed on main thread.
● LiveData is lifecycle aware by default.	● Flow is not lifecycle aware by default.
● LiveData receives value twice when observing again.	● Flow don’t re-emit value that is the same when collecting again.
● LiveData does not need coroutine environment to execute.	● Flow needs a coroutine environment to execute.

StateFlow and SharedFlow

With StateFlow and SharedFlow, flows can efficiently update states and values to multiple consumers.

StateFlow:

- It is a hot flow. Its active instance exists independently of the presence of collectors.
- It needs an initial value.
- We can create its variable like :
  val stateFlow = MutableStateFlow(0)
- The only value that is emitted is the last known value.
- The value property allows us to check the current value.
- It does not emit consecutive repeated values. When the value differs from the previous item, it emits the value.

```
class CounterModel {
    private val counter = MutableStateFlow(0) // private mutable state flow
    val counterValue = counter.asStateFlow() // public read-only state flow

    fun incrementAtomically() {
        counter.update { count -> { count + 1 }
    }

    fun incrementCounter() {
        val count = counter.value
        counterValue.value = count + 1
    }
}
```

SharedFlow:

- By default, it does not emit any value since it does not need an initial value.
- We can create its variable like :
  val sharedFlow = MutableSharedFlow<Int>()
- With the **replay** operator, it is possible to emit many previous values at once.
- It does not have a value property.
- The emitter emits all the values without caring about the distinct differences from the previous item. It emits consecutive repeated values also.
- It is useful for broadcasting events that happen inside an application to subscribers that can come and go.

```
class EventBus {
    private val events = MutableSharedFlow<Event>() // private mutable shared fl
    val eventsValue = events.asSharedFlow() // public read-only shared flow

    suspend fun produceEvent(event: Event) {
        events.emit(event) // suspends until all subscribers receive it
        }
    }
```

stateIn and shareIn

The ***shareIn*** and ***stateIn*** operators convert cold flows into hot flows.

The shareIn operator returns a SharedFlow instance whereas stateIn returns a StateFlow.

stateIn contains 3 parameters scope, started and initialValue.

- **scope** = the coroutine scope to define.
- **started** = *SharingStarted* strategy:
  — **Eagerly:** Sharing is started immediately and never stops.
  — **Lazily:** Sharing is started when the first subscriber appears and never stops.
  — **WhileSubscribed:** Sharing is started when the first subscriber appears, immediately stops when the last subscriber disappears (by default), keeping the replay cache forever (by default).
- **initialValue** = initial value.

```
val stateFlow: StateFlow<SomeState> = someFlow
    .stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5000),
        initialValue = someInitialValue,
    )
```

shareIn contains the same three parameters as stateIn, but instead of initialValue, it has a replay parameter.

- **replay** = how many times to emit the value?

```
val sharedFlow: SharedFlow<SomeState> = someFlow
    .shareIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5000),
        replay = 1,
    )
```

callbackFlow

callbackFlow is a flow builder that lets you convert callback-based API into flows.

The resulting flow is cold, which means that [block] is called every time a terminal operator is applied to the resulting flow.

The following example uses location callback of fusedLocationClient to convert it into callbackFlow.

```
// Get location updates of fusedLocationClient with callbackFlow
val locationUpdatesFlow = callbackFlow<Location> {
    val callback = object : LocationCallback() {
        override fun onLocationResult(result: LocationResult?) {
            result ?: return
            // Send the new location
            trySend(result.lastLocation)
        }
    }

    // Request location updates
    fusedLocationClient.requestLocationUpdates(
        locationRequest,
        callback,
        Looper.getMainLooper()
    ).addOnFailureListener { e ->
        close(e) // in case of exception, close the Flow
    }
```

```
        awaitClose {
            // Reomve location updates when Flow collection ends
            fusedLocationClient.removeLocationUpdates(callback)
        }
    }
```

Collecting Flow from View (i.e. Activity or Fragment)

```
// For Fragment use viewLifecycleOwner.lifecycleScope
lifecycleScope.launch {
    // For Fragment use viewLifecycleOwner.repeatOnLifecycle
    repeatOnLifecycle(LifeCycle.State.STARTED) {
        viewModel.flow.collect {
        // do something with values
        }
        // However, here if we start collecting flow then it will never going to
        // Therefore, to overcome this we should use launch { }
        launch {
            viewModel.userDetails.collect {
                // do something with user details
             }
        }
    }
}
```

To collect Flow in activity, we use *lifecycleScope.launch*.

In that block, we need to call *repeatOnLifecycle(LifeCycle.State.Started)* to collect flow safely when the lifecycle state is in started state.

repeatOnLifecycle establishes a suspending point that executes the block anytime the lifecycle enters the specified state and cancels it when it falls below it.

It requires a *Lifecycle.State* as a parameter. When the lifecycle reaches that state, it immediately creates and launches a new Coroutine with the block supplied to it, and it cancels the ongoing Coroutine that is running the block when the lifecycle falls below that state.

Collecting Flow in Compose

```
val someFlow by viewModel.flow.collectAsStateWithLifecycle()
```

To collect flow in compose, we use *collectAsStateWithLifecycle*.

Compose provides the *collectAsStateWithLifecycle* function, which collects values from a flow and gives the latest value to be used wherever needed. When a new flow value is emitted, we get the updated value, and re-composition takes place to update the state of the value.

It uses *LifeCycle.State.Started* by default to start collecting values when the lifecycle is in the specified state and stops when it falls below it.

Some useful points

- Flow can be used with Data Binding from Android Studio Arctic Fox | 2020.3.1 onwards.
- Room, DataStore, Paging3 and other various libraries provide support for Flow.
- Flow is extremely appropriate for data updates. For example, you can use flow with Room to be notified of changes in your database.
- For one-shot operations, LiveData is sufficient.

Summary

In conclusion, Kotlin Flow is a reactive programming library for Kotlin that provides a way to asynchronously process streams of data. Its concise and streamlined syntax, based on Coroutines, makes it easy to create and manipulate data streams.

In addition, Kotlin offers StateFlow and SharedFlow, which are useful for managing and sharing state across different parts of an application. StateFlow is ideal for managing stateful data, while SharedFlow can buffer and emit values to new subscribers when they first subscribe.

Overall, Kotlin Flow, StateFlow, and SharedFlow are powerful and flexible tools for managing and processing streams of data in Kotlin, and they are rapidly gaining popularity among developers as essential components of modern reactive programming.

Happy Coding!

Please don’t forget to give claps 👏 and share this with your fellow coder friends.

> *For more such insights and updates on the latest tools and technologies — follow the Simform engineering blog.*