



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

🚀 Day 21

Advanced

Asynchronous Programming

Unlocking the Hidden Secrets of Kotlin Coroutines & Flow! ⚡



Esmaeeil Moradi

@esmaeeil-moradi700



Day 21



Advanced Asynchronous Programming – Unlocking the Hidden Secrets of Kotlin Coroutines & Flow! ⚡

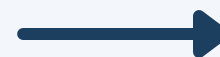
Hey Android Architects!

Think you know Coroutines and Flow? It's time to peel back the layers and dive deep into the hidden secrets of asynchronous programming on Android. From optimizing data streams to managing complex interactions, this is where the true power of Coroutines and Flow becomes evident! Today, on Day 21 of the #30DayAndroidChallenge, we're stepping into the world of advanced concurrency, going far beyond *launch* and *collect*. On Day 20, we laid a solid foundation for Coroutines and Flow. But real-world applications demand a deeper understanding and more precise tools to handle intricate scenarios, optimize performance, and ensure stability at scale. This article is designed for senior Android developers ready to elevate their skills and achieve true mastery in asynchronous programming.



Esmaeel Moradi

@esmaeel-moradi700



Introduction:

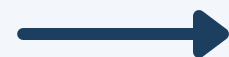
Beyond the Basics – Entering the Realm of Advanced Concurrency

If you're familiar with *suspend*, *Dispatchers*, and *Structured Concurrency*, you're ready to delve into the inner workings and complexities of Kotlin *Coroutines* and *Flow*. Today, we'll cover topics that will help you build significantly more stable, responsive, and maintainable applications. This journey includes exploring *Hot/Cold Flows*, *SharedFlow*, *StateFlow*, *Channels*, *Concurrency control* (*buffering*, *conflate*), and a *deep dive* into *Coroutine Context*.



Esmaeel Moradi
@esmaeel-moradi700

Flows:



From Cold to Hot – A Deeper Understanding of Streams

Understanding the behavior of data streams (Flows) is crucial for effectively managing data in complex applications.

Cold Streams in Flow:

By default, a *Flow* is **cold**. This means its **producer** only starts emitting values when a collector attaches to it, and each new *collector* triggers the *Flow* execution from the beginning.

```
fun coldFlowExample(): Flow<Int> = flow {
    println("Flow started: Emitting 1")
    emit(1)
    delay(100)
    println("Flow started: Emitting 2")
    emit(2)
}

// Collector 1
lifecycleScope.launch {
    println("Collector 1 starts")
    coldFlowExample().collect { value -> println("Collector 1: $value") }
}

// Collector 2 (starts later)
lifecycleScope.launch {
    delay(50) // Simulate delay before second collector starts
    println("Collector 2 starts")
    coldFlowExample().collect { value -> println("Collector 2: $value") }
}
```



Esmaeel Moradi

@esmaeel-moradi700



Why Cold?

This behavior offers high efficiency; resources are consumed only when needed. Also, each Collector receives its independent stream, enhancing safety. However, in scenarios requiring a single value broadcast to multiple Collectors without re-executing the producer (e.g., for UI events), it becomes inefficient or impossible.

Hot Streams: Introducing *SharedFlow* and *StateFlow*

To address the limitations of Cold Flow in specific scenarios, we need Hot Streams. Hot Streams produce values even without an active Collector and can broadcast values to multiple Collectors simultaneously.

- ***StateFlow***: *StateFlow* is a specialized *SharedFlow* that always has an initial value and only emits distinct values. These features make it an ideal choice for holding and emitting UI State. Unlike *LiveData*, *StateFlow* is Kotlin-native and offers deeper integration with *Coroutines* and other Flows.



Esmaeel Moradi

@esmaeel-moradi700



```
class MyViewModel : ViewModel() {
    private val _uiState = MutableStateFlow(MyUiState.Loading)
    val uiState: StateFlow<MyUiState> = _uiState.asStateFlow()

    init {
        viewModelScope.launch {
            delay(2000)
            _uiState.value = MyUiState.Success("Data Loaded!")
            delay(1000)
            _uiState.value = MyUiState.Success("Data Loaded!") // This won't
        }
    }
}

// In UI (Activity/Fragment)
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.uiState.collect { state ->
            when (state) {
                is MyUiState.Loading -> binding.progressBar.isVisible = true
                is MyUiState.Success -> binding.textView.text = state.data
                // ...
            }
        }
    }
}
```

- ***SharedFlow***:



SharedFlow is a more general-purpose and flexible Hot Flow. It's used for emitting events or data streams that don't require an initial value or distinctness checks. *SharedFlow* is highly suitable for scenarios like one-shot events (e.g., displaying a Toast, navigation) or broadcasting data to multiple Collectors without guaranteeing replay of past values.

- **Key parameters for *SharedFlow* include:**
- *replay*: The number of values new Collectors immediately receive upon connection.
- *extraBufferCapacity*: Additional buffer capacity to hold values before *onBufferOverflow* policy is applied.
- *onBufferOverflow*: The buffer overflow management policy (e.g., *SUSPEND*, *DROP_LATEST*, *DROP_OLDEST*).



Esmaeel Moradi

@esmaeel-moradi700



```
class EventViewModel : ViewModel() {
    private val _events = MutableSharedFlow<String>(
        replay = 0, // No replay for new collectors
        extraBufferCapacity = 1, // Buffer 1 event
        onBufferOverflow = BufferOverflow.DROP_OLDEST // Drop if buffer full
    )
    val events: SharedFlow<String> = _events.asSharedFlow()

    fun showToastEvent(message: String) {
        viewModelScope.launch {
            _events.emit(message)
        }
    }
}

// In UI (Activity/Fragment)
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.events.collect { message ->
            Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
        }
    }
}
```



Esmaeel Moradi
@esmaeeil-moradi700

Channels:

Powerful Interactions Between Coroutines

Channels offer a powerful way for two or more Coroutines to communicate and exchange data. They operate similarly to a *BlockingQueue* but with *suspend* capabilities, making them non-blocking.

Difference from *SharedFlow*: *SharedFlow* (and *StateFlow*) are designed for broadcasting events or state to multiple listeners; that is, each Collector receives a copy of the data, and data might persist in the Flow (depending on replay). In contrast, a Channel is for one-to-one (or many-to-many) communication with a consumer that consumes messages. Once a message is read from a Channel, it's removed.

Types of **Channels** (based on buffering):

- **Rendezvous** (default): Has no buffer. The sender suspends until the receiver is ready to receive (and vice-versa).
- **Buffered**: Has a specified buffer capacity.
- **Conflated**: Only keeps the latest element in the buffer, dropping older ones.
- **Unlimited**: An unbounded buffer (can lead to *OutOfMemoryError*).

Channels utilize two interfaces: *SendChannel* (for sending) and *ReceiveChannel* (for receiving).



Esmaeel Moradi
@esmaeeil-moradi700



```
fun main() = runBlocking {
    val channel = Channel<String>()

    // Sender Coroutine
    launch {
        channel.send("Message 1")
        channel.send("Message 2")
        println("Sent all messages")
        channel.close() // Important: Close the channel when done sending
    }

    // Receiver Coroutine
    launch {
        for (message in channel) { // Iterate over channel until it's closed
            println("Received: $message")
        }
        println("Channel closed and all messages received")
    }
}
```

Use Cases for Channels:

- **Pipelines:** Building multi-stage processing pipelines where data flows between different *Coroutines*.
- **Event Processing:** Queuing events for sequential processing within a single Coroutine.
- **Manual Backpressure:** Implementing complex backpressure logic with fine-grained control over data flow.



Esmaeel Moradi
@esmaeel-moradi700

Advanced Concurrency Control and Stream Optimization:

When the *producer* (data emitter) is faster than the *collector* (data consumer), controlling the data flow becomes critical. *Flow* offers powerful tools to manage this situation.

Buffering and Conflate in Flow:

- `buffer()`:

This operator creates an intermediate buffer between the *producer* and the *collector*. If the producer is faster than the collector, values are stored in this buffer.

```
flow {  
    for (i in 1..5) {  
        delay(100) // Producer emits every 100ms  
        emit(i)  
    }  
}  
.buffer(capacity = 2) // Buffer up to 2 items  
.collect { value ->  
    delay(300) // Collector consumes every 300ms  
    println("Collected: $value")  
}
```



Esmaeel Moradi
@esmaeel-moradi700

- *conflate()*:

This operator is extremely important, especially for UI updates. When the *producer* is faster than the *collector*, *conflate()* always preserves the latest value and drops older ones. This ensures the *collector* never processes "stale" values.

```
flow {
  repeat(5) {
    emit("New Data $it")
    delay(50) // Producer is fast
  }
}
.conflate() // Only process the latest
.collect { value ->
  delay(200) // Collector is slow
  println("Collected (conflated): $value")
}
```



Esmaeel Moradi

@esmaeel-moradi700

- *collectLatest()*:

This terminal operator has a very specific behavior. When *collectLatest* receives a value, if a new value arrives before the previous one finishes processing, it cancels the processing of the previous value and immediately starts processing the new one. This is ideal for scenarios like typing in a **search bar or making API calls on scroll**.

```
fun searchFlow(query: String) = flow {
    println("Searching for: $query")
    delay(500) // Simulate network call
    emit("Results for $query")
}

// In a ViewModel
fun onSearchQueryChanged(query: String) {
    viewModelScope.launch {
        // Cancel previous search if new query arrives
        searchFlow(query).collectLatest { results ->
            println("Displaying: $results")
        }
    }
}
```



Esmaeel Moradi
@esmaeeil-moradi700



- Using **debounce** and **throttleFirst**:

These operators are used to control the rate at which events are emitted from the producer.

- **debounce(timeoutMillis: Long):**

Delays emitted values for a specified *timeoutMillis*. If a new value is emitted within this duration, the previous value is discarded, and the timer restarts for the new value. This is excellent for preventing frequent calls (e.g., when a user types in a search field).

```
val searchInput = MutableStateFlow("")

// In a ViewModel
init {
    viewModelScope.launch {
        searchInput
            .debounce(300) // Wait 300ms after last input
            .onEach { query ->
                if (query.isNotBlank()) {
                    performSearch(query) // Call API
                }
            }
            .collect() // Terminal operator to activate the flow
    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700



- ***throttleFirst(windowDuration: Long):***

Allows only the first value emitted within a specified time window (*windowDuration*) to pass through, ignoring all other values in that window. This is used to prevent rapid, successive clicks on a button or similar events.

```
val clickEvents = MutableSharedFlow<Unit>()

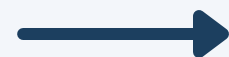
fun onClicked() {
    viewModelScope.launch { clickEvents.emit(Unit) }
}

// In a ViewModel
init {
    viewModelScope.launch {
        clickEvents
            .throttleFirst(500) // Only process one click every 500ms
            .onEach {
                println("Button clicked (throttled)")
                // Perform safe action
            }
            .collect()
    }
}
```



Esmaeel Moradi
@esmaeel-moradi700

Coroutine Context Deep Dive: Beyond Dispatchers



Coroutine Context is more than just Dispatchers. It's a persistent set of Elements that define a *Coroutine's* behavior. A deep understanding of it is vital for correctly managing *Coroutines* and Jobs in large applications.

- **What is Coroutine Context?**

A *Coroutine Context* is a collection of different Elements, each controlling a specific aspect of the *Coroutine*. These Elements include:

- **Dispatcher:** Determines which Thread or Thread Pool the *Coroutine* runs on.
- **Job:** The *Coroutine's* "handle" that allows you to cancel it, join (wait for its completion), or monitor its state.
- **CoroutineName:** A name for the *Coroutine* for debugging purposes.
- **CoroutineExceptionHandler:** For handling uncaught exceptions within *Coroutines*.

Job and Lifecycle Management:

Every *Coroutine* created by `launch` or `async` returns a Job. Jobs form a parent-child hierarchy in Structured Concurrency.

- When a parent Job is cancelled, all its child Jobs are also cancelled.
- If a child Job encounters an error, the error propagates up to the parent, which can cause the parent (and other children) to be cancelled (unless error handling is in place).



Esmaeel Moradi

@esmaeel-moradi700



```

val parentJob = Job()
val scope = CoroutineScope(Dispatchers.Default + parentJob)

scope.launch { // Child 1
    println("Child 1 started")
    delay(1000)
    println("Child 1 finished")
}

scope.launch { // Child 2
    println("Child 2 started")
    delay(500)
    parentJob.cancel() // Cancel the parent job
    println("Child 2 tried to finish") // This won't be printed
}

Thread.sleep(2000) // Keep main thread alive
println("Parent Job cancelled: ${parentJob.isCancelled}")

```

- ***Composing Coroutine Context:***

You can combine different *Coroutine* Context Elements using the + operator. This allows you to create custom-tailored Contexts for specific Coroutines.

```

val customContext = Dispatchers.IO + CoroutineName("MyCustomWorker") +
    println("Caught exception in custom Coroutine: $throwable")
}

val scope = CoroutineScope(Job()) // New scope for demonstration

scope.launch(customContext) {
    println("Running on ${coroutineContext[CoroutineName.Key]?.name}")
    println("Thread: ${Thread.currentThread().name}")
    throw IllegalStateException("Something went wrong!")
}

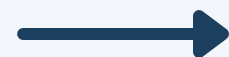
```



Esmaeel Moradi

@esmaeel-moradi700

withContext vs. **async/launch** vs. **flowOn**:



These functions are all used to change the Dispatcher (or Context), but with different purposes and behaviors:

- **withContext(context: CoroutineContext):**

Used to temporarily change the Dispatcher for a block of code. After the block completes, the Coroutine returns to its previous Dispatcher. This function suspends the code block and waits for its completion.

```
suspend fun fetchDataAndProcess() {  
    val result = withContext(Dispatchers.IO) {  
        // Network call or DB operation here  
        "Raw Data"  
    }  
    withContext(Dispatchers.Default) {  
        // Heavy computation here  
        "Processed Data from $result"  
    }  
    // Implicitly back on the original dispatcher (e.g., Main)  
}
```

- **launch(context: CoroutineContext)** and **async(context: CoroutineContext):**

These are used to launch new Coroutines. **launch** returns a Job (no result), while **async** returns a Deferred (a result that can be awaited later). They launch the Coroutine in a new Context, which can be independent of the calling Context.



Esmaeel Moradi

@esmaeel-moradi700



```
val job = CoroutineScope(Dispatchers.Main).launch(Dispatchers.IO) {  
    // This Coroutine runs on IO  
}  
val deferred = CoroutineScope(Dispatchers.Main).async(Dispatchers.Default) {  
    // This Coroutine runs on Default  
    "Result"  
}  
val result = deferred.await()
```

- **flowOn(context: CoroutineContext):**

Specific to Flows, it changes the Dispatcher for the upstream Flow. This means operations placed before flowOn run on the Dispatcher specified in flowOn, while downstream operations and collect run on the Collector's Dispatcher.

```
fun getDataStream(): Flow<String> = flow {  
    // This part runs on IO (due to flowOn)  
    delay(100)  
    emit("Data")  
}.flowOn(Dispatchers.IO) // Upstream moves to IO  
  
// In a ViewModel  
init {  
    viewModelScope.launch(Dispatchers.Main) { // Collector runs on Main  
        getDataStream().collect { data ->  
            // This part runs on Main  
            println("Collected on UI: $data")  
        }  
    }  
}
```



Esmaeel Moradi
@esmaeel-moradi700



Coroutine Context Deep Dive: Beyond Dispatchers

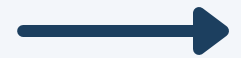
Error handling is critical in asynchronous systems. Flow provides robust tools for managing errors and implementing retry logic.

- `retry(retries: Long)` **and** `retryWhen(predicate: suspend FlowCollector<T>.(cause: Throwable, attempt: Long) -> Boolean)` **in Flow:**

`retry()` allows you to automatically retry an operation a specified number of times if an error occurs (e.g., in network calls). `retryWhen()` provides greater flexibility to implement custom backoff logic based on the error type or the number of attempts.



Esmaeel Moradi
@esmaeel-moradi700



```
var attempts = 0
fun unreliableApiCall(): Flow<String> = flow {
    if (attempts < 2) { // Simulate failure for first 2 attempts
        attempts++
        throw IOException("Network error attempt $attempts")
    }
    emit("Data from API after $attempts attempts")
}

// In a ViewModel
viewModelScope.launch {
    unreliableApiCall()
        .retryWhen { cause, attempt ->
            // Retry up to 3 times for IOException, with exponential backoff
            if (cause is IOException && attempt < 3) {
                delay(1000 * attempt) // Wait longer for each retry
                true // Retry
            } else {
                false // Do not retry
            }
        }
        .catch { e ->
            println("Final error: ${e.message}") // Catch any unhandled errors
        }
        .collect { data ->
            println("Successfully received: $data")
        }
}
```



Esmaeel Moradi
@esmaeel-moradi700



- **Using `catch { ... }` in Flow:**

The catch operator is used to handle exceptions that occur in the upstream Flow. You can handle the error, re-emit it downstream, or emit a fallback value.

```
flow {  
    emit(1)  
    throw IllegalStateException("Something bad happened!")  
    emit(2) // This won't be reached  
}  
  
.catch { e ->  
    println("Caught in catch: ${e.message}")  
    emit(0) // Emit a fallback value  
}  
  
.collect { value ->  
    println("Collected: $value")  
}
```



Esmaeel Moradi
@esmaeeil-moradi700

Conclusion:

Mastering Asynchronous Operations for Scalable Applications

Congratulations! You've now journeyed deep into Kotlin Coroutines and Flow, gaining insights into powerful tools that extend far beyond the basics. Understanding **Hot/Cold Flows**, **SharedFlow**, **StateFlow**, **Channels**, advanced flow controls (**buffering**, **conflate**, **collectLatest**, **debounce**, **throttleFirst**), and **Coroutine Context** alongside sophisticated error handling transforms you into a software architect capable of building robust and scalable systems on Android.

These concepts are vital tools for senior engineers seeking to build powerful and scalable applications. By applying this knowledge, you can enhance your app's stability, optimize performance, and gain precise control over complex data flows.

Coroutines and Flow have not only simplified asynchronous coding but have also opened the door to powerful and flawless system designs. This journey has just begun! In the coming days, we'll apply this knowledge to even more practical scenarios.

Happy coding until tomorrow! 🖥️



Esmaeel Moradi

@esmaeel-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



Esmaeeil Moradi

@esmaeeil-moradi700