

# Coroutine

Programmazione asincrona in Kotlin

# Introduzione

---

- Le **coroutines** sono un *concurrency design pattern* che si può usare su Kotlin (e su Android) per semplificare l'esecuzione di **codice asincrono**
- Sono state introdotte stabilmente nella versione 1.3 di Kotlin
- Le coroutine aiutano a gestire task di lunga durata (long-running tasks) che potrebbero altrimenti bloccare il main thread e, su Android, causare una perdita di responsività dell'app

# Kotlin le definisce..

---

- Una coroutine è un'istanza di calcolo sospendibile (*suspendable computation*)
- È concettualmente simile a un thread, nel senso che prevede l'esecuzione di un blocco di codice in maniera simultanea rispetto al resto del programma
- Tuttavia, una coroutine non è legata a nessun particolare thread: la sua esecuzione può essere sospesa in un thread e riprendere in un altro

# Funzionalità

Una coroutine si può sospendere (suspend) senza occupare il thread: quando incontra una chiamata suspend, il suo stato viene salvato su heap e il thread viene liberato per altri task. Al resume, lo stato viene ripristinato e l'esecuzione riprende (anche su un thread diverso). Poiché non serve un thread dedicato per ogni task, si possono lanciare migliaia di coroutine con un sovraccarico di poche decine di byte ciascuna. Ogni coroutine viene eseguita all'interno di uno CoroutineScope: la gerarchia genitore-figlio è automaticamente tracciata. Se un genitore termina (sia per completamento o per errore), tutti i figli vengono cancellati (structured concurrency), evitando "coroutine orfane" e potenziali memory leak.

- Le coroutine sono la soluzione consigliata per la programmazione asincrona su Android
- Le loro caratteristiche più rilevanti sono:
  - **Leggerezza:** si possono eseguire molte coroutine su un singolo thread grazie al supporto alla *sospensione*: l'esecuzione di una coroutine può essere sospesa e ripresa senza bloccare il thread in cui è in esecuzione. La sospensione consente di risparmiare memoria rispetto al blocco supportando al tempo stesso molte operazioni simultanee.
  - **Meno perdite di memoria:** tramite la concorrenza strutturata, ogni coroutine esegue operazioni all'interno di un determinato ambito (*scope*).
  - **Supporto alla cancellazione:** la cancellazione di una coroutine viene propagata automaticamente attraverso la gerarchia coroutine in esecuzione. fermando tutte le coroutine figlie in modo cooperativo
  - **Integrazione con Jetpack Compose:** molte librerie Jetpack includono estensioni che forniscono un supporto completo per le coroutine. Alcune librerie forniscono anche il proprio coroutine scope che è possibile utilizzare per la concorrenza strutturata.

Le coroutine devono essere lanciate all'interno di un CoroutineScope che ne delimita in modo esplicito il ciclo di vita. Ogni scope definisce un contesto (dispatcher, job padre, dispatcher, ecc.) e funge da contenitore: Il Job associato allo scope coordina tutte le coroutine figlie. Un scope viene solitamente creato tramite costruttori o attraverso extension. Lanciare coroutine in GlobalScope (che non ha padre) è sconsigliato perché sfugge al modello strutturato e rende impossibile un controllo centralizzato

# Approfondimento: structured concurrency

- Le coroutine seguono un principio di concorrenza strutturata (*structured concurrency*): una coroutine può essere eseguita solo in uno specifico CoroutineScope che ne delimita la durata
- In un'applicazione reale è comune lanciare varie coroutine. La concorrenza strutturata assicura che siano tracciate e non vadano perse portando a dei memory leak.  
è un particolare tipo di consumo non voluto di memoria dovuto alla mancata deallocazione dalla stessa di variabili o dati non più utilizzati da parte dei processi.
- Uno scope esterno non può essere completato finché tutte le sue coroutine figlie non sono state completate.
- La concorrenza strutturata garantisce inoltre che eventuali errori nel codice vengano segnalati correttamente e non vadano mai persi.

# Per capire meglio... Un esempio

- Nella nostra app abbiamo una schermata di login
- Vogliamo effettuare il login tramite una richiesta di rete e restituire il risultato al main thread, dove l'app può mostrare il risultato all'utente
- Abbiamo una **LoginRepository** con una funzione per effettuare la richiesta di login tramite HTTP
- E un **LoginViewModel** che richiama la funzione della repository, lanciando la richiesta di rete

# Esempio - LoginRepository

```
class LoginRepository(private val responseParser: LoginResponseParser) {
    companion object {
        private const val loginUrl = "https://example.com/login"
    }

    // Make the network request. This blocks the current thread.
    fun makeLoginRequest(jsonBody: String): Result<LoginResponse> {
        val url = URL(loginUrl)
        (url.openConnection() as? HttpURLConnection)?.run {
            requestMethod = "POST"
            setRequestProperty("Content-Type", "application/json; utf-8")
            setRequestProperty("Accept", "application/json")
            doOutput = true
            outputStream.write(jsonBody.toByteArray())
            return Result.Success(responseParser.parse(inputStream))
        }
        return Result.Error(Exception("Cannot open HttpURLConnection"))
    }
}
```

# Esempio - LoginViewModel

---

```
class LoginViewModel(  
    private val repository: LoginRepository  
) : ViewModel() {  
    fun login(username: String, token: String) {  
        val jsonBody = "{ username: \"$username\", token: \"$token\" }"  
        repository.makeLoginRequest(jsonBody)  
    }  
}
```



# Esempio

---

- Con il codice precedente, **LoginViewModel** blocca il thread dell'interfaccia utente per l'intera durata della richiesta di rete
- Poiché il thread è bloccato, il sistema operativo non è in grado di disegnare l'interfaccia, il che causa il blocco dell'app e potenzialmente porta alla finestra di dialogo «Application Not Responding»
- Per una migliore esperienza utente, bisogna eseguire l'operazione su un thread in background
- La soluzione più semplice per spostare l'esecuzione fuori dal main thread è creare una nuova coroutine ed eseguire la richiesta di rete su un thread I/O
- Ovvero...

# Esempio - LoginViewModel v2

```
class LoginViewModel(  
    private val repository: LoginRepository  
) : ViewModel() {  
    fun login(username: String, token: String) {  
        // Create a new coroutine to move the execution off the UI thread  
        viewModelScope.launch(Dispatchers.IO) {  
            val jsonBody = "{ username: \"$username\", token: \"$token\" }"  
            repository.makeLoginRequest(jsonBody)  
        }  
    }  
}
```

# Esempio

---

- **viewModelScope** è un CoroutineScope predefinito incluso nelle estensioni ViewModel KTX (KoTlin eXtension)
- **launch** è una funzione che crea una coroutine e invia l'esecuzione del corpo della sua funzione al dispatcher corrispondente
- **Dispatchers.IO** indica che questa coroutine deve essere eseguita su un thread riservato alle operazioni di I/O

# Ricapitolando

---

- La funzione login viene eseguita come segue:
  - L'app chiama la funzione di login dal View layer nel main thread
  - **launch** crea una nuova coroutine e la richiesta di rete viene effettuata in modo indipendente su un thread riservato alle operazioni di I/O
  - Mentre la coroutine è in esecuzione, la funzione login continua l'esecuzione e ritorna, probabilmente prima che la richiesta di rete sia terminata

# Coroutine e scope

- Poiché questa coroutine viene avviata con **viewModelScope**, viene eseguita nell'ambito del ViewModel
- Se il ViewModel viene distrutto, ad esempio perché l'utente naviga via dalla schermata di login, il **viewModelScope** viene terminato automaticamente, così come tutte le coroutine in esecuzione
- Questo nella maggior parte dei casi è il comportamento corretto, tuttavia è una cosa di cui è bene essere consapevoli

# Coroutine e main-safety

- Un problema con l'esempio precedente è che, al momento della chiamata a **makeLoginRequest**, bisogna sempre ricordarsi di spostare esplicitamente l'esecuzione fuori dal main thread (in questo caso tramite **Dispatchers.IO**)
- La funzione non è quindi *main-safe*, poiché chiamarla dal main thread blocca il rendering dell'interfaccia utente
- Una funzione si definisce *main-safe* quando **non blocca l'esecuzione del main thread** e quindi degli aggiornamenti dell'interfaccia utente

# Coroutine e main-safety

- Per risolvere questo problema, possiamo modificare la repository in modo da spostare l'esecuzione su **Dispatchers.IO** direttamente dal corpo di **makeLoginRequest**

```
// Make the network request, moving the execution to the I/O dispatcher.
suspend fun makeLoginRequest(jsonBody: String): Result<LoginResponse> =
    withContext(Dispatchers.IO) {
        val url = URL(loginUrl)
        (url.openConnection() as? HttpURLConnection)?.run {
            requestMethod = "POST"
            setRequestProperty("Content-Type", "application/json; utf-8")
            setRequestProperty("Accept", "application/json")
            doOutput = true
            outputStream.write(jsonBody.toByteArray())
            return@withContext Result.Success(responseParser.parse(inputStream))
        }
        return@withContext Result.Error(Exception("Cannot open HttpURLConnection"))
    }
```

# Coroutine e main-safety

- **withContext(Dispatchers.IO)** sposta l'esecuzione della coroutine in un thread di I/O, rendendo la nostra funzione main-safe
- **makeLoginRequest** è anche contrassegnata con **suspend**. Questa keyword obbliga a richiamare la <sup>funzione</sup> ~~funziona~~ dall'interno di una coroutine



# Controllo granulare

controllo preciso e dettagliato degli accessi a risorse o dati, dove ogni utente o gruppo riceve permessi specifici per le risorse di cui necessita, e niente più

- Questo meccanismo di switching con `withContext` offre un controllo estremamente granulare dell'esecuzione del codice
- Consente di controllare il pool di thread di qualsiasi riga di codice e si può applicare lettura da un database, esecuzione di una richiesta di rete e ogni sorta di funzionalità anche molto piccola

# Dispatcher

- Per specificare dove devono essere eseguite le coroutine, Kotlin fornisce tre dispatcher:
  - **Dispatchers.Main**: esegue una coroutine sul main thread. Dovrebbe essere usato solo per interagire con l'interfaccia utente ed eseguire task di breve durata
  - **Dispatchers.IO**: ottimizzato per eseguire I/O su disco o rete al di fuori del main thread. Alcuni esempi includono l'utilizzo di Room, la lettura o la scrittura su file e l'esecuzione di qualsiasi operazione di rete
  - **Dispatchers.Default**: ottimizzato per eseguire operazioni a uso intensivo della CPU al di fuori del main thread. Esempi di casi d'uso includono l'ordinamento di un elenco o il parsing di dati in JSON

# Coroutine e main-safety

- A questo punto, dato che `makeLoginRequest` sposta l'esecuzione fuori dal main thread, non è necessario fare la stessa cosa nel ViewModel

```
fun login(username: String, token: String) {  
    // Create a new coroutine to move the execution off the UI thread  
    viewModelScope.launch {  
        val jsonBody = "{ username: \"$username\", token: \"$token\" }"  
        repository.makeLoginRequest(jsonBody)  
    }  
}
```

Dispatchers.IO non è più necessario

# Una nota

"Sospensione" è diverso da "blocco": quando la coroutine incontra un punto di sospensione (`suspend/withContext`), salva il suo stato e rilascia il thread: il main thread può continuare a gestire il rendering UI o altre coroutines. Con le coroutines, ogni `suspend` è progettato per "mettere in pausa" solo la coroutine, non il thread, garantendo UI fluida e reattiva.

- Da notare che la coroutine è ancora necessaria poiché **makeLoginRequest** è una *suspend function* e tutte le funzioni `suspend` devono essere eseguite in una coroutine (quindi è necessario l'utilizzo di una coroutine per avviare una funzione `suspend`, cioè usare `viewModelScope.launch`)
- La funzione di login ora viene eseguita come segue:
  - L'app chiama la funzione **login()** dal livello View sul main thread
  - **launch** crea una nuova coroutine, la cui esecuzione inizia sul main thread
  - All'interno della coroutine, la chiamata a **repository.makeLoginRequest()**
  - sospende la coroutine fino a quando il blocco **withContext** in **makeLoginRequest()**, eseguito su un thread separato, non termina
  - Una volta terminato il blocco **withContext**, la coroutine in **login()** riprende l'esecuzione sul main thread con il risultato della richiesta di rete

# Più in generale

- L'uso di `suspend` non dice a Kotlin di eseguire una funzione su un thread in background, ma solo che l'esecuzione di quella funzione potrebbe essere sospesa
- È normale che le funzioni `suspend` operino sul main thread
- È anche comune lanciare coroutine sul main thread
- Se una funziona necessita di main-safety, allora va utilizzato **`withContext()`**
- Una limitazione di questo sistema è che non sempre è facile sapere se una funzione internamente utilizza già **`withContext()`** ed è quindi main-safe
  - Ad esempio, per le funzioni provenienti da librerie di terze parti è buona pratica verificare questa cosa consultando la relativa documentazione
- • Ma allora perché non usare **`withContext()`** in ogni suspend function che esegue operazioni di I/O, per sicurezza?
  - Non è una buona idea. Ogni chiamata a **`withContext()`** crea un punto di sospensione della coroutine e può portare allo switch su un altro thread, anche quando la coroutine è già assegnata al dispatcher corretto

Quando chiami `withContext`, la coroutine viene sospesa, il lavoro viene schedato su un altro thread e al termine si torna al contesto originario. Se fai 10 chiamate distinte a `withContext(Dispatchers.IO)` in 10 funzioni diverse, rischi fino a 10 sospensioni e 10 context-switch superflui, con un overhead cumulativo che in scenari ad alto throughput può degradare le performance anche del 20/30%

# Ultimi ritocchi

- Nel ViewModel, gestiamo eventuali errori ed aggiorniamo lo stato della UI di conseguenza

```
class LoginViewModel(
    private val repository: LoginRepository
): ViewModel() {
    fun login(username: String, token: String) {
        // Create a new coroutine to move the execution off the UI thread
        viewModelScope.launch {
            val jsonBody = """{ username: "$username", token: "$token"}"""
            val res = repository.makeLoginRequest(jsonBody)
            when (res) {
                is Result.Success<LoginResponse> -> // Happy path
                is Result.Error -> // Display error in the UI
            }
        }
    }
}
```

# Ultimi ritocchi

- Nel ViewModel, gestiamo eventuali errori ed aggiorniamo lo stato della UI di conseguenza

```
class LoginViewModel(
    private val repository: LoginRepository
): ViewModel() {
    fun login(username: String) {
        // Create a new coroutine on the UI thread
        viewModelScope.launch {
            val jsonBody = """{"username": "$username", "password": "$password", "token": "$token"}"""
            val res = repository.makeLoginRequest(jsonBody)
            when (res) {
                is Result.Success<LoginResponse> -> // Happy path
                is Result.Error -> // Display error in the UI
            }
        }
    }
}
```

FINE DELL'ESEMPIO

# Avvio di una coroutine

- Una coroutine può essere avviata in due modi:
  - Tramite **launch**: per task fire-and-forget, che **non restituiscono alcun valore**. **launch** restituisce un **Job** che può essere utilizzato per tracciare l'esecuzione della coroutine (attendere la terminazione, cancellarla, ecc.)
  - Tramite **async**: per task che **restituiscono un risultato**. **async** restituisce un **Deferred<T>**, e possiamo utilizzare **.await()** per attendere la terminazione della coroutine e recuperarne il valore di ritorno



# Gestione delle eccezioni

- **launch** e **async** gestiscono le eccezioni in modo diverso
  - Con **launch**, eventuali eccezioni vengono propagate immediatamente
  - Poiché **async** prevede un'eventuale chiamata ad **await**, eventuali eccezioni vengono lanciate solo al momento di tale chiamata. Ciò significa che utilizzando **async** in modo errato è possibile ignorare silenziosamente un'eccezione. È indispensabile chiamare sempre **await** (o funzioni simili che ne attendono il completamento, vedi prossima slide)

# Esempio - launch

- Definiamo un **coroutineScope** che effettua l'upload di due documenti in modo asincrono. Dato che non ci interessa il valore di ritorno delle funzioni, utilizziamo **launch**. Chiamando **join()** su ogni **Job**, blocchiamo l'esecuzione della funzione in attesa che le operazioni siano terminate

```
suspend fun uploadDocs() = coroutineScope {  
    val job1 = launch { uploadDoc(1) }  
    val job2 = launch { uploadDoc(2) }  
    job1.join()  
    job2.join()  
}
```

# Esempio - launch

- In alternativa, possiamo creare una lista di **Job** e attendere il completamento di tutti gli elementi con **joinAll()**

```
suspend fun uploadDocs() = coroutineScope {  
    val jobs = listOf(  
        launch { uploadDoc(1) },  
        launch { uploadDoc(2) }  
    )  
    jobs.joinAll()  
}
```

# Esempio - async

- Definiamo un **coroutineScope** che recupera due documenti in modo asincrono. Vogliamo accedere ai documenti restituiti dalle funzioni, quindi **utilizziamo async**. Chiamando **await()** su ogni **Deferred**, garantiamo che entrambe le operazioni asincrone terminino e recuperiamo il documento da ciascuna

```
suspend fun fetchDocs() = coroutineScope {  
    val deferred1 = async { fetchDoc(1) }  
    val deferred2 = async { fetchDoc(2) }  
    val doc1 = deferred1.await()  
    val doc2 = deferred2.await()  
}
```

# Esempio - async

- In alternativa, possiamo creare una lista di **Deferred** e attendere il completamento di tutti gli elementi con **awaitAll()**

```
suspend fun fetchDocs() = coroutineScope {  
    val deferreds = listOf(  
        async { fetchDoc(1) },  
        async { fetchDoc(2) }  
    )  
    val docs = deferreds.awaitAll()  
}
```

- Da notare che, nei quattro esempi precedenti, **uploadDoc(1)**, **uploadDoc(2)**, **fetchDoc(1)** e **fetchDoc(2)** vengono **eseguite in maniera concorrente**. I due documenti vengono quindi caricati (o scaricati) contemporaneamente.

# CoroutineScope

- Un **CoroutineScope** tiene traccia di qualsiasi coroutine creata al suo interno utilizzando **launch** o **async** e ne controlla l'avvio e la cancellazione
  - Non stabilisce, tuttavia, in quale thread vengono eseguite le coroutine. Questo compito è lasciato al dispatcher
- In qualsiasi momento, tutte le coroutine in esecuzione possono essere annullate chiamando **scope.cancel()**
- In Android, alcune librerie KTX forniscono il proprio **CoroutineScope** per determinate classi del ciclo di vita
  - Ad esempio, ViewModel ha un **viewModelScope**, Lifecycle ha **lifecycleScope** e i composable hanno **rememberCoroutineScope**

Le estensioni KTX forniscono automaticamente un CoroutineScope pre-configurato solo per quelle classi di Android i cui oggetti hanno un ciclo di vita tracciato dal framework

Ad esempio: Ogni ViewModel riceve in dotazione, grazie a lifecycle-viewmodel-ktx, un scope legato al suo Job interno: tutte le coroutine lanciate lì vengono cancellate automaticamente quando il ViewModel viene distrutto (metodo onCleared()).

# Esempio - CoroutineScope e ViewModel

```
class LoginViewModel(  
    private val repository: LoginRepository  
) : ViewModel() {  
    fun login(username: String, token: String) {  
        // Launch a coroutine in the ViewModel's scope  
        viewModelScope.launch {  
            val jsonBody = """{ username: "$username", token: "$token"}"""  
            val res = repository.makeLoginRequest(jsonBody)  
            when (res) {  
                is Result.Success<LoginResponse> -> // Happy path  
                is Result.Error -> // Display error in the UI  
            }  
        }  
    }  
}
```

# Esempio - CoroutineScope e Composable

```
@Composable
fun MyScreen(snackbarHostState: SnackbarHostState) {
    // Creates a CoroutineScope bound to MyScreen's lifecycle
    val scope = rememberCoroutineScope()

    Scaffold(
        snackbarHost = { SnackbarHost(hostState = snackbarHostState) }
    ) { innerPadding ->
        Column(Modifier.padding(innerPadding)) {
            Button(onClick = {
                // Launch a coroutine in the created scope
                scope.launch {
                    snackbarHostState.showSnackbar("Something happened!")
                }
            }) { Text("Press me") }
        }
    }
}
```



# CoroutineContext

- Un **CoroutineContext** è un set di elementi che definisce il comportamento di una coroutine. Include:
  - **Job**: traccia il ciclo di vita della coroutine
  - **CoroutineDispatcher**: determina in quale thread viene eseguita la coroutine
  - **CoroutineName**: utile in fase di debugging
  - **CoroutineExceptionHandler**: gestisce le eccezioni non rilevate
- Alla creazione di una coroutine in un certo scope, una nuova istanza di **Job** viene assegnata alla coroutine. Gli altri elementi del context vengono ereditati dallo scope, o possono essere specificati manualmente

```
val scope = CoroutineScope(Dispatchers.Main + Job())
scope.launch { println("Running on Main Thread") }
scope.launch(Dispatchers.Default + CoroutineName("BackgroundCoroutine")) {
    println("Running in the background")
}
```

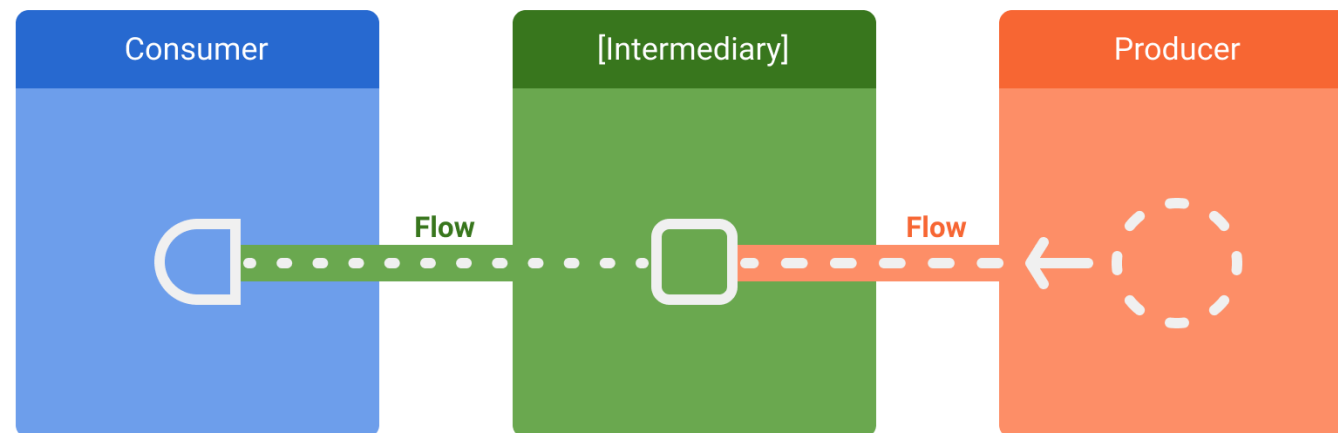
# Flow, StateFlow e Compose State

# Coroutine che emettono valori multipli

- E se volessimo eseguire una coroutine che propaga verso l'esterno più valori nel corso della sua esecuzione?
- Potremmo farlo con un **Flow**
- Un **Flow** è un tipo di dato che rappresenta uno stream di valori, emessi sequenzialmente nel tempo, ciascuno dei quali proviene dall'esecuzione di una suspend function che non blocca quindi il thread chiamante
  - A differenza delle suspend function, può emettere più di un valore
  - È una sorta di versione asincrona e con supporto multi-thread delle **Sequence** di Java

# Flow

- Un **Flow** coinvolge tre tipologie di entità:
  - Un **producer**, che genera i dati che vengono emessi dal flusso
  - (Facoltativo) Uno o più **intermediari**, che modificano ciascun valore emesso nel flusso o il flusso stesso  
Sono gli step di trasformazione che agiscono sui valori emessi dal producer, rimandando un nuovo Flow downstream. Consentono di combinare, filtrare, ritardare o raggruppare i dati prima che arrivino al consumer.
  - Un **consumer** che riceve i valori emessi dal flusso e manipolati dai ~~consumer~~ <sup>intermediari</sup> e agisce di conseguenza  
È il terminale dello stream, che riceve i valori "finali" e li elabora.



# Esempio - Flow

Quando chiami `emit(latestNews)` stai in quel preciso istante trasferendo il valore downstream, cioè verso tutti gli operatori intermedi e infine verso il consumer che sta raccogliendo il flusso. Non c'è accumulo o "coda interna". Se non chiami mai un terminal operator (ad esempio `collect`), il blocco flow non viene eseguito e nessun `emit` avviene mai: i Flow sono cold per default.

- Utilizziamo un **Flow** per creare una data source che recupera automaticamente le notizie più recenti a intervalli prefissati
  - Poiché **fetchLatestNews** non può restituire più valori consecutivi, utilizziamo un **Flow** per soddisfare questo requisito. In questo caso **NewsRemoteDataSource** è il nostro producer

```
class NewsRemoteDataSource(  
    private val newsApi: NewsApi,  
    private val refreshIntervalMs: Long = 5000  
) {  
    val latestNews: Flow<List<Article>> = flow {  
        while(true) {  
            val latestNews = newsApi.fetchLatestNews() // Call the suspend function  
            emit(latestNews) // Emit the result of the request to the flow  
            delay(refreshIntervalMs) // Suspend the coroutine for some time  
        }  
    }  
}
```

```
interface NewsApi {  
    suspend fun fetchLatestNews(): List<Article>  
}
```

# Collecting da un Flow (consumer)

- Per ottenere i valori nel **Flow** man mano che vengono emessi ed agire di conseguenza, possiamo usare la suspend function **collect**, che prende un lambda come parametro e la esegue su ogni nuovo valore emesso dal **Flow**
- Nel seguente esempio, un ViewModel consuma i dati dal livello del repository

```
class LatestNewsViewModel(private val repository: NewsRepository) : ViewModel() {  
    // Run some code when the ViewModel is instantiated  
    init {  
        viewModelScope.launch {  
            // Trigger the flow and consume its elements using collect  
            repository.favoriteLatestNews.collect { news ->  
                // Update View with the latest favorite news  
            }  
        }  
    }  
}
```

# StateFlow

- Ma come facciamo, in questo caso, a memorizzare lo stato dell'interfaccia nel ViewModel?
- Possiamo utilizzare uno **StateFlow**
- Come **Flow**, **StateFlow** rappresenta un flusso di dati, ma con alcune importanti differenze che lo rendono molto indicato all'uso in un ViewModel:
  - Ha un valore iniziale
  - Tiene traccia dell'ultimo valore emesso
  - Può essere condiviso da più consumer ed emettere valori anche quando non ha nessun consumer
  - Smette di emettere valori solo quando viene raccolto dal garbage collector, ossia quando non ci sono più riferimenti ad esso
  - Ogni consumer che sottoscrive al flusso riceve l'ultimo valore e tutti i successivi

# Da Flow a StateFlow v1

- Arricchiamo l'ultimo esempio utilizzando StateFlow

```
data class LatestNewsState(val news: List<Article>)

class LatestNewsViewModel(private val repository: NewsRepository) : ViewModel() {
    // Backing property to avoid state updates from other classes
    private val _state = MutableStateFlow(LatestNewsState(emptyList()))
    // Read-only flow, collectable from the UI
    val state = _state.asStateFlow()

    init {
        viewModelScope.launch {
            repository.favoriteLatestNews.collect { favoriteNews ->
                // Update the flow with the latest favorite news
                _state.value = LatestNewsState(favoriteNews)
            }
        }
    }
}
```



# Da Flow a StateFlow v2

- In alternativa, tramite **stateIn**, possiamo convertire direttamente il **Flow** in **StateFlow**

```
class LatestNewsViewModel(private val repository: NewsRepository) : ViewModel() {  
    val state = repository.favoriteLatestNews.stateIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed(),  
        initialValue = LatestNewsState(emptyList())  
    )  
}
```

# State e lifecycle-aware collect

- Ma che fine hanno fatto **State** e **MutableState**? Possiamo farne a meno?
- No! **State** resta l'unico metodo di tracciare i cambiamenti di stato all'interno di Jetpack Compose
- Salvo casi particolari, è necessario convertire ogni **Flow** e **StateFlow** in **State** prima di utilizzarlo in un composable
- La conversione è semplice, basta utilizzare la funzione **collectAsStateWithLifecycle()**

# Da StateFlow a State

- Creiamo un composable che utilizza **LatestNewsViewModel**

```
@Composable
fun NewsApp() {
    val viewModel = viewModel<LatestNewsViewModel>()
    val state by viewModel.state.collectAsStateWithLifecycle()
    LazyColumn {
        items(state.news) { article ->
            ArticleCard(article)
        }
    }
}
```

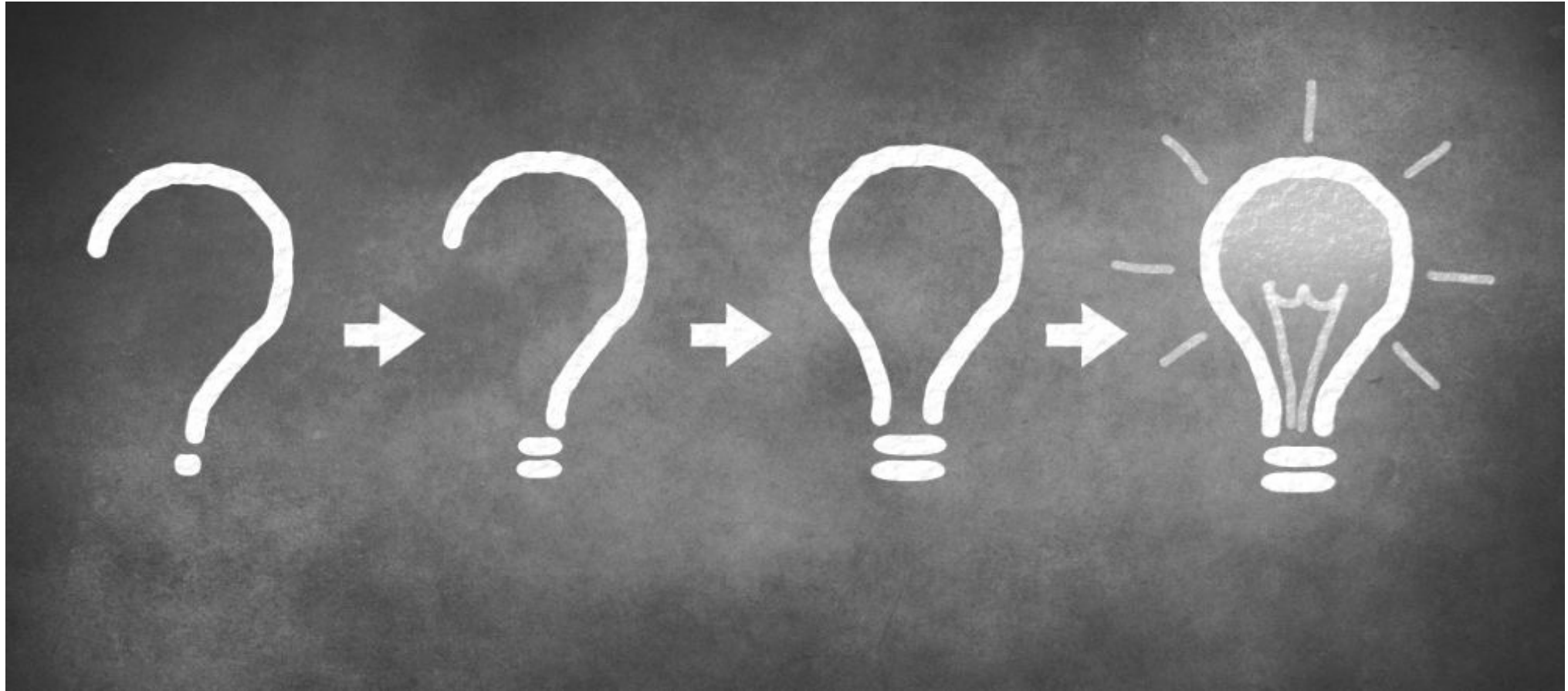
Colleziona un StateFlow o Flow in modo lifecycle-aware (ossia: osserva solo quando il composable è attivo).

- state rappresenta un State<NewsUiState> derivato da un Flow.
- Quando il ViewModel emette un nuovo NewsUiState nel state (che sarà un StateFlow), collectAsStateWithLifecycle() lo intercetta.
- Jetpack Compose nota il cambiamento e ricompone solo la parte interessata del tree — in questo caso, il LazyColumn.

# Recap: Flow vs. StateFlow vs. Compose State

Caratteristica	Flow	StateFlow	State
Package	kotlinx.coroutines.flow	kotlinx.coroutines.flow	androidx.compose.runtime
Thread-safe	✓	✓	✗
Cache	✗ Non tiene traccia dei valori emessi	✓ Traccia l'ultimo valore	✓ Memorizza un valore
Valore iniziale	✗ Nessun valore iniziale	✓ Richiede un valore iniziale	✓ Richiede un valore iniziale
Reattività	❄️ Cold: emette valori solo quando ha dei consumer	🔥 Hot: sempre attivo	🔥 Hot: sempre attivo
Consumer multipli	✓ Emissioni indipendenti per ogni consumer	✓ Tutti i consumer accedono allo stesso valore	✗ Non supportati
Compose lifecycle awareness	✗	✗	✓ Recomposition al cambio di stato
Operatori per la manipolazione	✓ emit, map, filter, ...	✓ value, update, map, filter, ...	✗ Solo getter e setter
Per cosa usarlo	Dati asincroni (API, database, ...)	Stato condiviso tra più consumer	Stato di un composable
Dove usarlo	Repository / data source	ViewModel	Composable function

# Domande?



# Riferimenti e Approfondimenti

---

- <https://developer.android.com/kotlin/coroutines>
- <https://medium.com/androiddevelopers/coroutines-first-things-first-e6187bf3bb21>
- <https://medium.com/androiddevelopers/cancellation-in-coroutines-aa6b90163629>
- <https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471>
- <https://kotlinlang.org/docs/coroutines-guide.html>
- <https://developer.android.com/kotlin/flow>