**Kotlin Coroutine**

Mari kita bahas sedikit tentang Coroutine sebelum masuk lebih dalam tentang Flow. Coroutine adalah library yang digunakan untuk melakukan proses asynchronous pada bahasa Kotlin Native. Ia mengubah callback yang biasanya digunakan untuk menerima data menjadi kode yang lebih mudah dibaca, caranya yaitu dengan menerapkan **suspend function** seperti berikut:

1. // Tanpa coroutine
2. fun loadUser() {
3. api.fetchUser { user -> //callback
4. textView.text = user.name
5. }
6. }
8. // Dengan coroutine
9. suspend fun loadUser() = coroutineScope {
10. val user = async { api.fetchUser() }.await
11. textView.text = user.name
12. }

Ada dua jenis fungsi yang bisa digunakan untuk menggunakan coroutine, yaitu:

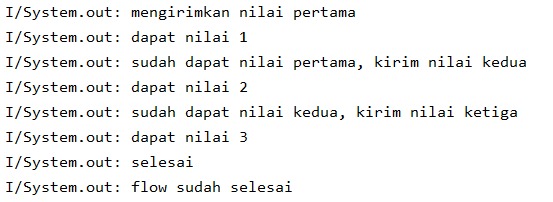
* **async**: untuk proses dengan nilai kembalian.
* **launch**: untuk proses tanpa nilai kembalian.

**Konsep Dasar**

Coroutine memang library yang keren, namun Coroutine hanya bisa mengembalikan *single value* saja. Bagaimana jika ingin mengembalikan *multiple-value* seperti data stream? Di sinilah Flow hadir sebagai solusi.  
  
Berikut ini adalah contoh simpel penggunaan Flow. Anda dapat mencobanya di Android Studio dan menjalankannya.

|  |
| --- |
| 1. fun makeFlow() = flow { 2. println("mengirimkan nilai pertama") 3. emit(1) 4. println("sudah dapat nilai pertama, kirim nilai kedua") 5. emit(2) 6. println("sudah dapat nilai kedua, kirim nilai ketiga") 7. emit(3) 8. println("selesai") 9. } 11. GlobalScope.launch { 12. makeFlow().collect { value -> 13. println("dapat nilai $value") 14. } 15. println("flow sudah selesai") 16. } |

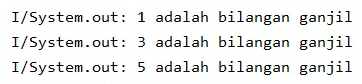
* flow{}: adalah builder untuk membuat Flow.
* emit : operator untuk mengirimkan value ke dalam Flow.
* collect (wajib ada) : terminal operator (operator terakhir) untuk mengambil hasil akhir data (hampir sama dengan subscribe pada RxJava).

Hasil dari kode di atas adalah seperti berikut:  
[](https://www.dicoding.com/academies/165/tutorials/10344?from=10339)

Anda dapat melihat bahwa nilai dikirimkan satu per satu sehingga menghasilkan log yang berurutan seperti di atas.  
Flow memiliki fungsi yang mirip dengan tipe Flowable pada RxJava sehingga Anda dapat membuat dan mengubah data yang dikirim dengan operator seperti map, filter, dan operator lainnya seperti berikut.

1. GlobalScope.launch {
2. flowOf("1", "2", "3", "4", "5", "6")
3. .map { string -> string.toInt() }
4. .filter { number -> number % 2 == 1 }
5. .collect { result -> println("$result adalah bilangan ganjil") }
6. }

* flowOf: adalah builder untuk mendeklarasikan Flow dengan lebih cepat.
* map, filter : operator untuk transformasi dan filter data.
* collect (wajib ada) : salah satu terminal operator untuk mengambil setiap data.

Berikut adalah hasil dari kode di atas jika dilihat dari Logcat:  
[](https://www.dicoding.com/academies/165/tutorials/10344?from=10339)

Output 3 baris awal adalah hasil log pada fungsi collect jika dilihat hasil keluar satu per satu. Kenapa ya? Tentu karena konsep reactive sendiri adalah data di-*emit*satu per satu.

Selain **collect**, ada juga **terminal operator** lain yang berfungsi untuk menjalankan flow dan mengumpulkan hasil akhir sesuai dengan kebutuhan Anda. Contohnya yaitu seperti berikut:

* **toList** & **toSet**: untuk mengubah hasil akhir menjadi list.
* **first**: untuk mengambil value pertama saja.
* **single** : untuk memastikan value pertama saja.
* **reduce** & **fold**: untuk mengurangi banyak flow menjadi satu value saja.

Berikut ini adalah contoh penerapan terminal operator count:

1. GlobalScope.launch {
2. val total = flowOf("1", "2", "3", "4", "5", "6")
3. .map { string -> string.toInt() }
4. .count { number -> number % 2 == 1 }
5. println("Total bilangan ganjil : $total")
6. }
7. /\* output:
8. Total bilangan ganjil : 3
9. \*/

Selain operator di atas, Anda dapat melihat fungsi lainnya secara lengkap di tautan berikut:

* [Flow - Extension Functions](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/index.html)

**Multi-Threading dengan Flow**

Okay, sekarang saatnya bermain dengan multi-threading. Seperti yang dijelaskan di awal modul, kita memerlukan multi-threading untuk membuat proses yang tidak memblok UI alias *freeze*. Dengan menggunakan Flow, Anda dapat berpindah antar thread dengan mudah dengan memanfaatkan **Dispatcher**dan bantuan operator **flowOn**.

Sebagai contohnya Anda ingin mengambil data karyawan dari API seperti ini:

1. CoroutineScope(Dispatchers.Main).launch {
2. getEmployeeNames()
3. .flowOn(Dispatchers.IO)
4. .collect { name -> println(name) }
5. }
6. ...
7. private fun getEmployeeNames(): Flow<String> {
8. val names = mutableListOf("Buchori", "Dimas", "Gheddi", "Gilang", "Widy")
9. return names.asFlow() //untuk membuat Flow dari Array
10. }
12. /\*Output :
13. Buchori
14. Dimas
15. Gheddi
16. Gilang
17. Widy
18. \*/

Anggaplah fungsi getEmployeeNames merupakan fungsi yang mengambil data dari server, yang tentunya memerlukan waktu untuk mengambilnya dan kemudian menampilkannya di UI. Maka kode yang perlu Anda tambahkan adalah :

* flowOn : untuk mendefinisikan proses mengirim data dilakukan di mana.
* CoroutineScope : untuk mendefinisikan proses menerima data dilakukan di mana.

Untuk menentukan thread yang digunakan, Anda bisa menggunakan beberapa pilihan Dispatcher berikut:

* **Dispatchers.Default**, adalah default dispatcher jika tidak ada Dispatcher yang didefinisikan, cocok untuk fungsi yang membutuhkan proses CPU yang tinggi, seperti parsing 100 data.
* **Dispatchers.IO**, untuk menjalankan fungsi yang berisi read-write data ke Network/Disk, seperti menulis atau mengambil data ke database dan ke server.
* **Dispatchers.Main**, untuk menjalankan fungsi di Main Thread, biasanya digunakan untuk mengupdate View UI.

Pada kode di atas proses emitting dilakukan di Thread IO dan hasilnya ditampilkan pada Main Thread. Seperti itulah biasanya proses multi-threading untuk mengambil data dari server dilakukan.

Selamat! Anda telah selangkah lebih maju untuk belajar menggunakan library Coroutine Flow dan penerapannya. Seperti biasa kami menyarankan Anda mendalami lebih jauh tentang Coroutine Flow melalui tautan di bawah ini:

* [Coroutine Flow Documentation](https://kotlinlang.org/docs/reference/coroutines/flow.html)
* [Understand Kotlin Coroutines on Android (Google I/O'19)](https://www.youtube.com/watch?v=BOHK_w09pVA)
* [KotlinConf 2019: Asynchronous Data Streams with Kotlin Flow by Roman Elizarov](https://www.youtube.com/watch?v=tYcqn48SMT8)
* [Kotlin Flow API ~ Android cheat sheet by Remy Benza](https://drive.google.com/file/d/1abawxLC2KkYds3wuRkS4oEOXde1eDYo-/view)