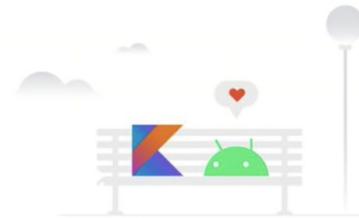


1. The alternative of AsyncTask
 - a. Finally deprecated in Android 11!

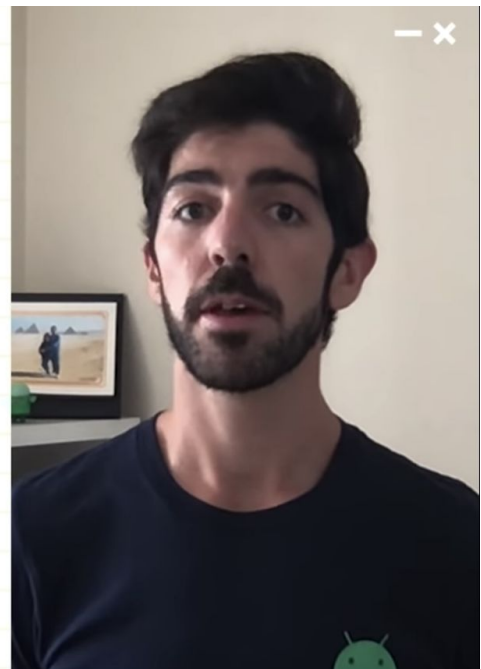
Kotlin coroutines are the recommended solution for async code.



- b. Android official document says to use Concurrent Class of Java or Kotlin concurrency utilities. (Kotlin Coroutines)

Android ❤️ coroutines

- **Structured concurrency**
- **Callback-free code**
- **Cancellation support**
- **Lightweight**



- c. Why coroutines?


- i. The unique properties of coroutines -> make it apps well-suited
- ii. Structured Concurrency
 - Helps developers scope their works to application's component and prevent memory leaks
- iii. Callback-free code
 - Higher readability and understandability
- iv. Cancellation Support and natural exception handling
- v. Lightweight
 - Many coroutines run on the single thread, suspending their work instead of blocking
 - Making them fast and lightweight

2. Practical Examples

WorkManager

Worker

```
override fun doWork(): ListenableWorker.Result {  
    try {  
        val dbData = readFromDb()  
        val serverData = uploadToServer(data)  
        writeToDb(serverData)  
    } catch (e: IOException) {  
        return ListenableWorker.Result.failure()  
    }  
  
    return ListenableWorker.Result.success()  
}
```



- a. WorkManager with synchronous job
 - But we can't stop the work at any moment.

```

override fun doWork(): ListenableWorker.Result {
    try {
        val dbData = readFromDb()
        if (isStopped()) return ListenableWorker.Result.retry()
        val serverData = uploadToServer(data)
        if (isStopped()) return ListenableWorker.Result.retry()
        writeToDb(serverData)
    } catch (e: IOException) {
        return ListenableWorker.Result.failure()
    }

    return ListenableWorker.Result.success()
}

```



- b. WorkManager with code handling 'stopped case'
 - i. But how to pass the signal to stop?

WorkManager

ListenableWorker

```

override fun startWork(): ListenableFuture<Result> {
    val futureDb: ListenableFuture<Data> = executor.submit{ readFromDb() }
    val futureSrv = Futures.transformAsync(futureDb, { uploadToServer(it) }, executor)
    return Futures.transformAsync(
        futureSrv, {
            writeToDb(it)
            ListenableWorker.Result.success()
        },
        executor
    )
}

```



- c. The other way to do asynchronous request using ListeneableFuture
 - i. Cancellation, and error propagation
 - ii. But it's hard to find where the actual work is being done.

WorkManager

RxWorker

```
override fun createWork(): Single<Result> {  
    return Single.just(readFromDb())  
        .subscribeOn(Schedulers.io())  
        .flatMap { dbData -> uploadToServer(dbData) }  
        .flatMap { serverData -> writeToDatabase(serverData) }  
        .map {  
            Result.success()  
        }  
}
```



WorkManager

RxWorker

```
override fun createWork(): Single<Result> {  
    return Single.just(readFromDb())  
        .subscribeOn(Schedulers.io())  
        .flatMap { dbData -> uploadToServer(dbData) }  
        .flatMap { serverData -> writeToDatabase(serverData) }  
        .map {  
            Result.success()  
        }  
}
```



- d. Still need a Java way, chaining calls.
- e. Have to understand the operators of RxJava
- f. Less readability

WorkManager

CoroutineWorker

```
override suspend fun doWork(): Result {  
    val data = readFromDb()  
    val serverData = uploadToServer(data)  
    writeToDb(serverData)  
  
    return Result.success()  
}
```



- g. Finally we use Coroutines!
 - i. Everything looks sequential and is readable
 - ii. Error propagation is more natural
 - iii. Try-catch if needed

Room

Database queries

```
@Dao  
interface UsersDao {  
    @Query("SELECT * FROM users")  
    suspend fun getUsers(): List<User>  
}
```




- h. Assuming that these operations shouldn't be run in the main thread.

- i. Adopting the same approaches in Room and other Jetpack Libraries, and 3rd party libraries such as Retrofit.

Your API

Make it main-safe

```
suspend fun mySuspendFunction() = withContext(Dispatchers.IO) {  
    // you can do blocking stuff here  
}  
  
// usage:  
viewModelScope.launch {  
    ...  
    mySuspendFunction()  
    ...  
}
```




- j. Suspend function is main-safe
 - i. safe to call from the main thread or the main dispatcher
 - ii. Non-blocking the thread

Room

Database

```
RoomDatabase.Builder.setQueryExecutor(executor: Executor)  
RoomDatabase.Builder.setTransactionExecutor(executor: Executor)  
  
// in coroutines library:  
fun Executor.asCoroutineDispatcher(): CoroutineDispatcher  
fun CoroutineDispatcher.asExecutor(): Executor
```



- k. Executor's rule is almost identical to the coroutine Dispatchers.

- I. There are methods to translate between them.

Your API

Injecting dispatchers

```
// provide a mechanism to configure the dispatcher somewhere
var dispatcher = Dispatchers.IO

suspend fun mySuspendFunction() = withContext(dispatcher) {
    // you can do blocking stuff here
}
```



Coroutine adapters

For common future APIs

```
//Awaits for completion of the completion stage without blocking a thread.
suspend fun <T> CompletionStage<T>.await(): T // (for CompletableFuture)

// Awaits completion of this ListenableFuture without blocking a thread.
suspend fun <T> ListenableFuture<T>.await(): T

// Awaits completion of this Task without blocking a thread.
suspend fun <T> Task<T>.await(): T
```




```
suspend fun <T> Task<T>.await(): T =
    suspendCancellableCoroutine { continuation ->
        addOnSuccessListener { result ->
            continuation.resume(result)
        }
        addOnFailureListener { exception ->
            continuation.resumeWithException(exception)
        }
    }
}
```



```
suspend fun <T> Task<T>.await(): T {
    if (isComplete) { // eagerly resume coroutine if Task is ready
        if (isSuccessful) {
            return result
        } else {
            throw exception!!
        }
    }
    return suspendCancellableCoroutine { continuation ->
        addOnSuccessListener { result ->
            continuation.resume(result)
        }
        addOnFailureListener { exception ->
            continuation.resumeWithException(exception)
        }
    }
}
```



- m. This `suspendCancellableCoroutine` is **to wrap** the callback-based libraries with **suspend functions**. It transforms callback functions to coroutine corresponding.