Best practices

> This is a chapter from the book Kotlin Coroutines. You can find it on LeanPub or Amazon.

I will finish this book with my humble set of best practices. They've all been discussed in the book already, so you might treat this as a brief summary, but I hope it will help you remember and apply them in your everyday practice.

Don't use async with an immediate await

It makes no sense to define an asynchronous task with `async` if we want to await its completion without doing any operations during that time.

```
// Don't
suspend fun getUser(): User = coroutineScope {
    val user = async { repo.getUser() }.await()
    user.toUser()
}

// Do
suspend fun getUser(): User {
```

```
    val user = repo.getUser()
    return user.toUser()
}
```

There are cases where this transformation is not so simple. If you need a scope, instead of `async { ... }.await()`, use `coroutineScope`. If you need to set a context, use `withContext`.

When you start a few async tasks, all of them except the last one need to use `async`. In this case, I suggest using `async` for all of them for readability.

```
fun showNews() {
    viewModelScope.launch {
        val config = async { getConfigFromApi() }
        val news = async { getNewsFromApi(config.await()) }
        val user = async { getUserFromApi() } // async not
        // necessary here, but useful for readability
        view.showNews(user.await(), news.await())
    }
}
```

Use `coroutineScope` instead of `withContext(EmptyCoroutineContext)`

The only difference between `withContext` and `coroutineScope` is that `withContext` can override context, so instead of `withContext(EmptyCoroutineContext)`, use `coroutineScope`.

Use awaitAll

The `awaitAll` function should be preferred over `map { it.await() }` because it stops waiting when the first async task throws an exception, while `map { it.await() }` awaits these coroutines one after another until this process reaches one that fails.

Suspending functions should be safe to call from any thread

When you call a suspending function, you shouldn't be worried that it might block the thread you're currently using. This is especially important on Android, where we often use `Dispatchers.Main`; however, it is also important on the backend, where you might be using a dispatcher limited to a single thread for synchronization.

Each suspending function that needs to make blocking calls should use `Dispatchers.IO` or a custom dispatcher that is designed to be blocked. Each dispatcher that might be CPU-intensive should use `Dispatchers.Default` or `Dispatchers.Default` with limited parallelism. These dispatchers should be set using `withContext` so that function calls don't need to set these dispatchers themselves.

```
class DiscSaveRepository(
    private val discReader: DiscReader
) : SaveRepository {

    override suspend fun loadSave(name: String): SaveData =
        withContext(Dispatchers.IO) {
            discReader.read("save/$name")
        }
}
```

Functions that return `Flow` should specify a dispatcher using `flowOn`, which changes the context for all the steps **above** it, so it is typically used as the last step in a function.

Whether or not `Dispatchers.Main.immediate` should be used explicitly in suspending functions that update Android views is a controversial topic. Your decision should depend on your project's policy. We don't need to use it in layers where `Dispatchers.Main` is considered the default dispatcher, like in the presentation layer in many Android projects.

If you want to unit test these classes, remember that you need to inject a dispatcher so it can be overridden for unit testing.


```
class DiscSaveRepository(
    private val discReader: DiscReader,
    private val dispatcher: CoroutineContext = Dispatchers.IO
) : SaveRepository {

    override suspend fun loadSave(name: String): SaveData =
        withContext(dispatcher) {
            discReader.read("save/$name")
        }
}
```

Use `Dispatchers.Main.immediate` instead of `Dispatchers.Main`

`Dispatchers.Main.immediate` is an optimized version of `Dispatchers.Main` that avoids coroutine redispatching if it isn't necessary. We generally prefer to use it.

```
suspend fun showUser(user: User) =
    withContext(Dispatchers.Main.immediate) {
        userNameElement.text = user.name
        // ...
    }
```

Remember to use `yield` in heavy functions

It is good practice to use `yield` in suspending functions between blocks of non-suspended CPU-intensive, blocking or time-intensive operations. This function suspends and immediately resumes the coroutine, thus it supports cancellation. Calling `yield` also allows redispatching, thanks to which one process will not starve other processes.

```
suspend fun cpuIntensiveOperations() =
    withContext(Dispatchers.Default) {
        cpuIntensiveOperation1()
        yield()
        cpuIntensiveOperation2()
        yield()
        cpuIntensiveOperation3()
    }

suspend fun blockingOperations() =
    withContext(Dispatchers.IO) {
        blockingOperation1()
        yield()
        blockingOperation2()
        yield()
        blockingOperation3()
    }
```

Inside coroutine builders, you can also use `ensureActive`.

Understand that suspending functions await completion of their children

A parent coroutine cannot complete before its children, and coroutine scope functions, like `coroutineScope` or `withContext`, suspend their parent until

their coroutines are completed. As a result, they await all the coroutines they've started.

```
suspend fun longTask() = coroutineScope {
    launch {
        delay(1000)
        println("Done 1")
    }
    launch {
        delay(2000)
        println("Done 2")
    }
}

suspend fun main() {
    println("Before")
    longTask()
    println("After")
}
// Before
// (1 sec)
// Done 1
// (1 sec)
// Done 2
// After
```

Open in Playground → Target: JVM Running on v.1.9.20

Notice that ending a coroutine scope function with `launch` makes no sense because nothing would change if we deleted it.

```
suspend fun updateUser() = coroutineScope {
    // ...

    // Don't
    launch { sendEvent(UserSunchronized) }
}
```

We expect suspending functions to await completion of coroutines they've started. They can overcome this expectation by using an external scope, but we should avoid doing this if there is no good reason.

```
suspend fun updateUser() = coroutineScope {
    // ...

    eventsScope.launch { sendEvent(UserSunchronized) }
}
```

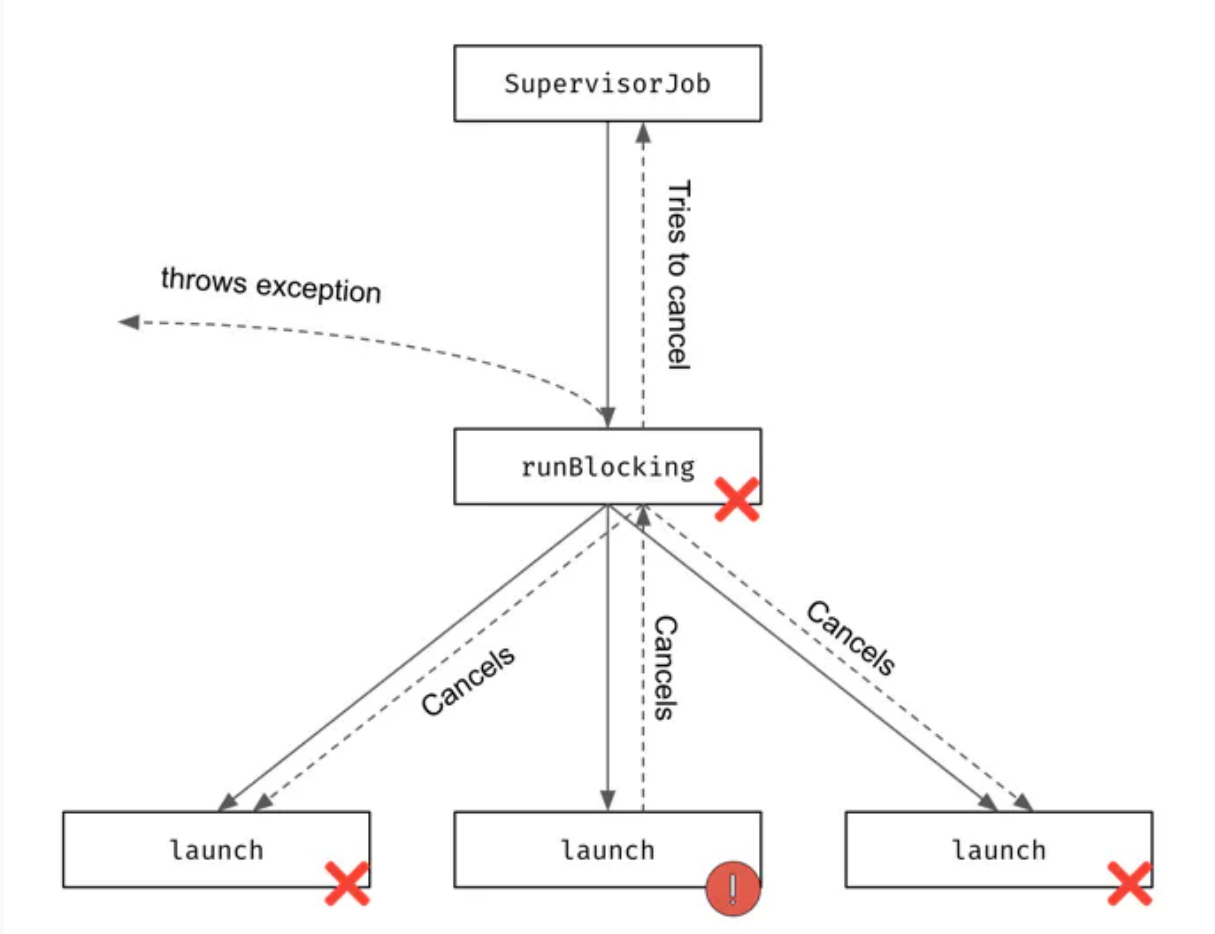
Understand that Job is not inherited: it is used as a parent

One of the biggest misunderstandings which causes mistakes in projects using Kotlin Coroutines comes from the fact that the `Job` context is the only context that is not inherited. Instead, a `Job` from a parent or argument is used as the parent of a coroutine.

Let's take a look at some examples. Adding `SupervisorJob` as a coroutine builder argument is pointless as it changes nothing.

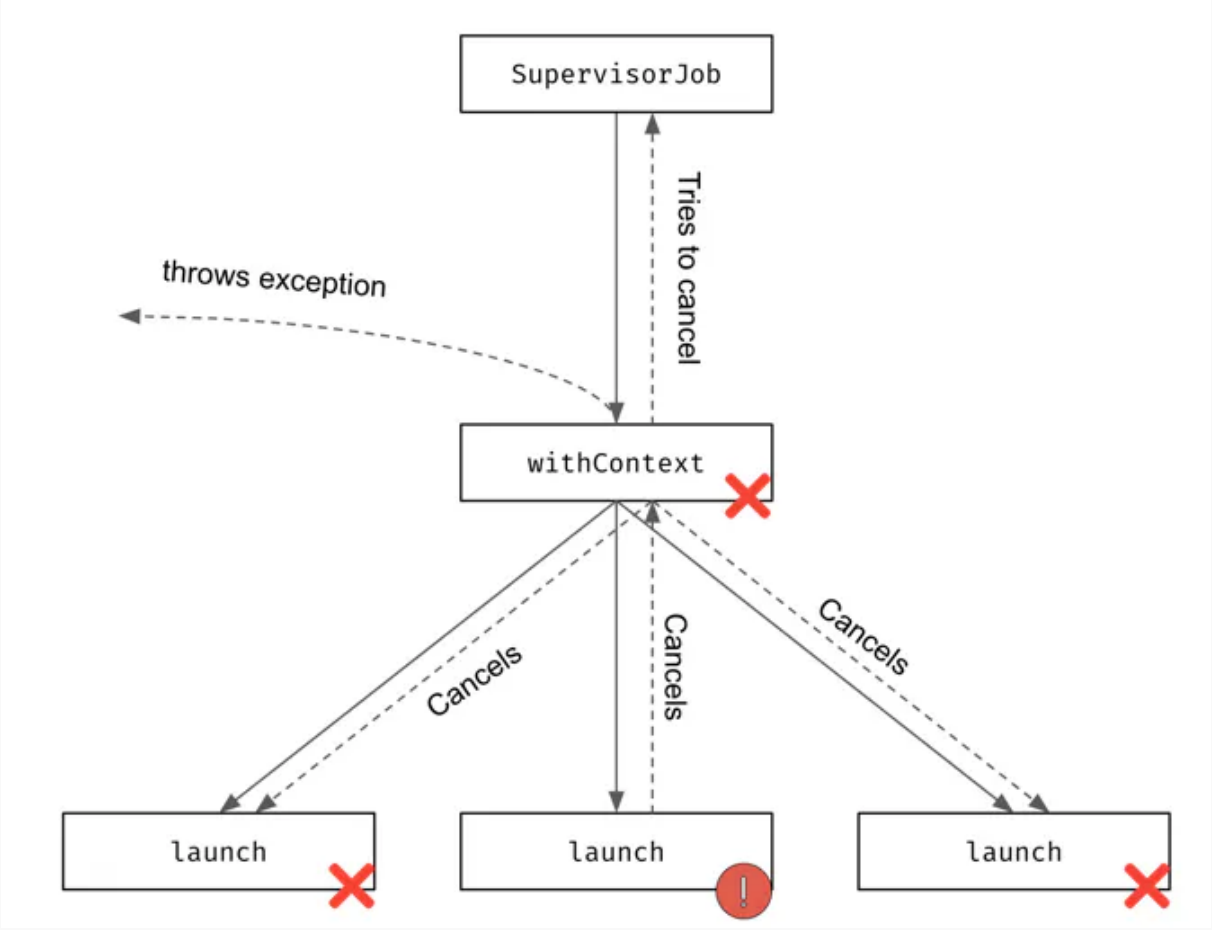
```
// Don't
fun main() = runBlocking(SupervisorJob()) {
    launch {
        delay(1000)
        throw Error()
    }
    launch {
        delay(2000)
        println("Done")
    }
    launch {
        delay(3000)
        println("Done")
    }
}
// (1 sec)
// Error...
```

`Job` is the only context that is not inherited. Each coroutine needs its own job, and passing a job to a coroutine makes the passed job the **parent** of this coroutine job. So, in the snippet above, `SupervisorJob` is the parent of `runBlocking`. When a child has an exception, this exception propagates to the `runBlocking` coroutine, breaks the `Job` coroutine, cancels its children, and throws an exception. The fact that `SupervisorJob` is a parent has no practical implication.



I see a similar mistake even more often when`withContext` is used together with `SupervisorJob`.

```
// Don't
suspend fun sendNotifications(
    notifications: List<Notification>
) = withContext(SupervisorJob()) {
    for (notification in notifications) {
        launch {
            client.send(notification)
        }
    }
}
```



Using `SupervisorJob` this way is pointless. When I see this, I can generally guess that the intention is to silence exceptions in children. The proper way to do this is to use `supervisorScope`, which ignores exceptions in its direct children.

```
// Do
suspend fun sendNotifications(
    notifications: List<Notification>
) = supervisorScope {
    for (notification in notifications) {
        launch {
            client.send(notification)
        }
    }
}
```

Using `withContext(Job())` is pointless and should also be considered a mistake.

Don’t break structured concurrency

The mistakes we presented above are not only pointless but also harmful. Whenever you set an explicit `Job` as a coroutine context, you break the relationship with this coroutine's parent. Take a look at the example below. The

problem with using `Job` as an argument to the coroutine is that it is set as the parent of this coroutine. As a result, `withContext` is not a child of the coroutine that called this function. When this coroutine is cancelled, our coroutine will not be, therefore the processes inside it will continue and waste our resources. Using an external job or scope breaks structured concurrency, prevents proper cancellation, and leads to memory leaks as a result.

```
// Don't
suspend fun getPosts() = withContext(Job()) {
    val user = async { userService.currentUser() }
    val posts = async { postsService.getAll() }
    posts.await()
        .filterCanSee(user.await())
}
```

Use SupervisorJob when creating CoroutineScope

When we create a scope, we can freely assume that we don't want an exception in one coroutine that was started using this scope to cancel all the other coroutines. For that, we need to use `SupervisorJob` instead of `Job`, which is used by default.

```
// Don't
val scope = CoroutineScope(Job())

// Do
val scope = CoroutineScope(SupervisorJob())
```

Consider cancelling scope children

Once a scope has been cancelled, it cannot be used again. If you want to cancel all the tasks started on a scope but you might want to keep this scope active, cancel its children. Keeping a scope active costs nothing.

```
fun onCleared() {
    // Consider doing
    scope.coroutineContext.cancelChildren()

    // Instead of
    scope.cancel()
}
```

On Android, instead of defining and cancelling custom scopes, you should use the `viewModelScope`, `lifecycleScope` and lifecycle-aware coroutine scopes

from the ktx libraries because these are cancelled automatically.

Before using a scope, consider under which conditions it is cancelled

One of my favorite heuristics for using Kotlin Coroutines on Android is "choosing what scope you should use is choosing when you want this coroutine cancelled". Each view model provides its own `viewModelScope`, which is cancelled when this view model is finalized. Each lifecycle owner has its own `lifecycleScope`, which is cancelled when this lifecycle is completed. We use these scopes instead of some shared global scope because we want our coroutines cancelled when they are not needed. Starting a coroutine on another scope means it will be cancelled under other conditions. Coroutines started on `GlobalScope` will never be cancelled.

```
class MainViewModel : ViewModel() {
    val scope = CoroutineScope(SupervisorJob())

    fun onCreate() {
        viewModelScope.launch {
            // Will be cancelled with MainViewModel
            launch { task1() }
            // Will never be cancelled
            GlobalScope.launch { task2() }
            // Will be cancelled when we cancel scope
            scope.launch { task2() }
        }
    }
}
```

Don’t use GlobalScope

It is too easy to use `GlobalScope`, so it might be tempting, but I would avoid doing that and instead create at least a very simple scope with only `SupervisorJob` as its context.


```
val scope = CoroutineScope(SupervisorJob())

fun example() {
    // Don't
    GlobalScope.launch { task() }

    // Do
```

```
    scope.launch { task() }
}
```

`GlobalScope` means no relation, no cancellation, and is hard to override for testing. Even if `GlobalScope` is all you need now, defining a meaningful scope might be helpful in the future.

```
// GlobalScope definition
public object GlobalScope : CoroutineScope {
    override val coroutineContext: CoroutineContext
        get() = EmptyCoroutineContext
}
```

Avoid using `Job` builder, except for constructing a scope

When you create a job using the `Job` function, it is created in the active state regardless of the state of its children. Even if some children have completed, this doesn't mean their parents have also completed.

```
suspend fun main(): Unit = coroutineScope {
    val job = Job()
    launch(job) {
        delay(1000)
        println("Text 1")
    }
    launch(job) {
        delay(2000)
        println("Text 2")
    }
    job.join() // Here we will await forever
    println("Will not be printed")
}
// (1 sec)
// Text 1
// (1 sec)
// Text 2
// (runs forever)
```

It is possible for such a `Job` to complete, but only if its `complete` method is first called and its state is then changed from "Active" to "Completing", where it waits until its children are finished. However, you cannot start new coroutines on completing or completed jobs. A more practical approach is to use a job reference to await its children (`job.children.forEach { it.join() }`). In most cases, the simplest solution is to await the job returned by a coroutine builder. Most common cases include storing the active task job in a variable, or collecting the jobs of all the started coroutines.

```
class SomeService {
```

```
    private var job: Job? = null
    private val scope = CoroutineScope(SupervisorJob())

    // Every time we start a new task,
    // we cancel the previous one.
    fun startTask() {
        cancelTask()
        job = scope.launch {
            // ...
        }
    }

    fun cancelTask() {
        job?.cancel()
    }
}
```

```
class SomeService {
    private var jobs: List<Job> = emptyList()
    private val scope = CoroutineScope(SupervisorJob())

    fun startTask() {
        jobs += scope.launch {
            // ...
        }
    }

    fun cancelTask() {
        jobs.forEach { it.cancel() }
    }
}
```

My general recommendation is to avoid using `Job` builder, except when constructing a scope.

Functions that return `Flow` should not be suspending

A flow represents a certain process that is started using the `collect` function. Functions that return `Flow` define such processes, and their execution is postponed until these processes are started. This is very different from suspending functions, which are supposed to execute processes themselves. Mixing these two concepts is counterintuitive and problematic.

As an example, consider that you need a function that fetches services to observe and then observes them. This is a problematic implementation:

```
// Don't use suspending functions returning Flow
suspend fun observeNewsServices(): Flow<News> {
    val newsServices = fetchNewsServices()
    return newsServices
        .asFlow()
        .flatMapMerge { it.observe() }
}

suspend fun main() {
    val flow = observeNewsServices() // Fetching services
```

```
    // ...
    flow.collect { println(it) } // Start observing
}
```

Open in Playground → Target: JVM Running on v.1.9.20

It is counterintuitive that part of the process is executed when `observeNewsServices` is called, and part is executed when we start collecting. Also, if we collect later, we will still use news items that were fetched in the past. This is problematic and counterintuitive. We expect that functions that return `Flow` pack the whole process into this flow.

To improve the function above, the most common intervention involves packing suspend calls into a flow.

```
fun observeNewsServices(): Flow<News> {
    return flow { emitAll(fetchNewsServices().asFlow()) }
        .flatMapMerge { it.observe() }
}

suspend fun main() {
    val flow = observeNewsServices()
    // ...
    flow.collect { println(it) }
    // Fetching services
    // Start observing
}
```

Open in Playground → Target: JVM Running on v.1.9.20

An alternative is, as always, to make a suspending function that awaits the completion of its process.

```
suspend fun fetchNewsFromServices(): List<News> {
    return fetchNewsServices()
        .mapAsync { it.observe() }
        .flatten()
}

suspend fun main() {
    val news = fetchNewsFromServices()
    // Fetching services
    // Start observing
    // ...
}
```

Open in Playground → Target: JVM Running on v.1.9.20

Prefer a suspending function instead of Flow when you expect only one value

I will finish this collection with the most controversial suggestion. Consider the function below. What values do you expect its flow to emit?

```
interface UserRepository {
    fun getUser(): Flow<User>
}
```

I would expect it to emit a user whenever it is changed, not only the current state. This is because the `Flow` type represents a source of values. To represent a single deferred value, we use suspending functions.

```
interface UserRepository {
    suspend fun getUser(): User
}
```

Contrary to this rule, many applications, especially Android applications, use Flow instead of suspending functions wherever possible. I understand the reasons behind this: some teams have used RxJava before and they don't want to change their habits. As my friend said, "I'm a RxJava senior, but a Kotlin Coroutines junior. I don't like being a junior, but Flow is like RxJava, so maybe I am a Flow mid".

On Android, developers have one more reason. It's become popular to represent mutable states using StateFlow, and Flow can be easily transformed into StateFlow using the `stateIn` function. So, operating on Flow is convenient.

```
class LocationsViewModel(
    locationService: LocationService
) : ViewModel() {

    private val location = locationService.observeLocations()
        .map { it.toLocationsDisplay() }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.Lazily,
            initialValue = LocationsDisplay.Loading,
        )

    // ...
}
```

If you join a team that uses Flow everywhere, it’s best to follow your team’s conventions. Every team has its own style and practices. However, if you can choose - maybe because you're designing a greenfield project, or maybe because you’ve just introduced coroutines - I suggest you don’t use flow where you expect only a single value. This will make your code simpler, easier to understand, and more efficient.

Before we close this chapter, I want you to remember this sentence: Best practices sometimes need to be violated; they are guidelines for standard situations, not rules for every situation.

The author:

Marcin Moskała

Marcin Moskala is an experienced developer and Kotlin trainer. He is the founder of the Kt. Academy, an official JetBrains partner for Kotlin training, author of the books Effective Kotlin, Kotlin Coroutines, Functional Kotlin and Android Development with Kotlin. He is also the main author of the biggest medium publication about Kotlin and a speaker invited to many programming conferences.

Add a comment

Write here...

Submit

Comments

Marc Tatham 2023-11-07T07:32:33.746Z

An excellent read, thank you for sharing your insights.