rules.sonarsource.com

Kotlin static code analysis: Coroutine usage should adhere to structured concurrency principles

3-4 minutes

Kotlin coroutines follow the principle of structured concurrency. This helps in preventing resource leaks and ensures that scopes are only exited once all child coroutines have exited. Hence, structured concurrency enables developers to build concurrent applications while having to worry less about cleaning up concurrent tasks manually.

It is possible to break this concept of structured concurrency in various ways. Generally, this is not advised, as it can open the door to coroutines being leaked or lost. Ask yourself if breaking structured concurrency here is really necessary for the application's business logic, or if it could be avoided by refactoring parts of the code.

This rule raises an issue when it detects that the structured concurrency principles are violated. It avoids reporting on valid use cases and in situations where developers have consciously opted into using delicate APIs (e.g. by using the `@OptIn` annotation) and hence should be aware of the possible pitfalls.

Noncompliant Code Example

GlobalScope:

```
fun main() {
  GlobalScope.launch { // Noncompliant: no explicit opt-in to
DelicateCoroutinesApi
    // Do some work
  }.join()
}
```

Manual job instantiation:

```
fun startLongRunningBackgroundJob(job: Job) {
    val coroutineScope = CoroutineScope(job)
    coroutineScope.launch(Job()) { // Noncompliant: new job
instance passed to launch()
        // Do some work
    }
}
```

Manual supervisor instantiation:

```
    coroutineScope {
        launch(SupervisorJob()) { // Noncompliant: new supervisor
instance passed to launch()
            // Do some work
        }
    }
```

Compliant Solution

In many situations, a good pattern is to use `coroutineScope` as provided in suspending functions:

```
suspend fun main() {
    worker()
}


suspend fun worker() {
    coroutineScope {
        launch { // Compliant: no manually created job/supervisor
instance passed to launch()
            // Do some work
        }
    }
}
```

GlobalScope:

```
@OptIn(DelicateCoroutinesApi::class)
fun main() {
```

```
    GlobalScope.launch { // Compliant: explicit opt-in to
DelicateCoroutinesApi via method annotation
        // Do some work
    }.join()
}
```

No manual job instantiation:

```
fun startLongRunningBackgroundJob(job: Job) {
    val coroutineScope = CoroutineScope(job)
    coroutineScope.launch { // Compliant: no manually created
job/supervisor instance passed to launch()
        // Do some work
    }
}
```

Using a supervisor scope instead of manually instantiating a supervisor:

```
 supervisorScope {
    launch {
      // Do some work
    }
 }
```

See

- Structured concurrency in the Kotlin docs
- GlobalScope documentation
- coroutineScope documentation
- Android coroutines best practices