rules.sonarsource.com

Kotlin static code analysis: The return value of functions returning "Deferred" should be used

3-4 minutes

Some functions return `kotlinx.coroutines.Deferred` to eventually communicate the result of an asynchronous operation. This is necessary, as callers do not wait for the result of an operation when it is launched asynchronously. Even if the result may be relevant later on, other tasks can be completed while waiting for the asynchronous operation's result to become available. Hence, to avoid blocking the caller, functions can trigger a task to be run and then immediately return a `Deferred` instance, which will contain the result once the background task is complete.

`Deferred` (aka Future, Promise, etc) provides an `await()` function, which suspends the caller coroutine until the asynchronous task is complete and returns the result of the execution. By not using the `Deferred` return value, the result of the corresponding asynchronously launched task is lost. This could point to an issue in the code, where data is not passed along as intended.

For instance, the Kotlin coroutines API provides both the functions `async` and `launch` as ways to launch asynchronous work. The key difference here is their return type. `launch` starts a new coroutine without blocking the current thread and returns a reference to the coroutine as a Job. This function is not designed to return a result, i.e. follows the idea of "fire and forget". `async` creates a coroutine and returns its future result as an implementation of `Deferred`.

Ask yourself whether:

- You really need whatever result is calculated by the asynchronous operation. If not, you may be better off using a function that does not return `Deferred`.
- You should be using the `Deferred` return value and fetching some data from it later on, for instance by calling `await()` on it.

This rule raises an issue when a function returning the type `kotlinx.coroutines.Deferred` is used without the result of the operation being retrieved.

Noncompliant Code Example

Here `coroutineScope` returns the `Deferred` instance initially returned by `async`. It is not used in any way afterwards, which could point to an issue in the business logic.

```
suspend fun doSomething() {
    coroutineScope { // Noncompliant
        async {
            // Do some work
            "result"
        }
    }
}
```

Compliant Solution

Using `launch`:

```
suspend fun doSomething() {
    coroutineScope {
        launch {
            // Do some work
        }
    }
}
```

Using `await` to retrieve the deferred result of the asynchronous operation:

```
suspend fun doSomething(): String {
    return coroutineScope {
        val asyncTask = async {
            // Do some work
            "result"
        }
        // Return the result, possibly with some other processing
before, by calling await() on the Deferred instance
        asyncTask.await()
    }
}
```

See

- Concurrent using async