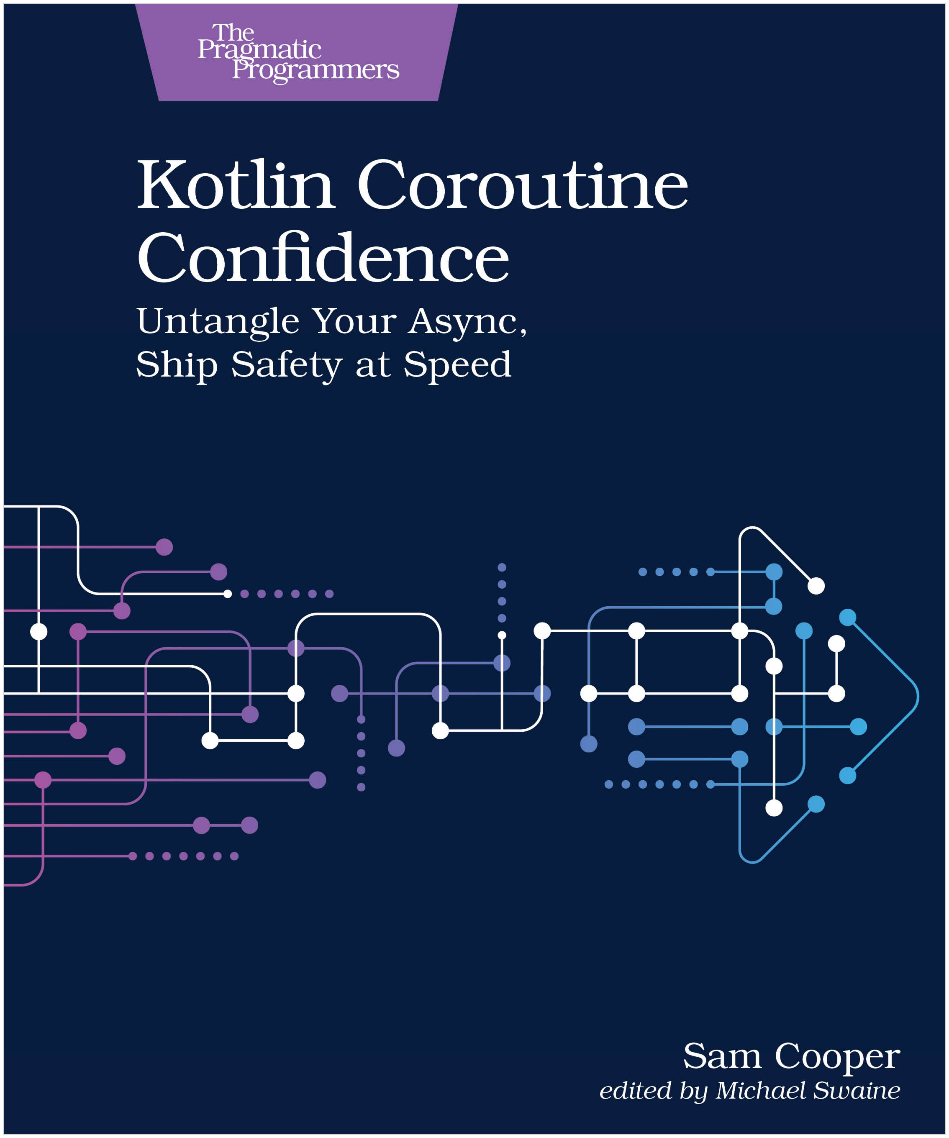


The Pragmatic Programmers

Kotlin Coroutine Confidence

Untangle Your Async, Ship Safety at Speed

Sam Cooper

edited by Michael Swaine

How Slow Can You Go?

Our password hashing function creates security by taking a long time to run. But as we've seen, a more powerful computer with extra CPU cores can make the job easier. Let's make our hashing algorithm configurable, so that hackers can't overcome all our defenses just by buying a more powerful computer.

parallel/v6/src/main/kotlin/com/example/parallel/Hash.kt

```
➤ const val iterations = 200_000_000

  fun slowHash(password: String, salt: ByteArray = salt()): String {
    val sha256 = MessageDigest.getInstance("sha256")

    var hash = salt + password.encodeToByteArray()
➤   repeat(iterations) {
      hash = sha256.digest(hash)
    }

    return salt.toHexString() + hash.toHexString()
  }
```

Two hundred million iterations should slow it down plenty! Okay, maybe that's a bit extreme—but with an adjustable number of iterations, we can tune it later to find the right balance between performance and security.

Let's test our new slowHash() function on just one password, and see how long it takes now.

parallel/v6/src/main/kotlin/com/example/parallel/MultithreadingApplication.kt

```
fun main() {
  val (value, duration) = measureTimedValue {
    slowHash(password = "kotlin1")
  }

  println("Computed the hash in $duration")
  println(value)
}
```

```
Computed the hash in 8.370272792s
0ba2d16763766162a702a8adfab7d39e07aa5b31ae54b019…
```

Great—that should keep those hackers busy.

Play by the Rules

Now that this function takes longer to run, there's a new problem we need to think about. Let's see what happens to this program if we try to add a time limit using the withTimeoutOrNull() function.

parallel/v7/src/main/kotlin/com/example/parallel/MultithreadingApplication.kt

```
suspend fun main() {
  val (value, duration) = measureTimedValue {
    withTimeoutOrNull(5.seconds) {
      slowHash("kotlin1")
    }
  }

  println("Computed the hash in $duration")
  println(value)
}
```

What do you think is going to happen when you run this code? Can you predict the output?

```
Computed the hash in 8.494025s
0e8ba295bcb4a52d823ef0814cc7583573de604a60081252…
```

Huh? The five-second timeout didn't seem to do anything at all. The output is exactly the same as it would have been if the withTimeoutOrNull() function wasn't there at all.

What went wrong?

In a way, that's a trick question. The withTimeoutOrNull() function is designed for use with suspending operations, like the long-running network requests and timed delays that made up our museum tour program in the last chapter. It'll wake up a suspended coroutine if it's been waiting for too long.

But our slowHash() function isn't a suspending function at all, and it doesn't include any waiting. It doesn't have a clue that it's running in a coroutine, and with no suspension points or other cancellation checks, it'll never find out that the coroutine has been cancelled. Remember, cancellation is cooperative—the only way to stop a task is to ask it nicely, and hope it's paying attention.

That's not great! This function uses a lot of resources, and since we're going to be using coroutines to run it in parallel, we want it to play by the rules. If its parent Job fails or gets cancelled, the task should exit promptly, instead of continuing to spend CPU time on a calculation that will never be used.

There are a few ways we could think about adding cancellation checks to our slowHash() function. Maybe you already have an idea of how you might modify the function and its loop condition to solve the problem. Before we take care of it, though, let's have a look at another problem we might run into when we use this code in a coroutine.

Don't Be Selfish

Let's add some parallelization back into our program. That's the reason we want coroutines in the mix, after all.

Our function will take much longer to run now, so we'll reduce the number of passwords we're processing. Even so, we're just going to get a blank console until it's done. Maybe we could add some sort of output to keep track of progress, and show how long the program's been working?

We'll start with one password-hashing task for each CPU core. On top of that, we'll add one additional coroutine, to measure the elapsed time and display our output. This extra task will only wake up once every second, so it'll barely add any extra demand on our system's resources. Even so, we're not going to get the behavior we want when we try running this code. Can you guess what the problem is?

parallel/v8/src/main/kotlin/com/example/parallel/MultithreadingApplication.kt

```
suspend fun main() = withContext(Dispatchers.Default) {
  val startTime = TimeSource.Monotonic.markNow()
  val numberOfCores = Runtime.getRuntime().availableProcessors()

  val passwordJob = async {
    passwords.take(numberOfCores) // one task for each CPU core
      .map { async { slowHash(it) } }.awaitAll()
  }

  launch {
    while (passwordJob.isActive) {
      delay(1.seconds)
      println("Time taken so far: ${startTime.elapsedNow()}")
    }
  }

  val values = passwordJob.await()
  println("Computed ${values.size} hashes in ${startTime.elapsedNow()}")
}
```

Our new timer coroutine will run for as long as the passwordJob is still doing work. Notice how we're putting our structured concurrency skills to use by using that Job as a parent for each of our other async() tasks!

All our coroutines are supposed to be running at once, and so we should start seeing the output from our timer coroutine right away. But instead, it doesn't seem to do anything at all until the program's almost over.

```
Time taken so far: 27.5s
Time taken so far: 28.5s
Computed 10 hashes in 29.4s
Time taken so far: 29.5s
```

Why did the final coroutine take more than twenty seconds to start?

It's because all our dispatcher's threads were already busy doing other work. Our slowHash() function is selfish—once it's got hold of a thread, it's going to keep it. As you learned way back at the start of our journey, an ordinary non-suspending function like this one can't give up its thread until it reaches the end and exits.

So by the time our final coroutine comes along, all the threads are already taken! The default dispatcher has just the right number of threads to match our system's processor cores—any more would be wasteful. But with all those threads already busy running uncooperative coroutines, there's no way for other tasks to get a look in.

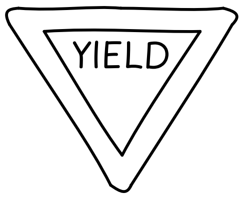
If we were using a dedicated thread for each task, this wouldn't be a problem. Instead of relying on cooperative suspension points, threads allow for *preemptive multitasking*, where the system can swap between tasks whenever it likes. But it can only work with the threads we give it, and all our dispatcher's threads are already loaded up with password-hashing tasks. The timer coroutine was the last to be started, and it won't be allocated to a thread at all until the dispatcher has completed at least one of its other tasks.

Sharing Is Caring

You've just seen two problems with running our `slowHash()` function in a coroutine. First, it can't exit early when the coroutine is cancelled, because it doesn't know anything about coroutines or cancellation. And second, it can't cooperate by sharing its thread with other tasks. As an ordinary function with no suspension capabilities, it's bound to its thread until it's completed its entire execution.

We can fix both of these problems by giving the `slowHash()` some suspension points. That might sound strange, since it doesn't have any asynchronous operations or outside events to wait for. But suspending is just as much about cooperating with other coroutines as it is about waiting for outside events. If there are several coroutines waiting to execute, we can suspend the current function and let another task run for a while.

That's the job of the `yield()` function.



Think of it like the *yield* or *give way* sign at a traffic intersection. It marks a point where you need to pause and check for other vehicles—but you only have to stop if there's actually something coming. If there aren't any other coroutines waiting, the yield() function doesn't suspend at all, and the current task just keeps right on running.

Let's try it out in our code. We'll call yield() once every two hundred thousand iterations.

parallel/v8/src/main/kotlin/com/example/parallel/Hash.kt

```
➤ suspend fun slowHash(password: String, salt: ByteArray = salt()): String {
    val sha256 = MessageDigest.getInstance("sha256")

    var hash = salt + password.encodeToByteArray()
    repeat(iterations) { i ->
➤     if (i % 200_000 == 0) yield()
      hash = sha256.digest(hash)
    }

    return salt.toHexString() + hash.toHexString()
  }
```

Run either of the two previous programs again with this upgraded slowHash() function, and everything works the way we wanted. Our extra timer coroutine can print its output every one second, and our timeout code can cancel the work promptly.

Striking a Balance

Consider using yield() to keep everything running smoothly if you're doing CPU work for more than a few dozen milliseconds. But don't go overboard, or you might end up with more task-switching than actual work.

Any suspension point, whether it's a delay() or a network request, creates a gap where the task can give up its thread and let other coroutines take a turn. So the yield() function is only necessary when a coroutine runs for a long time without encountering any other suspending functions. Like any other suspension point, it also checks for cancellation, throwing a CancellationException when the task is no longer needed.

Notice how we had to add the suspend modifier directly to our slowHash() function this time—not just to the main() function that's going to wait for its parallel execution. That's because yield() is a suspending function—even if it only ever waits for other coroutines, and not outside events. Since coroutines use suspension points for cooperative multitasking, being able to suspend is a useful trait for any long-running function that wants to be a good neighbor while running on a coroutine dispatcher.