Kotlin Coroutines(Part — 2): The 'Suspend' Function

What are suspend functions, and how to make an API call using coroutines?

Aditi Katiyar · Follow
Published in DeHaat · 3 min read · Sep 3, 2021

Photo by Kelly Lacy from Pexels

Before moving further, if you want to have a basic idea of coroutines, check this out: Kotlin Coroutines(Part — 1): The Basics.

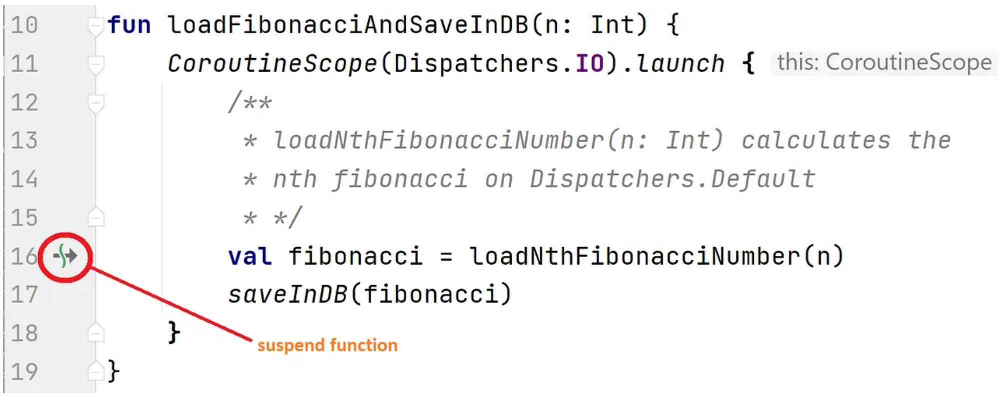
The most common use-case of background processing is fetching data from servers (API calls). The most common approach will be to implement callbacks for success and failure cases. But what if we have to make 2 or more API calls one after another? We will be implementing callbacks for every API call within another API call. This is known as 'callback hell'.

With coroutines, we can avoid 'callback hell' and make our code easily testable and readable!

What is a suspend function?

A 'suspend' function forces the caller coroutine to **wait** for the function completion without blocking itself. While the suspend function is executing, the coroutine releases the thread on which it was running and allows other coroutines to access that thread(Because coroutines are **cooperative**).

Let's look at an example:

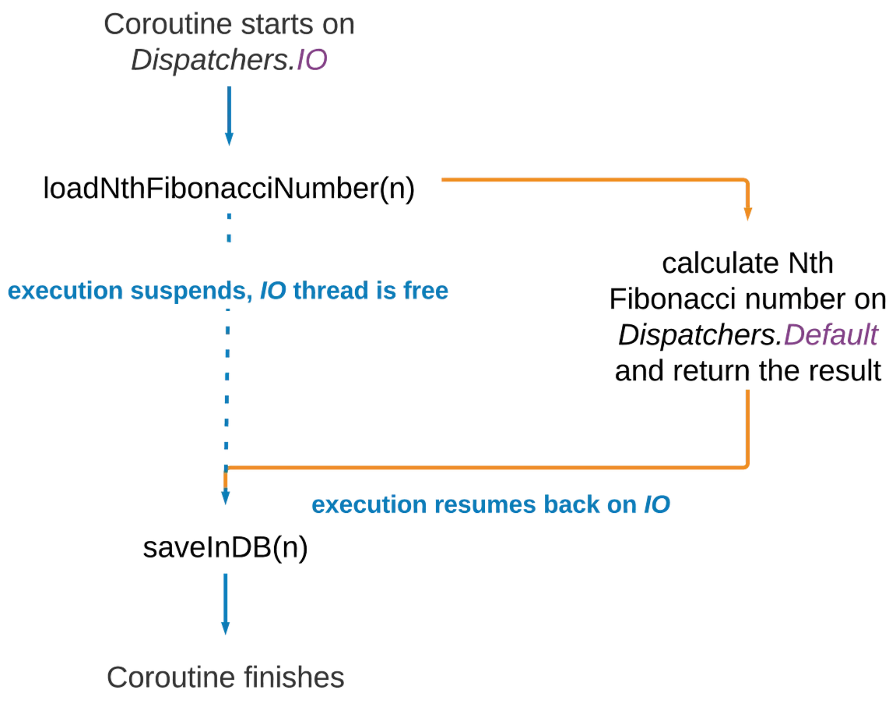


```
fun loadFibonacciAndSaveInDB(n: Int) {
    CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope
        /**
         * loadNthFibonacciNumber(n: Int) calculates the
         * nth fibonacci on Dispatchers.Default
         * */
        val fibonacci = loadNthFibonacciNumber(n)
        saveInDB(fibonacci)
    }
}
```

We start a coroutine on *Dispatchers.IO* and call *loadNthFibonacciNumber(n).* We will learn about the underlying implementation of *loadNthFibonacciNumber(n: Int)* in the future(when we have a relatively better understanding of coroutines). For now, all we know is that it is calculated on *Dispatchers.Default* and is a suspend function.

While the function *loadNthFibonacciNumber* is executing on *Dispatchers.Default,* it 'suspends' the execution of the caller coroutine. The coroutine will **wait** for *loadNthFibonacciNumber* to finish. Meanwhile, this coroutine will release the *IO* thread and allow other coroutines to access that thread.

This is the reason why *loadNthFibonacciNumber* is a suspend function(because it suspends/halts the execution of caller coroutine and forces it to yield its resources so that others can use those resources).


Coroutine starts on Dispatchers.IO
loadNthFibonacciNumber(n)
execution suspends, IO thread is free
calculate Nth Fibonacci number on Dispatchers.Default and return the result
execution resumes back on IO
saveInDB(n)
Coroutine finishes

A suspend function can also be called from another suspend function.

```kotlin
private suspend fun getFibonacci(n: Int) =
    loadNthFibonacciNumber(n)
```

How to make API calls in Retrofit with coroutines?

Let us fetch a list of Dogs from an API. The data class looks like this:

```kotlin
data class Dog(
    val name: String,
    val weight: Int
)
```

Dog.kt hosted with ❤ by GitHub view raw

Make the API call in the service class a 'suspend' function.

```kotlin
interface ApiService {
    @GET("/doggos")
    suspend fun getDoggos(): List<Dog>
}
```

ApiService.kt hosted with ❤ by GitHub view raw

Here, we wish to get a list of dogs from the API. Now make the call in the repository.

```kotlin
class DogsRepository @Inject constructor(
    private val apiService: ApiService
) {

    suspend fun fetchFromServer(): List<Dog> =
        try {
            val dogs = apiService.getDoggos()
            dogs // return fetched data
        } catch (e: Exception) {
            // Log exception to Firebase, etc.
            listOf() // return an empty list
        }
}
```

Here, 'ApiService' is the same retrofit object that we used when calls were handled using callback listeners.

Since we can call a suspend function only from a suspend function or a coroutine scope, we added the 'suspend' modifier to the function '*fetchFromServer()*' here. This function is supposed to return *List<Dog>*. We put the API call — '*apiService.getDoggos()*' in a try-catch block to catch the exceptions like — IOException, HttpException, etc. In case of exceptions, we can log them and return an empty list.

Now, we call the function '*fetchFromServer()*' from the ViewModel as follows:

```kotlin
class DogsViewModel(
    private repository: DogsRepository
): ViewModel() {

    val doggos = MutableLiveData<List<Dog>>()

    fun loadDoggos() {
        viewModelScope.launch {
            doggos.value = repository.fetchFromServer() // set the list in LiveData
        }
    }
}
```

This is where we initiate the API call. We create a coroutine in *viewModelsScope* (so that the call cancels when ViewModel clears), and call our API through the repository's code. Simply call the function *loadDoggos()* from Fragment/Activity to load data from the server and receive it by

Retrofit implicitly handles it on a background thread(and returns the result on the caller thread, which is the Main thread in this case) the moment we make the calls using suspend functions(Coroutines).

Writing API calls this way takes lesser time and less code, and hence becomes easier to test.

To learn more about coroutines, check these out:

- Kotlin Coroutines(Part — 3): Coroutine Context
- Kotlin Coroutines(Part — 4): Cancellation
- Kotlin Coroutines(Part — 5): Exception Handling

Thanks for reading! If you liked it, please give it a clap! Keep Learning!

Engineering Android App Development Coroutine App Development Kotlin

Written by Aditi Katiyar

Follow

90 Followers · Writer for DeHaat

Android developer @DeHaat