CMPS 312 Mobile App Development
Lab 9 – Web API with Coroutines and Ktor

Objective

In this Lab, you will **extend the Banking App** to communicate with the Bank Web API. You will be using the **Ktor** library, asynchronous suspend functions, and coroutines to get, add, update, and delete transfers and beneficiaries.

PART A: Warmup Coroutines Exercises

1. Create a new Android Project named **coroutines** under **Lab9-Coroutines** in your GitHub repo.
2. Create a new file koltin file named **CoroutinesTest** and the following code

```
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one is de
}
```

You should see the following output

```
Hello
World!
```

Note : If you **remove or forget runBlocking** in this code, you'll get an error on the launch call, since launch is declared only in the CoroutineScope. Therefor the **runBlocking** is also a coroutine builder that bridges the non-coroutine world of a regular fun main()

3. Let's modify the code above by extracting the launch into a separate function. You should get an error saying you need to make the fun suspend. This means, the suspend function allows us to call asynchronous functions.

```
fun main() = runBlocking { // this: CoroutineScope
    launch { doWorld() }
    println("Hello")
}

// this is your first suspending function
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

4. Let us create our first coroutine scope. The coroutine scope will allow us to execute tasks in a non sequential concurent manner. When you run the code below you should see **Hello** followed by **World**. Even though, the sequence in which we wrote them is different.

```
fun main() = runBlocking {
    doWorld()
}

suspend fun doWorld() = coroutineScope {  // this: CoroutineScope
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

5. Create a multiple concurrent print task by typing the following. You can add a simple for loop to see even more interesting effect.

```
// Sequentially executes doWorld followed by "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}

// Concurrently executes both sections
suspend fun doWorld() = coroutineScope { // this: CoroutineScope
    launch {
        delay(2000L)
        println("World 2")
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
```

6. Run the code and try to see the output

```
Hello
World 1
World 2
Done
```

7. When we launch coroutine we can capture the coroutine as a job. This allows us to have a better control of the way the coroutine behaves. For example type the following code and run it.

```
val job = launch { // launch a new coroutine and keep a reference to its Jo
    delay(1000L)
    println("World!")
}
println("Hello")
job.join() // wait until child coroutine completes
println("Done")
```

You should see the following output on the screen. This means, we were able to wait our job to finish before displaying the done. Sometimes, in our code, we might need such scenarios. Specially when we do not want to continue until a coroutine finishes its task. Like adding a user on the server and waiting the newly generated ID of the user.

```
Hello
World!
Done
```

8. Coroutines are very lightweight, which means we can start 100 thousand of them in parallel without affecting the performance of the user's phone. If we do the same using normal threads our phone will crush.

```
//sampleStart
fun main() = runBlocking {
    repeat(100_000) { // launch a lot of coroutines
        launch {
            delay(5000L)
            print(".")
        }
    }
}
```

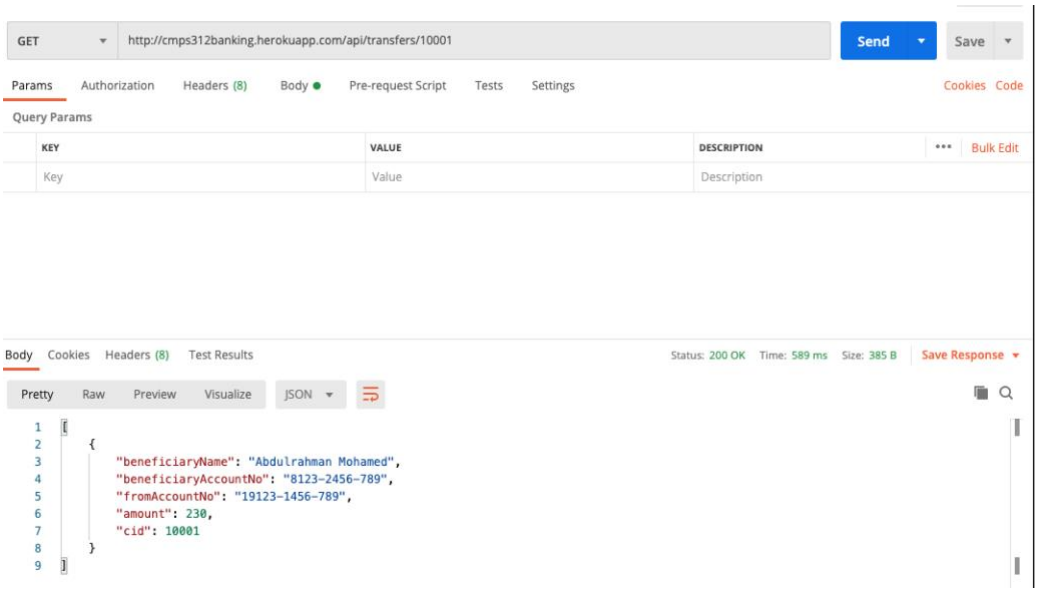
In the second part of the Lab, we will use coroutines and implement a real world client-server application. The coroutines will give us all the benefit of asynchronous and non-blocking behavior.

PART B: Implement the Bank Web API using Coroutines

1. Sync the Lab GitHub repo and copy the **Lab9-Coroutines** folder into your repository.
2. Open the ***Banking App*** project on Android Studio. This project has the complete implementation of **Lab7-BankingApp** with some minor modifications such as delete button and new properties added to the Account class such as cid (i.e., Customer id).
3. Your task is to implement calling the Bank Web API using coroutines to read/write data from/to the remote bank service. You will be using Ktor with coroutines to achieve this.
4. Download postman from https://www.postman.com/downloads/ and test the following Banking Service Web API available at https://cmps312banking.herokuapp.com

Available API

Description	Endpoint	Possible Methods
GET Accounts	https://cmps312banking.herokuapp.com/api/accounts/:cid	GET
GET Transfers	https://cmps312banking.herokuapp.com/api/transfers/:cid	GET
ADD Transfers	https://cmps312banking.herokuapp.com/api/transfers/:cid	POST
DELETE Transfers	https://cmps312banking.herokuapp.com/api/transfers/:cid/:transferId	DELETE
GET Beneficiaries	https://cmps312banking.herokuapp.com/api/beneficiaries/:cid	GET
ADD Beneficiary	https://cmps312banking.herokuapp.com/api/beneficiaries/:cid	POST [Required cid in the URL]
UPDATE Beneficiary	https://cmps312banking.herokuapp.com/api/beneficiaries/:cid	POST [Requires cid in the URL]
DELETE Beneficiary	https://cmps312banking.herokuapp.com/api/beneficiaries/:cid/:accounNo	DELETE [Requires cid and accountNo in the URL]
Local Banks	https://cmps312banking.herokuapp.com/api/banks	GET



1. Add the following dependencies for ktor and coroutines in your build.gradle app module.

```
//For Ktor Client
def ktor_version = "1.6.4"
implementation "io.ktor:ktor-client-android:$ktor_version"
implementation "io.ktor:ktor-client-serialization:$ktor_version"
```

2. Inside the **webapi** package, create an **interface** called **BankService**

   List all the interfaces methods that allow the app to communicate with Banking Web API available at https://cmps312banking.herokuapp.com

```
interface BankService {
    suspend fun getTransfers(cid: Int): List<Transfer>
    suspend fun addTransfer(transfer: Transfer): Transfer?
    suspend fun deleteTransfer(cid: Int, transferId: String): String
    suspend fun getAccounts(cid: Int): List<Account>
    suspend fun getBeneficiaries(cid: Int): List<Beneficiary>
    suspend fun updateBeneficiary(cid: Int, beneficiary: Beneficiary): String?
    suspend fun deleteBeneficiary(cid: Int, accountNo: Int): String
}
```

3. Create a QuBankService class under the webapi package that implements all the methods that allow the app to communicate with the Bank Web API.
   - Declare the BASE_URL constant and set it to https://cmps312banking.herokuapp.com/api
   - Create a HTTP client and add JsonFeature auto-parse from/to json when sending and receiving data from the Web API.

```
private val BASE_URL = "https://cmps312banking.herokuapp.com/api"
private val client = HttpClient {  this: HttpClientConfig<*>
    //This will auto-parse from/to json when sending and receiving data from Web API
    install(JsonFeature) { serializer = KotlinxSerializer() }
}
```

   - Implement the **BankService** interface including getTransfers , addTransfer , deleteTransfer , getBeneficiaries , updateBeneficiary , deleteBeneficiary. Create BankServiceTest class and add a main method to test your implementation as you make progress.

   For example: the following getTransfers() method sends a get request to url https://cmps312banking.herokuapp.com/api/accounts/100101 and returns the list of transfers for customer 10001.

```
override suspend fun getTransfers(cid: Int): List<Transfer> {
   val url = "$BASE_URL/transfers/$cid"
   println(url)
   return client.get(url)
}
```

PART C: Change the App ViewModels to use QuBankService implemented in Part B

Your task is to change the app view models to use the **new QuBankService** implemented in Part B.

1. Modify the **TransferViewModel** to use QuBankService and implement the following methods

```
fun getTransfers() {}
fun getAccounts()
fun addTransfer(transfer: Transfer) {}
fun getBeneficiaries() {}
fun deleteTransfer(transferId: String) {}
```

2. Modify the **BeneficiaryViewModel** methods to use QuBankService. You should not change anything else in the app, and it should work as before. **This is the fruit of using MVVM!!!** 👍👌