Network Call with Kotlin Coroutines

What is a Network Call?

A **network call** is when your app **communicates with a server** (usually over the internet) to **send or receive data**. Common use cases include:

- Fetching user data from a remote API
- Sending form data to a server
- Getting a list of articles or images

In Android, you can't do this on the **main thread** because it would freeze the UI — that's where tools like **Retrofit**, **OkHttp**, and **Kotlin Coroutines** come in.

What is Retrofit?

Retrofit is a powerful library by Square that makes it super easy to **call REST APIs**.

Key Features:

Feature	What it does
Type-safe	You define your API endpoints in an interface
Automatic parsing	Converts JSON into Kotlin data classes
Works with coroutines	No need for callbacks or boilerplate
Pluggable	You can add interceptors, logging, converters, etc.

How it Works:

1. You define your API like this:

```
@GET("users")
suspend fun getUsers(): List<User>
```

2. Retrofit turns that into a real HTTP call behind the scenes.
3. You call `api.getUsers()` and it fetches and parses the data!

What is OkHttp?

OkHttp is the **underlying HTTP client** used by Retrofit (also made by Square).

Retrofit is like the **friendly face**, and OkHttp is the **muscle behind it**.

What OkHttp Does:

Feature	Description
Makes HTTP requests	GET, POST, PUT, DELETE
Handles connection pooling	Reuses sockets to save bandwidth
Supports interceptors	You can log requests, add headers, or authenticate
Built-in caching	Helps apps work faster offline

Why use OkHttp directly?

You might use OkHttp directly if:

- You want to customize request behavior
- You need advanced logging or caching
- You're not using Retrofit

> **Retrofit uses OkHttp internally** unless you plug in something else.

What are Kotlin Coroutines?

Kotlin Coroutines are a **lightweight, modern way** to do background work **without freezing the UI**.

Why do we need them?

In Android:

- You **can't do network calls on the main thread**
- Traditionally, people used `AsyncTask` or callbacks (which are messy)

Coroutines make this **easy and readable**.

Concepts:

Keyword	What it Means
`suspend`	A function that can run "asynchronously" without blocking

Keyword	What it Means
`launch`	Starts a coroutine in a scope (like `viewModelScope`)
`viewModelScope.launch`	Starts background work tied to ViewModel lifecycle

Example:

```
viewModelScope.launch {
    val users = repository.getUsers() // network call, safe & non-blocking
}
```

- The code *looks synchronous* — but it runs in the background!
- No freezing, no lag, no messy callbacks

Bringing it Together (Analogy)

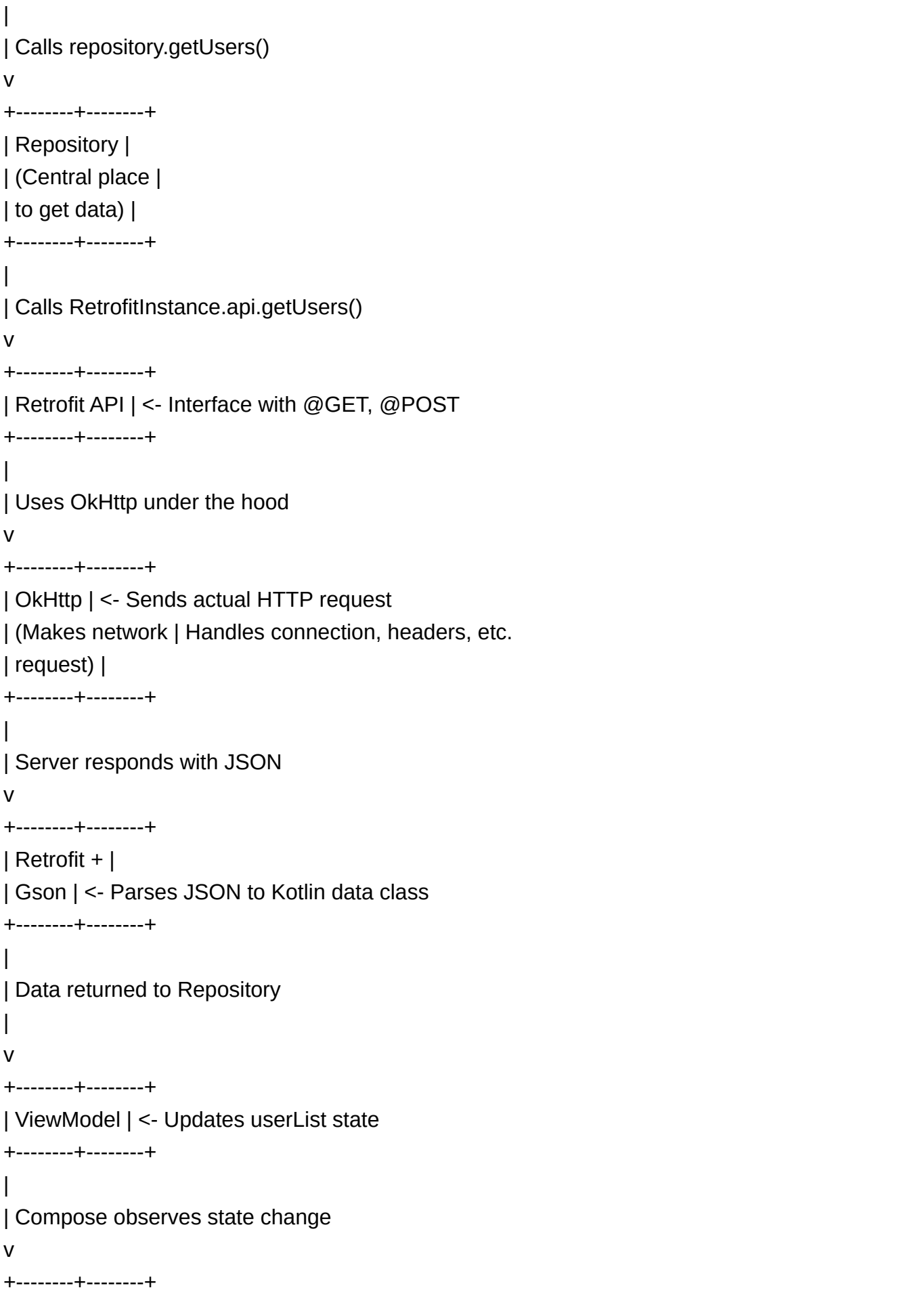
Imagine you're a chef (your app), and:

- The **server API** is your grocery store
- **Retrofit** is your grocery delivery app — you just click a button and get your groceries (data)
- **OkHttp** is the delivery driver — it knows the streets and delivers the request
- **Coroutines** make sure while you're waiting for the groceries, you can still cook something else — you don't freeze your whole kitchen!

What You'll Build

A simple app with a **button** that fetches **user data** from a remote API and shows it in a list — using **Jetpack Compose**, **Retrofit**, **Coroutines**, and **ViewModel**.

+-----------------+
| UI (Jetpack |
| Compose) |
+--------+--------+
|
| User taps "Fetch Users"
|
v
+--------+--------+
| ViewModel |
| (Holds State + |
| launches coroutine) <- Uses viewModelScope.launch { ... }
+--------+--------+


| Calls repository.getUsers()
v
+--------+--------+
| Repository |
| (Central place |
| to get data) |
+--------+--------+
| Calls RetrofitInstance.api.getUsers()
v
+--------+--------+
| Retrofit API | <- Interface with @GET, @POST
+--------+--------+
| Uses OkHttp under the hood
v
+--------+--------+
| OkHttp | <- Sends actual HTTP request
| (Makes network | Handles connection, headers, etc.
| request) |
+--------+--------+
| Server responds with JSON
v
+--------+--------+
| Retrofit + |
| Gson | <- Parses JSON to Kotlin data class
+--------+--------+
| Data returned to Repository
v
+--------+--------+
| ViewModel | <- Updates userList state
+--------+--------+
| Compose observes state change
v
+--------+--------+

| UI Rebuilds | <- Shows list of users on screen
+----------------+

Dependency	Purpose
Retrofit	Makes it easy to call REST APIs
Gson	Parses JSON response into Kotlin objects
OkHttp	Adds logging/debugging for network requests
Coroutines	Lets you call APIs without freezing the UI
ViewModel	Keeps UI state alive during screen rotation
Compose UI	The modern way to build Android UIs

Step 1: Create a New Android Project

- Open Android Studio
- Create a **New Project** → Choose **Empty Compose Activity**
- Name it `ComposeNetworkApp`
- Language: **Kotlin**
- Click **Finish**

This gives you a minimal app using **Jetpack Compose**, Google's modern UI toolkit.

Step 2: Add Dependencies (with Explanation)

In `build.gradle.kts` (Module: app):

```
dependencies {
    // Retrofit - to make API calls
    implementation("com.squareup.retrofit2:retrofit:2.9.0")
    implementation("com.squareup.retrofit2:converter-gson:2.9.0")

    // OkHttp logging - to see API logs (optional)
    implementation("com.squareup.okhttp3:logging-interceptor:4.12.0")

    // Coroutines - for background threading
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3")

    // ViewModel - to hold and manage UI data
    implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.7.0")
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")

    // Jetpack Compose UI
```

```
    implementation("androidx.activity:activity-compose:1.8.2")
    implementation("androidx.compose.ui:ui:1.6.0")
    implementation("androidx.compose.material:material:1.6.0")
    implementation("androidx.compose.ui:ui-tooling-preview:1.6.0")
}
```

Step 3: Define the API Model and Service

`User.kt`

```
data class User(
    val id: Int,
    val name: String,
    val email: String
)
```

This class maps to the **JSON object** you get from the API.

`ApiService.kt`

```
import retrofit2.http.GET

interface ApiService {
    @GET("users")
    suspend fun getUsers(): List<User>
}
```

- `@GET("users")` : Tells Retrofit to make a GET request to `/users`
- `suspend fun` : Means this can be called from a coroutine
- `List<User>` : Retrofit automatically converts the JSON array to a Kotlin list

Step 4: Set Up Retrofit

`RetrofitInstance.kt`

```
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object RetrofitInstance {
    val api: ApiService by lazy {
        Retrofit.Builder()
            .baseUrl("https://jsonplaceholder.typicode.com/")
```

```
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(ApiService::class.java)
    }
}
```

- `baseUrl` : API root URL
- `addConverterFactory` : Tells Retrofit how to parse JSON (Gson here)
- `.create(ApiService::class.java)` : Gives us the implementation of the interface

Step 5: Create the Repository and ViewModel

`UserRepository.kt`

```
class UserRepository {
    suspend fun getUsers(): List<User> {
        return RetrofitInstance.api.getUsers()
    }
}
```

A **Repository** separates data handling from the UI. It talks to the API and gives results to the ViewModel.

`UserViewModel.kt`

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch
import androidx.compose.runtime.*

class UserViewModel : ViewModel() {
    private val repository = UserRepository()

    var userList by mutableStateOf<List<User>>(emptyList())
        private set

    var isLoading by mutableStateOf(false)
        private set

    fun fetchUsers() {
        viewModelScope.launch {
            isLoading = true
            try {
```

```
                userList = repository.getUsers()
            } catch (e: Exception) {
                e.printStackTrace()
            }
            isLoading = false
        }
    }
}
```

- `viewModelScope.launch` : Launches a coroutine for safe background work
- `mutableStateOf` : Used for Compose to reactively update UI when data changes

Step 6: Build the UI in Compose

`MainActivity.kt`

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material.*
import androidx.compose.runtime.*
import androidx.lifecycle.viewmodel.compose.viewModel
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.layout.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                UserScreen()
            }
        }
    }
}
```

`UserScreen.kt`

```
@Composable
fun UserScreen(viewModel: UserViewModel = viewModel()) {
    val users = viewModel.userList
```

```
    val isLoading = viewModel.isLoading

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
    ) {
        Button(onClick = { viewModel.fetchUsers() }) {
            Text("Fetch Users")
        }

        Spacer(modifier = Modifier.height(16.dp))

        if (isLoading) {
            CircularProgressIndicator()
        } else {
            LazyColumn {
                items(users) { user ->
                    Text(text = "${user.name} - ${user.email}")
                    Divider()
                }
            }
        }
    }
}
```

- `Button` : Calls `fetchUsers()` when clicked
- `LazyColumn` : Displays a scrollable list
- `CircularProgressIndicator` : Shows a loading spinner while fetching data
- `Text` : Displays user info
- `viewModel()` : Gets your `UserViewModel` and links it to this UI