Working with Kotlin App

Tushar B. Kute,
http://tusharkute.com

RESTful APIs

- Working with RESTful APIs and JSON is a core task in modern development, especially for mobile and web applications.
- Kotlin, combined with popular libraries, makes this process efficient and safe.
- Think of a RESTful API as a restaurant's menu for a web service. It defines a set of rules and endpoints (like menu items) that you can use to request (GET), create (POST), update (PUT), or delete (DELETE) data.
- The data is most commonly returned in JSON (JavaScript Object Notation) format, which is a lightweight, human-readable text format for structuring data.

Making Network Requests (The HTTP Client)

- Kotlin doesn't have a built-in, high-level HTTP client for all platforms, so you'll rely on libraries. These libraries handle the complexities of making network requests, like opening connections and managing headers.
- The most popular choices are:
  - Retrofit: The industry standard for Android development. It's a type-safe client that lets you define your API as a simple Kotlin interface.
  - Ktor Client: A modern, multiplatform client from JetBrains. It's an excellent choice for Kotlin Multiplatform Mobile (KMM) and server-side applications.
- Analogy: An HTTP client is like a waiter. You give your order (the API request) to the waiter, who takes it to the kitchen (the server) and brings back your food (the JSON response).

Parsing JSON Data (The Serializer)

- Once you get a response from the API, it's typically a raw string of JSON text. To use this data in a structured way, you need to convert (or deserialize) it into Kotlin objects. This is where JSON parsing libraries come in.
- The best choices for Kotlin are:
  - kotlinx.serialization: The official serialization library from JetBrains. It's multiplatform, type-safe, and integrates directly with the Kotlin compiler. This is the recommended modern approach.
  - Moshi: A popular and highly performant library from Square, with excellent support for Kotlin's features.
  - Gson: An older library from Google. While still functional, it's less Kotlin-idiomatic and can require more boilerplate.

Parsing JSON Data (The Serializer)

- The key is to create a Kotlin data class that exactly matches the structure of the JSON object you expect to receive.
- The library then automatically maps the JSON fields to the properties of your data class.
- For example, for this JSON:

```
{
  "id": 1,
  "title": "My First Post",
  "isPublished": true
}
```

Parsing JSON Data (The Serializer)

- You would create this data class using kotlinx.serialization:

```
import kotlinx.serialization.Serializable

@Serializable // This annotation tells the
library to generate a serializer for this class

data class Post(
    val id: Int,
    val title: String,
    val isPublished: Boolean
)
```

Putting It All Together

- Let's fetch data from a public API using Retrofit and kotlinx.serialization. This is a very common pattern in Android development.

  Imagine we want to fetch a user from the JSONPlaceholder API.

  Step 1: Define the Data Class

  First, define a data class that mirrors the JSON structure.

```
import kotlinx.serialization.Serializable
@Serializable
data class User(
    val id: Int,
    val name: String,
    val username: String,
    val email: String
)
```

Putting It All Together

- Step 2: Define the API Interface with Retrofit
- Create an interface that describes the API endpoints. Retrofit uses annotations to turn this interface into a working HTTP client. A suspend function indicates that this is an asynchronous call that should be run inside a coroutine.

```
import retrofit2.http.GET
import retrofit2.http.Path
interface ApiService {
    @GET("users/{userId}") // The endpoint path
    suspend fun getUser(@Path("userId") id:
Int): User // The function to call it
}
```

Putting It All Together

- Step 3: Create the API Client
- Now, build the Retrofit instance. You tell it the base URL of the API and which library to use for converting JSON into your data classes.

```
import com.jakewharton.retrofit2.converter.kotlinx.serialization.asConverterFactory
import kotlinx.serialization.json.Json
import okhttp3.MediaType.Companion.toMediaType
import retrofit2.Retrofit
// The base URL for all API calls
val baseUrl = "https://jsonplaceholder.typicode.com/"
// The JSON converter
val json = Json { ignoreUnknownKeys = true } // ignore keys in JSON not in our data class
// The Retrofit instance
val retrofit = Retrofit.Builder()
    .baseUrl(baseUrl)
    .addConverterFactory(json.asConverterFactory("application/json".toMediaType()))
    .build()
// Create an implementation of our API service
val apiService = retrofit.create(ApiService::class.java)
```

Putting It All Together

- Step 4: Make the API Call
- Finally, use a coroutine to make the network request. Network calls can't run on the main thread, so coroutines are perfect for this.

```
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking
fun main() = runBlocking { // Start a coroutine scope
    launch {
        try {
            // Make the actual network request
            val user = apiService.getUser(1)
            println("Successfully fetched user: ${user.name} (${user.email})")
        } catch (e: Exception) {
            println("Error fetching user: ${e.message}")
        }
    }
}
```

- // Expected Output:
- // Successfully fetched user: Tushar Kute (tushar@tusharkute.com)

Summary

- This example demonstrates the core workflow:
  - Model your data with a data class.
  - Describe your API with an interface.
  - Build a client to handle requests and parsing.
  - Execute the call asynchronously using coroutines and handle potential errors.

Retrofit and OkHttp

- Retrofit and OkHttp are two powerful libraries from Square that work together to make network calls in your app robust and easy to manage. They have distinct but complementary roles.
  - OkHttp is the engine. It's a powerful and efficient HTTP client that handles the low-level work of making requests and receiving responses over the network. It manages things like connection pooling, response caching, retries, and handling network failures.
  - Retrofit is the dashboard. It's a type-safe REST client that sits on top of OkHttp. It lets you define your API as a simple Kotlin interface with annotations, turning complex HTTP API calls into straightforward function calls.
- You almost always use them together. Retrofit provides the clean API, and OkHttp does the heavy lifting in the background.

How They Work Together

- The standard workflow is to create and configure an OkHttpClient instance and then pass it to your Retrofit.Builder.
- This gives you full control over the networking layer while keeping your API call definitions clean.
- The key benefit of this separation is customization. You can configure OkHttp with powerful features like interceptors without cluttering your Retrofit API definitions.

How They Work Together

- Interceptors are a core feature of OkHttp. They are powerful middleware that can observe, modify, and even short-circuit requests going out and responses coming back. Common uses include:
  - Logging: To log all request and response details for debugging.
  - Authentication: To automatically add an API key or an Authorization header to every request.
  - Caching: To define how responses should be cached.
  - Retrying: To automatically retry failed requests.

Example:

- Practical Example: Adding a Logging Interceptor
  - Let's expand on the previous example by building a custom OkHttpClient with a logging interceptor and attaching it to Retrofit.
  - This is one of the most common and useful configurations.

Example:

- Step 1: Add Dependencies
- You'll need dependencies for Retrofit, OkHttp, and OkHttp's logging interceptor in your build.gradle.kts file.

```
// For Retrofit
implementation("com.squareup.retrofit2:retrofit:2.9.0")
// For OkHttp and its logging interceptor
implementation("com.squareup.okhttp3:okhttp:4.12.0")
implementation("com.squareup.okhttp3:logging-
interceptor:4.12.0")
// A JSON converter (e.g., kotlinx.serialization)
implementation("com.jakewharton.retrofit2.converter.kotlinx.s
erialization:retrofit2-kotlinx-serialization-
converter:1.0.0")
implementation("org.jetbrains.kotlinx:kotlinx-serialization-
json:1.6.0")
```

Example:

- Step 2: Create a Custom OkHttpClient
- Here, we create an OkHttpClient and add the HttpLoggingInterceptor, which will print detailed logs of our network calls to Logcat.

```
import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
// 1. Create the logging interceptor
val loggingInterceptor = HttpLoggingInterceptor().apply {
    level = HttpLoggingInterceptor.Level.BODY // Log request and
response bodies
}
// 2. Create the OkHttp client and add the interceptor
val okHttpClient = OkHttpClient.Builder()
    .addInterceptor(loggingInterceptor)
    // You can add other configurations here, like timeouts
    // .connectTimeout(30, TimeUnit.SECONDS)
    .build()
```

Example:

- Step 3: Build Retrofit with the Custom Client
- Now, simply pass your custom okHttpClient to the Retrofit.Builder using the .client() method.

```
import
com.jakewharton.retrofit2.converter.kotlinx.serialization.asConverterFactory
import kotlinx.serialization.json.Json
import okhttp3.MediaType.Companion.toMediaType
import retrofit2.Retrofit
val retrofit = Retrofit.Builder()
    .baseUrl("https://jsonplaceholder.typicode.com/")
    .client(okHttpClient) // <-- Attach our custom OkHttp client here!

.addConverterFactory(Json.asConverterFactory("application/json".toMediaType(
)))
    .build()
// The rest of your code (defining the ApiService interface, data classes,
// and making calls) remains exactly the same.
val apiService = retrofit.create(ApiService::class.java)
```

Example:

- Now, when you run your app and make an API call using this apiService, the logging interceptor will automatically print detailed information about the request (URL, method, headers, body) and the response (status code, headers, body) to your console, which is incredibly useful for debugging.

Summary

Library	Role	Key Features
OkHttp	The low-level HTTP client (engine)	Interceptors, caching, connection pooling, retries, protocol support
Retrofit	The type-safe layer on top (dashboard)	Declarative API via interfaces, automatic request/response parsing

ViewModel and LiveData

- Implementing ViewModel and LiveData is fundamental to building modern, robust Android apps. They are part of the Android Architecture Components and work together to manage UI-related data in a lifecycle-aware way.
- Think of it like this:
  - ViewModel is the brain. It holds and processes the data for the UI, surviving events like screen rotation that would normally destroy that data.
  - LiveData is the nervous system. It's a special data holder that automatically notifies the UI when the data in the ViewModel changes, but only when the UI is active and ready to be updated.

ViewModel and LiveData

- This solves two major problems:
  - Data loss on configuration changes:
    - When you rotate your phone, Android destroys and recreates your Activity. A ViewModel preserves your data across this change.
  - Memory leaks and crashes:
    - LiveData is aware of the UI's lifecycle. It won't try to update a screen that has been closed, preventing common crashes and memory leaks.

The ViewModel: Storing Your Data

- A ViewModel is a simple class designed to store and manage UI-related data. Its lifecycle is tied to a UI controller (like an Activity or Fragment), but it outlives the specific instances of that controller.
- The key principle is separation of concerns. The ViewModel handles the data, while the Activity/Fragment handles displaying the data and capturing user input.
- A common pattern is to expose data from the ViewModel using LiveData.

LiveData: Observing Your Data

- LiveData is an observable data holder. "Observable" means your UI can "subscribe" or "observe" it. Whenever the data held by LiveData changes, it notifies all its active observers.
- A crucial best practice is to use:
  - MutableLiveData: This is an editable version of LiveData (it has value and postValue methods). You should keep this private within your ViewModel so only the ViewModel can change the data.
  - LiveData: This is the public, read-only version. You expose this to your UI (Activity/Fragment) so it can observe the data but cannot change it directly. This enforces a clean data flow.

A Practical Example: A Simple Counter App

- Let's build a simple app with a button that increments a number displayed in a TextView.
- Step 1: Add Dependencies

  Make sure you have the necessary libraries in your app's build.gradle.kts file:

```
// ViewModel
implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.3")
// LiveData
implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.8.3")
// Fragment KTX for the 'by viewModels()' delegate
implementation("androidx.fragment:fragment-ktx:1.8.1")
```

A Practical Example: A Simple Counter App

- Step 2: Create the ViewModel

```
This class will hold the counter's value.
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
class CounterViewModel : ViewModel() {
    // 1. Private MutableLiveData that can be changed within the ViewModel
    private val _count = MutableLiveData<Int>()
    // 2. Public, read-only LiveData that the UI will observe
    val count: LiveData<Int>
        get() = _count
    init {
        // Initialize the counter value
        _count.value = 0
    }
    // 3. A function the UI can call to update the data
    fun incrementCount() {
        _count.value = (_count.value ?: 0) + 1
    }
}
```

A Practical Example: A Simple Counter App

- Step 3: Use the ViewModel and LiveData in Your Fragment/Activity

```
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.TextView
import androidx.fragment.app.Fragment
import androidx.fragment.app.viewModels
class CounterFragment : Fragment(R.layout.fragment_counter) {
    // 1. Get a reference to the ViewModel using the 'by viewModels()' delegate.
    // This automatically handles creating/retaining the ViewModel instance.
    private val viewModel: CounterViewModel by viewModels()
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        val countTextView: TextView = view.findViewById(R.id.countTextView)
        val incrementButton: Button = view.findViewById(R.id.incrementButton)
        // 2. Observe the LiveData.
        // The code inside the observer lambda will run whenever the count changes
        // AND this Fragment is in an active state.
        viewModel.count.observe(viewLifecycleOwner) { newCount ->
            // Update the UI with the new data
            countTextView.text = newCount.toString()
        }
        // 3. Set up a click listener to call the ViewModel's function.
        incrementButton.setOnClickListener {
            viewModel.incrementCount()
        }
    }
}
```

The Data flow

- The flow is simple and unidirectional:
  - User Action: The user presses the incrementButton.
  - UI to ViewModel: The button's onClickListener calls the viewModel.incrementCount() method.
  - ViewModel Logic: The ViewModel updates the private _count (MutableLiveData).
  - LiveData to UI: Because its value changed, the LiveData object automatically notifies its observer in CounterFragment.
  - UI Update: The observer's code (countTextView.text = ...) runs, and the screen updates with the new count.

The Data flow

- If you rotate the screen, the CounterFragment instance is destroyed and recreated, but the ViewModel instance persists.
- The new Fragment instance just reconnects to the existing ViewModel and immediately gets the current count (1, 2, 3, etc.), preventing any data loss.

Thank you

@mitu_skillologies

@mITuSkillologies

@mitu_group

@mitu-skillologies

@MITUSkillologies

kaggle

@mituskillologies

Web Resources
https://mitu.co.in
http://tusharkute.com

@mituskillologies

contact@mitu.co.in

tushar@tusharkute.com