COMP304
Summer 2025

COMP304
Summer 2025

Android App Architecture

Objectives:

- ❑ **Discuss Android Networking Capabilities**
- ❑ **Perform network calls with Retrofit**
- ❑ **Apply Kotlin coroutines to perform asynchronous network requests in Android apps**

Introduction to Android Networking

- Networking in Android refers to the capability of Android applications to connect over a network to send or receive data.
  - This could involve interacting with a remote server or API, downloading or uploading files, etc.
- Importance of internet data in modern Android applications.
  - Common uses: fetching data, interacting with APIs, real-time updates.
- Tools and libraries commonly used: HttpURLConnection, Retrofit, OkHttp, Volley

Android Networking Permissions

- Android requires applications to **declare permissions in the manifest file to access the internet or check network state**, ensuring that the user's privacy preferences are respected.
- Declare internet permission in **AndroidManifest.xml**:

```
<uses-permission android:name="android.permission.INTERNET" />
```

- Importance of checking permissions at runtime for sensitive data.
- Best practices for handling permissions responsibly.

Basic Network Operations with HttpURLConnection

- **HttpURLConnection** is a Java class **used for sending and receiving data over the web**.
- It's part of the Java standard library and is used for handling HTTP requests in a relatively low-level, manual way.
- The following code creates a simple network request to fetch data.
  - Handling network responses.
  - Perform a GET request:

```
val url = URL("http://example.com")
val urlConnection = url.openConnection() as HttpURLConnection
try {
    val inputStream: InputStream = urlConnection.inputStream
    // process input stream
} finally {
    urlConnection.disconnect()
}
```

Handling POST Requests with HttpURLConnection

- Demonstrates how to send data to a server by making POST requests using HttpURLConnection, which involves setting the request method to "POST" and writing data to the request body.
  - Differences between GET and POST requests.
  - Constructing a POST request with HttpURLConnection.
    - Setting request method and headers.
    - Sending data to the server.
- Example of sending a POST request:

Handling POST Requests with HttpURLConnection

```
val url = URL("http://example.com/api/data")
val urlConnection = url.openConnection() as HttpURLConnection
urlConnection.requestMethod = "POST"
urlConnection.doOutput = true
urlConnection.outputStream.write(postDataBytes)
urlConnection.connect()
```

Introduction to Retrofit

- Retrofit is a **type-safe REST client for Android** (and Java) developed by Square.
- It simplifies the process of interacting with RESTful web services, turning your HTTP API into a Java interface.
- Advantages over HttpURLConnection: simplicity, scalability, maintainability.
- Setting up Retrofit with a converter (e.g., Gson).
- Defining a service interface for API calls.
- Example of Retrofit configuration:

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

Creating API Interfaces with Retrofit

- Retrofit works by **defining an interface for HTTP operations**.
- **Annotations on the interface methods** and its parameters indicate how requests are made and data is sent.
  - Defining endpoints using annotations.
  - Passing parameters to queries.
  - Handling synchronous and asynchronous calls.
- Example API interface:

```
interface ApiService {
    @GET("users/{user}/repos")
    fun listRepos(@Path("user") user: String): Call<List<Repo>>
}
```

Making Asynchronous Requests with Retrofit

- Asynchronous execution is used to prevent blocking the main thread during network operations, crucial for maintaining a smooth user interface.
- Importance of asynchronous operations in networking.
  - Using Call<T> and Callback<T> to handle asynchronous requests.
  - Handling responses and failures.
- Example of an asynchronous call:

Making Asynchronous Requests with Retrofit

```
val call: Call<List<Repo>> = apiService.listRepos("octocat")
call.enqueue(object : Callback<List<Repo>> {
    override fun onResponse(call: Call<List<Repo>>, response:
Response<List<Repo>>) {
        if (response.isSuccessful) {
            // Handle successful response
        }
    }

    override fun onFailure(call: Call<List<Repo>>, t: Throwable) {
        // Handle error
    }
})
```

Error Handling in Retrofit

- Handling errors involves **interpreting the response object to decide if the request was successful** or if errors like a network issue or server problem occurred.
- Common errors in network calls: HTTP errors, network failures, serialization issues.
- Handling errors using Response<T> and checking isSuccessful.
- Custom error handling strategies.
- Example of error handling:

```
if (response.isSuccessful) {
    val repos = response.body()
} else {
    val errorBody = response.errorBody()
}
```

Introduction to OkHttp

- **OkHttp** is another library from Square, designed to be an **efficient HTTP client**.
- It **handles network requests more directly** and can be used as the engine driving Retrofit underneath or standalone.
- OkHttp as a **low-level HTTP client**.
- Advantages: speed, efficiency, direct control over network operations.
- Configuring OkHttp with **logging**, **caching**, and **timeouts**.
- Example of setting up OkHttp client:

```
val client = OkHttpClient.Builder()
    .addInterceptor(HttpLoggingInterceptor().setLevel(Level.BODY))
    .connectTimeout(30, TimeUnit.SECONDS)
    .build()
```

Making Requests with OkHttp

- Constructing and sending HTTP requests using OkHttp.
- Handling responses and parsing data.
- Example of a GET request with OkHttp:

```
val request = Request.Builder()
    .url("https://api.example.com/data")
    .build()

client.newCall(request).execute().use { response ->
    if (!response.isSuccessful) throw IOException("Unexpected
code $response")
```

Introduction to Volley

- Volley is a networking library developed by Google that manages network requests queue automatically, facilitating the fetching and uploading of data over the network.
- Overview of Volley library for network operations.
- Features: automatic scheduling, multiple concurrent network connections.
- Setting up Volley in an Android project.
- Example of initializing a Volley request queue:

```
val queue = Volley.newRequestQueue(context)
```

Making Requests with Volley

- Use Volley to perform **simple network operations like a GET request**, with automatic thread management and response handling.
  - Creating a simple **StringRequest**.
  - Handling responses and errors with Volley.
- Example of using Volley for a GET request:

```
val stringRequest = StringRequest(Request.Method.GET, url,
    Response.Listener<String> { response ->
        // Display the first 500 characters of the response string.
        textView.text = "Response is: ${response.substring(0, 500)}"
    },
    Response.ErrorListener { error -> textView.text = "That didn't work!" })
queue.add(stringRequest)
```

Advanced Features of Volley

- Explores deeper functionalities of Volley such as custom requests, setting request priorities, and caching responses for efficiency.
  - Custom requests: **creating a custom request class for specific needs**.
  - **Setting priorities** of requests.
  - **Caching responses** to improve performance.
- Example of a custom Volley request:

```
class CustomRequest(
    method: Int,
    url: String,
    private val headers: Map<String, String>,
    private val listener: Response.Listener<JSONObject>,
    errorListener: Response.ErrorListener
) : JsonObjectRequest(method, url, null, listener, errorListener) {
    override fun getHeaders(): Map<String, String> {
      return headers
    }
}
```

Handling JSON with Networking Libraries

- Parsing JSON data is a common format for data exchange in networking.
- Libraries like Gson can be used with Retrofit to automate the conversion process between JSON strings and Java/Kotlin objects.
  - Parsing JSON data from network responses.
  - Using Gson with Retrofit to automatically convert JSON to Java/Kotlin objects.
- Example of Gson converter with Retrofit:

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

Security Best Practices in Android Networking

- Some of the best practices for securing network communications, include using HTTPS, managing API keys securely, and implementing features like certificate pinning.
  - Importance of using HTTPS for secure data transmission.
  - Tips on securing API keys and sensitive data.
  - Handling user data responsibly.
  - Using certificate pinning to prevent man-in-the-middle attacks.
- Example of certificate pinning with OkHttp:

```
val client = OkHttpClient.Builder()
    .certificatePinner(CertificatePinner.Builder()
        .add("hostname.com",
"sha256/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=")
        .build())
    .build()
```

Networking in the Background

- **WorkManager** is a part of the Android Jetpack library suite that **manages deferred and reliable background work**.
  - It **handles API calls or data synchronization tasks** that should continue even if the app is closed or the device is restarted.
  - Use WorkManager for deferred and reliable background work.
  - Set up WorkManager with network constraints.
- Example of a network-bound work request:

```
val constraints = Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .build()
val uploadWorkRequest =
OneTimeWorkRequest.Builder(UploadWorker::class.java)
    .setConstraints(constraints)
    .build()
WorkManager.getInstance(context).enqueue(uploadWorkRequest)
```

Best Practices for Efficient Networking

- Tips and strategies to reduce the impact of networking on user experience and device resources:
  - Minimizing the number of network calls
  - Using efficient data structures and algorithms for data handling
  - Caching strategies to reduce network use.
  - Monitoring and optimizing network performance

Testing Networking Code in Android

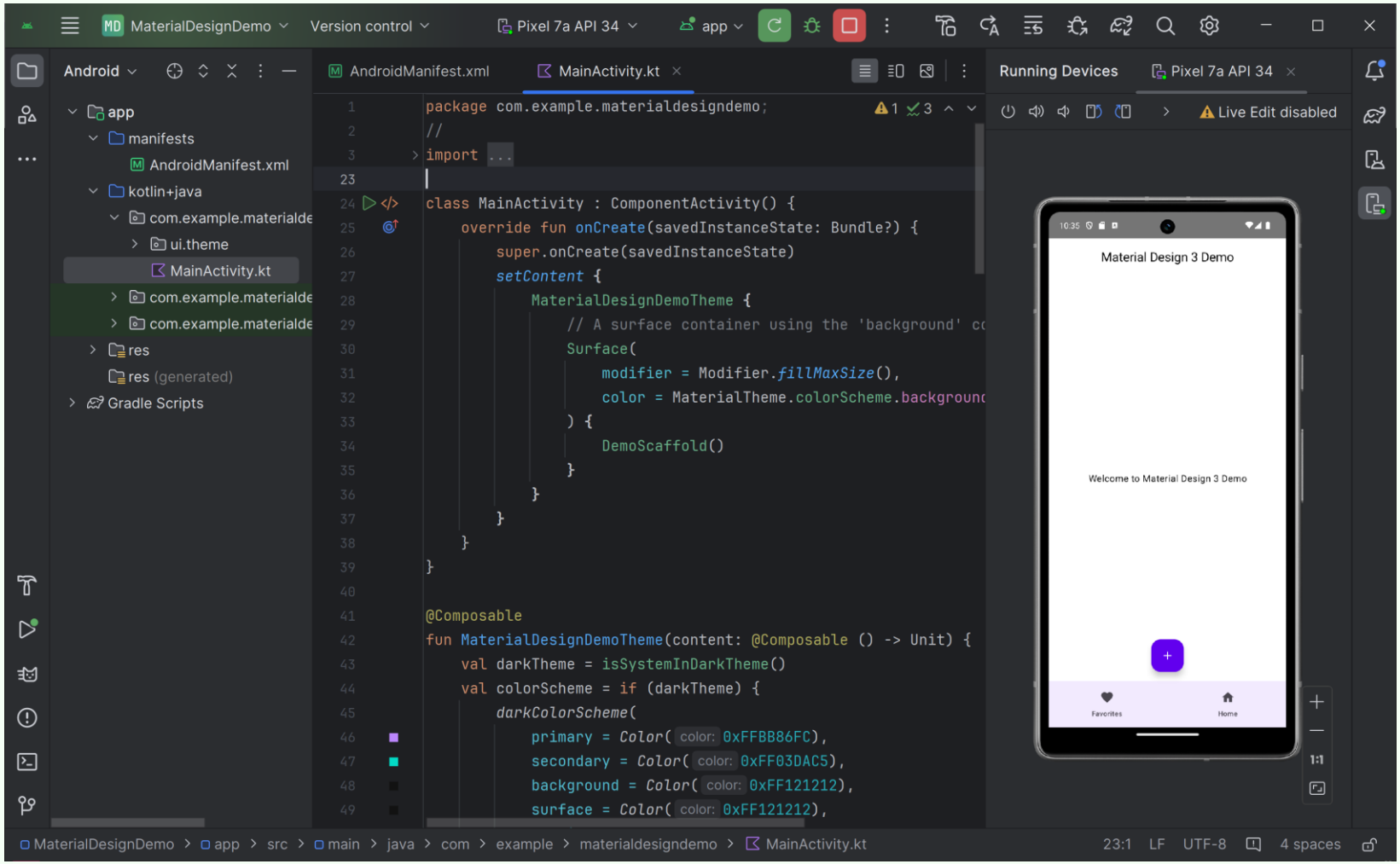
- Methodologies and tools to test networking logic in Android apps, ensuring that network interactions work as expected under various conditions:
  - Unit testing with **MockWebServer**.
  - Integration testing strategies.
  - Tools for performance and security testing.
- Example of setting up MockWebServer for tests:

```
val server = MockWebServer()
server.start()
val url = server.url("/v1/chat/")
// Perform network operations using the URL
```

Libraries vs. Native API Usage

- The pros and cons of using third-party libraries versus native APIs for network operations are influenced by project needs and developer preference.
  - When to use native APIs like HttpURLConnection.
  - Advantages of using third-party libraries like Retrofit and OkHttp.
  - Deciding factors: project size, complexity, and maintenance concerns.

chapter06 Example



References

- Textbook Chapter 6